Prólogo de James O. Coplien

Robert C. Martin
Coautor del ***Manifiesto Ágil***

Código Limpio

Manual de estilo para el desarrollo ágil de software

Cada año, se invierten innumerables horas y se pierden numerosos recursos debido a código mal escrito, ralentizando el desarrollo, disminuyendo la productividad, generando graves fallos e incluso pudiendo acabar con la organización o empresa.

El reconocido experto de software Robert C. Martin, junto con sus colegas de Object Mentor, nos presentan sus óptimas técnicas y metodologías ágiles para limpiar el código sobre la marcha y crearlo de forma correcta, de este modo mejorará como programador.

Esta obra se divide en tres partes. La primera describe los principios, patrones y prácticas para crear código limpio. La segunda incluye varios casos de estudio cuya complejidad va aumentando. Cada ejemplo es un ejercicio de limpieza y transformación de código con problemas. La tercera parte del libro contiene una lista de heurística y síntomas de código erróneo (smells) confeccionada al crear los casos prácticos. El resultado es una base de conocimientos que describe cómo pensamos cuando creamos, leemos y limpiamos código.

Imprescindible para cualquier desarrollador, ingeniero de software, director de proyectos, jefe de equipo o analista de sistemas interesado en crear código de mejor calidad.

¡El libro que todo programador debe leer!

Robert Cecil Martin

Código limpio

Manual de estilo para el desarrollo ágil de software

ePub r1.1
XcUiDi 21.03.2018

Título original: *Clean code: A handbook of agile software craftsmanship*
Robert Cecil Martin, 2009
Traducción: José Luis Gómez Celador
Ilustraciones: Jeniffer Kohnke & Angela Brooks

Editor digital: XcUiDi

Colaborador: Mario J. C. (PDF-Español)
ePub base r1.2

Este libro se ha maquetado siguiendo los estándares de calidad de www.epublibre.org. La página, y sus editores, no obtienen ningún tipo de beneficio económico por ello. Si ha llegado a tu poder desde otra web debes saber que seguramente sus propietarios sí obtengan ingresos publicitarios mediante archivos como este

Para Ann Marie: El verdadero amor de mi vida.

Agradecimientos

Me gustaría dar las gracias a mis dos artistas, Jeniffer Kohnke y Angela Brooks. Jennifer es la encargada de las impresionantes ilustraciones del inicio de cada capítulo y también de los retratos de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers y el mío propio.

Angela se encarga de las ilustraciones internas de los capítulos. Ha realizado muchos dibujos para mí en los últimos años, incluidos muchos de los del libro *Agile Software Development: Principles, Patterns, and Practices*. También es mi primogénita.

Un agradecimiento especial a los revisores Bob Bogetti, George Bullock, Jeffrey Overbey y especialmente Matt Heusser. Han sido increíbles. Han sido inmisericordes. Han sido minuciosos. Me han forzado al máximo para realizar las mejoras necesarias.

Gracias a mi editor, Chris Guzikowski, por su apoyo, aliento y amistad. Gracias a todo el personal de la editorial, incluida Raina Chrobak, que se encargó de que fuera honesto y cumpliera los plazos.

Gracias a Micah Martin y a todos los de 8th Light (www.8thlight.com) por sus críticas y su apoyo.

Gracias a todos los Object Mentor, pasados, presentes y futuros, incluidos Bob Koss, Michael Feathers, Michael Hill, Erik Meade, Jeff Langr, Pascal Roy, David Farber, Brett Schuchert, Dean Wampler, Tim Ottinger, Dave Thomas, James Grenning, Brian Button, Ron Jeffries, Lowell Lindstrom, Angelique Martin, Cindy Sprague, Libby Ottinger, Joleen Craig, Janice

Brown, Susan Rosso y el resto.

Gracias Jim Newkirk, mi amigo y socio, que me ha enseñado más de lo que cree. Mi agradecimiento a Kent Beck, Martin Fowler, Ward Cunningham, Bjarne Stroustrup, Grady Booch y todos mis mentores, compatriotas y colegas. Gracias a John Vlissides por estar ahí cuando lo necesitaba. Gracias a todos los de Zebra por permitirme despotricar sobre la extensión que debe tener una función.

Y, por último, darle las gracias por leer estos agradecimientos.

Prólogo

Una de nuestras golosinas preferidas en Dinamarca es *Ga-Jol*, con un fuerte sabor a regaliz, que constituye un complemento perfecto para nuestro húmedo y frío clima. Parte del encanto de *Ga-Jol* para los daneses es la frase que suele incluir en el envoltorio. Esta mañana compré un paquete de dos y me encontré con este antiguo dicho danés:

Ærlighed i små ting er ikke nogen lille ting.

«La honestidad por las cosas pequeñas no es algo menor». Perfecto para que lo que pensaba escribir. Las cosas pequeñas importan. Este libro trata sobre humildes preocupaciones cuyo valor dista mucho de ser menor.

Dios está en los detalles, afirmó el arquitecto Ludwig mies van der Rohe. Esta cita recuerda argumentos contemporáneos sobre el papel de la arquitectura en el desarrollo de *software*, en especial en el universo ágil. Bob y yo hemos tenido esta conversación muchas veces. Y sí, mies van der Rohe se fijaba en la utilidad y la forma atemporal de la construcción que subyace a las grandes creaciones arquitectónicas. Por otra parte, seleccionaba personalmente los pomos de todas las puertas de todas las casas que diseñaba. ¿Por qué? Porque las cosas pequeñas importan.

En nuestro interminable debate sobre TDD, Bob y yo coincidimos en que la arquitectura del *software* desempeña una importante labor en el desarrollo, aunque tenemos diferentes visiones de lo que esto significa. Estas diferencias carecen de importancia, ya que podemos aceptar que los profesionales responsables dedican parte de su tiempo a planificar un proyecto antes de comenzarlo. Las nociones de diseño controlado únicamente por pruebas y el código, propias de finales de la década de 1990, ya no son válidas. Y la atención al detalle es un pilar fundamental de los profesionales, casi como cualquier visión. Por un lado, la práctica en los detalles otorga dominio a los profesionales, y aumenta su confianza para la práctica a mayor escala. Por otra parte, el más mínimo fallo de construcción, una puerta que no cierre bien o un baldosín mal colocado, acaba con el encanto del todo. De eso se trata el código limpio.

Pero la arquitectura es sólo una metáfora del desarrollo de *software* y en concreto de la parte del *software* que ofrece el producto inicial, de la misma forma que un arquitecto entrega un edificio inmaculado. Hoy en día, el objetivo es comercializar rápidamente los productos. Queremos que las fábricas produzcan *software* a pleno rendimiento. Se trata de fábricas humanas, programadores que piensan, que sienten y que trabajan para crear un producto. La metáfora de la manufacturación es incluso más evidente en este pensamiento. Los aspectos productivos de las fábricas de automóviles japonesas fueron una gran inspiración para Serum.

Pero incluso en la industria automovilística, gran parte del trabajo no radica en la fabricación sino en el mantenimiento, o más bien en cómo evitarlo. En el *software*, el 80 por 100 o más de lo que hacemos se denomina cuantitativamente mantenimiento, el acto de reparar. En lugar de optar por la

típica costumbre occidental de crear *software* de calidad, debemos pensar como reparadores o mecánicos. ¿Qué piensan los directores japoneses de todo esto?

En 1951, un enfoque de calidad denominado TPM (*Total Productive Maintenance* o Mantenimiento productivo total) apareció en escena. Se centraba en el mantenimiento y no en la producción. Uno de los pilares de TPM es el conjunto de principios denominados 5S, una serie de disciplinas. Estos principios 5S son en realidad la base Lean, otro conocido término en la escena occidental, y cada vez más presente en el mundo del *software*. Estos principios no son opcionales. Como indica Uncle Bob, la práctica del *software* correcto requiere disciplina. No siempre se trata de hacer, de producir a la velocidad óptima.

La filosofía 5S incluye estos conceptos:

- ***Seiri* u organización**: Es fundamental saber dónde están las cosas, mediante enfoques como el uso de nombres correctos. ¿Cree que los nombres de los identificadores no son relevantes? Lea los siguientes capítulos.
- ***Seiton* o sistematización**: Existe un antiguo dicho norteamericano: un sitio para cada cosa y cada cosa en su sitio. Un fragmento de código debe estar donde esperamos encontrarlo; en caso contrario, refactorice hasta conseguirlo.
- ***Seiso* o limpieza**: Mantenga limpio el lugar de trabajo. ¿Qué dicen los autores sobre inundar el código de comentarios y líneas que capturan historias o deseos futuros? Elimínelos.
- ***Seiketsu* o estandarización**: El grupo decide cómo mantener limpio el lugar de trabajo. ¿Cree que este libro habla sobre tener un estilo de código coherente y una serie de prácticas dentro del grupo? ¿De dónde provienen esos estándares? Siga leyendo.
- ***Shutsuke* o disciplina**: Significa ser disciplinado en la aplicación de las prácticas y reflejarlas en el trabajo y aceptar los cambios.

Si acepta el reto, ha leído bien, el reto, de leer y llevar a la práctica este libro, podrá comprender y apreciar el último punto. Aquí nos acercamos a la

raíz de la profesionalidad responsable de una profesión que debería preocuparse del ciclo vital de un producto. Al igual que mantenemos coches y otras máquinas, el mantenimiento divisible, esperar a que surjan los errores, es la excepción. Por el contrario, ascendemos un nivel: inspeccionamos diariamente las máquinas y arreglamos los componentes gastados antes de que se rompan, o cambiamos el aceite cada varios miles de kilómetros para evitar problemas. En el código, debemos refactorizar sin compasión. Puede ascender otro nivel más, como hizo el movimiento TPM hace 50 años: crear máquinas que se pueden mantener mejor. Crear código legible es tan importante como crear código ejecutable. La práctica definitiva, que apareció en los círculos TPM en 1960, es la que se centra en introducir nuevas máquinas o sustituir las antiguas. Como Fred Brooks nos advirtió, deberíamos rehacer el *software* cada siete años para eliminar los problemas latentes. Tendríamos que actualizar este plazo por semanas, días e incluso horas en lugar de años. Ahí es donde se encuentra el detalle.

El detalle tiene un gran poder, y es un enfoque vital humilde y profundo, como es de esperar de cualquier enfoque de origen japonés. Pero no es sólo la visión oriental de la vida; también lo encontramos en el pueblo norteamericano. La cita *seiton* anterior proviene de la pluma de un ministro de Ohio que, literalmente, consideraba la limpieza como un remedio para todas las formas del mal. ¿Y *seiso*? *La limpieza es la pureza*. Aunque una casa sea bella, el mobiliario inadecuado acaba con su encanto. ¿Y la opinión de *shutsuke* al respecto? *El que confíe en lo pequeño confiará en lo superior*. ¿Y la predisposición a refactorizar en el momento adecuado, reforzando nuestra posición para las posteriores grandes decisiones, en lugar de dejarlo pasar? *Una puntada a tiempo ahorra ciento. Al que madruga, Dios le ayuda. No dejes para mañana lo que puedas hacer hoy* (éste era el sentido original de la frase «en el momento adecuado» de Lean hasta que cayó en manos de consultores de *software*). ¿Y sobre calibrar la importancia de los pequeños esfuerzos individuales en un todo mayor? *De pequeñas semillas crecen grandes árboles*. ¿Y la integración de sencillas tareas preventivas en la vida diaria? *Más vale prevenir que curar*. El código limpio honra las raíces de la sabiduría popular, de antes o de ahora, y se puede aplicar con atención al detalle.

Incluso en la literatura arquitectónica encontramos ejemplos de estos detalles. Piense en los pomos de mies van der Rohe. Eso es *seiri*. Es la atención a todos los nombres de variables. Debe bautizar a una variable con el mismo cuidado como si fuera su primogénito.

Y como todo sabemos, este cuidado no acaba nunca. El arquitecto Christopher Alexander, padre de patrones y lenguajes de patrones, considera todo acto de diseño como un pequeño acto local de reparación, y considera la maestría de la estructura como competencia única del arquitecto; las formas mayores se ceden a los patrones y su aplicación a los habitantes. El diseño es interminable no sólo al añadir una nueva habitación a una casa, sino al prestar atención a la pintura, a cambiar las alfombras o a instalar un nuevo fregadero en la cocina. Otras artes muestran sentimientos análogos. En nuestra búsqueda por la importancia de los detalles, nos topamos con el autor francés del siglo XIX Gustav Flaubert. El poeta francés Paul Valery afirma que un poema no se acaba nunca y que tiene que retocarse continuamente, y que dejar de trabajar en el poema es señal de abandono. Tal preocupación por el detalle es común en todas las empresas de excelencia. Puede que esto no sea nada nuevo, pero al leer este libro sentirá la necesidad de adoptar disciplinas rechazadas en su momento por apatía o por un deseo de espontaneidad o una simple respuesta al cambio.

Desafortunadamente, no solemos considerar estas preocupaciones la clave del arte de la programación. Renunciamos pronto a nuestro código, no porque lo hayamos completado, sino porque nuestro sistema de valores se centra en el futuro más que en la sustancia de nuestros productos.

Esto tiene un precio final: *hierba mala nunca muere*. La investigación, ni en el mundo industrial ni en el académico, se reduce a mantener limpio el código. Cuando trabajaba en la organización Bell Labs Software Production Research (sin duda de producción) comprobamos que un estilo de sangrado coherente era uno de los mayores indicadores estadísticamente significativos de una baja densidad de errores. Queremos que una arquitectura, un lenguaje de programación u otra noción superior sea el motivo de la calidad; como seres cuya supuesta profesionalidad se debe al dominio de herramientas y métodos de diseño, nos sentimos insultados por el valor que los programadores añaden con tan sólo aplicar un estilo de sangrado coherente.

Para citar mi propio libro de hace 17 años, dicho estilo distingue la excelencia de la simple competencia. La visión japonesa comprende el verdadero valor del trabajador cotidiano y, en especial, de los sistemas de desarrollo que dependen de las sencillas acciones diarias de tales trabajadores. La calidad es el resultado de un millón de acciones cuidadosas, no de un método magnífico caído del cielo. Que dichas acciones sean simples no significa que sean simplistas, y mucho menos que sean sencillas. Son la base de la grandeza y, cada vez más, de cualquier empresa humana. Ignorarlas no es humano en absoluto.

Evidentemente, todavía defiendo el pensamiento global, en especial el valor de los enfoques arquitectónicos cimentados en el conocimiento de los dominios y la capacidad de uso del *software*. Este libro no versa sobre esto, al menos no de forma evidente. Este libro transmite un mensaje más sutil cuya profundidad no debe menospreciarse. Coincide con la visión de gente como Peter Sommerlad, Kevlin Henney y Giovanni Asproni, cuyos mantras son «El código es el diseño» y «Código simple». Aunque debemos recordar que la interfaz es el programa y que sus estructuras dicen mucho sobre la propia estructura del programa, es fundamental adoptar de forma continuada la humilde posición de que el diseño vive en el código. Y aunque los cambios y la metáfora de la fábrica supongan costes, los cambios de diseño suponen valor. Debemos considerar al código como la articulación de los esfuerzos de diseño, visto como un proceso, no como algo estático. Es en el código donde se desarrollan los conceptos arquitectónicos de conexión y cohesión. Si escucha a Larry Constantine describir la conexión y la cohesión, lo hace en términos del código, no desde conceptos abstractos propios de UML. En su ensayo *Abstraction Descant*, Richard Gabriel afirma que la abstracción es el mal. El código es el remedio al mal y el código limpio puede que sea divino.

Volviendo a mi caja de *Ga-Jol*, considero importante recordar que la sabiduría danesa nos recomienda no sólo prestar atención a las pequeñas cosas, sino también ser *honestos* con ellas. Esto significa ser honesto con el código, con nuestros colegas sobre el estado del código y, en especial, con nosotros mismos. ¿Hemos hecho todo lo posible para dejar las cosas mejor que como las encontramos? ¿Hemos refactorizado el código antes de terminarlo? No se trata de preocupaciones periféricas, sino que se encuentran

en la base misma de los valores Agile. En Serum se recomienda que la refactorización sea parte del concepto de Terminado. Ni la arquitectura ni el código limpio insisten en la perfección, sino en la honestidad y en hacerlo lo mejor posible. *Errar es humano; perdonar es divino*. En Serum, todo lo hacemos de forma visible. Aireamos los trapos sucios. Somos honestos sobre el estado de nuestro código ya que nunca es perfecto. Nos hemos hecho más humanos, más merecedores de lo divino y estamos más próximos a la grandeza de los detalles.

En nuestra profesión, necesitamos desesperadamente toda la ayuda posible. Si un suelo seco reduce el riesgo de resbalones y las herramientas bien organizadas aumentan la productividad, es nuestra meta. Y en cuanto al libro, es la mejor aplicación pragmática de los principios Lean de *software* que he visto nunca en formato impreso. No esperaba menos de este grupo de individuos que durante años se han esforzado no sólo por mejorar sino en ofrecer sus conocimientos a la industria mediante obras como la que ahora tiene entre manos. Hace que el mundo sea un poco mejor que antes de que Uncle Bob me enviara el manuscrito.

Y tras completar este ejercicio, me dispongo a limpiar mi escritorio.

James O. Coplien
Mørdrup, Dinamarca

Introducción

Reproducido con permiso de Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m. © 2008 Focus Shift.

¿Qué puerta representa su código? ¿Qué puerta representa a su equipo o a su

empresa? ¿Por qué estamos en esa habitación? ¿Es una sencilla revisión del código o hemos detectado un sinfín de problemas terribles? ¿Depuramos presas del pánico el código que pensábamos que funcionaba? ¿Los clientes huyen despavoridos y los directores nos pisan los talones? ¿Cómo asegurarnos de que abrimos la puerta correcta cuando las cosas se ponen feas? La respuesta: *la maestría*.

La maestría se consigue de dos formas: conocimientos y trabajo. Debe adquirir el conocimiento de los principios, patrones, prácticas y heurística propios de un maestro, y dominar dichos conocimientos a través de la práctica.

Puedo enseñarle la teoría de montar en bicicleta. De hecho, los conceptos matemáticos clásicos son muy sencillos. Gravedad, fricción, velocidad angular, centro de masa, etc., se pueden demostrar en menos de una página repleta de ecuaciones. Con esas fórmulas, puedo demostrar que montar en bicicleta es práctico y proporcionarle los conocimientos necesarios para conseguirlo. Pero la primera vez que se monte en una bici se caerá al suelo.

El diseño de código no es diferente. Podríamos enumerar todos los principios del código limpio y confiar en que se *encargue* del resto (es decir, dejar que se cayera de la bici) pero entonces la pregunta sería qué clase de profesores somos y qué clase de alumno sería.

No. Así no funciona este libro.

Aprender a crear código limpio es *complicado*. Requiere algo más que conocer principios y patrones. Tiene que sudar. Debe practicarlo y fallar. Debe ver cómo otros practican y fallan. Debe observar cómo se caen y recuperan el paso. Debe ver cómo agonizan en cada decisión y el precio que pagan por tomar decisiones equivocadas.

Para leer este libro, prepárese a trabajar duro. No es un libro que se pueda leer en un avión y terminarlo antes de aterrizar. Este libro le hará trabajar, y mucho. ¿Y qué tipo de trabajo? Tendrá que leer código, en abundancia. Y se le pedirá que piense en qué acierta el código y en qué falla. Se le pedirá que siga nuestras descripciones mientras despedazamos módulos y los volvemos a ensamblar. Para ello necesitará tiempo y esfuerzo, pero creemos que merece la pena.

Hemos dividido el libro en tres partes. Los primeros capítulos describen

los principios, patrones y prácticas para crear código limpio. Incluyen abundante código y resultan difíciles de leer. Sirven como preparación a la segunda parte. Si abandona tras leer la primera sección, que tenga buena suerte.

La segunda parte del libro es la más difícil. Incluye varios casos de estudio cuya complejidad va aumentando. Cada ejemplo es un ejercicio de limpieza de código, transformar código con problemas para que tenga menos problemas. El detalle de esta parte es *abundante*. Tendrá que alternar entre el texto y los listados de código. Tendrá que analizar y entender el código, y comprender el razonamiento de cada cambio realizado. Piense en que esta parte *le llevará varios días*.

La tercera parte del libro es un único capítulo que contiene una lista de heurística y síntomas de código erróneo (*smells*) confeccionada al crear los casos prácticos. Al analizar y limpiar el código de los ejemplos, documentamos el motivo de todas nuestras acciones como heurística o síntoma. Intentamos comprender nuestras reacciones al código que leíamos y modificábamos, y nos esforzamos por capturar las sensaciones que tuvimos y las decisiones que adoptamos. El resultado es una base de conocimientos que describe cómo pensamos cuando creamos, leemos y limpiamos código.

Esta base de conocimientos no le servirá de mucho si no lee atentamente los casos de la segunda parte del libro. En esos capítulos hemos anotado con precisión todos los cambios realizados con referencias a la heurística. Estas referencias se muestran entre corchetes, como [H22]. De este modo puede ver el contexto en el que se ha aplicado y creado dicha heurística. No importa tanto el propio valor de las heurísticas sino *la relación entre ellas y las decisiones adoptadas al limpiar el código en los ejemplos*.

Si lee la primera y la tercera parte y se salta los casos prácticos, habrá leído otro libro distinto sobre cómo crear código correcto, pero si dedica tiempo a analizar los casos, sigue todos y cada uno de los pasos, cada una de las decisiones, si se pone en nuestro lugar y se obliga a pensar tal y como lo hicimos nosotros, entonces comprenderá mucho mejor todos los principios, patrones, prácticas y heurística. Ya no será un conocimiento superficial. Se convertirá en algo profundo. Lo integrará de la misma forma que una bicicleta se convierte en una extensión propia una vez dominada la forma de

montar.

Sobre la imagen de cubierta

La imagen de la portada es M104: la Galaxia del Sombrero. M104 se encuentra en Virgo, a unos 30 millones de años luz, y su núcleo es un súper agujero negro que pesa aproximadamente mil millones de masas solares.

¿La imagen le recuerda la explosión de la luna Praxis de Klingon? Recuerdo la escena de Star Trek VI en la que se mostraba un anillo de restos flotando tras la explosión. Tras esa escena, el anillo se ha convertido en un elemento habitual de las explosiones de ciencia ficción. Incluso se añadió a la explosión de Alderaan en ediciones posteriores de la primera película de La Guerra de las Galaxias.

¿Qué provocó la formación de este anillo alrededor de M104? ¿Por qué tiene un centro de tales dimensiones y un núcleo tan brillante y diminuto? Parece como si el agujero negro central hubiera provocado un orificio de 30 000 años luz en el centro de la galaxia. La desgracia caería sobre las civilizaciones que se encontraran en el camino de este desastre cósmico.

Los súper agujeros negros desayunan estrellas y convierten parte de su masa en energía. $E=MC^2$ puede bastar, pero cuando M es una masa estelar hay que tener cuidado. ¿Cuántas estrellas habrá engullido este monstruo antes de saciar su apetito? El tamaño del vacío central podría ser una pista.

La imagen de M104 de la portada es una combinación de la famosa fotografía del Hubble (imagen superior) y la reciente imagen de infrarrojos del observatorio orbital Spitzer (inferior).

Esta última muestra claramente la forma de anillo de la galaxia.

A la luz, sólo vemos el borde frontal de la silueta del anillo.

La masa central oculta el resto.

Pero en la imagen de infrarrojos, las partículas calientes del anillo brillan a través de la masa central. Las dos imágenes combinadas nos ofrecen una vista desconocida hasta ahora e implican que hace tiempo era un auténtico infierno activo.

Imagen de portada: © Spitzet Space Telescope.

1
Código Limpio

Está leyendo este libro por dos motivos. Por un lado, es programador. Por otro, quiere ser mejor programador. Perfecto. Necesitamos mejores

programadores.

Este libro trata sobre programación correcta. Está repleto de código. Lo analizaremos desde todas las direcciones. Desde arriba, desde abajo y desde dentro. Cuando terminemos, sabremos mucho sobre código y, en especial sabremos distinguir entre código correcto e incorrecto. Sabremos cómo escribir código correcto y cómo transformar código incorrecto en código correcto.

Hágase el código

Se podría afirmar que un libro sobre código es algo obsoleto, que el código ya no es el problema y que deberíamos centrarnos en modelos y requisitos. Hay quienes sugieren que el final del código está próximo. Que los programadores ya no serán necesarios porque los empresarios generarán programas a partir de especificaciones.

No es cierto. El código nunca desaparecerá, ya que representa los detalles de los requisitos. En algún nivel, dichos detalles no se pueden ignorar ni abstraer; deben especificarse, y para especificar requisitos de forma que un equipo pueda ejecutarlos se necesita la programación. Dicha especificación es el código.

Espero que el nivel de abstracción de nuestros lenguajes siga aumentando. También espero que aumente el número de lenguajes específicos de dominios. Será algo positivo, pero no eliminará el código. De hecho, todas las especificaciones creadas en estos lenguajes de nivel superior y específicos de los dominios serán código, y tendrá que ser riguroso, preciso, formal y detallado para que un equipo pueda entenderlo y ejecutarlo.

El que piense que el código va a desaparecer es como el matemático que espera que un día las matemáticas no sean formales. Esperan descubrir una forma de crear máquinas que hagan lo que queramos en lugar de lo que digamos. Esas máquinas tendrían que entendernos de tal forma que puedan traducir necesidades ambiguas en programas perfectamente ejecutados que satisfagan dichas necesidades a la perfección.

Esto nunca sucederá. Ni siquiera los humanos, con toda su intuición y

creatividad, han sido capaces de crear sistemas satisfactorios a partir de las sensaciones de sus clientes. En realidad, si la disciplina de la especificación de requisitos nos ha enseñado algo es que los requisitos bien especificados son tan formales como el código y que pueden actuar como pruebas ejecutables de dicho código.

Recuerde que el código es básicamente el lenguaje en el que expresamos los requisitos en última instancia. Podemos crear lenguajes que se asemejen a dichos requisitos. Podemos crear herramientas que nos permitan analizar y combinar dichos requisitos en estructuras formales, pero nunca eliminaremos la precisión necesaria; por ello, siempre habrá código.

Código Incorrecto

Recientemente leí el prólogo del libro *Implementation Pattern*[1] de Kent Beck, donde afirmaba que «…este libro se basa en una frágil premisa: que el código correcto es relevante…». ¿Una *frágil* premisa? En absoluto. Considero que es una de las más robustas, admitidas e importantes de nuestro sector (y creo que Kent lo sabe). Sabemos que el código correcto es relevante porque durante mucho tiempo hemos tenido que sufrir su ausencia.

Sé de una empresa que, a finales de la década de 1980, creó una *magnífica* aplicación, muy popular y que muchos profesionales compraron y utilizaron. Pero los ciclos de publicación empezaron a distanciarse. No se corrigieron los errores entre una versión y la siguiente. Crecieron los tiempos de carga y aumentaron los fallos. Todavía recuerdo el día en que apagué el producto y nunca más lo volví a usar.

Poco después, la empresa desapareció.

Dos décadas después conocí a uno de los empleados de la empresa y le pregunté sobre lo que había pasado. La respuesta confirmó mis temores. Habían comercializado el producto antes de tiempo con graves fallos en el código. Al añadir nuevas funciones, el código empeoró hasta que ya no pudieron controlarlo. *El código incorrecto fue el motivo del fin de la empresa*.

¿En alguna ocasión el código incorrecto le ha supuesto un obstáculo? Si es programador seguramente sí. De hecho, tenemos una palabra que lo describe: *sortear*. Tenemos que sortear el código incorrecto. Nos arrastramos por una maraña de zarzas y trampas ocultas. Intentamos buscar el camino, una pista de lo que está pasando, pero lo único que vemos es más y más código sin sentido.

Sin duda el código incorrecto le ha supuesto un obstáculo. Entonces, ¿por qué lo escribió?

¿Tenía prisa? ¿Plazos de entrega? Seguramente. Puede que pensara que no tenía tiempo para hacer un buen trabajo; que su jefe se enfadaría si necesitaba tiempo para limpiar su código. O puede que estuviera cansado de trabajar en ese programa y quisiera acabar cuanto antes. O que viera el trabajo pendiente y tuviera que acabar con un módulo para pasar al siguiente. A todos nos ha pasado.

Todos hemos visto el lío en el que estábamos y hemos optado por dejarlo para otro día. Todos hemos sentido el alivio de ver cómo un programa incorrecto funcionaba y hemos decidido que un mal programa que funciona es mejor que nada. Todos hemos dicho que lo solucionaríamos después. Evidentemente, por aquel entonces, no conocíamos la ley de LeBlanc: *Después es igual a nunca*.

El coste total de un desastre

Si es programador desde hace dos o tres años, probablemente haya sufrido los desastres cometidos por otros en el código. Si tiene más experiencia, lo habrá sufrido en mayor medida. El grado de sufrimiento puede ser significativo. En un periodo de un año o dos, los equipos que avancen rápidamente al inicio de

un proyecto pueden acabar a paso de tortuga. Cada cambio en el código afecta a dos o tres partes del mismo. Ningún cambio es trivial. Para ampliar o modificar el sistema es necesario comprender todos los detalles, efectos y consecuencias, para de ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar. Es imposible.

Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el director hace lo único que puede: ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esa nueva plantilla no conoce el diseño del sistema. No conocen la diferencia entre un cambio adecuado al objetivo de diseño y otro que lo destroce. Por tanto, todos se encuentran sometidos a una gran presión para aumentar la productividad. Por ello, cometen más errores, aumenta el desastre y la productividad se acerca a cero cada vez más (véase la figura 1.1).

Figura 1.1. Productividad frente a tiempo.

El gran cambio de diseño

En última instancia, el equipo se rebela. Informan al director que no pueden seguir trabajando con ese código. Exigen un cambio de diseño. La dirección no requiere invertir en un cambio de diseño del proyecto, pero no puede ignorar el bajo nivel de productividad.

Acaba por ceder a las exigencias de los programadores y autoriza el gran cambio de diseño. Se selecciona un nuevo equipo. Todos quieren formar parte del nuevo equipo por ser un lienzo en blanco. Pueden empezar de cero y crear algo realmente bello, pero sólo los mejores serán elegidos para el nuevo equipo. Los demás deben continuar con el mantenimiento del sistema actual.

Ahora los dos equipos compiten. El nuevo debe crear un sistema que haga lo que el antiguo no puede. Además, deben asumir los cambios que continuamente se aplican al sistema antiguo. La dirección no sustituirá el sistema antiguo hasta que el nuevo sea capaz de hacer todo lo que hace el antiguo.

Esta competición puede durar mucho tiempo. Conozco casos de casi 10 años. Y cuando acaba, los miembros originales del equipo nuevo han desaparecido y los miembros actuales exigen un cambio de diseño del nuevo sistema porque es un desastre.

Si ha experimentado alguna fase de esta historia, ya sabrá que dedicar tiempo a que el código sea correcto no sólo es rentable, es una cuestión de supervivencia profesional.

Actitud

¿Alguna vez ha tenido que superar un desastre tan grave que ha tardado semanas en lo que normalmente hubiera tardado horas? ¿Ha visto un cambio que debería haberse realizado en una línea, aplicado en cientos de módulos distintos? Son síntomas demasiado habituales.

¿Por qué sucede en el código? ¿Por qué el código de calidad se transforma tan rápidamente en código incorrecto? Hay muchas explicaciones. Nos quejamos de que los requisitos cambian de forma que comprometen el diseño original, de que los plazos de entrega son demasiado exigentes para hacer las cosas bien. Culpamos a directores incompetentes, a usuarios intolerantes y a comerciales sin sentido. Pero la culpa, querido Dilbert, es nuestra. No somos profesionales.

Puede que resulte duro escucharlo. ¿Cómo es posible que *seamos responsables* de tales desastres? ¿Qué pasa con los requisitos? ¿Y los plazos de entrega? ¿Y los directores incompetentes y los comerciales sin sentido?

¿No es también culpa suya?

No. Los directores y los comerciales *nos* exigen la información que necesitan para realizar sus promesas y compromisos, e incluso cuando no recurren a nosotros, no debemos tener miedo a decirles lo que pensamos. Los usuarios acuden a nosotros para validar la forma de encajar los requisitos en el sistema. Los directores de proyectos acuden a nosotros para determinar los objetivos. Somos cómplices en la programación del proyecto y compartimos gran parte de la responsabilidad de los fallos, en especial si tienen que ver con código incorrecto.

Seguramente piense que, si no hace lo que su jefe le dice, le despedirán. Es improbable. Muchos jefes sólo quieren la verdad, aunque lo disimulen. Muchos quieren código correcto, aunque estén obsesionados con los objetivos. Pueden defender apasionadamente los objetivos y los requisitos, pero es su trabajo. El *nuestro* es defender el código con la misma intensidad.

Para resumir, imagine que es médico y un paciente le exige que no se lave las manos antes de una operación porque se pierde demasiado tiempo[2]. En este caso, el paciente es el jefe, pero el médico debe negarse a lo que pide. ¿Por qué? Porque el médico sabe más que el paciente sobre los riesgos de infecciones. No sería profesional (incluso sería ilegal) que el médico cediera a las exigencias del paciente.

Tampoco sería profesional que los programadores cedieran a la voluntad de los jefes que no entienden los riesgos de un posible desastre.

El enigma

Los programadores se enfrentan a un enigma de valores básicos. Los que tienen años de experiencia saben que un desastre ralentiza su trabajo, y aun así todos los programadores sienten la presión de cometer errores para poder cumplir los plazos de entrega. En definitiva, no toman el tiempo necesario para avanzar.

Los verdaderos profesionales saben que la segunda parte del enigma no es cierta. No se cumple un plazo de entrega cometiendo un error. De hecho, el error nos ralentiza de forma inmediata y hace que no lleguemos al plazo de entrega. La *única* forma de cumplirlo, la única forma de avanzar, es intentar

que el código siempre sea limpio.

¿El arte del código limpio?

Imagine que cree que el código incorrecto es un obstáculo significativo. Imagine que acepta que la única forma de avanzar es mantener el código limpio. Entonces se preguntará cómo crear código limpio. No tiene sentido intentar crearlo si no sabe lo que es.

La mala noticia es que crear código limpio es como pintar un cuadro. Muchos sabemos si un cuadro se ha pintado bien o no, pero poder reconocer la calidad de una obra no significa que sepamos pintar. Por ello, reconocer código limpio no significa que sepamos cómo crearlo.

Para crearlo se requiere el uso disciplinado de miles de técnicas aplicadas mediante un detallado sentido de la «corrección». Este sentido del código es la clave.

Algunos nacen con este sentido. Otros han de luchar para conseguirlo. No sólo permite distinguir entre código correcto e incorrecto, sino que también muestra la estrategia para aplicar nuestra disciplina y transformar código incorrecto en código correcto.

Un programador sin este sentido puede reconocer el desastre cometido en un módulo, pero no saber cómo solucionarlo. Un programador con este sentido verá las posibles opciones y elegirá la variante óptima para definir una secuencia de cambios.

En definitiva, un programador que cree código limpio es un artista que puede transformar un lienzo en blanco en un sistema de código elegante.

Concepto de código limpio

Existen tantas definiciones como programadores. Por ello, he consultado la opinión de conocidos y experimentados programadores.

Bjarne Stroustrup, inventor de C++ y autor de *The C++ Programming Language*

Me gusta que mi código sea elegante y eficaz. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser mínimas para facilitar el mantenimiento, el procesamiento de errores completo y sujeto a una estrategia articulada, y el rendimiento debe ser óptimo para que los usuarios no tiendan a estropear el código con optimizaciones sin sentido. El código limpio hace bien una cosa.

Bjarne usa la palabra «elegante». Menuda palabra.

Según el diccionario, «elegante» significa «*dotado de gracia, nobleza y sencillez*». Aparentemente Bjarne piensa que el código limpio es un placer a la hora de leerlo. Su lectura debe hacernos sonreír, como una caja de música o un coche bien diseñado.

Bjarne también menciona la eficacia, en *dos ocasiones*. No debería sorprendemos viniendo del inventor de C++; pero considero que hay algo más que el mero deseo de velocidad. Los ciclos malgastados no son elegantes, no son un placer. Y fíjese en la palabra empleada por Bjarne para describir la consecuencia de esta falta de elegancia. Usa *tiendan*. Una gran verdad. El código incorrecto *tiende* a aumentar el desastre. Cuando otros cambian código incorrecto, tienden a empeorarlo.

Dave Thomas y Andy Hunt lo expresan de forma diferente. Usan la metáfora de las ventanas rotas[3]. Un edificio con ventanas rotas parece abandonado. Y hace que otros lo abandonen. Dejan que se rompan otras ventanas. E incluso las rompen a propósito. La fachada se ensucia con pintadas y se acumula la basura. Una ventana rota inicia el proceso de la decadencia.

Bjarne también menciona que el procesamiento de errores debe ser completo, lo que se relaciona con la disciplina de prestar atención a los detalles. El procesamiento de errores abreviado es una forma de ignorar los detalles. Otras son las fugas de memoria, las condiciones de carrera o el uso

incoherente de los nombres. En definitiva, el código limpio muestra gran atención al detalle.

Bjarne termina afirmando que el *código limpio hace una cosa bien*. No es accidental que existan tantos principios de diseño de *software* que se puedan reducir a esta sencilla máxima. Muchos escritores han tratado de comunicar este pensamiento. El código incorrecto intenta hacer demasiadas cosas y su cometido es ambiguo y enrevesado. El código limpio es *concreto*. Cada función, cada clase y cada módulo muestran una única actitud que se mantiene invariable y no se contamina por los detalles circundantes.

Grady Booch, autor de *Object Oriented Analysis and Design with Applications*

El código limpio es simple y directo. El código limpio se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador, sino que muestra nítidas abstracciones y líneas directas de control.

Grady mantiene las mismas ideas que Bjarne, pero adopta una perspectiva de *legibilidad*. Me gusta especialmente que el código limpio se pueda leer como un texto bien escrito. Piense en un buen libro. Recordará que las palabras desaparecen y se sustituyen por imágenes, como ver una película.

Mejor todavía. Es ver los caracteres, escuchar los sonidos, experimentar las sensaciones.

Leer código limpio nunca será como leer *El Señor de los Anillos*. Pero esta metáfora literaria no es incorrecta. Como una buena novela, el código limpio debe mostrar de forma clara el suspense del problema que hay que resolver. Debe llevar ese suspense hasta un punto álgido para después demostrar al lector que los problemas y el suspense se han solucionado de forma evidente.

La frase «nítida abstracción» de Grady es un oxímoron fascinante. Nítido es casi un sinónimo de concreto, con un potente mensaje. El código debe ser específico y no especulativo. Sólo debe incluir lo necesario. Nuestros lectores deben percibir que hemos tomado decisiones.

«Big» Dave Thomas, fundador de OTI, el padrino de la estrategia Eclipse

El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas de unidad y de aceptación. Tiene nombres con sentido. Ofrece una y no varias formas de hacer algo. Sus dependencias son mínimas, se definen de forma explícita y ofrece una API clara y mínima. El código debe ser culto en función del lenguaje, ya que no toda la información necesaria se puede expresar de forma clara en el código.

Big Dave comparte el deseo de Grady de la legibilidad, pero con una importante variante. Dave afirma que el código limpio facilita las labores de mejora de *otros*. Puede parecer evidente pero no debemos excedernos. Después de todo, existe una diferencia entre el código fácil de leer y el código fácil de cambiar.

Dave vincula la limpieza a las pruebas. Hace 10 años esto hubiera provocado cierta controversia. Pero la disciplina del Desarrollo controlado por pruebas ha tenido un gran impacto en nuestro sector y se ha convertido en uno de sus pilares. Dave tiene razón. El código, sin pruebas, no es limpio. Independientemente de su elegancia, legibilidad y accesibilidad, si no tiene pruebas, no será limpio.

Dave usa dos veces la palabra *mínimo*. Valora el código de tamaño

reducido, una opinión habitual en la literatura de *software* desde su concepción. Cuanto más pequeño, mejor.

También afirma que el código debe ser *culto*, una referencia indirecta a la programación de Knuth[4] y que en definitiva indica que el código debe redactarse de una forma legible para los humanos.

Michael Feathers, autor de *Working Effectively with Legacy Code*

Podría enumerar todas las cualidades del código limpio, pero hay una principal que engloba a todas ellas. El código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que hacer para mejorarlo. El autor del código pensó en todos los aspectos posibles y si intentamos imaginar alguna mejora, volvemos al punto de partida y sólo nos queda disfrutar del código que alguien a quien le importa realmente nos ha proporcionado.

Una palabra; dar importancia. Es el verdadero tema de este libro, que incluso podría usar el subtítulo «*Cómo dar importancia al código*».

Michael ha acertado de pleno. El código limpio es aquél al que se le ha dado importancia. Alguien ha dedicado su tiempo para que sea sencillo y ha prestado atención a los detalles. Se ha preocupado.

Ron Jeffries, autor de *Extreme Programming Installed* y *Extreme Programming Adventures in C#*

Ron comenzó su carrera como programador con Fortran en Strategic Air Command y ha escrito código para la práctica totalidad de lenguajes y equipos. Merece la pena fijarse en sus palabras:

> En los últimos años, comencé y prácticamente terminé con las reglas de código simple de Beck.
> En orden de prioridad, el código simple:
>
> - Ejecuta todas las pruebas.
> - No contiene duplicados.
> - Expresa todos los conceptos de diseño del sistema.
> - Minimiza el número de entidades como clases, métodos, funciones y similares.
>
> De todos ellos, me quedo con la duplicación. Cuando algo se repite una y otra vez, es una señal de que tenemos una idea que no acabamos de representar correctamente en el código. Intento determinar cuál es y, después, expresar esa idea con mayor claridad. Para mí, la expresividad debe incluir nombres con sentido y estoy dispuesto a cambiar los nombres de las cosas varias veces. Con las modernas herramientas de creación de código como Eclipse, el cambio de nombres es muy sencillo, por lo que no me supone problema alguno.
> La expresividad va más allá de los nombres. También me fijo si un objeto o un método hacen más de una cosa. Si se trata de un objeto, probablemente tenga que dividirse en dos o más. Si se trata de un método, siempre recurro a la refactorización de extracción de métodos para generar un método que exprese con mayor claridad su cometido y varios métodos secundarios que expliquen cómo lo hace.
> La duplicación y la expresividad son dos factores que permiten mejorar considerablemente código que no sea limpio. Sin embargo,

existe otra cosa que también hago conscientemente, aunque sea más difícil de explicar.

Tras años en este trabajo, creo que todos los programas están formados de elementos muy similares. Un ejemplo es la búsqueda de elementos en una colección. Independientemente de que sea una base de datos de registros de empleados o un mapa de claves y valores, o una matriz de elementos, por lo general tenemos que buscar un elemento concreto de esa colección. Cuando esto sucede, suelo incluir esa implementación concreta en un método o una clase más abstractos. De ese modo disfruto de una serie de interesantes ventajas.

Puedo implementar la funcionalidad con algo sencillo, como un mapa hash, por ejemplo, pero como ahora todas las referencias a la búsqueda se ocultan en mi pequeña abstracción, puedo modificar la implementación siempre que desee. Puedo avanzar rápidamente al tiempo que conservo la posibilidad de realizar cambios posteriores.

Además, la abstracción de la colección suele captar mi atención en lo que realmente sucede e impide que implemente comportamientos de colecciones arbitrarias si lo que realmente necesito es una forma sencilla de localizar un elemento.

Reducir los duplicados, maximizar la expresividad y diseñar sencillas abstracciones en las fases iniciales. Para mí, es lo que hace que el código sea limpio.

En estos breves párrafos, Ron resume el contenido de este libro. Nada de duplicados, un objetivo, expresividad y pequeñas abstracciones. Todo está ahí.

Ward Cunningham, inventor de Wiki, Fit, y uno de los inventores de la programación eXtreme. Uno de los impulsores de los patrones de diseño. Una de las mentes tras Smalltalk y la programación orientada a objetos. El padrino de todos a los que les importa el código.

Sabemos que estamos trabajando con código limpio cuando cada rutina que leemos resulta ser lo que esperábamos. Se puede denominar código atractivo cuando el código hace que parezca que el lenguaje se ha creado para el problema en cuestión.

Este tipo de afirmaciones son características de Ward. Las leemos, asentimos y pasamos a la siguiente. Es tan razonable y evidente que apenas parece profundo. Incluso podemos pensar que es lo que esperábamos. Pero preste atención.

«… resulta ser lo que esperábamos». ¿Cuándo fue la última vez que vio un módulo que fuera más o menos lo que esperaba? ¿Lo habitual no es ver módulos complicados y enrevesados? ¿No es esta falta de concreción lo habitual? ¿No está acostumbrado a intentar extraer el razonamiento de un sistema para llegar al módulo que está leyendo? ¿Cuándo fue la última vez que leyó un código y asintió como seguramente haya hecho al leer la afirmación de Ward?

Ward espera que al leer código limpio no le sorprenda. De hecho, ni siquiera tendrá que esforzarse. Lo leerá y será prácticamente lo que esperaba. Será evidente, sencillo y atractivo. Cada módulo prepara el camino del siguiente. Cada uno indica cómo se escribirá el siguiente. Los programas limpios están tan bien escritos que ni siquiera se dará cuenta. El diseñador consigue simplificarlo todo enormemente, como sucede con todos los diseños excepcionales.

¿Y la noción de atractivo de Ward? Todos hemos criticado que nuestros lenguajes no se hayan diseñado para nuestros problemas. Pero la afirmación de Ward hace que ahora la responsabilidad sea nuestra. Afirma que *el código atractivo hace que el lenguaje parezca creado para el problema*. Por tanto, somos responsables de que el lenguaje parezca sencillo. No es el lenguaje el que hace que los programas parezcan sencillos, sino el programador que consigue que el lenguaje lo parezca.

Escuelas de pensamiento

¿Y yo (Uncle Bob)? ¿Qué es para mí el código limpio? En este libro le contaremos, con todo detalle, lo que yo y mis colegas pensamos del código limpio. Le contaremos lo que pensamos que hace que un nombre de variable, una función o una clase sean limpias.

Presentaremos estas opiniones de forma absoluta, sin disculparnos. En este punto de nuestra carrera, ya son absolutas. Son *nuestra escuela de pensamiento* del código limpio.

Los especialistas de las artes marciales no se ponen de acuerdo sobre cuál es la mejor de todas, ni siquiera sobre cuál es la mejor técnica de un arte marcial. Es habitual que los maestros de las artes marciales creen sus propias escuelas de pensamiento y los alumnos aprendan de ellos. De esta forma nació *Gracie Jiu Jitsu*, creada e impartida por la familia Gracie en Brasil; *Hakkoryu Jiu Jitsu*, fundada e impartida por Okuyama Ryuho en Tokio o *Jeet Kune Do*, fundada e impartida por Bruce Lee en Estados Unidos.

Los alumnos de estas disciplinas se sumergen en las enseñanzas del fundador. Se dedican a aprender lo que su maestro les enseña y suelen excluir las enseñanzas de otros maestros. Después, cuando han mejorado su arte, pueden convertirse en alumnos de otro maestro diferente para ampliar sus conocimientos y su experiencia. Algunos seguirán mejorando sus habilidades, descubriendo nuevas técnicas y fundando sus propias escuelas.

Ninguna de estas escuelas tiene la *razón* absoluta pero dentro de cada una actuamos como si las enseñanzas y las técnicas fueran correctas. Después de todo, existe una forma correcta de practicar Hakkoryu Jiu Jitsu o Jeet Kune Do, pero esta corrección dentro de una escuela determinada no anula las enseñanzas de otra diferente.

Imagine que este libro es una descripción de la *Escuela de mentores del código limpio*. Las técnicas y enseñanzas impartidas son la forma en la que practicamos nuestro arte. Podemos afirmar que, si sigue nuestras enseñanzas, disfrutará de lo que hemos disfrutado nosotros, y aprenderá a crear código limpio y profesional. Pero no cometa el error de pensar que somos los únicos que tenemos razón. Existen otras escuelas y otros maestros tan profesionales como nosotros, y su labor es aprender de ellos también.

De hecho, muchas de las recomendaciones del libro son controvertidas, seguramente no esté de acuerdo con muchas de ellas y puede que rechace algunas de forma definitiva. Es correcto. No somos la autoridad final. Pero, por otra parte, las recomendaciones del libro son algo en lo que hemos pensado mucho. Las hemos aprendido tras décadas de experiencia y ensayo y error. Por lo tanto, esté o no de acuerdo, sería una lástima que no apreciara, y respetara, nuestro punto de vista.

Somos autores

El campo `@author` de un Javadoc indica quiénes somos. Somos autores. Y los autores tienen lectores. De hecho, los autores son *responsables* de comunicarse correctamente con sus lectores. La próxima vez que escriba una línea de código, recuerde que es un autor y que escribe para que sus lectores juzguen su esfuerzo.

Seguramente se pregunte qué cantidad de código se lee realmente y si la mayor parte del esfuerzo no se concentra en crearlo.

¿Alguna vez ha reproducido una sesión de edición? En las décadas de 1980 y 1990 teníamos editores como Emacs que controlaban cada pulsación de tecla. Se podía trabajar durante una hora y después reproducir la sesión de edición completa como una película a alta velocidad. Cuando lo hice, los resultados fueron fascinantes.

La mayor parte de la reproducción eran desplazamientos entre módulos.

Bob accede al módulo.
Se desplaza hasta la función que tiene que cambiar.

Se detiene y piensa en las posibles opciones.
Oh, vuelve al inicio del módulo para comprobar la inicialización de una variable.
Ahora vuelve a bajar y comienza a escribir.
Vaya, borra lo que había escrito.
Vuelve a escribirlo.
Lo vuelve a borrar.
Escribe algo diferente pero también lo borra.
Se desplaza a otra función que invoca la función que está modificando para comprobar cómo se invoca.
Vuelve a subir y escribe el mismo código que acaba de borrar.
Se detiene.
Vuelve a borrar el código.
Abre otra ventana y examina las subclases. ¿Se ha reemplazado esa función?
…

Se hace una idea. En realidad, la proporción entre tiempo dedicado a leer frente a tiempo dedicado a escribir es de más de 10:1. *Constantemente* tenemos que leer código antiguo como parte del esfuerzo de crear código nuevo.

Al ser una proporción tan elevada, queremos que la lectura del código sea sencilla, aunque eso complique su creación. Evidentemente, no se puede escribir código sin leerlo, de modo que *si es más fácil de leer será más fácil de escribir*.

Es una lógica sin escapatoria. No se puede escribir código si no se puede leer el código circundante. El código que intente escribir hoy será fácil o difícil de escribir en función de lo fácil o difícil de leer que sea el código circundante. Si quiere avanzar rápidamente, terminar cuanto antes y que su código sea fácil de escribir, haga que sea fácil de leer.

La regla del Boy Scout

No basta con escribir código correctamente. El código debe limpiarse con el tiempo. Todos hemos visto que el código se corrompe con el tiempo, de modo que debemos adoptar un papel activo para evitarlo.

Los Boy Scouts norteamericanos tienen una sencilla regla que podemos aplicar a nuestra profesión:

Dejar el campamento más limpio de lo que se ha encontrado[5].

Si todos entregamos el código más limpio de lo que lo hemos recibido, no se corromperá. No hace falta que la limpieza sea masiva. Cambie el nombre de una variable, divida una función demasiado extensa, elimine elementos duplicados, simplifique una instrucción `if` compuesta.

¿Se imagina trabajar en un proyecto en el que el código *mejorara* con el tiempo? ¿Cree que hay otras opciones que puedan considerarse profesionales? De hecho, ¿la mejora continuada no es una parte intrínseca de la profesionalidad?

Precuela y principios

En muchos aspectos, este libro es una «precuela» de otro que escribí en 2002 titulado *Agile Software Development: Principles, Patterns, and Practices* (PPP). El libro PPP trata sobre los principios del diseño orientado a objetos y muchas de las técnicas empleadas por desarrolladores profesionales. Si no ha leído PPP, comprobará que continúa la historia contada en este libro. Si lo ha leído, encontrará muchas de las sensaciones de ese libro reproducidas en éste a nivel del código.

En este libro encontrará referencias esporádicas a distintos principios de diseño como SRP (*Single Responsibility Principle* o Principio de responsabilidad única), OCP (*Open Closed Principle* o Principio Abierto/Cerrado) y DIP (*Dependency Inversion Principle* o Principio de inversión de dependencias) entre otros. Todos estos principios se describen detalladamente en PPP.

Conclusión

Los libros sobre arte no le prometen que se convertirá en artista. Solamente pueden mostrarle herramientas, técnicas y procesos de pensamiento que otros artistas hayan utilizado. Del mismo modo, este libro no puede prometer que se convierta en un buen programador, que tenga sentido del código. Sólo puede mostrarle los procesos de pensamiento de buenos programadores y los trucos, técnicas y herramientas que emplean.

Al igual que un libro sobre arte, este libro está repleto de detalles. Encontrará mucho código. Verá código correcto y código incorrecto. Verá código incorrecto transformado en código correcto. Verá listas de heurística, disciplinas y técnicas. Verá un ejemplo tras otro. Y después de todo, será responsabilidad suya.

¿Recuerda el chiste sobre el violinista que se pierde camino de un concierto? Se cruza con un anciano y le pregunta cómo llegar al Teatro Real. El anciano mira al violinista y al violín que lleva bajo el brazo y le responde: «Practique joven, practique».

Bibliografía

- **[Beck07]**: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.
- **[Knuth92]**: *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

2
Nombres con sentido

por Tim Ottinger

Introducción

En el *software*, los nombres son omnipresentes. Aparecen en variables,

funciones, argumentos, clases y paquetes. Asignamos nombres a archivos y a directorios, a archivos jar, war y ear. Usamos nombres constantemente. Por ello, debemos hacerlo bien. A continuación, veremos algunas reglas básicas para crear nombres correctos.

Usar nombres que revelen las intenciones

Es fácil afirmar que los nombres deben revelar nuestras intenciones. Lo que queremos recalcar es la *importancia* de hacerlo. Elegir nombres correctos lleva tiempo, pero también ahorra trabajo. Por ello, preste atención a los nombres y cámbielos cuando encuentre otros mejores. Todo el que lea su código se lo agradecerá.

El nombre de una variable, función o clase debe responder una serie de cuestiones básicas. Debe indicar por qué existe, qué hace y cómo se usa. Si un nombre requiere un comentario, significa que no revela su cometido.

```
int d; // tiempo transcurrido en días
```

El nombre `d` no revela nada. No evoca una sensación de tiempo transcurrido, ni de días. Debe elegir un nombre que especifique lo que se mide y la unidad de dicha medida:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

La elección de nombres que revelen intenciones facilita considerablemente la comprensión y la modificación del código. ¿Para qué sirve el siguiente código?

```
public List<int[]> getThem() {
   List<int[]> list1 = new ArrayList<int[]>();
   for (int[] x : theList)
     if (x[0] == 4)
       list1.add(x);
   return list1;
}
```

¿Por qué es complicado saber la función de este código? No hay expresiones complejas. Los espacios y el sangrado son razonables. Sólo hay tres variables y dos constantes. Ni siquiera contiene clases complejas o

métodos polimórficos, sólo una lista de matrices (o eso parece).

El problema no es la simplicidad del código sino su carácter *implícito*: el grado en el que el contexto no es explícito en el propio código. Implícitamente, el código requiere que sepamos las respuestas a las siguientes preguntas:

- ¿Qué contiene `theList`?
- ¿Qué significado tiene el subíndice cero de un elemento de `theList`?
- ¿Qué importancia tiene el valor `4`?
- ¿Cómo se usa la lista devuelta?

Las respuestas a estas preguntas no se encuentran en el código, pero se podrían haber incluido. Imagine que trabaja en un juego de buscar minas. El tablero es una lista de celdas llamada `theList`. Cambiemos el nombre por `gameBoard`.

Cada celda del teclado se representa por medio de una matriz. El subíndice cero es la ubicación de un valor de estado que, cuando es `4`, significa que se ha detectado. Al asignar nombres a estos conceptos mejoramos considerablemente el código:

```
public List<int[]> getFlaggedCells() {
   List<int[]> flaggedCells = new ArrayList<int[]>();
   for (int[] cell : gameBoard)
     if (cell[STATUS_VALUE] == FLAGGED)
       flaggedCells.add(cell);
   return flaggedCells;
}
```

La simplicidad del código no ha cambiado. Sigue teniendo los mismos operadores y constantes y el mismo número de niveles anidados, pero ahora es mucho más explícito. Podemos crear una sencilla clase para celdas en lugar de usar una matriz de elementos `int`. Puede incluir una función que revele el objetivo (con el nombre `isFlagged`) para ocultar los números. El resultado es una nueva versión de la función:

```
public List<Cell> getFlaggedCells() {
   List<Cell> flaggedCells = new ArrayList<Cell>();
   for (Cell cell : gameBoard)
     if (cell.isFlagged())
       flaggedCells.add(cell);
```

```
    return flaggedCells;
  }
```

Con estos sencillos cambios de nombre, es fácil saber qué sucede. Es la ventaja de seleccionar nombres adecuados.

Evitar la desinformación

Los programadores deben evitar dejar pistas falsas que dificulten el significado del código. Debemos evitar palabras cuyo significado se aleje del que pretendemos. Por ejemplo, `hp`, `aix` y `sco` son nombres de variables pobres ya que son los nombres de plataformas o variantes de Unix. Aunque se trate del código de una hipotenusa y `hp` parezca la abreviatura correcta, puede no serlo.

No haga referencia a un grupo de cuentas como `accountList` a menos que realmente sea una lista (`List`). La palabra lista tiene un significado concreto para los programadores. Si el contenedor de las cuentas no es realmente una lista, puede provocar falsas conclusiones[6]. Por tanto, resulta más adecuado usar `accountGroup`, `bunchOfAccounts` o simplemente `accounts`.

Evite usar nombres con variaciones mínimas. ¿Cuánto se tarda en apreciar la sutil diferencia entre `XYZControllerForEfficientHandlingOfStrings` y `XYZControllerForEfficientStorageOfStrings` en un módulo? Ambas palabras tienen una forma similar.

La ortografía similar de conceptos parecidos es información; el uso de ortografía incoherente es desinformación. En los entornos modernos de Java, el código se completa de forma automática. Escribimos varios caracteres de un nombre y pulsamos varias teclas para obtener una lista de posibles opciones de un nombre. Es muy útil si los nombres de elementos similares se ordenan alfabéticamente de forma conjunta y si las diferencias son muy evidentes, ya que es probable que el programador elija un objeto por nombre sin fijarse en los comentarios o la lista de métodos proporcionados por una clase.

Un ejemplo de nombre desinformativo sería el uso de la `L` minúscula o la `O` mayúscula como nombres de variables, sobre todo combinados. El problema, evidentemente, es que se parecen a las constantes `1` y `0` respectivamente:

```
int a = l;
if ( O == l )
   a = O1;
else
   l = 01;
```

El lector puede pensar que es una invención, pero hemos visto código con abundancia de estos elementos. En un caso, el autor del código, sugirió usar una fuente distinta para que las diferencias fueran más evidentes, una solución que se hubiera transmitido a todos los futuros programadores como tradición oral o en un documento escrito. El problema se resolvió con carácter definitivo y sin necesidad de crear nuevos productos, con tan sólo cambiar los nombres.

Realizar distinciones con sentido

Los programadores se crean un problema al crear código únicamente dirigido a un compilador o intérprete. Por ejemplo, como se puede usar el mismo nombre para hacer referencia a dos elementos distintos en el mismo ámbito, puede verse tentado a cambiar un nombre de forma arbitraria. En ocasiones se hace escribiéndolo incorrectamente, lo que provoca que los errores ortográficos impidan la compilación[7].

No basta con añadir series de números o palabras adicionales, aunque eso satisfaga al compilador. Si los nombres tienen que ser distintos, también deben tener un significado diferente.

Los nombres de series numéricas (`a1`, `a2`… `aN`) son lo contrario a los nombres intencionados. No desinforman, simplemente no ofrecen información; son una pista sobre la intención del autor. Fíjese en lo siguiente:

```
public static void copyChars(char a1[], char a2[]) {
   for (int i = 0; i < a1.length; i++) {
     a2[i] = a1[i];
   }
}
```

Esta función se lee mejor cuando se usa `source` y `destination` como nombres de argumentos.

Las palabras adicionales son otra distinción sin sentido. Imagine que tiene la clase Product. Si tiene otra clase con el nombre `ProductInfo` o `ProductData`, habrá creado nombres distintos, pero con el mismo significado. `Info` y `Data` son palabras adicionales, como `a`, `an` y `the`.

No es incorrecto usar prefijos como `a` y `the` mientras la distinción tenga sentido. Imagine que usa `a` para variables locales y `for` para argumentos de funciones[8]. El problema aparece cuando decide invocar la variable `theZork` porque ya tiene otra variable con el nombre `zork`.

Las palabras adicionales son redundantes. La palabra `variable` no debe incluirse nunca en el nombre de una variable. La palabra `table` no debe incluirse nunca en el nombre de una tabla. ¿Es mejor `NameString` que `Name`? ¿Podría ser `Name` un número de coma flotante? En caso afirmativo, incumple la regla anterior sobre desinformación. Imagine que encuentra una clase con el nombre `Customer` y otra con el nombre `CustomerObject`. ¿Cuál sería la distinción? ¿Cuál representa mejor el historial de pagos de un cliente?

Existe una aplicación que lo ilustra. Hemos cambiado los nombres para proteger al culpable. Veamos el error exacto:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

¿Cómo saben los programadores de este proyecto qué función deben invocar?

En ausencia de convenciones concretas, la variable `moneyAmount` no se distingue de `money`, `customerInfo` no se distingue de `customer`, `accountData` no se distingue de `account` y `theMessage` no se distingue de `message`. Debe diferenciar los nombres de forma que el lector aprecie las diferencias.

Usar nombres que se puedan pronunciar

A los humanos se nos dan bien las palabras. Gran parte de nuestro cerebro se dedica al concepto de palabras. Y, por definición, las palabras son pronunciables. Sería una pena malgastar esa parte de nuestro cerebro dedicada al lenguaje hablado. Por tanto, cree nombres pronunciables. Si no lo puede pronunciar, no podrá explicarlo sin parecer tonto. Es un factor importante, ya que la programación es una actividad social.

Conozco una empresa que usa `genymdhms` (fecha de generación, año, mes, día, hora, minuto y segundo) y lo pronuncian tal cual. Yo tengo la costumbre de pronunciar todo tal y como lo veo escrito, de forma que muchos analistas y diseñadores acabaron por llamarme algo como «genimedemes». Era un chiste y nos parecía divertido, pero en realidad estábamos tolerando el uso de nombres pobres. Teníamos que explicar las variables a los nuevos programadores y cuando las pronunciaban, usaban palabras inventadas en lugar de nombres correctos. Compare:

```
class DtaRcrd102 {
   private Date genymdhms;
   private Date modymdhms;
   private final String pszqint = “102”;
   /*… */
};
```

con:

```
class Customer {
   private Date generationTimestamp;
   private Date modificationTimestamp;
   private final String recordId = “102”;
   /*… */
};
```

Ahora se puede mantener una conversación inteligente: «Eh, Mikey, fíjate en este registro. La marca de tiempo de generación es para mañana. ¿Cómo es posible?»

Usar nombres que se puedan buscar

Los nombres de una letra y las constantes numéricas tienen un problema: no son fáciles de localizar en el texto. Se puede detectar `MAX_CLASSES_PER_STUDENT`, pero el número 7 resulta más complicado. Las búsquedas pueden devolver el dígito como parte de nombres de archivo, otras definiciones de constantes o expresiones en las que se use con otra intención. Mucho peor si la constante es un número extenso y alguien ha intercambiado los dígitos, lo que genera un error inmediato y no aparece en la búsqueda.

Del mismo modo, el nombre `e` es una opción muy pobre para variables que el programador tenga que buscar. Es la letra más usada en inglés y aparece en la práctica totalidad de los textos de un programa. A este respecto, los nombres extensos superan a los breves y cualquier nombre que se pueda buscar supera a una constante en el código.

Personalmente prefiero nombres de una letra que sólo se puedan usar como variables locales dentro de métodos breves. *La longitud de un nombre debe corresponderse al tamaño de su ámbito* [N5]. Si una variable o constante se usa en varios puntos del código, debe asignarle un nombre que se pueda buscar. Compare:

```
for (int j=0; j<34; j++) {
   s += (t[j]*4)/5;
}
```

con:

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j = 0; j < NUMBER_OF_TASKS; j++) {
   int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
   int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
   sum += realTaskWeeks;
}
```

En este ejemplo, `sum` no es un nombre especialmente útil, pero al menos se puede buscar. Se usa una función más extensa, pero comprobará que resulta mucho más fácil buscar `WORK_DAYS_PER_WEEK` que todas las instancias de `5` y filtrar la lista a los casos con el significado adecuado.

Evitar codificaciones

Ya tenemos suficientes codificaciones como para tener que añadir otras nuevas. Al codificar información de tipos o ámbitos en un nombre se dificulta la descodificación. No parece razonable que todos los nuevos empleados tengan que aprender otro lenguaje de codificación además del código con el que van a trabajar. Es una carga mental innecesaria a la hora de intentar solucionar un problema. Los nombres codificados resultan impronunciables y suelen escribirse de forma incorrecta.

Notación húngara

Antiguamente, cuando trabajábamos con lenguajes en los que la longitud de los nombres era un reto, incumplíamos esta regla a nuestro pesar. Fortran forzaba las codificaciones convirtiendo la primera letra de un tipo en código. En sus primeras versiones, BASIC sólo se permitía una letra y un dígito. La notación húngara (HN) lo llevó a un nuevo nivel.

HN se consideraba muy importante en el API C de Windows, donde todo era un control entero, un puntero `long`, un puntero `void` o una de varias implementaciones de `string` (con diferentes usos y atributos). Por aquel entonces, el compilador no comprobaba los tipos, de modo que los programadores tenían que recordarlos.

En los lenguajes modernos disponemos de sistemas de tipos más completos y los compiladores recuerdan los tipos y los aplican. Es más, existe una tendencia a usar clases y funciones más breves para que los usuarios aprecien la declaración de las variables que usan.

Los programadores de Java no tienen que codificar tipos. Los objetos son de tipos fuertes y los entornos de edición han avanzado tanto que detectan un error de tipo antes de ejecutar la compilación. Por ello, en la actualidad HN y otras formas de codificación de tipos no son más que un impedimento. Hacen que sea más complicado cambiar el nombre o el tipo de una variable o clase. Dificultan la legibilidad del código y pueden hacer que el sistema de codificación confunda al lector:

```
PhoneNumber phoneString;
// el nombre no cambia cuando cambia el tipo
```

Prefijos de miembros

Tampoco es necesario añadir `m_` como prefijo a los nombres de variables. Las clases y funciones tienen el tamaño necesario para no tener que hacerlo, y debe usar un entorno de edición que resalte o coloree los miembros para distinguirlos.

```
public class Part {
   private String m_dsc; // La descripción textual
   void setName(String name) {
     m_dsc = name;
   }
}
```

```
public class Part {
   String description;
   void setDescription(String description) {
     this.description = description;
   }
}
```

Además, los usuarios aprenden rápidamente a ignorar el prefijo (o sufijo) y fijarse en la parte con sentido del nombre. Cuanto más código leemos, menos nos fijamos en los prefijos. En última instancia, los prefijos son un indicio de código antiguo.

Interfaces e Implementaciones

Existe un caso especial para usar codificaciones. Imagine por ejemplo que crea una factoría abstracta para crear formas. Esta factoría será una interfaz y se implementará por medio de una clase concreta. ¿Qué nombres debe asignar? ¿`IShapeFactory` y `ShapeFactory`? Prefiero las interfaces sin adornos. La I inicial, tan habitual en los archivos de legado actuales es, en el mejor de los casos, una distracción, y en el peor, un exceso de información. No quiero que mis usuarios sepan que se trata de una interfaz, solamente que se trata de `ShapeFactory`. Si tengo que codificar la interfaz o la implementación, opto por ésta última. Es mejor usar `ShapeFactoryImp` o incluso `CShapeFactory`, que codificar la interfaz.

Evitar asignaciones mentales

Los lectores no tienen que traducir mentalmente sus nombres en otros que ya conocen. Este problema suele aparecer al elegir entre no usar términos de dominio de problemas o de soluciones.

Es un problema de los nombres de variables de una sola letra. Un contador de bucles se podría bautizar como `i`, `j` o `k` (pero nunca `l`) si su ámbito es muy reducido y no hay conflictos con otros nombres, ya que los nombres de una letra son tradicionales en contadores de bucles. Sin embargo, en otros contextos, un nombre de una letra es una opción muy pobre: es como un marcador de posición que el lector debe asignar mentalmente a un concepto real. No hay peor motivo para usar el nombre `c` que `a` y `b` ya estén seleccionados.

Por lo general, los programadores son gente inteligente. A la gente inteligente le gusta presumir de sus habilidades mentales. Si puede recordar que `r` es la versión en minúscula de una URL sin el host y el sistema, debe ser muy listo.

Una diferencia entre un programador inteligente y un programador profesional es que este último sabe que la *claridad es lo que importa*. Los profesionales usan sus poderes para hacer el bien y crean código que otros puedan entender.

Nombres de clases

Las clases y los objetos deben tener nombres o frases de nombre como `Customer`, `WikiPage`, `Account` y `AddressParser`. Evite palabras como `Manager`, `Processor`, `Data`, o `Info` en el nombre de una clase. El nombre de una clase no debe ser un verbo.

Nombres de métodos

Los métodos deben tener nombres de verbo como `postPayment`, `deletePage` o `save`. Los métodos de acceso, de modificación y los predicados deben tener como nombre su valor y usar como prefijo `get`, `set` e `is` de acuerdo al estándar de javabean[9].

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())…
```

Al sobrecargar constructores, use métodos de factoría estáticos con nombres que describan los argumentos. Por ejemplo:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

es mejor que:

```
Complex fulcrumPoint = new Complex(23.0);
```

Refuerce su uso convirtiendo en privados sus constructores correspondientes.

No se exceda con el atractivo

Si los nombres son demasiado inteligentes, sólo los recordarán los que compartan el sentido del humor de su autor, y sólo mientras se acuerden del chiste. ¿Sabrán qué significa la función `HolyHandGrenade`? Sin duda es atractiva, pero en este caso puede que `DeleteItems` fuera más indicado. Opte por la claridad antes que por el entretenimiento. En el código, el atractivo suele aparecer como formas coloquiales o jergas. Por ejemplo, no use `whack()` en lugar de `kill()`. No recurra a bromas culturales como `eatMyShorts()` si quiere decir `abort()`.

Diga lo que piense. Piense lo que diga.

Una palabra por concepto

Elija una palabra por cada concepto abstracto y manténgala. Por ejemplo,

resulta confuso usar `fetch`, `retrieve` y `get` como métodos equivalentes de clases distintas. ¿Cómo va a recordar qué método se corresponde a cada clase? Desafortunadamente, tendrá que recordar qué empresa, grupo o individuo ha creado la biblioteca o clase en cuestión para recordar qué término se ha empleado. En caso contrario, perderá mucho tiempo buscando en encabezados y fragmentos de código.

Los entornos de edición modernos como Eclipse e IntelliJ ofrecen pistas sensibles al contexto, como la lista de métodos que puede invocar en un determinado objeto. Pero esta lista no suele incluir los comentarios de nombres de funciones y listas de parámetros. Tendrá suerte si muestra los nombres de parámetros de las declaraciones de funciones. Los nombres de funciones deben ser independientes y coherentes para que pueda elegir el método correcto sin necesidad de búsquedas adicionales.

Del mismo modo, resulta confuso tener un controlador, un administrador y un control en la misma base de código. ¿Cuál es la diferencia entre `DeviceManager` y `ProtocolController`? ¿Por qué no son los dos controladores o administradores? ¿Son controladores? El nombre hace que espere que dos objetos tengan un tipo diferente y clases diferentes.

Un léxico coherente es una gran ventaja para los programadores que tengan que usar su código.

No haga juegos de palabras

Evite usar la misma palabra con dos fines distintos. Suele hacerse en juegos de palabras. Si aplica la regla de una palabra por conceptos, acabará con muchas clases que por ejemplo tengan un método `add`. Mientras las listas de parámetros y los valores devueltos de los distintos métodos `add` sean semánticamente equivalentes, no hay problema.

Sin embargo, alguien puede decidir usar la palabra `add` por motivos de coherencia, aunque no sea en el mismo sentido. Imagine que hay varias clases en las que `add` crea un nuevo valor sumando o concatenando dos valores existentes. Imagine ahora que crea una nueva clase con un método

que añada su parámetro a una colección. ¿Este método debe tener el método `add`? Parece coherente ya que hay otros muchos métodos `add`, pero en este caso hay una diferencia semántica, de modo que debemos usar un nombre como `insert` o `append`. Llamar `add` al nuevo método sería un juego de palabras.

Nuestro objetivo, como autores, es facilitar la comprensión del código. Queremos que el código sea algo rápido, no un estudio exhaustivo. Queremos usar un modelo en el que el autor sea el responsable de transmitir el significado, no un modelo académico que exija investigar el significado mostrado.

Usar nombres de dominios de soluciones

Recuerde que los lectores de su código serán programadores. Por ello, use términos informáticos, algoritmos, nombres de patrones, términos matemáticos y demás. No conviene extraer todos los nombres del dominio de problemas ya que no queremos que nuestros colegas tengan que preguntar el significado de cada nombre en especial cuando ya conocen el concepto bajo otro nombre diferente.

El nombre `AccountVisitor` tiene mucho significado para un programador familiarizado con el patrón `VISITOR`. ¿Qué programador no sabe lo que es `JobQueue`? Hay cientos de cosas técnicas que los programadores tienen que hacer y elegir nombres técnicos para dichas cosas suele ser lo más adecuado.

Usar nombres de dominios de problemas

Cuando no exista un término de programación para lo que esté haciendo, use el nombre del dominio de problemas. Al menos el programador que mantenga su código podrá preguntar el significado a un experto en dominios.

Separar los conceptos de dominio de soluciones y de problemas es parte del trabajo de un buen programador y diseñador. El código que tenga más relación con los conceptos del dominio de problemas tendrá nombres

extraídos de dicho dominio.

Añadir contexto con sentido

Algunos nombres tienen significado por sí mismos, pero la mayoría no. Por ello, debe incluirlos en un contexto, en clases, funciones y espacios de nombres con nombres adecuados. Cuando todo lo demás falle, pueden usarse prefijos como último recurso.

Imagine que tiene las variables `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` y `zipcode`. Si las combina, es evidente que forman una dirección. Pero si la variable `state` se usa de forma aislada en un método, ¿sabría que forma parte de una dirección? Puede añadir contexto por medio de prefijos: `addrFirstName`, `addrLastName`, `addrState`, etc. Al menos los lectores comprenderán que estas variables forman parte de una estructura mayor. Evidentemente, es mejor crear la clase `Address`. De ese modo, incluso el compilador sabrá que las variables pertenecen a un concepto más amplio.

Fíjese en el método del Listado 2-1. ¿Las variables necesitan un contexto con más sentido? El nombre de la función sólo ofrece parte del contexto, el resto se obtiene del algoritmo. Tras leer la función, verá que las tres variables `number`, `verb` y `pluralModifier` forman parte del mensaje `guess statistics`. Desafortunadamente, es necesario inferir el contexto. Al leer el método, el significado de las variables no es evidente.

Listado 2-1
Variables en un contexto ambiguo.

```
private void printGuessStatistics(char candidate, int count) {
  String number;
  String verb;
  String pluralModifier;
  if (count == 0) {
    number = “no”;
    verb = “are”;
    pluralModifier = “s”;
  } else if (count == 1) {
    number = “1”;
```

```
      verb = “is”;
      pluralModifier = “”;
    } else {
      number = Integer.toString(count);
      verb = “are”;
      pluralModifier = “s”;
    }
    String guessMessage = String.format(
      “There %s %s %s%s”, verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

La función es demasiado extensa y las variables aparecen por todas partes. Para dividir la función en fragmentos más reducidos necesitamos crear una clase `GuessStatisticsMessage` y convertir a las tres variables en campos de la misma. De este modo contamos con un contexto más obvio para las tres variables. Forman parte sin duda de `GuessStatisticsMessage`. La mejora del contexto también permite que el algoritmo sea más limpio y se divida en funciones más reducidas (véase el Listado 2-2).

Listado 2-2
Variables con un contexto.

```
  public class GuessStatisticsMessage (
    private String number;
    private String verb;
    private String pluralModifier;

    public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      “There %s %s %s%s,
      verb, number, candidate, pluralModifier);
    }

    private void createPluralDependentMessageParts(int count) {
      if (count == 0) {
        thereAreNoLetters();
      } else if (cout == 1) {
        thereIsOneLetter();
      } else {
        thereAreManyLetters(count);
      }
    }

    private void thereAreManyLetters(int count) {
      number = “1”;
```

```
      verb = “is”;
      pluralModifier = “”;
    }

    private void thereIsOneLetter() {
      number = “1”;
      verb = “is”;
      pluralModifier = “”;
    }

    private void thereAreNoLetters() {
      number = “no”;
      verb = “are”;
      pluralModifier = “s”;
    }
  }
```

No añadir contextos innecesarios

En la aplicación imaginaria Gas Station Deluxe, no es aconsejable usar el prefijo `GSD` en todas las clases. Es trabajar contra las herramientas proporcionadas. Introduzca `G` y pulse la tecla de finalización para acceder a una lista interminable de todas las clases del sistema. ¿Es lo correcto? ¿Por qué dificultar la ayuda del IDE?

Del mismo modo, imagine que ha creado la clase `MailingAddress` en un módulo de contabilidad de `GSD`, con el nombre `GSDAccountAddress`. Después, necesita una dirección de correo para la aplicación de contacto con el cliente. ¿Usará `GSDAccountAddress`? ¿Le parece el nombre correcto? 10 de los 17 caracteres son redundantes o irrelevantes.

Los nombres breves suelen ser más adecuados que los extensos, siempre que sean claros. No añada más contexto del necesario a un nombre. Los nombres `accountAddress` y `customerAddress` son perfectos para instancias de la clase `Address` pero no sirven como nombres de clase. `Address` sirve como nombre de clase. Para distinguir entre direcciones MAC, direcciones de puertos y direcciones Web, podría usar `PostalAddress`, `MAC` y `URI`. Los nombres resultantes son más precisos, el objetivo de cualquier nombre.

Conclusión

Lo más complicado a la hora de elegir un buen nombre es que requiere habilidad descriptiva y acervo cultural. Es un problema de formación más que técnico, empresarial o administrativo. Como resultado, mucha gente del sector no aprende a hacerlo bien.

La gente teme que al cambiar los nombres otros programadores se quejen. Nosotros no compartimos ese temor y agradecemos los cambios de nombre (siempre que sean a mejor). En muchos casos no memorizamos los nombres de clases y métodos. Usamos herramientas modernas para estos detalles y así poder centrarnos en si el código se lee como frases o párrafos, o al menos como tablas y estructuras de datos (una frase no siempre es la mejor forma de mostrar datos). Seguramente acabará sorprendiendo a alguien cuando cambie los nombres, como puede suceder con cualquier otra mejora del código. No deje que le detenga.

Aplique estas reglas y compruebe si mejora o no la legibilidad de su código. Si es el encargado de mantener código de terceros, use herramientas para solucionar estos problemas. Obtendrá recompensas a corto y largo plazo.

3
Funciones

En los inicios de la programación, creábamos sistemas a partir de rutinas y subrutinas. Después, en la época de Fortran y PL/1, creábamos nuestros sistemas con programas, subprogramas y funciones. En la actualidad, sólo las funciones han sobrevivido. Son la primera línea organizativa en cualquier programa. En este capítulo veremos cómo crearlas.

Fíjese en el código del Listado 3-1. Es complicado encontrar una función extensa en FitNesse[10], pero acabé encontrando ésta. No sólo es extensa, sino que también contiene código duplicado, muchas cadenas y tipos de datos extraños, además de API poco habituales y nada evidentes. Intente

comprenderlo en los próximos tres minutos.

Listado 3-1
HtmlUtil.java (FitNesse 20070619).

```
public static String testableHtml {
   PageData pageData,
   boolean includeSuiteSetup
} throws Exception {
   WikiPage wikiPage = pageData.getWikiPage();
   StringBuffer buffer = new StringBuffer();
   if (pageData.hasAttribute("Test")) {
     if (includeSuiteSetup) {
       WikiPage suiteSetup =
         PageCrawlerImpl.getInheritedPage(
           SuiteResponder.SUITE_SETUP_NAME, wikiPage
         );
       if (suiteSetup != null) {
         WikiPagePath pagePath =
           suiteSetup.getPageCrawler().getFullPath (suiteSetup);
         String pagePathName = PathParser.render(pagePath);
         buffer.append("!include -setup .")
           .append(pagePathName)
           .append("\n");
       }
     }
     WikiPage setup =
       PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
     if (setup != null) {
       WikiPagePath setupPath =
         wikiPage.getPageCrawler().getFullPath(setup);
       String setupPathName = PathParser.render(setupPath);
       buffer.append("!include -setup .")
         .append(setupPathName)
         .append("\n");
     }
   }
   buffer.append(pageData.getContent());
   if (pageData.hasAttribute("Test")) {
     WikiPage teardown =
       PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
     if (teardown != null) {
       WikiPagePath tearDownPath =
         wikiPage.getPageCrawler().getFullPath(teardown);
       String tearDownPathName = PathParser.render(tearDownPath);
       buffer.append("\n")
         .append("!include -teardown .")
         .append(tearDownPathName)
         .append("\n");
     }
     if (includeSuiteSetup) {
```

```
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
          SuiteResponder.SUITE_TEARDOWN_NAME,
          wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
          .append(pagePathName)
          .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

¿Tras tres minutos entiende la función? Seguramente no. Pasan demasiadas cosas y hay demasiados niveles de abstracción diferentes. Hay cadenas extrañas e invocaciones de funciones mezcladas en instrucciones `if` doblemente anidadas controladas por indicadores. Sin embargo, con sencillas extracciones de código, algún cambio de nombres y cierta reestructuración, pude capturar la intención de la función en las nueve líneas del Listado 3-2. Compruebe si ahora la entiende.

Listado 3-2
HtmlUtil.java (refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages (testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

A menos que sea un alumno de FitNesse, seguramente no entienda los

detalles. Entenderá que la función se encarga de añadir páginas de configuración y detalles en una página de prueba, que después muestra en HTML. Si está familiarizado con JUnit[11], verá que esta función pertenece a algún tipo de estructura de pruebas basada en la Web y, evidentemente, es correcto. Resulta sencillo adivinar esta información del Listado 3-2 pero no del Listado 3-1. ¿Qué tiene la función del Listado 3-2 para que resulte sencilla de leer y entender? ¿Qué hay que hacer para que una función transmita su intención? ¿Qué atributos podemos asignar a nuestras funciones para que el lector pueda intuir el tipo de programa al que pertenecen?

Tamaño reducido

La primera regla de las funciones es que deben ser de tamaño reducido. La segunda es que *deben ser todavía más reducidas*. No es una afirmación que pueda justificar. No puedo mostrar referencias a estudios que demuestren que las funciones muy reducidas sean mejores. Lo que sí puedo afirmar es que durante casi cuatro décadas he creado funciones de diferentes tamaños. He creado monstruos de casi 3000 líneas y otras muchas funciones de entre 100 y 300 líneas. También he creado funciones de 20 a 30 líneas de longitud. Esta experiencia me ha demostrado, mediante ensayo y error, que las funciones deben ser muy reducidas.

En la década de 1980 se decía que una función no debía superar el tamaño de una pantalla. Por aquel entonces, las pantallas VT100 tenían 24 líneas por 80 columnas, y nuestros editores usaban 4 líneas para tareas administrativas. En la actualidad, con una fuente mínima y un monitor de gran tamaño, se pueden encajar 150 caracteres por línea y 100 líneas o más en una pantalla. Las líneas no deben tener 150 caracteres. Las funciones no deben tener 100 líneas de longitud. Las funciones deben tener una longitud aproximada de 20 líneas.

¿Qué tamaño mínimo debe tener una función? En 1999 visité a Kent Beck en su casa de Oregon. Nos sentamos y comenzamos a programar. Me enseñó un atractivo programa de Java/Swing que había llamado *Sparkle*. Generaba un efecto visual en pantalla, similar a la varita mágica del hada de

Cenicienta. Al mover el ratón, salían estrellitas del cursor, y descendían a la parte inferior de la pantalla en un campo gravitatorio simulado. Cuando Kent me enseñó el código, me sorprendió la brevedad de las funciones. Estaba acostumbrado a ver programas de Swing con funciones que ocupaban kilómetros de espacio vertical. En este programa, las funciones tenían dos, tres o cuatro líneas de longitud. Todas eran obvias. Todas contaban una historia y cada una llevaba a la siguiente en un orden atractivo. ¡Así de breves deberían ser todas las funciones![12]

¿Qué tamaño mínimo deben tener sus funciones? Deberían ser más breves que las del Listado 3-2. De hecho, el Listado 3-2 debería reducirse como el Listado 3-3.

Listado 3-3
HtmlUtil.java (nueva refactorización).

```
public static String renderPageWithSetupsAndTeardowns(
   PageData pageData, boolean isSuite) throws Exception {
   if (isTestPage(pageData))
     includeSetupAndTeardownPages(pageData, isSuite);
   return pageData.getHtml();
}
```

Bloques y sangrado

Esto implica que los bloques en instrucciones `if`, `else`, `while` y similares deben tener una línea de longitud que, seguramente, sea la invocación de una función. De esta forma, no sólo se reduce el tamaño de la función, sino que también se añade valor documental ya que la función invocada desde el bloque puede tener un nombre descriptivo. También implica que las funciones no deben tener un tamaño excesivo que albergue estructuras anidadas. Por tanto, el nivel de sangrado de una función no debe ser mayor de uno o dos. Evidentemente, de esta forma las funciones son más fáciles de leer y entender.

Hacer una cosa

Es evidente que el Listado 3-1 hace más de una cosa. Crea búferes, obtiene páginas, busca páginas heredadas, añade cadenas antiguas y genera HTML. El Listado 3-1 está muy ocupado realizando varias tareas. Por su parte, el Listado 3-3 sólo hace una cosa: incluye configuraciones y detalles en páginas de prueba.

El siguiente consejo lleva vigente, de una u otra forma, durante más de 30 años:

LAS FUNCIONES SÓLO DEBEN HACER UNA COSA. DEBEN HACERLO BIEN Y DEBE SER LO ÚNICO QUE HAGAN.

El problema de esta afirmación es saber qué es una cosa. ¿El Listado 3-3 hace una cosa? Se podría pensar que hace tres:

1. Determinar si la página es una página de prueba.
2. En caso afirmativo, incluir configuraciones y detalles.
3. Representar la página en HTML.

¿Cuál será de las tres? ¿La función hace una o tres cosas? Los tres pasos de la función se encuentran un nivel de abstracción por debajo del nombre de la función. Podemos describir la función como un breve párrafo TO (PARA) [13]:

> *Para `renderPageWithSetupsAndTeardowns`, comprobamos si la página es de prueba y, en caso afirmativo, incluimos las configuraciones y los detalles. En ambos casos, la representamos en HTML.*

Si una función sólo realiza los pasos situados un nivel por debajo del nombre de la función, entonces hace una cosa. En definitiva, creamos funciones para descomponer conceptos más amplios (es decir, el nombre de

la función) en un conjunto de pasos en el siguiente nivel de abstracción. Es evidente que el Listado 3-1 contiene pasos en distintos niveles de abstracción, por lo que es obvio que hace más de una cosa. Incluso el Listado 3-2 tiene tres niveles de abstracción, como ha demostrado la capacidad de reducirlo, pero sería complicado reducir con sentido el Listado 3-3. Podríamos extraer la instrucción `if` en la función `includeSetupsAndTeardownsIfTestPage`, pero sólo reduciríamos el código sin cambiar el nivel de abstracción.

Por ello, otra forma de saber que una función hace más de una cosa es extraer otra función de la misma con un nombre que no sea una reducción de su implementación [G34].

Secciones en funciones

Fíjese en el Listado 4-7. Verá que la función `generatePrimes` se divide en secciones como declaraciones, inicializaciones y filtros. Es un síntoma evidente de que hace más de una cosa. Las funciones que hacen una sola cosa no se pueden dividir en secciones.

Un nivel de abstracción por función

Para que las funciones realicen «una cosa», asegúrese de que las instrucciones de la función se encuentran en el mismo nivel de abstracción. El Listado 3-1 incumple esta regla. Incluye conceptos a un elevado nivel de abstracción, como `getHtml();` otros se encuentran en un nivel intermedio, como `StringpagePathName = PathParser.render(pagePath)` y hay otros en un nivel especialmente bajo, como `.append("\n").`

La mezcla de niveles de abstracción en una función siempre resulta confusa. Los lectores no sabrán si una determinada expresión es un concepto esencial o un detalle. Peor todavía, si se mezclan detalles con conceptos esenciales, aumentarán los detalles dentro de la función.

Leer código de arriba a abajo: la regla descendente

El objetivo es que el código se lea como un texto de arriba a abajo[14]. Queremos que tras todas las funciones aparezcan las del siguiente nivel de abstracción para poder leer el programa, descendiendo un nivel de abstracción por vez mientras leemos la lista de funciones. Es lo que denomino la regla descendente.

Para decirlo de otra forma, queremos leer el programa como si fuera un conjunto de párrafos `TO`, en el que cada uno describe el nivel actual de abstracción y hace referencia a los párrafos `TO` posteriores en el siguiente nivel.

> *Para incluir configuraciones y detalles, incluimos configuraciones, después del contenido de la página de prueba, y por último los detalles.*
> *Para incluir las configuraciones, incluimos la configuración de suite si se trata de una suite, y después la configuración convencional.*
> *Para incluir la configuración de suite; buscamos la jerarquía principal de la página `SuiteSetUp` y añadimos una instrucción `include` con la ruta de dicha página.*
> *Para buscar la jerarquía principal…*

A los programadores les resulta complicado aprender esta regla y crear funciones en un único nivel de abstracción, pero es un truco importante. Es la clave para reducir la longitud de las funciones y garantizar que sólo hagan una cosa. Al conseguir que el código se lea de arriba a abajo, se mantiene la coherencia de los niveles de abstracción.

Fíjese en el Listado 3-7 del final del capítulo. Muestra la función `testableHtml` modificada de acuerdo a estos principios. Cada función presenta a la siguiente y se mantiene en un nivel de abstracción coherente.

Instrucciones Switch

Es complicado usar una instrucción `switch` de tamaño reducido[15]. Aunque sólo tenga dos casos, es mayor de lo que un bloque o función debería ser. También es complicado crear una instrucción `switch` que haga una sola cosa. Por su naturaleza, las instrucciones `switch` siempre hacen N cosas. Desafortunadamente, no siempre podemos evitar las instrucciones `switch` pero podemos asegurarnos de incluirlas en una clase de nivel inferior y de no repetirlas. Para ello, evidentemente, recurrimos al polimorfismo.

Fíjese en el Listado 3-4. Muestra una de las operaciones que pueden depender del tipo de empleado.

Listado 3-4
Payroll.java.

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType (
   switch (e.type) {
     case COMMISSIONED:
       return calculateCommissionedPay(e);
     case HOURLY:
       return calculateHourlyPay(e);
     case SALARIED:
       return calculateSalariedPay(e);
     default:
       throw new InvalidEmployeeType(e.type);
   }
}
```

Esta función tiene varios problemas. Por un lado, es de gran tamaño y cuando se añadan nuevos tipos de empleado, aumentará más. Por otra parte, hace más de una cosa. También incumple el Principio de responsabilidad única (*Single Responsibility Principie* o SRP)[16] ya que hay más de un motivo para cambiarla. Además, incumple el Principio de abierto/cerrado (*Open Closed Principle* u OCP)[17], ya que debe cambiar cuando se añadan nuevos tipos, pero posiblemente el peor de los problemas es que hay un número ilimitado de funciones que tienen la misma estructura.

Por ejemplo, podríamos tener:

```
isPayday(Employee e, Date date),
```

o

```
deliverPay(Employee e, Date date),
```

o muchas otras, todas con la misma estructura.

La solución al problema (véase el Listado 3-5) consiste en ocultar la instrucción `switch` en una factoría abstracta[18] e impedir que nadie la vea. La factoría usa la instrucción `switch` para crear las instancias adecuadas de los derivados de Employee y las distintas funciones, como `calculatePay`, `isPayday` y `deliverPay`, se entregarán de forma polimórfica a través de la interfaz `Employee`.

Listado 3-5
Employee y Factory.

```
public abstract class Employee {
   public abstract boolean isPayday();
   public abstract Money calculatePay();
   public abstract void deliverPay(Money pay);
}
```

```
public interface EmployeeFactory {
   public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
```

```
public class EmployeeFactoryImpl implements EmployeeFactory {
   public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType
{
     switch (r.type) {
       case COMMISSIONED:
         return new CommissionedEmployee(r);
       case HOURLY:
         return new HourlyEmployee(r);
       case SALARIED:
         return new SalariedEmployee(r);
       default:
         throw new InvalidEmployeeType(r.type);
     }
   }
}
```

Mi regla general para las instrucciones `switch` es que se pueden tolerar si sólo aparecen una vez, se usan para crear objetos polimórficos y se ocultan tras una relación de herencia para que el resto del sistema no las pueda ver [G23]. Evidentemente, cada caso es diferente y en ocasiones se puede

incumplir una o varias partes de esta regla.

Usar nombres descriptivos

En el Listado 3-7, hemos cambiado el nombre de la función de ejemplo de `testableHtml` a `SetupTeardownIncluder.render`. Es un nombre más apropiado ya que describe mejor el cometido de la función. También hemos asignado a los métodos privados un nombre descriptivo como `isTestable` o `includeSetupAndTeardownPages`. No hay que olvidar el valor de los nombres correctos. Recuerde el principio de Ward: «Sabemos que trabajamos con código limpio cuando cada rutina es más o menos lo que esperábamos». Para alcanzar este principio, gran parte del esfuerzo se basa en seleccionar nombres adecuados para pequeñas funciones que hacen una cosa. Cuanto más reducida y concreta sea una función, más sencillo será elegir un nombre descriptivo. No tema los nombres extensos. Un nombre descriptivo extenso es mucho mejor que uno breve pero enigmático. Use una convención de nombres que permita leer varias palabras en los nombres de las funciones y use esas palabras para asignar a la función un nombre que describa su cometido.

No tema dedicar tiempo a elegir un buen nombre. De hecho, debería probar con varios nombres y leer el código con todos ellos. Los IDE modernos como Eclipse o IntelliJ facilitan el cambio de nombres. Use uno de estos IDE y experimente con diferentes nombres hasta que encuentre uno que sea lo bastante descriptivo.

La elección de nombres descriptivos clarifica el diseño de los módulos y le permite mejorarlos. No es extraño que la búsqueda de nombres adecuados genere una reestructuración favorable del código. Sea coherente con los nombres. Use las mismas frases, sustantivos y verbos en los nombres de función que elija para los módulos. Pruebe, por ejemplo, con `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` e `includeSetupPage`. La estructura similar de estos nombres permite que la secuencia cuente una historia. En realidad, si ve la secuencia anterior, seguramente se pregunte qué ha pasado con

`includeTeardownPages`, `includeSuiteTeardownPage` e `includeTeardownPage`.

Argumentos de funciones

El número ideal de argumentos para una función es cero. Después uno (monádico) y dos (diádico). Siempre que sea posible, evite la presencia de tres argumentos (triádico). Más de tres argumentos (poliádico) requiere una justificación especial y no es muy habitual.

Los argumentos son complejos ya que requieren un gran poder conceptual. Por ello suelo evitarlos en los ejemplos. Fíjese en `StringBuffer`. Podríamos haberlo pasado como argumento en lugar de como variable de instancia, pero los lectores habrían tenido que interpretarlo cada vez que lo vieran. Al leer la historia que cuenta el módulo, `includeSetupPage()` es más sencillo de interpretar que `includeSetupPageInto(newPageContent)`. El argumento se encuentra en un nivel de abstracción diferente que el nombre de la función y nos obliga a conocer un detalle (`StringBuffer`) que no es especialmente importante en ese momento.

Los argumentos son todavía más complicados desde un punto de vista de pruebas. Imagine la dificultad de crear todos los casos de prueba para garantizar el funcionamiento de las distintas combinaciones de argumentos. Si no hay argumentos, todo es más sencillo. Si hay uno, no es demasiado difícil. Con dos argumentos el problema es más complejo. Con más de dos argumentos, probar cada combinación de valores adecuados es todo un reto. Los argumentos de salida son más difíciles de entender que los de entrada. Al leer una función, estamos acostumbrados al concepto de información añadida

a la función a través de argumentos y extraída a través de un valor devuelto. No esperamos que la información se devuelva a través de los argumentos. Por ello, los argumentos de salida suelen obligamos a realizar una comprobación doble.

Un argumento de salida es la mejor opción, después de la ausencia de argumentos. `SetupTeardownIncluder.render(pageData)` se entiende bien. Evidentemente, vamos a representar los datos en el objeto `pageData`.

Formas monádicas habituales

Hay dos motivos principales para pasar un solo argumento a una función. Puede que realice una pregunta sobre el argumento, como en `boolean fileExists("MyFile"),` o que procese el argumento, lo transforme en otra cosa y lo devuelva. Por ejemplo, `InputStream fileOpen("MyFile")` transforma un nombre de archivo `String` en un valor devuelto `InputStream`. Los usuarios esperan estos dos usos cuando ven una función. Debe elegir nombres que realicen la distinción con claridad y usar siempre ambas formas en un contexto coherente (consulte el apartado sobre separación de consultas de comandos).

Una forma menos habitual pero muy útil para un argumento es un evento. En esta forma, hay argumento de entrada pero no de salida. El programa debe interpretar la invocación de la función como evento y usar el argumento para alterar el estado del sistema, por ejemplo, `void passwordAttemptFailedNtimes(int attempts)`. Use esta forma con precaución. Debe ser claro para el lector que se trata de un evento. Elija nombres y contextos con atención. Intente evitar funciones monádicas que no tengan estas formas, por ejemplo, `void includeSetupPageInto(StringBuffer pageText)`. El uso de un argumento de salida en lugar de un valor devuelto para realizar transformaciones resulta confuso. Si una función va a transformar su argumento de entrada, la transformación debe aparecer como valor devuelto. Sin duda `StringBuffertransform(StringBuffer in)` es mejor que `void`

`transform(StringBuffer out)`, aunque la implementación del primer caso devuelva solamente el argumento de entrada. Al menos se ajusta a la forma de la transformación.

Argumentos de indicador

Los argumentos de indicador son horribles. Pasar un valor Booleano a una función es una práctica totalmente desaconsejable. Complica inmediatamente la firma del método e indica que la función hace más de una cosa. Hace algo si el indicador es `true` y otra cosa diferente si es `false`. En el Listado 3-7 no se puede evitar, porque los invocadores ya pasan el indicador y el objetivo era limitar el ámbito a la función y después, pero la invocación de `render (true)` es confusa para el lector. Si se desplaza el ratón sobre la invocación vemos que `render (boolean isSuite)` puede ayudar, pero no demasiado. Tendremos que dividir la función en dos: `renderForSuite()` y `renderForSingleTest()`.

Funciones diádicas

Una función con dos argumentos es más difícil de entender que una función monádica. Por ejemplo `writeField(name)` es más fácil de entender que `writeField (outputStream, name)`[19]. Aunque en ambos casos el significado es evidente, la primera se capta mejor visualmente. La segunda requiere una breve pausa hasta que ignoramos el segundo parámetro, lo que en última instancia genera problemas ya que no debemos ignorar esa parte del código. Las partes que ignoramos son las que esconden los errores. Pero en ocasiones se necesitan dos argumentos. Por ejemplo. `Point p = new Point(0,0);` es totalmente razonable. Los puntos cartesianos suelen adoptar dos argumentos. De hecho, sería muy sorprendente ver `Point(0)`. Sin embargo, en este caso ambos argumentos son componentes ordenados de un mismo valor, mientras que `outputStream` y `name` carecen de una cohesión o un orden natural.

Incluso funciones diádicas evidentes como `assertEquals(expected, actual)` resultan problemáticas. ¿Cuántas veces ha incluido el valor real en su posición esperada? Los dos argumentos carecen de un orden natural. El orden real y esperado es una convención que se adquiere gracias a la práctica.

Las combinaciones diádicas no son el mal en persona y tendrá que usarlas. Sin embargo, recuerde que tienen un precio y que debe aprovechar los mecanismos disponibles para convertirlas en unitarias. Por ejemplo, puede hacer que el método `writeField` sea un miembro de `outputStream` para poder usar `outputStream.writeField(name)`, o podría convertir `outputStream` en una variable miembro de la clase actual para no tener que pasarla. Incluso podría extraer una nueva clase como `FieldWriter` que usara `outputStream` en su constructor y tuviera un método `write`.

Triadas

Las funciones que aceptan tres argumentos son sin duda mucho más difíciles de entender que las de dos. Los problemas a la hora de ordenar, ignorar o detenerse en los argumentos se duplican. Piense atentamente antes de crear una triada.

Por ejemplo, fíjese en la sobrecarga de `assertEquals` que acepta tres argumentos: `assertEquals(message, expected, actual)`. ¿Cuántas veces lee el mensaje y piensa que es lo esperado? He visto esta triada en concreto muchas veces. De hecho, siempre que la veo, tengo que repasarla antes de ignorar el mensaje.

Por otra parte, hay otra triada que no es tan negativa: `assertEquals(1.0, amount, .001)`. Aunque también exija doble atención, merece la pena. Conviene recordar siempre que la igualdad de los valores de coma flotante es algo relativo.

Objeto de argumento

Cuando una función parece necesitar dos o más argumentos, es probable que

alguno de ellos se incluya en una clase propia. Fíjese en la diferencia entre las dos siguientes declaraciones:

```
Circle makeCircle (double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

La reducción del número de argumentos mediante la creación de objetos puede parecer una trampa pero no lo es. Cuando se pasan grupos de variables de forma conjunta, como x e y en el ejemplo anterior, es probable que formen parte de un concepto que se merece un nombre propio.

Listas de argumentos

En ocasiones tendremos que pasar un número variable de argumentos a una función. Fíjese en el método `String.format`:

```
String.format ("%s worked %.2f hours.", name, hours);
```

Si los argumentos variables se procesan de la misma forma, como en el ejemplo anterior, serán equivalentes a un único argumento de tipo `List`. Por tanto, `String.format` es en realidad diádico. De hecho, la siguiente declaración de `String.format` es claramente diádica.

```
public String format(String format, Object… args)
```

Así pues, se aplican las mismas reglas. Las funciones que aceptan argumentos variables pueden ser monádicas, diádicas o incluso triádicas, pero sería un error asignar más argumentos.

```
void monad(Integer… args);
void dyad(String name, Integer… args);
void triad(String name, int count, Integer… args);
```

Verbos y palabras clave

La selección de nombres correctos para una función mejora la explicación de su cometido, así como el orden y el cometido de los argumentos. En formato monádico, la función y el argumento deben formar un par de verbo y sustantivo. Por ejemplo, `write(name)` resulta muy evocador. Sea lo que sea `name`, sin duda se escribe `(write)`.

Un nombre más acertado podría ser `writeField(name)`, que nos dice que `name` es un campo `(field)`. Éste es un ejemplo de palabra clave como nombre de función. Con este formato codificamos los nombres de los argumentos en el nombre de la función. Por ejemplo, `assertEquals` se podría haber escrito como `assertExpectedEqualsActual(expected, actual)`, lo que mitiga el problema de tener que recordar el orden de los argumentos.

Sin efectos secundarios

Los efectos secundarios son mentiras. Su función promete hacer una cosa, pero también hace otras cosas ocultas. En ocasiones realiza cambios inesperados en las variables de su propia clase. En ocasiones las convierte en las variables pasadas a la función o a elementos globales del sistema. En cualquier caso, se comete un engaño que suele provocar extrañas combinaciones temporales y dependencias de orden.

Fíjese en la función del Listado 3-6, aparentemente inofensiva. Usa un algoritmo estándar para comparar `userName` con `password`. Devuelve `true` si coinciden y `false` si hay algún problema, pero también hay un efecto secundario. ¿Lo detecta?

Listado 3-6
UserValidator.java.

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if (“Valid Password”.equals(phrase)){
        Session.initialize();
        return true;
      }
    }
    return false;
  }
```

```
}
```

El efecto secundario es la invocación de `Session.initialize()`. La función `checkPassword`, por su nombre, afirma comprobar la contraseña. El nombre no implica que inicialice la sesión. Por tanto, un invocador que se crea lo que dice el nombre de la función se arriesga a borrar los datos de sesión actuales cuando decida comprobar la validez del usuario. Este efecto secundario genera una combinación temporal. Es decir, sólo se puede invocar `checkPassword` en determinados momentos (cuando se pueda inicializar la sesión). Si no se invoca en orden, se pueden perder los datos de la sesión. Las combinaciones temporales son confusas, en especial cuando se ocultan como efecto secundario. Si tiene que realizar una combinación temporal, hágalo de forma clara en el nombre de la función. En este caso, podríamos cambiar el nombre de la función por `checkPasswordAndInitializeSession`, pero incumpliría la norma de hacer una sola cosa.

Argumentos de salida

Los argumentos suelen interpretarse como entradas de una función. Si lleva varios años programando, estoy seguro de que habrá visto un argumento que en vez de ser de entrada era de salida. Por ejemplo;

```
appendFooter(s);
```

¿Está función añade `s` al final de algo? ¿O añade el final de algo a `s`? ¿`s` es una entrada o una salida? Lo sabemos al ver la firma de la función:

```
public void appendFooter(StringBuffer report)
```

Esto lo aclara todo, pero para ello hay que comprobar la declaración de la función. Todo lo que le obligue a comprobar la firma de la función es un esfuerzo doble. Es una pausa cognitiva y debe evitarse.

Antes de la programación orientada a objetos, era necesario tener argumentos de salida. Sin embargo, gran parte de su necesidad desaparece en los lenguajes orientados a objetos, pensados para actuar como argumento de salida. Es decir, sería más indicado invocar `appendFooter` como `report.appendFooter();`.

Por lo general, los argumentos de salida deben evitarse. Si su función

tiene que cambiar el estado de un elemento, haga que cambie el estado de su objeto contenedor.

Separación de consultas de comando

Las funciones deben hacer algo o responder a algo, pero no ambas cosas. Su función debe cambiar el estado de un objeto o devolver información sobre el mismo, pero ambas operaciones causan confusión. Fíjese en la siguiente función:

```
public boolean set(String attribute, String value);
```

Esta función establece el valor de un atributo y devuelve `true` en caso de éxito o `false` si el atributo no existe. Esto provoca la presencia de una extraña instrucción como la siguiente:

```
if (set("username", "unclebob"))…
```

Imagínelo desde el punto de vista del lector. ¿Qué significa? ¿Pregunta si el atributo «`username`» se ha establecido antes en «`unclebob`», o si el atributo «username» se ha establecido correctamente en «`unclebob`»? Es complicado saberlo por la invocación ya que no es evidente si set es un verbo o un adjetivo.

El autor pretendía que `set` fuera un verbo, pero el contexto de la instrucción `if` parece un adjetivo. La instrucción se lee como «si el atributo `username` se ha establecido previamente en `unclebob`», no como «establecer el atributo `username` en `unclebob` y si funciona, entonces…». Podríamos solucionarlo si cambiamos el nombre de la función `set` por `setAndCheckIfExists`, pero no mejoraría la legibilidad de la instrucción `if`. La verdadera solución es separar el comando de la consulta para evitar la ambigüedad.

```
if (attributeExists("username")) {
   setAttribute("username", "unclebob");
   …
}
```

Mejor excepciones que devolver códigos de error

Devolver códigos de error de funciones de comando es un sutil incumplimiento de la separación de comandos de consulta. Hace que los comandos usados asciendan a expresiones en los predicados de las instrucciones `if`.

```
if (deletePage(page) == E_OK)
```

No padece la confusión entre verbo y adjetivo, pero genera estructuras anidadas. Al devolver un código de error se crea un problema: el invocador debe procesar el error de forma inmediata.

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
     if (configKeys.deleteKey(page.name.makeKey()) == E_OK) {
       logger.log("page deleted");
     } else {
       logger.log("configKey not deleted");
     }
   } else {
     logger.log("deleteReference from registry failed");
   }
} else {
   logger.log("delete failed");
   return E_ERROR;
}
```

Por otra parte, si usa excepciones en lugar de códigos de error, el código de procesamiento del error se puede separar del código de ruta y se puede simplificar:

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
   logger.log(e.getMessage());
}
```

Extraer bloques Try/Catch

Los bloques `try/catch` no son atractivos por naturaleza. Confunden la estructura del código y mezclan el procesamiento de errores con el normal.

Por ello, conviene extraer el cuerpo de los bloques `try` y `catch` en funciones individuales.

```
public void delete(Page page) {
   try {
     deletePageAndAllReferences(page);
   }
   catch (Exception e) {
     logError(e);
   }
}

private void deletePageAndAllReferences(Page page) throws Exception {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
   logger.log(e.getMessage());
}
```

En este caso, la función `delete` es de procesamiento de errores. Es fácil de entender e ignorar. La función `deletePageAndAllReferences` es para los procesos de borrar una página. El procesamiento de errores se puede ignorar. De este modo, la separación facilita la comprensión y la modificación del código.

El procesamiento de errores es una cosa

Las funciones sólo deben hacer una cosa y el procesamiento de errores es un ejemplo. Por tanto, una función que procese errores no debe hacer nada más. Esto implica (como en el ejemplo anterior) que, si una función incluye la palabra clave `try`, debe ser la primera de la función y que no debe haber nada más después de los bloques `catch/finally`.

El imán de dependencias Error.java

La devolución de códigos de error suele implicar que existe una clase o enumeración en la que se definen los códigos de error.

```
public enum Error {
```

```
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
  }
```

Clases como ésta son un *imán para las dependencias*; otras muchas clases deben importarlas y usarlas. Por ello, cuando cambia la enumeración `Error`, es necesario volver a compilar e implementar dichas clases[20]. Esto añade presión a la clase `Error`. Los programadores no quieren añadir nuevos errores porque tendrán que volver a generar e implementarlo todo. Por ello, reutilizan códigos de error antiguos en lugar de añadir otros nuevos.

Al usar excepciones en lugar de códigos de error, las nuevas excepciones son derivaciones de la clase de la excepción. Se pueden añadir sin necesidad de volver a compilar o implementar[21].

No repetirse[22]

Fíjese de nuevo en el Listado 3-1; verá que hay un algoritmo que se repite cuatro veces, en los casos `SetUp`, `SuiteSetUp`, `TearDown` y `SuiteTearDown`. No es fácil detectar esta repetición ya que las cuatro instancias se mezclan con otro código, pero la duplicación es un problema ya que aumenta el tamaño del código y requerirá una modificación cuádruple si alguna vez cambia el algoritmo.

También se cuadriplica el riesgo de errores.

Esta duplicación se remedia gracias al método `include` del Listado 3-7. Vuelva a leer el código y fíjese en cómo se ha mejorado la legibilidad del código reduciendo la duplicación.

La duplicación puede ser la raíz de todos los problemas del *software*. Existen numerosos principios y prácticas para controlarla o eliminarla. Imagine que todas las formas normales de la base de datos de Codd sirvieran

para eliminar la duplicación de datos. Imagine también cómo la programación orientada a objetos concentra el código en clases base que en otros casos serian redundantes. La programación estructurada, la programación orientada a aspecto y la orientada a componentes son, en parte, estrategias para eliminar duplicados. Parece que, desde la aparición de las subrutinas, las innovaciones en desarrollo de *software* han sido un intento continuado por eliminar la duplicación de nuestro código fuente.

Programación estructurada

Algunos programadores siguen las reglas de programación estructurada de Edsger Dijkstra[23]. Dijkstra afirma que todas las funciones y todos los bloques de una función deben tener una entrada y una salida. Estas reglas implican que sólo debe haber una instrucción `return` en una función, que no debe haber instrucciones `break` o `continue` en un bucle y nunca, bajo ningún concepto, debe haber instrucciones `goto`.

Aunque apreciemos los objetivos y disciplinas de la programación estructurada, no sirven de mucho cuando las funciones son de reducido tamaño. Su verdadero beneficio se aprecia en funciones de gran tamaño.

Por tanto, si sus funciones son de tamaño reducido, una instrucción `return`, `break` o `continue` no hará daño alguno y en ocasiones puede resultar más expresiva que la regla de una entrada y una salida. Por otra parte, `goto` sólo tiene sentido en funciones de gran tamaño y debe evitarse.

Cómo crear este tipo de funciones

La creación de *software* es como cualquier otro proceso creativo. Al escribir un informe o un artículo, primero se estructuran las ideas y después el mensaje hasta que se lea bien. El primer borrador puede estar desorganizado, de modo que lo retoca y mejora hasta que se lea de la forma adecuada.

Cuando creo funciones, suelen ser extensas y complicadas, con abundancia de sangrados y bucles anidados. Con extensas listas de

argumentos, nombres arbitrarios y código duplicado, pero también cuento con una serie de pruebas de unidad que abarcan todas y cada una de las líneas de código.

Por tanto, retoco el código, divido las funciones, cambio los nombres y elimino los duplicados. Reduzco los métodos y los reordeno. En ocasiones, elimino clases enteras, mientras mantengo las pruebas.

Al final, consigo funciones que cumplen las reglas detalladas en este capítulo. No las escribo al comenzar y dudo que nadie pueda hacerlo.

Conclusión

Todo sistema se crea a partir de un lenguaje específico del dominio diseñado por los programadores para describir dicho sistema. Las funciones son los verbos del lenguaje y las clases los sustantivos. No es volver a la noción de que los sustantivos y verbos de un documento de requisitos son las clases y funciones de un sistema. Es una verdad mucho más antigua. El arte de la programación es, y ha sido siempre, el arte del diseño del lenguaje.

Los programadores experimentados piensan en los sistemas como en historias que contar, no como en programas que escribir. Recurren a las prestaciones del lenguaje de programación seleccionado para crear un lenguaje expresivo mejor y más completo que poder usar para contar esa historia. Parte de ese lenguaje es la jerarquía de funciones que describen las acciones que se pueden realizar en el sistema. Dichas acciones se crean para usar el lenguaje de dominio concreto que definen para contar su pequeña parte de la historia.

En este capítulo hemos visto la mecánica de la creación de funciones correctas. Si aplica estas reglas, sus funciones serán breves, con nombres correctos, y bien organizadas, pero no olvide que su verdadero objetivo es contar la historia del sistema y que las funciones que escriba deben encajar en un lenguaje claro y preciso que le sirva para contar esa historia.

SetupTeardownIncluder

Listado 3-7
SetupTeardownIncluder.java.

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
   private PageData pageData;
   private boolean isSuite;
   private WikiPage testPage;
   private StringBuffer newPageContent;
   private PageCrawler pageCrawler;

   public static String render(PageData pageData) throws Exception {
     return render(pageData, false);
   }

   public static String render(PageData pageData, boolean isSuite)
     throws Exception {
     return new SetupTeardownIncluder(pageData).render(isSuite);
   }

   private SetupTeardownIncluder(PageData pageData) {
     this.pageData = pageData;
     testPage = pageData.getWikiPage();
     pageCrawler = testPage.getPageCrawler();
     newPageContent = new StringBuffer();
   }

   private String render(boolean isSuite) throws Exception {
     this.isSuite = isSuite;
     if (isTestPage())
       includeSetupAndTeardownPages();
     return pageData.getHtml();
   }

   private boolean isTestPage() throws Exception {
     return pageData.hasAttribute("Test");
   }

   private void includeSetupAndTeardownPages() throws Exception {
     includeSetupPages();
     includePageContent();
     includeTeardownPages();
     updatePageContent();
   }

   private void includeSetupPages() throws Exception {
     if (isSuite)
       includeSuiteSetupPage();
     includeSetupPage();
   }
```

```
private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, “-setup”);
}

private void includeSetupPage() throws Exception {
  include(“SetUp”, “-setup”);
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include(“TearDown”, “-teardown”);
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, “-teardown”);
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception (
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
    String pagePathName = getPathNameForPage(inheritedPage);
    buildIncludeDirective(pagePathName, arg);
  }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
  return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
  WikiPagePath pagePath = pageCrawler.getFullPath(page);
  return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
  newPageContent
    .append(“\n!include ”)
    .append(arg)
    .append(“ .”)
    .append(pagePathName)
    .append(“\n”);
}
```

```
}
```

Bibliografía

- **[KP78]**: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.
- **[PPP02]**: Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.
- **[GOF]**: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison Wesley, 1996.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **[SP72]**: *Structured Programming*, O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

4
Comentarios

«No comente el código incorrecto, reescríbalo».
Brian W. Kernighan y P. J. Plaugher[24]

No hay nada más útil que un comentario bien colocado. No hay nada que colapse más un módulo que comentarios dogmáticos innecesarios. No hay nada más dañino que un comentario antiguo que propague mentiras y desinformación.

Los comentarios no son como la Lista de Schindler. No son pura bondad. De hecho, en el mejor de los casos, son un mal necesario. Si los lenguajes de

programación fueran más expresivos o si pudiéramos dominarlos para expresar nuestras intenciones, no necesitaríamos demasiados comentarios, puede que incluso ninguno.

El uso correcto de los comentarios permite compensar nuestra incapacidad para expresarnos en el código. He usado la palabra incapacidad, a propósito. Los comentarios siempre son fallos. Debemos usarlos porque no siempre sabemos cómo expresarnos sin ellos pero su uso no es motivo de celebración.

Cuando tenga que escribir un comentario, piense si no existe otra forma de expresarse en el código. Siempre que se exprese en el código, debe felicitarse. Siempre que escriba un comentario, debe hacer un gesto de desaprobación y sentir su incapacidad para expresarse.

¿Por qué estoy en contra de los comentarios? Porque mienten. No siempre y no siempre intencionadamente, pero lo hacen. Cuando más antiguo es un comentario y más se aleja del código que describe, mayor es la probabilidad de que sea equivocado. El motivo es sencillo. Los programadores no los pueden mantener.

El código cambia y evoluciona. Los fragmentos cambian de lugar, se bifurcan, se reproducen y se vuelven a combinar para crear quimeras. Desafortunadamente, los comentarios no siempre siguen el ritmo, no siempre pueden hacerlo y suelen separarse del código que describen y se convierten en huérfanos sin precisión alguna. Por ejemplo, fíjese en lo que sucede con este comentario y la línea que pretendía describir:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
   “[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s” +
   “[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT”;
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Ejemplo: «Tue, 02 Apr 2003 22:18:49 GMT»
```

Seguramente se añadieron después otras variables de instancia entre la constante `HTTP_DATE_REGEXP` y su comentario explicativo.

Se podría afirmar que los programadores deben ser lo bastante disciplinados como para mantener los comentarios actualizados, relevantes y precisos. De acuerdo, debería, pero esa energía debería invertirse en crear

código claro y expresivo que no necesite comentario alguno.

Los comentarios imprecisos son mucho peor que la ausencia de comentarios. Suelen confundir al usuario. Generan expectativas que nunca se cumplen. Definen reglas que no deben seguirse en absoluto.

La verdad sólo se encuentra en un punto: el código. Sólo el código puede contar lo que hace. Es la única fuente de información precisa. Por tanto, aunque los comentarios sean necesarios en ocasiones, dedicaremos nuestra energía a minimizarlos.

Los comentarios no compensan el código incorrecto

Una de las principales motivaciones para crear comentarios es el código incorrecto. Creamos un módulo y sabemos que es confuso y está desorganizado. Sabemos que es un desastre y entonces decidimos comentarlo. Error. Mejor límpielo.

El código claro y expresivo sin apenas comentarios es muy superior al código enrevesado y complejo con multitud de comentarios. En lugar de perder tiempo escribiendo comentarios que expliquen el desastre cometido, dedíquelo a solucionarlo.

Explicarse en el código

En ocasiones, el código es un pobre vehículo de expresión. Desafortunadamente, muchos programadores lo entienden como que el código no es un buen medio de expresión. Esto es falso. ¿Qué prefiere ver? Esto:

```
// Comprobar si el empleado tiene derecho a todos los beneficios
if ((employee.flags & HOURLY_FLAG) &&
   (employee.age > 65))
```

o esto otro:

```
if (employee.isEligibleForFullBenefits())
```

Apenas se tardan unos segundos en explicar nuestras intenciones en el código. En muchos casos, basta con crear una función que diga lo mismo que

el comentario que pensaba escribir.

Comentarios de calidad

Algunos comentarios son necesarios o beneficiosos. Veremos algunos de los que considero válidos. No obstante, recuerde que el único comentario realmente bueno es el que no tiene que escribir.

Comentarios legales

En ocasiones, nuestros estándares corporativos de creación de código nos obligan a crear determinados comentarios por motivos legales. Por ejemplo, los comentarios de derechos de autor son necesarios y deben incluirse al inicio de cada archivo.

El siguiente encabezado de comentario se incluye de forma estándar al inicio de todos los archivos fuente de FitNesse. Nuestro IDE evita que este comentario parezca sobrante replegándolo de forma automática.

```
  // Copyright (C) 2003,2004,2005 de Object Mentor, Inc. Todos los derechos reservados.
  // Publicado bajo las condiciones de la Licencia pública general GNU versión 2 o posterior.
```

Este tipo de comentarios no deben ser contratos ni tomos legales. Siempre que sea posible, haga referencia a una licencia estándar o a otro documento externo en lugar de incluir todos los términos y condiciones en el comentario.

Comentarios informativos

En ocasiones es útil proporcionar información básica con un comentario. Por ejemplo, el siguiente comentario explica el valor devuelto por un método abstracto:

```
  // Devuelve una instancia del elemento Responder probado.
  protected abstract Responder responderInstance();
```

Estos comentarios pueden ser útiles, pero es mejor usar el nombre de la función para transmitir la información siempre que sea posible. Por ejemplo,

en este caso el comentario sería redundante si cambiamos el nombre de la función por responderBeingTested. Veamos un ejemplo mejor:

```
// el formato coincide con kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
   "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

En este caso, el comentario nos indica que la expresión regular debe coincidir con una fecha y una hora con el formato aplicado por la función `SimpleDateFormat.format` con la cadena de formato especificada.

Hubiera resultado mejor y más claro si el código se hubiera cambiado a una clase especial que convirtiera los formatos de fechas y horas. De ese modo el comentario habría sido superfluo.

Explicar la intención

En ocasiones, un comentario es algo más que información útil sobre la implementación y proporciona la intención de una decisión. En el siguiente caso, vemos una interesante decisión documentada por un comentario. Al comparar dos objetos, el autor decidió ordenar los objetos de su clase por encima de los objetos de otra.

```
public int compareTo(Object o)
{
   if (o instanceof WikiPagePath)
   {
     WikiPagePath p = (WikiPagePath) o;
     String compressedName = StringUtil.join(names, "");
     String compressedArgumentName = StringUtil.join(p.names, "");
     return compressedName.compareTo(compressedArgumentName);
   }
   return 1; // somos mayores porque somos el tipo correcto.
}
```

Veamos otro ejemplo mejor. Puede que no esté de acuerdo con la solución del programador, pero al menos sabe lo que intentaba hacer.

```
public void testConcurrentAddWidgets() throws Exception {
   WidgetBuilder widgetBuilder =
     new WidgetBuilder(new Class[](BoldWidget.class));
   String text = "''''bold text'''";
   ParentWidget parent =
     new BoldWidget(new MockWidgetRoot(), "''''bold text''''");
   AtomicBoolean failFlag = new AtomicBoolean();
   failFlag.set(false);
```

```
    //Nuestro mejor intento de obtener una condición de carrera
    //creando un gran número de procesos.
    for (int i = 0; i < 25000; i++) {
      WidgetBuilderThread widgetBuiIderThread =
        new WidgetBuilderThread(WidgetBuilder, text, parent, failFlag);
      Thread thread = new Thread(WidgetBuilderThread);
      thread.start();
    }
    assertEquals(false, failFlag.get());
  }
```

Clarificación

En ocasiones, basta con traducir el significado de un argumento o valor devuelto en algo más legible. Por lo general, conviene buscar la forma de que el argumento o el valor devuelto sean claros por sí mismos; pero cuando forma parte de una biblioteca estándar o de código que no se puede alterar, un comentario aclarativo puede ser muy útil.

```
  public void testCompareTo() throws Exception
  {
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1); // b > a
    assertTrue(ab.compareTo (aa) == 1); // ab > aa
    assertTrue(bb.compareTo(ba) == 1); // bb > ba
  }
```

Pero también existe el riesgo de que un comentario aclarativo sea incorrecto. En el ejemplo anterior, compruebe lo difícil que resulta comprobar si los comentarios son correctos. Esto explica por qué la clarificación es necesaria y también arriesgada. Por ello, antes de escribir estos comentarios, asegúrese de que no hay una solución mejor y también de que sean precisos.

Advertir de las consecuencias

En ocasiones es muy útil advertir a otros programadores de determinadas consecuencias. Por ejemplo, el siguiente comentario explica por qué un determinado caso de prueba está desactivado:

```
// No ejecutar a menos
// que le sobre tiempo.
public void _testWithReallyBigFile()
{
   writeLinesToFile(10000000);

   response.setBody(testFile);
   response.readyToSend(this);
   String responseString = output.toString();
   assertSubString("Content-Length: 1000000000", responseString);
   assertTrue(bytesSent > 1000000000);
}
```

En la actualidad, evidentemente, desactivaríamos la prueba por medio del atributo `@Ignore` con la correspondiente cadena explicativa: `@Ignore("Takes too long to run")`, pero antes de la aparición de JUnit 4, era habitual añadir un guion bajo delante del nombre del método. El comentario realizaba su cometido. Veamos otro ejemplo:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
   //SimpleDataFormat no es compatible con procesos,
   //por lo que debe crear cada instancia de forma independiente.
   SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
   df.setTimeZone (TimeZone.getTimeZone ("GMT"));
   return df;
}
```

Seguramente conozca soluciones mejores para este problema. Estoy de acuerdo, pero el comentario es perfectamente razonable. Evita que un programador use un inicializador estático por motivos de eficacia.

Comentarios TODO

En ocasiones conviene usar notas con forma de comentarios `//TODO`. En el

siguiente caso, el comentario `TODO` explica por qué la función tiene una implementación incorrecta y cuál debe ser su futuro.

```
// TODO-MdM no son necesarios
// Esperamos que desaparezca en el modelo definitivo
protected VersionInfo makeVersion() throws Exception
{
   return null;
}
```

`TODO` son tareas que el programador piensa que debería haber hecho pero que no es así. Pueden ser un recordatorio para eliminar una función obsoleta o una petición para resolver un problema.

Pueden ser una solicitud para buscar un nombre más adecuado o para realizar un cambio que dependa de un evento planeado. Sea lo que sea, no es excusa para mantener código incorrecto en el sistema.

En la actualidad, muchos IDE cuentan con funciones especiales para localizar comentarios `TODO`, por lo que seguramente no se pierda. Sin embargo, no colapse el código con estos comentarios. Examínelos y elimine todos los que pueda.

Amplificación

Se puede usar un comentario para amplificar la importancia de algo que, en caso contrario, parecería irrelevante.

```
String listItemContent = match.group(3).trim();
// el recorte es importante. Elimina los espacios iniciales
// que harían que el elemento se reconociera como
// otra lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadoc en API públicas

No hay nada más útil y satisfactorio que una API pública bien descrita. Los javadoc de la biblioteca estándar de Java son un ejemplo. Sería muy complicado crear programas de Java sin ellos.

Si usa una API pública, debe crear javadoc de calidad para la misma, pero recuerde el siguiente consejo a lo largo del capítulo: los javadoc pueden ser

tan ambiguos, amplios y descorteses como cualquier otro tipo de documento.

Comentarios incorrectos

Muchos comentarios pertenecen a esta categoría. Suelen ser excusas de código pobre o justificaciones de decisiones insuficientes, algo así como si el programador se hablara a sí mismo.

Balbucear

Añadir un comentario sin razón o porque el proceso lo requiere es un error. Si decide escribir un comentario, tómese el tiempo necesario para asegurarse de que sea el mejor que puede redactar.

El siguiente ejemplo es de FitNesse, donde un comentario sin duda sería de utilidad, pero el autor tenía prisa o no prestó demasiada atención. Su balbuceo generó un enigma:

```
public void loadProperties()
{
   try
   {
     String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
     FileInputStream propertiesStream = new FileInputStream(propertiesPath);
     loadedProperties.load(propertiesStream);
   }
   catch(IOException e)
   {
     // Si no hay archivos de propiedades significan que cargan las predeterminadas
   }
}
```

¿Qué significa el comentario del bloque `catch`? Seguro que algo para el autor, pero el significado no está claro. Aparentemente, si se genera `IOException`, significa que no hay archivo de propiedades y, en ese caso, se cargan los valores predeterminados. ¿Pero quién carga los valores predeterminados? ¿Se cargan antes de la invocación de `loadProperties.load` o `loadProperties.load` captura la excepción, carga los valores predeterminados y después nos pasa la excepción para que la ignoremos? ¿O será que `loadProperties.load` carga todos los valores

predeterminados antes de intentar abrir el archivo? ¿Intentaba el autor consolarse por dejar el bloque catch vacío? Ésta es la posibilidad más temida, ¿se estaba diciendo que volviera más tarde para crear el código para cargar los valores predeterminados?

Nuestro único recurso es examinar el código en otras partes del sistema para determinar qué sucede. Cualquier comentario que le obligue a buscar su significado en otro módulo ha fallado en su intento de comunicación y no merece los bits que consume.

Comentarios redundantes

El Listado 4-1 muestra una sencilla función con un comentario de encabezado totalmente redundante. Seguramente se tarde más en leer que el propio código.

Listado 4-1
waitForClose.

```
// Método de utilidad devuelto cuando this.closed es true. Genera una excepción
// si se alcanza el tiempo de espera.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
   if (!closed)
   {
     wait(timeoutMillis);
     if(!closed)
       throw new Exception (“MockResponseSender could not be closed”);
   }
}
```

¿Para qué sirve este comentario? No es más informativo que el código. No lo justifica ni transmite la intención ni la lógica. No es más fácil de leer que el código. De hecho, es menos preciso y obliga al lector a aceptar la falta de precisión en lugar de a entenderlo. Es como un vendedor de coches de segunda menos que le asegura que no hace falta revisar el motor.

Fíjese ahora en la legión de javadoc inútiles y redundantes del Listado 4-2, obtenido de Tomcat. Estos comentarios únicamente ensucian y oscurecen el código. No tienen ninguna función documental. Para empeorar las cosas,

sólo le mostramos algunos. El módulo tiene muchos más.

Listado 4-2
ContainerBase.java (Tomcat).

```
public abstract class ContainerBase
   implements Container, Lifecycle, Pipeline,
   MBeanRegistration, Serializable {

   /**
   * Retardo del procesador para este componente.
   */
   protected int backgroundProcessorDelay = -1;

   /**
   * Compatibilidad con eventos de ciclo vital de este componente.
   */
   protected LifecycleSupport lifecycle =
     new LifecycleSupport(this);

   /**
   * Escuchadores de eventos de contenedor de este contenedor.
   */
   protected ArrayList listeners = new ArrayList();

   /**
   * Implementación Loader a la que se asocia este contenedor.
   */
   protected Loader loader = null;

   /**
   * Implementación Logger a la que se asocia este contenedor.
   */
   protected Log logger = null;

   /**
   * Nombre de registrador asociado.
   */
   protected String logName = null;

   /**
   * Implementación Manager a la que se asocia este contenedor.
   */
   protected Manager manager = null;

   /**
   * Clúster al que se asocia este contenedor.
   */
   protected Cluster cluster = null;

   /**
   * Nombre legible de este contenedor.
```

```
    */
    protected String name = null;

    /**
    * Contenedor principal de este contenedor.
    */
    protected Container parent = null;

    /**
    * Cargador de clase principal que configurar al instalar un elemento
    * Loader.
    */
    protected ClassLoader parentClassLoader = null;

    /**
    * Objeto Pipeline al que se asocia este contenedor.
    */
    protected Pipeline pipeline = new StandardPipeline(this);

    /**
    * Objeto Realm al que se asocia este contenedor.
    */
    protected Realm realm = null;

    /**
    * Objeto DirContext de recursos al que se asocia este contenedor.
    */
    protected DirContext resources = null;
```

Comentarios confusos

En ocasiones, a pesar de las buenas intenciones, un programador realiza una afirmación en sus comentarios que no es del todo precisa. Fíjese otra vez en el comentario redundante y confuso del Listado 4-1.

¿Sabe por qué es confuso? El método no devuelve nada cuando `this.closed` se convierte en `true`. Devuelve algo si `this.closed` es `true`; en caso contrario, espera y genera una excepción si `this.closed` no es `true`.

Este sutil fragmento, oculto en un comentario más difícil de leer que el cuerpo del código, puede hacer que otro programador invoque la función con la esperanza de que devuelva algo cuando `this.closed` sea `true`. Ese pobre programador se encontrará en una sesión de depuración intentando determinar por qué el código se ejecuta tan lentamente.

Comentarios obligatorios

Es una locura tener una regla que afirme que todas las funciones deben tener un javadoc o que todas las variables deben tener un comentario. Este tipo de comentarios ensucian el código y generan confusión y desorganización. Por ejemplo, los javadoc obligatorios para todas las funciones crean abominaciones como el Listado 4-3. No sirven de nada, complican el código y constituyen posibles engaños y desorientaciones.

Listado 4-3

```
/**
   *
   * @param title El título del CD
   * @param author El autor del CD
   * @param tracks El número de pistas del CD
   * @param durationInMinutes La duración del CD en minutos
   */
   public void addCD(String title, String author,
        int tracks, int durationInMinutes) {
     CD cd = new CD();
     cd.title = title;
     cd.author = author;
     cd.tracks = tracks;
     cd.duration = duration;
     cdList.add(cd);
   }
```

Comentarios periódicos

En ocasiones, se añade un comentario al inicio de un módulo cada vez que se edita. Estos comentarios acumulan una especie de registro de todos los cambios realizados. He visto módulos con decenas de páginas con estas entradas.

```
* Cambios (11-Oct-2001)
* -----------------------------
* 11-Oct-   : Reorganización de la clase y cambio a un nuevo paquete
2001
*             com.jrefinery.date (DG);
* 05-Nov-   : Se añade un método getDescription() y se elimina la clase
2001          NotableDate (DG);
```

```
* 12-Nov-   : IBD requiere el método setDescription(), una vez eliminada la
2001          clase NotableDate
*             (DG); se cambian getPreviousDayOfWeek(), getFollowingDayOfWeek()
*             y getNearestDayOfWeek() para corregir errores (DG);
* 05-Dic-   : Error corregido en la clase SpreadsheetDate (DG);
2001
* 29-May-   : Se transfieren todas las constantes de mes a una interfaz
2002
*             independiente (MonthConstants) (DG);
* 27-Ago-   : Error corregido en el método addMonths(), gracias a Nálevka Petr
2002          (DG);
* 03-Oct-   : Errores indicados por Checkstyle (DG) corregidos;
2002
* 13-Mar-   : Implementación de Serializable (DG);
2003
* 29-May-   : Error corregido en el método addMonths (DG);
2003
* 04-Sep-   : Implementación de Comparable. Actualización de los javadoc
2003          isInRange (DG);
* 05-Ene-   : Error corregido en el método addYears() (1096202) (DG);
2005
```

Hace tiempo hubo una buena razón para crear y mantener estas entradas de registro al inicio de cada módulo. Carecíamos de sistemas de control de código fuente que se encargaran de ello, pero en la actualidad, estas entradas son elementos sobrantes que complican los módulos. Debe eliminarlas totalmente.

Comentarios sobrantes

En ocasiones vemos comentarios que simplemente sobran. Restan importancia a lo evidente y no ofrecen información nueva.

```
/**
* Constructor predeterminado.
*/
protected AnnualDateRule() {
}
```

¿En serio? ¿Y este otro?:

```
/** Día del mes. */
private int dayOfMonth;
```

Y aquí el parangón de la redundancia:

```
/**
* Devuelve el día del mes.
```

```
 *
 * @return el día del mes.
 */
 public int getDayOfMonth() {
    return dayOfMonth;
 }
```

Estos comentarios son tan inservibles que aprendemos a ignorarlos. Al leer el código, la vista los salta. Con el tiempo, los comentarios empiezan a mentir cuando cambia el código que les rodea.

El primer comentario del Listado 4-4 parece correcto[25]. Explica por qué se ignora el bloque `catch`, pero el segundo comentario sobra. Parece que el programador estaba tan frustrado con crear bloques `try/catch` en la función que necesitaba explotar.

Listado 4-4
startSending.

```
 private void startSending()
 {
    try
    {
      doSending();
    }
    catch(SocketException e)
    {
      // normal, alguien ha detenido la solicitud.
    }
    catch(Exception e)
    {
      try
      {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
      }
      catch(Exception e1)
      {
        //¡Un respiro!
      }
    }
 }
```

En lugar de explotar en un comentario sin sentido, el programador debería haber sabido que su frustración se podría aliviar mejorando la estructura del código. Tendría que haber centrado su energía en extraer el último bloque `try/catch` en una función independiente, como muestra el

Listado 4-5.

Listado 4-5
startSending (refactorizado).

```
private void startSending()
{
   try
   {
     doSending();
   }
   catch(SocketException e)
   {
     // normal. Alguien ha detenido la solicitud.
   }
   catch(Exception e)
   {
     addExceptionAndCloseResponse(e);
   }
}

private void addExceptionAndCloseResponse(Exception e)
{
   try
   {
     response.add(ErrorResponder.makeExceptionString(e));
     response.closeAll();
   }
   catch(Exception e1)
   {
   }
}
```

Cambie la tentación de crear elementos sobrantes por la determinación de limpiar su código. Mejorará como programador y será más fácil.

Comentarios sobrantes espeluznantes

Los javadoc también pueden ser innecesarios. ¿Para qué sirven los siguientes javadoc (de una conocida biblioteca) de código abierto? La respuesta: para nada. Son comentarios redundantes creados en un intento equivocado de redactar documentación.

```
/** El nombre. */
private String name;

/** La versión. */
```

```
private String version;

/** El licenceName. */
private String licenceName;

/** La versión. */
private String info;
```

Vuelva a leer los comentarios. ¿Detecta el error de corta y pega? Si los autores no prestan atención al escribir sus comentarios (o al pegarlos), ¿por qué se espera que sean de utilidad para los lectores?

No usar comentarios si se puede usar una función o una variable

Fíjese en el siguiente código:

```
// ¿el módulo de la lista global <mod> depende del
// subsistema del que formamos parte?
if (smodule.getDependSubsystems().contains(subSysMod.getSubsystem()))
```

Se podría cambiar sin el comentario de esta forma:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

El autor del código original seguramente escribió primero el comentario (improbable) y después el código para ajustarlo al comentario. Sin embargo, el autor tendría que haber refactorizado el código, como hice yo, para poder eliminar el comentario.

Marcadores de posición

En ocasiones los programadores marcan una determinada posición en un archivo. Por ejemplo, recientemente encontré esto en un programa:

```
// Acciones ////////////////////////////////
```

Son escasas las ocasiones en las que tiene sentido agrupar funciones bajo esta estructura. Por lo general, debe eliminarse, sobre todo la molesta hilera de barras al final.

Piénselo de esta forma. Estas estructuras son atractivas si no las usa demasiado. Por ello, úselas esporádicamente y sólo cuando el beneficio sea

significativo. Si las usa en exceso, acabarán por ser ignoradas.

Comentarios de llave de cierre

En ocasiones, los programadores incluyen comentarios especiales en llaves de cierre, como en el Listado 4-6. Aunque pueda tener sentido en funciones extensas con estructuras anidadas, únicamente estorba a las funciones encapsuladas y de pequeño tamaño que nos gustan. Por ello, si siente el deseo de marcar sus llaves de cierre, pruebe a reducir el tamaño de sus funciones.

Listado 4-6
wc.java.

```
public class wc {
   public static void main(String[] args) {
     BufferedReader in = new BufferedReader(new InputstreamReader(System.in));
     String line;
     int lineCount = 0;
     int charCount = 0;
     int wordCount = 0;
     try {
       while ((line = in.readLine()) != null) {
         lineCount++;
         charCount += line.length();
         String words[] = line.split("\\W");
         wordCount += words.length;
       } //while
       System.out.println("wordCount = " + wordCount);
       System.out.println("lineCount = " + lineCount);
       System.out.println("charCount = " + charCount);
     } // try
     catch (IOException e) {
       System.err.println("Error: " + e.getMessage());
     } //catch
   } //main
}
```

Asignaciones y menciones

```
/* Añadido por Rick */
```

Los sistemas de control de código fuente recuerdan a la perfección quién ha añadido qué y cuándo. No es necesario plagar el código con pequeñas

menciones. Puede pensar que estos comentarios son útiles y que ayudan a otros a hablar sobre el código, pero en realidad sobreviven durante años y cada vez son menos precisos y relevantes. El sistema de control de código fuente es el punto idóneo para este tipo de información.

Código comentado

No hay nada más odioso que el código comentado. ¡No lo haga!

```
InputStreamResponse response = new inputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Los lectores que vean código comentado no tendrán el valor de borrarlo. Pensarán que está ahí por algo y que es demasiado importante para borrarlo. Por ello, el código comentado se acumula como los sedimentos en una botella de vino malo.

Fíjese en este fragmento de apache commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
   writeEnd();
   this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
   this.pngBytes = null;
}
return this.pngBytes;
```

¿Por qué hay dos líneas comentadas? ¿Son importantes? ¿Se han conservado como recordatorio de un cambio inminente o es algo que alguien comentó hace años y no se ha preocupado de limpiar? Hubo una época, en la década de 1960, en la que el código comentado pudo ser útil, pero hace tiempo que contamos con buenos sistemas de control de código fuente, sistemas que recuerdan el código por nosotros. Ya no tenemos que comentarlo. Elimínelo. No lo perderá. Se lo aseguro.

Comentarios HTML

El HTML en comentarios de código fuente es una aberración, como puede apreciar en el siguiente fragmento. Dificulta la lectura de los comentarios donde debería ser más fácil; el editor o IDE. Si los comentarios se van a extraer con una herramienta (como Javadoc) para mostrarlos en una página Web, debe ser responsabilidad de dicha herramienta y no del programador el adornar los comentarios con el correspondiente HTML.

```
/**
*   Tarea para ejecutar pruebas de aceptación.
*   Esta tarea ejecuta pruebas de aceptación y publica los resultados.
*   <p/>
*   <pre>
*   Uso:
*   &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
*             classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
*             classpathref=&quot;classpath&quot; /&gt;
*   OR
*   &lt;taskdef classpathref=&quot;classpath&quot;
*                  resource=&quot;tasks.properties&quot; /&gt;
*   <p/>
*   &lt;execute-fitnesse-tests
*             suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
*             fitnesseport=&quot;8082&quot;
*             resultsdir=&quot;$(results.dir)&quot;
*             resultshtmlpage=&quot;fit-results.html&quot;
*             classpathref=&quot;classpath&quot; /&gt;
*   </pre>
*/
```

Información no local

Si tiene que escribir un comentario, asegúrese de que describa el código que le rodea. No ofrezca información global del sistema en el contexto de un comentario local. Fíjese en el siguiente comentario javadoc. Aparte de su terrible redundancia, también ofrece información sobre el puerto predeterminado y la función no tiene control alguno sobre el puerto predeterminado. El comentario no describe la función sino otra parte distinta

del sistema. Evidentemente, no hay garantías de que el comentario cambie cuando lo haga el código que contiene el valor predeterminado.

```
/**
* Puerto para ejecutar fitnesse. El predeterminado es <b>8082</b>.
*
* @param fitnessePort
*/
public void setFitnessePort(int fitnessePort)
{
   this.fitnessePort = fitnessePort;
}
```

Demasiada información

No incluya en sus comentarios interesantes reflexiones históricas ni irrelevantes descripciones de detalles. El siguiente comentario se ha extraído de un módulo diseñado para probar que una función puede codificar y descodificar base64. Aparte del número RFC, el lector de este código no necesita la información obsoleta que contiene el comentario.

```
/*
   RFC 2045 - Extensiones Multipropósito de correo de Internet (MIME)
   Primera parte: Formato del Cuerpo de los Mensajes de Internet
   sección 6.8. Codificación de transferencia de contenidos Base64
   El proceso de codificación representa grupos de 24 bits de la entrada
   como cadenas de salida de 4 caracteres codificados. Procediendo de
   izquierda a derecha, se forma un grupo de 24 bits de entrada
   concatenando 3 grupos de 8 bits de entrada. Estos 24 bits se tratan
   como 4 grupos concatenados de 6 bits, cada uno de los cuales se
   traduce en un solo dígito del alfabeto base64. Cuando se codifica un
   flujo de bits mediante la codificación base64, el flujo de bits se
   debe considerar ordenado con el bit más significativo primero. Esto
   es, el primer bit del flujo será el bit de orden más alto en el
   primer byte de 8 bits, y el octavo bit será el de orden más bajo en
   el primer byte de 8 bits, y así sucesivamente.
*/
```

Conexiones no evidentes

La conexión entre un comentario y el código que describe debe ser evidente. Si se ha preocupado de escribir un comentario, lo mínimo es que el lector que lo vea entienda a qué se refiere. Fíjese en este comentario obtenido de apache commons:

```
  /*
  * comienza con una matriz de tamaño suficiente para albergar todos los
pixeles
  * (más bytes de filtro), y 200 bytes adicionales para la información de
encabezado
  */
  this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

¿Qué es un *byte* de filtro? ¿Está relacionado con `+1`? ¿O con `*3`? ¿Con ambos? ¿Es un pixel un *byte*? ¿Por qué 200? La función de un comentario es explicar código que no se explica por sí mismo. Es una lástima que un comentario requiera su propia explicación.

Encabezados de función

Las funciones breves apenas requieren explicación. Un nombre bien elegido para una función que hace una cosa suele ser mejor que un encabezado de comentario.

Javadocs en código no público

A pesar de la utilidad de los javadoc para las API públicas, no sirven para código no dirigido a consumo público. La generación de páginas javadoc para clases y funciones de un sistema no suele ser útil y la formalidad adicional de los comentarios javadoc no es más que una distracción.

Ejemplo

Escribí el módulo del Listado 4-7 para la primera versión de *XP Immersion*. Debía ser un ejemplo de estilo incorrecto de creación de código y comentarios. Después, Kent Beck refactorizó este código en algo mucho más atractivo delante de varios alumnos. Posteriormente, adapté el ejemplo para mi libro *Agile Software Development, Principles, Patterns, and Practices* y para el primero de mis artículos *Craftsman* publicados en la revista *Software Development*.

Lo que me fascina de este módulo es que hubo un tiempo en que muchos

lo hubiéramos considerado bien documentado. Ahora vemos que es un auténtico desastre. A ver cuántos problemas detecta en los comentarios.

Listado 4-7
GeneratePrimes.java.

```
/**
* Esta clase genera números primos hasta la cantidad máxima especificada por
el
* usuario. El algoritmo usado es la Criba de Eratóstenes.
* <p>
* Eratóstenes de Cirene, 276 a. C., Cirene, Libia -
* 194 a. C., Alejandría. El primer hombre en calcular la
* circunferencia de la Tierra. También trabajó con calendarios
* con años bisiestos y fue responsable de la Biblioteca de Alejandría.
* <p>
* El algoritmo es muy simple. Dada una matriz de enteros
* empezando por el 2, se tachan todos los múltiplos de 2. Se busca el
siguiente
* entero sin tachar y se tachan todos sus múltiplos.
* Repetir hasta superar la raíz cuadrada del valor
* máximo.
*
* ©author Alphonse
* ©version 13 Feb 2002 atp
*/
import java.util.*;

public class GeneratePrimes
{
   /**
   * @param maxValue es el límite de generación.
   */
   public static int[] generatePrimes(int maxValue)
   {
     if (maxValue >= 2) //el único caso válido
     {
       // declaraciones
       int s = maxValue + 1; // tamaño de la matriz
       boolean[] f = new boolean[s];
       int i;

       // inicializar la matriz en true.
       for (i = 0; i < s; i++)
         f[i] = true;

       // eliminar los números no primos conocidos
       f[0] = f[1] = false;

       // cribar
       int j;
```

```
        for (i = 2; i < Math.sqrt(s) + 1; i++)
        {
          if (f[i]) // si no está tachado, tachar sus múltiplos.
          {
            for (j = 2 * i; j < s; j += i)
              f[j] = false; // el múltiplo no es primo
          }
        }

        // ¿cuántos primos hay?
        int count = 0;
        for (i = 0; i < s; i++)
        {
          if (f[i])
            count++; // contador.
        }

        int[] primes = new int[count];

        // enviar primos al resultado
        for (i = 0, j = 0; i < s; i++)
        {
          if (f[i]) // si es primo
            primes[j++] = i;
        }

        return primes; // devolver los primos
      }
      else // maxValue < 2
        return  new  int[0];  //  devolver  matriz  null  si  la  entrada  no  es
correcta.
    }
  }
```

En el Listado 4-8 puede ver una versión refactorizada del mismo módulo. Se ha limitado considerablemente el uso de comentarios. Hay sólo dos en todo el módulo y ambos claramente descriptivos.

Listado 4-8
PrimeGenerator.java (refactorizado).

```
  /**
  * Esta clase genera números primos hasta la cantidad máxima especificada por
el
  * usuario. El algoritmo usado es la Criba de Eratóstenes. Dada una matriz de
enteros
  * empezando por el 2: buscar el primer entero sin tachar y tachar todos sus
  * múltiplos. Repetir hasta que no haya más múltiplos en la matriz.
  */
  public class PrimeGenerator
  {
```

```
    private static boolean[] crossedOut;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
      if (maxValue < 2)
        return new int[0];
      else
      {
        uncrossIntegersUpTo(maxValue);
        crossOutMultiples();
        putUncrossedIntegersIntoResult();
        return result;
      }
    }

    private static void uncrossIntegersUpTo(int maxValue)
    {
      crossedOut = new boolean[maxValue + 1];
      for (int i = 2; i < crossedOut.length; i++)
        crossedOut[i] = false;
    }

    private static void crossOutMultiples()
    {
      int limit = determineIterationLimit();
      for (int i = 2; i <= limit; i++)
        if (notCrossed(i))
          crossOutMultiplesOf(i);
    }

    private static int determineIterationLimit()
    {
      // Cada múltiplo en la matriz tiene un factor primordial que
      // es menor o igual que la raíz del tamaño de la matriz,
      // entonces no tenemos que tachar múltiplos de números
      // más grande que esa raíz.
      double iterationLimit = Math.sqrt(crossedOut.length);
      return (int) iterationLimit;
    }

    private static void crossOutMultiplesOf (int i)
    {
      for (int multiple = 2 * i;
          multiple < crossedOut.length;
          multiple += i)
        crossedOut[multiple] = true;
    }

    private static boolean notCrossed(int i)
    {
      return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult()
```

```
    {
      result = new int[numberOfUncrossedIntegers()];
      for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          result[j++] = i;
    }

    private static int numberOfUncrossedIntegers()
    {
      int count = 0;
      for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          count++;

      return count;
    }
  }
```

Se podría decir que el primer comentario es redundante ya que es muy similar a la función `generatePrimes`, pero creo que muestra mejor el algoritmo al lector, motivo por el que lo he mantenido. El segundo argumento es sin duda necesario. Explica la lógica del uso de la raíz cuadrada como límite del bucle. No encontré otro nombre de variable más sencillo ni otra estructura de código que lo aclarara más. Por otra parte, el uso de la raíz cuadrada podría resultar presuntuoso. ¿Realmente se ahorra tanto tiempo limitando la iteración a la raíz cuadrada? ¿El cálculo de la raíz cuadrada llevaría más tiempo del que se ahorra? Conviene analizarlo. El uso de la raíz cuadrada como límite de iteración satisface al viejo *hacker* de C y de lenguajes de ensamblado de mi interior, pero no estoy convencido de que merezca el tiempo y el esfuerzo que los demás puedan dedicar a entenderlo.

Bibliografía

- **[KP78]**: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

5
Formato

Cuando los usuarios miran entre bastidores, queremos que queden

impresionados por el atractivo, la coherencia y la atención al detalle que perciben. Queremos que el orden les sorprenda, que abran los ojos con asombro cuando se desplacen por los módulos. Queremos que aprecien que se trata de un trabajo de profesionales. Si ven una masa amorfa de código que parece escrito por un grupo de marineros borrachos, es probable que piensen que sucederá lo mismo en otros aspectos del proyecto.

Debe preocuparse por el formato de su código. Debe elegir una serie de reglas sencillas que controlen el formato del código y después aplicarlas de forma coherente. Si trabaja en equipo, debe acordar una serie de reglas que todos los miembros deben cumplir. También es muy útil usar una herramienta automatizada que se encargue de aplicar las reglas.

La función del formato

En primer lugar, debe ser claro. El formato de código es importante, demasiado importante como para ignorarlo y también demasiado importante como para tratarlo de forma religiosa. El formato del código se basa en la comunicación y la comunicación debe ser el principal pilar de un desarrollador profesional.

Puede que piense que conseguir que algo funcione es la principal preocupación de un programador profesional. Espero que este libro le haga cambiar de idea. La funcionalidad que cree hoy es muy probable que cambie en la siguiente versión, pero la legibilidad de su código afectará profundamente a todos los cambios que realice. El estilo del código y su legibilidad establecen los precedentes que afectan a la capacidad de mantenimiento y ampliación mucho después de que el código cambie. Su estilo y su disciplina sobrevivirán, aunque el código no lo haga.

Veamos qué aspectos del formato nos permiten comunicarnos mejor.

Formato vertical

Comencemos por el tamaño vertical. ¿Qué tamaño debe tener un archivo

fuente? En Java, el tamaño de los archivos está relacionado con el tamaño de las clases, como veremos más adelante. Por el momento, nos detendremos en el tamaño de los archivos.

¿Qué tamaño tienen la mayoría de archivos fuente de Java? Existe una amplia gama de tamaños e importantes diferencias de estilo, como se aprecia en la figura 5.1.

Figura 5.1. Escala LOG de distribuciones de longitud de archivos (altura del cuadro = sigma).

Se describen siete proyectos: Junit, FitNesse, testNG, Time and Money, JDepend, Ant y Tomcat. Las líneas que cruzan los cuadros muestran la longitud máxima y mínima de cada proyecto. El cuadro muestra aproximadamente un tercio (una desviación estándar[26]) de los archivos. La parte central del cuadro es la media. Por tanto, el tamaño de archivo medio del proyecto FitNesse es de 65 líneas y un tercio de los archivos ocupan entre 40 y 100+ líneas.

El mayor archivo de FitNesse tiene unas 400 líneas y el de menor tamaño, 6. Es una escala de registro, de modo que la pequeña diferencia de posición vertical supone una gran diferencia en tamaño absoluto.

Junit, FitNesse y Time and Money tienen archivos relativamente pequeños. Ninguno supera las 500 líneas y la mayoría tienen menos de 200. Tomcat y Ant, por su parte, tienen archivos con varios miles de líneas de

longitud y más de la mitad superan las 200.

¿Qué significa todo esto? Aparentemente se pueden crear sistemas (FitNesse se aproxima a las 50 000 líneas) a partir de archivos de unas 200 líneas de longitud, con un límite máximo de 500. Aunque no debería ser una regla, es un intervalo aconsejable. Los archivos de pequeño tamaño se entienden mejor que los grandes.

La metáfora del periódico

Piense en un artículo de periódico bien escrito. En la parte superior espera un titular que indique de qué se trata la historia y le permita determinar si quiere leerlo o no. El primer párrafo ofrece una sinopsis de la historia, oculta los detalles y muestra conceptos generales. Al avanzar la lectura, aumentan los detalles junto con todas las fechas, nombres, citas y otros elementos.

Un archivo de código debe ser como un artículo de periódico. El nombre debe ser sencillo pero claro. Por sí mismo, debe bastar para indicarnos si estamos o no en el módulo correcto. Los elementos superiores del archivo deben proporcionar conceptos y algoritmos de nivel superior. Los detalles deben aumentar según avanzamos, hasta que en la parte final encontremos las funciones de nivel inferior del archivo.

Un periódico se compone de varios artículos, algunos muy reducidos y otros de gran tamaño. No hay muchos que ocupen toda la página con texto, para que el periódico sea manejable. Si el periódico fuera un único y extenso texto con una aglomeración desorganizada de hechos, fechas y nombres, no lo leeríamos.

Apertura vertical entre conceptos

La práctica totalidad del código se lee de izquierda a derecha y de arriba a abajo. Cada línea representa una expresión o una cláusula, y cada grupo de líneas representa un pensamiento completo. Estos pensamientos deben separarse mediante líneas en blanco.

Fíjese en el Listado 5-1. Hay líneas en blanco que separan la declaración

del paquete, las importaciones y las funciones. Es una regla muy sencilla con un profundo efecto en el diseño visual del código. Cada línea en blanco es una pista visual que identifica un nuevo concepto independiente. Al avanzar por el listado, la vista se fija en la primera línea que aparece tras una línea en blanco.

Listado 5-1
BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
   public static final String REGEXP = “’’’.+?’’’”;
   private static final Pattern pattern = Pattern.compile (“’’’(.+?)’’’”,
     Pattern.MULTILINE + Pattern.DOTALL
   );

   public BoldWidget(ParentWidget parent, String text) throws Exception {
     super(parent);
     Matcher match = pattern.matcher(text);
     match.find();
     addChildWidgets(match.group(1));
   }

   public String render() throws Exception {
     StringBuffer html = new StringBuffer(“<b>”);
     html.append(childHtml()).append (“</b>”);
     return html.toString();
   }
}
```

Si eliminamos las líneas en blanco, como en el Listado 5-2, se oscurece la legibilidad del código.

Listado 5-2
BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
   public static final String REGEXP = “’’’.+?’’’”;
   private static final Pattern pattern = Pattern.compile(“’’’(.+?)’’’”,
     Pattern.MULTILINE + Pattern.DOTALL);
   public BoldWidget(ParentWidget parent, String text) throws Exception {
```

```
      super(parent);
      Matcher match = pattern.matcher(text);
      match.find();
      addChildWidgets(match.group(1));}
    public String render() throws Exception {
      StringBuffer html = new StringBuffer("<b>");
      html.append(childHtml()).append("</b>");
      return html.toString();
    }
  }
```

Este efecto aumenta todavía más si no centramos la vista. En el primer ejemplo, los distintos grupos de líneas saltan a la vista, mientras que en el segundo es una mezcla amorfa. La diferencia entre ambos listados es una ligera apertura vertical.

Densidad vertical

Si la apertura separa los conceptos, la densidad vertical implica asociaciones. Por tanto, las líneas de código con una relación directa deben aparecer verticalmente densas. Fíjese en cómo los comentarios sin sentido del Listado 5-3 anulan la asociación entre las dos variables de instancia.

Listado 5-3

```
public class ReporterConfig {

   /**
   * Nombre de clase del escuchador
   */
   private String m_className;

   /**
   * Propiedades del escuchador
   */
   private List<Property> m_properties = new ArrayList<Property>();

   public void addProperty(Property property) {
     m_properties.add(property);
   }
```

El Listado 5-4 es mucho más fácil de leer. Lo apreciamos a simple vista o al menos yo lo hago. Al mirarlo, veo que es una clase con dos variables y un método, sin tener que mover la cabeza ni la vista. El listado anterior nos

obliga a forzar la vista y a mover la cabeza para alcanzar el mismo nivel de comprensión.

Listado 5-4

```
public class ReporterConfig {
   private String m_className;
   private List<Property> m_properties = new ArrayList<Property>();

   public void addProperty(Property property) {
     m_properties.add(property);
   }
```

Distancia vertical

¿Alguna vez ha tenido que recorrer una clase, saltando de una función a otra, desplazándose por el código para intentar adivinar la relación y el funcionamiento de las funciones, y acabar totalmente confundido? ¿Alguna vez ha escudriñado la cadena de herencia buscando la definición de una variable o función? Resulta frustrante porque intenta comprender lo que hace el sistema, pero pierde el tiempo y su energía mental en intentar localizar y recordar sus elementos.

Los conceptos relacionados entre sí deben mantenerse juntos verticalmente [G10]. Esta regla no funciona con conceptos de archivos independientes. Por lo tanto, no debe separar conceptos relacionados en archivos independientes a menos que tenga un motivo de peso. De hecho, es uno de los motivos por los que se debe evitar el uso de variables protegidas.

Para los conceptos relacionados que pertenecen al mismo archivo, su separación vertical debe medir su importancia con respecto a la legibilidad del otro. Debe evitar que el lector deambule entre archivos y clases.

Declaraciones de variables

Las variables deben declararse de la forma más aproximada a su uso. Como las funciones son muy breves, las variables locales deben aparecer en la parte superior de cada función, como en este ejemplo de Junit4.3.1.

```
private static void readPreferences() {
   InputStream is = null;
   try {
     is = new FileInputStream(getPreferencesFile());
     setPreferences(new Properties(getPreferences()));
     getPreferences().load(is);
   } catch (IOException e) {
     try {
       if (is != null)
         is.close();
     } catch (IOException e1) {
     }
   }
}
```

Las variables de control de bucles deben declararse en la instrucción del bucle, como en esta pequeña función del mismo código fuente:

```
public int countTestCases() {
   int count= 0;
   for (Test each : tests)
     count += each.countTestCases();
   return count;
}
```

En casos excepcionales, una variable puede declararse en la parte superior de un bloque o antes de un bucle en una función extensa. Puede ver este tipo de variable en la siguiente función de TestNG.

```
…
for (XmlTest test: m_suite.getTests()) {
   TestRunner tr = m_runnerFactory.newTestRunner(this, test);
   tr.addListener(m_textReporter);
   m_testRunners.add(tr);

   invoker = tr.getInvoker();

   for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
     beforeSuiteMethods.put(m.getMethod(), m);
   }

   for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
     afterSuiteMethods.put(m.getMethod(), m);
   }
}
…
```

Variables de instancia

Las variables de instancia, por su parte, deben declararse en la parte superior de la clase. Esto no debe aumentar la distancia vertical de las variables, ya

que en una clase bien diseñada se usan en muchos sino en todos sus métodos.

Existen discrepancias sobre la ubicación de las variables de instancia. En C++ suele aplicarse la denominada regla de las tijeras, que sitúa todas las variables de instancia en la parte inferior. En Java, sin embargo, es habitual ubicarlas en la parte superior de la clase. No veo motivos para no hacerlo. Lo importante es declarar las variables de instancia en un punto conocido para que todo el mundo sepa dónde buscarlas.

Fíjese en el extraño caso de la clase `TestSuite` de JUnit 4.3.1. He atenuado considerablemente esta clase para ilustrar este concepto. Si se fija en la mitad del listado, verá dos variables de instancia declaradas. Resultaría complicado ocultarlas en un punto mejor. Cualquiera que lea este código tendría que toparse con las declaraciones por casualidad (como me pasó a mí).

```
public class TestSuite implements Test {
   static public Test createTest(Class<? extends TestCase> theClass,
   String name) {
     …
   }

   public static Constructor<? extends TestCase>
   getTestConstructor(Class<? extends TestCase> theClass)
   throws NoSuchMethodException {
     …
   }

   public static Test warning(final String message) {
     …
   }

   private static String exceptionToString(Throwable t) {
     …
   }

   private String fName;

   private Vector<Test> fTests = new Vector<Test>(10);

   public TestSuite() {
   }

   public TestSuite(final Class<? extends TestCase> theClass) {
     …
   }

   public TestSuite(Class<? extends TestCase> theClass, String name) {
     …
   }
```

```
    … … … …
  }
```

Funciones dependientes

Si una función invoca otra, deben estar verticalmente próximas, y la función de invocación debe estar por encima de la invocada siempre que sea posible. De este modo el programa fluye con normalidad. Si la convención se sigue de forma fiable, los lectores sabrán que las definiciones de función aparecen después de su uso. Fíjese en el fragmento de FitNesse del Listado 5-5.

La función superior invoca las situadas por debajo que, a su vez, invocan a las siguientes. Esto facilita la detección de las funciones invocadas y mejora considerablemente la legibilidad del módulo completo.

Listado 5-5
WikiPageResponder.java.

```
  public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;

    public Response makeResponse(FitNesseContext context, Request request)
      throws Exception {
      String pageName = getPageNameOrDefault(request, “Frontpage”);
      LoadPage(pageName, context);
      if (page == null)
        return notFoundResponse(context, request);
      else
        return makePageResponse(context);
    }

    private     String     getPageNameOrDefault(Request     request,     String
defaultPageName)
    {
      String pageName = request.getResource();
      if (StringUtil.isBlank(pageName))
        pageName = defaultPageName;

      return pageName;
    }

    protected void loadPage(String resource, FitNesseContext context)
      throws Exception {
      WikiPagePath path = PathParser.parse(resource);
```

```
      crawler = context.root.getPageCrawler();
      crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
      page = crawler.getPage(context.root, path);
      if (page != null)
        pageData = page.getData();
    }

    private   Response   notFoundResponse(FitNesseContext   context,   Request
request)
      throws Exception {
      return new NotFoundResponder().makeResponse(context, request);
    }

    private SimpleResponse makePageResponse(FitNesseContext context)
      throws Exception {
      pageTitle = PathParser.render(crawler.getFullPath(page));
      String html = makeHtml(context);

      SimpleResponse response = new SimpleResponse();
      response.setMaxAge(0);
      response.setContent(html);
      return response;
    }
  …
```

Además, este fragmento es un buen ejemplo de ubicación de constantes en un nivel correcto [G35]. La constante `FrontPage` se podría haber ocultado en la función `getPageNameOrDefault`, pero eso habría ocultado una constante conocida y esperada en una función de nivel inferior de forma incorrecta. Es mejor pasar la constante desde un punto en el que tiene sentido a la posición en la que realmente se usa.

Afinidad conceptual

Determinados conceptos de código *deben* estar próximos a otros. Tienen una afinidad conceptual concreta. Cuanto mayor sea esta afinidad, menor distancia vertical debe existir entre ellos.

Como hemos visto, esta afinidad se puede basar en una dependencia directa, como cuando una función invoca a otra, o cuando usa una variable. Pero hay otras causas de afinidad. Puede generarse porque un grupo de funciones realice una operación similar. Fíjese en este fragmento de código de Junit 4.3.1:

```
  public class Assert {
    static public void assertTrue(String message, boolean condition) {
```

```
      if (!condition)
        fail(message);
    }

    static public void assertTrue(boolean
condition) {
      assertTrue (null, condition);
    }

    static public void assertFalse(String
message, boolean condition) {
      assertTrue(message, !condition);
    }

    static public void assertFalse(boolean
condition) {
      assertFalse(null, condition);
    }
  …
```

Estas funciones tienen una elevada afinidad conceptual ya que comparten un sistema de nombres común y realizan variantes de la misma tarea básica. El hecho de que se invoquen unas a otras es secundario. Aunque no lo hicieran, deberían seguir estando próximas entre ellas.

Orden vertical

Por lo general, las dependencias de invocaciones de funciones deben apuntar hacia abajo. Es decir, la función invocada debe situarse por debajo de la que realice la invocación[27]. Esto genera un agradable flujo en el código fuente, de los niveles superiores a los inferiores.

Como sucede en los artículos del periódico, esperamos que los conceptos más importantes aparezcan antes y que se expresen con la menor cantidad de detalles sobrantes. Esperamos que los detalles de nivel inferior sean los últimos. De este modo, podemos ojear los archivos de código y captar el mensaje en las primeras funciones sin necesidad de sumergirnos en los detalles. El Listado 5-5 se organiza de esta forma. Puede que otros ejemplos mejores sean los listados 15-5 y 3-7.

Formato horizontal

¿Qué ancho debe tener una línea? Para responderlo, fíjese en la anchura de las líneas de un programa convencional. De nuevo, examinamos siete proyectos diferentes. En la figura 5.2 puede ver la distribución de longitud de todos ellos. La regularidad es impresionante, en especial en tomo a los 45 caracteres. De hecho, los tamaños entre 20 y 60 representan un uno por cien del número total de líneas. ¡Eso es un 40 por 100! Puede que otro 30 por 100 sea menos de 10 caracteres de ancho. Recuerde que es una escala de registro, de modo que la apariencia lineal es muy significativa. Es evidente que los programadores prefieren líneas menos anchas.

Figura 5.2. Distribución de anchura de líneas en Java.

Esto sugiere que debemos intentar reducir las líneas de código. El antiguo límite Hollerith de 80 es un tanto arbitrario y no me opongo a líneas que tienen 100 o incluso 120, pero no más.

Como norma, no debe tener que desplazarse hacia la derecha. Los monitores modernos son más anchos y los programadores noveles pueden reducir la fuente para encajar hasta 200 caracteres en la pantalla. No lo haga. Mi límite personal es de 120.

Apertura y densidad horizontal

Usamos el espacio en blanco horizontal para asociar elementos directamente relacionados y separar otros con una relación menos estrecha. Fíjese en la siguiente función:

```
private void measureLine(String line) {
   lineCount++;
   int lineSize = line.length();
   totalChars += lineSize;
   lineWidthHistogram.addLine(lineSize, lineCount);
   recordWidestLine(lineSize);
}
```

Hemos rodeado los operadores de asignación con espacios en blanco para destacarlos. Las instrucciones de asignación tienen dos elementos principales: el lado izquierdo y el derecho. Los espacios acentúan esta separación.

Por otra parte, no hemos incluido espacios entre los nombres de las funciones y el paréntesis de apertura, ya que la función y sus argumentos están estrechamente relacionados. Su separación los desconectaría. Separo los argumentos en los paréntesis de invocación de la función para acentuar la coma e indicar que los argumentos son independientes. El espacio en blanco también se usa para acentuar la precedencia de los operadores:

```
public class Quadratic {
   public static double root1(double a, double b, double c) {
     double determinant = determinant(a, b, c);
     return (-b + Math.sqrt(determinant)) / (2*a);
   }

   public static double root2(int a, int b, int c) {
     double determinant = determinant(a, b, c);
     return (-b - Math.sqrt(determinant)) / (2*a);
   }

   private static double determinant(double a, double b, double c) {
     return b*b - 4*a*c;
   }
}
```

Fíjese en lo bien que se leen las ecuaciones. Los factores carecen de espacios en blanco ya que tienen una mayor precedencia. Los términos se separan mediante espacios en blanco ya que la suma y la resta son de precedencia inferior.

Desafortunadamente, muchas herramientas de formato de código ignoran la precedencia de los operadores e imponen un espaciado uniforme. Por ello, separaciones sutiles como las anteriores suelen perderse tras modificar el

formato del código.

Alineación horizontal

Cuando era programador de lenguajes de ensamblado[28], usaba la alineación horizontal para acentuar determinadas estructuras. Cuando comencé a programar en C, C++ y Java, seguía intentando alinear los nombres de variables en un conjunto de declaraciones o todos los valores en un grupo de instrucciones de asignación. El aspecto de mi código era el siguiente:

```
public class FitNesseExpediter implements ResponseSender
{
     private     Socket            socket;
     private     InputStream       input;
     private     OutputStream      output;
     private     Request           request;
     private     Response          response;
     private     FitNesseContext   context;
     protected   long              requestParsingTimeLimit;
     private     long              requestProgress;
     private     long              requestParsingDeadline;
     private     boolean           hasError;
     public FitNesseExpediter(     Socket s,
                                   FitNesseContext context) throws Exception
     {
       this.context =              context;
       socket =                    s;
       input =                     s.getInputStream();
       output =                    s.getOutputStream();
       requestParsingTimeLimit =   10000;
     }
```

Sin embargo, este tipo de alineación no es útil. Parece enfatizar los elementos incorrectos y aleja la vista de la verdadera intención. Por ejemplo, en la lista anterior de declaraciones, nos vemos tentados a leer la lista de nombres de variables sin fijarnos en sus tipos. Del mismo modo, en la lista de instrucciones de asignación, nos fijamos en los valores sin ver el operador. Para empeorarlo todo, las herramientas automáticas de formato suelen eliminar este tipo de alineación. Por tanto, al final, ya no lo uso. Ahora prefiero declaraciones y asignaciones sin alinear, como se muestra a

continuación, ya que resaltan una deficiencia importante. Si tengo listas extensas que deben alinearse, el problema es la longitud de las listas, no la falta de alineación. La longitud de la siguiente lista de declaraciones de `FitNesseExpediter` sugiere que esta clase debe dividirse.

```
  public class FitNesseExpediter implements ResponseSender
  {
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long request Progress;
    private long requestParsingDeadline;
    private boolean hasError;

    public   FitNesseExpediter(Socket   s,   FitNesseContext   context)   throws
Exception
    {
      this.context = context;
      socket = s;
      input = s.getInputStream();
      output = s.getOutputStream();
      requestParsingTimeLimit = 10000;
    }
```

Sangrado

Un archivo de código es una jerarquía más que un contorno. Incluye información que pertenece a la totalidad del archivo, a sus clases individuales, a los métodos de las clases, a los bloques de los métodos y a los bloques de los bloques. Cada nivel de esta jerarquía es un ámbito en el que se pueden declarar nombres y en el que se interpretan declaraciones e instrucciones ejecutables.

Para que esta jerarquía de ámbitos sea visible, sangramos las líneas de código fuente de acuerdo a su posición en la jerarquía. Las instrucciones al nivel del archivo, como las declaraciones de clases, no se sangran. Los métodos de una clase se sangran un nivel a la derecha de la clase. Las implementaciones de dichos métodos se implementan un nivel a la derecha de la declaración de los métodos. Las implementaciones de bloques se

implementan un nivel a la derecha de su bloque contenedor y así sucesivamente.

Los programadores dependen de este sistema de sangrado. Alinean visualmente las líneas a la izquierda para ver el ámbito al que pertenece. De este modo pueden acceder rápidamente a los ámbitos, como por ejemplo a implementaciones de instrucciones `if` o `while`, que no son relevantes para la situación actual. Buscan en la izquierda nuevas declaraciones de métodos, variables e incluso clases. Sin el sangrado, los programas serian prácticamente ilegibles.

Fíjese en los siguientes programas, sintáctica y semánticamente idénticos:

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer {
   private FitNesseContext context;

   public FitNesseServer(FitNesseContext context) {
     this.context = context;
   }

   public void serve(Socket s) {
     serve (s, 10000);
   }

   public void serve(Socket s, long requestTimeout) {
     try {
       FitNesseExpediter sender = new FitNesseExpediter(s, context);
       sender.setRequestParsingTimeLimit(requestTimeout);
       sender.start();
     }
     catch (Exception e) {
       e.printStackTrace();
     }
   }
}
```

A la vista puede incluso apreciar la estructura del archivo sangrado. Detectamos inmediatamente las variables, constructores y métodos de acceso. En cuestión de segundos vemos que es una especie de interfaz de conexión,

con un tiempo de espera. La versión sin sangrar, por su parte, es prácticamente impenetrable.

Romper el sangrado

En ocasiones tenemos la tentación de romper la regla de sangrado con instrucciones `if` breves, bucles `while` breves o funciones breves. Siempre que he sucumbido a esta tentación, he acabado por volver a aplicar el sangrado. Por ello, evito replegar ámbitos a una línea, como en este ejemplo:

```
public class CommentWidget extends TextWidget
{
   public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

   public  CommentWidget(ParentWidget  parent,  String  text){super  (parent,
text);}
   public String render() throws Exception { return ""; }
}
```

Prefiero desplegar y sangrar los ámbitos:

```
public class CommentWidget extends TextWidget {
   public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

   public CommentWidget(ParentWidget parent, String text) {
     super(parent, text);
   }

   public String render() throws Exception {
     return "";
   }
}
```

Ámbitos ficticios

En ocasiones, el cuerpo de una instrucción `while` o `for` es ficticio, como se muestra a continuación. No me gustan estas estructuras y prefiero evitarlas. En caso de no poder hacerlo, me aseguro de sangrar el cuerpo ficticio y de incluirlo entre paréntesis. No sabría decir cuántas veces me ha engañado un punto y coma situado al final de un bucle `while` en la misma línea. A menos que lo haga visible y lo sangre en una línea propia, es difícil de ver.

```
while (dis.read(buf, 0, readBufferSize) != -1)
   ;
```

Reglas de equipo

Todo programador tiene sus reglas de formato preferidas, pero si forma parte de un equipo, el equipo manda.

Un equipo de programadores debe acordar un único estilo de formato y todos los integrantes del equipo deben aplicarlo. El objetivo es que el *software* tenga un estilo coherente. No queremos que parezca escrito por individuos enfrentados.

Cuando comencé el proyecto FitNesse en 2002, me reuní con el equipo para definir un estilo de código. Tardamos 10 minutos. Decidimos dónde añadir las llaves, qué tamaño de sangrado utilizar, los nombres de clases, variables y métodos, y demás. Tras ello, codificamos las reglas en el IDE y las cumplimos desde entonces. No son las reglas que prefiero, son las que el equipo decidió. Y como miembro de ese equipo, las apliqué cuando creamos el código del proyecto FitNesse.

Recuerde que un buen sistema de *software* se compone de una serie de documentos que se leen fácilmente. Deben tener un estilo coherente y dinámico. El lector debe confiar en que los formatos que ve en nuestro archivo de código significarán lo mismo para otros. Lo último que queremos es aumentar la complejidad del código creando una mezcla de estilos diferentes.

Reglas de formato de Uncle Bob

Las reglas que uso personalmente son sencillas y se ilustran en el código del Listado 5-6. Considérelo un ejemplo de documento estándar de código óptimo.

Listado 5-6

CodeAnalyzer.java.

```
  public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram lineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
      lineWidthHistogram = new LineWidthHistogram();
    }

    public static List<File> findJavaFiles(File parentDirectory) {
      List<File> files = new ArrayList<File>();
      findJavaFiles(parentDirectory, files);
      return files;
    }

    private static void findJavaFiles(File parentDirectory, List<File> files)
{
      for (File file : parentDirectory.listFiles()) {
        if (file.getName().endsWith(".java"))
          files.add(file);
        else if (file.isDirectory())
          findJavaFiles(file, files);
      }
    }

    public void analyzeFile(File javaFile) throws Exception {
      BufferedReader br = new BufferedReader(new FileReader(javaFile));
      String line;
      while ((line = br.readLine()) != null)
        measureLine(line);
    }

    private void measureLine(String line) {
      lineCount++;
      int lineSize = line.length();
      totalChars += lineSize;
      lineWidthHistogram.addLine(lineSize, lineCount);
      recordWidestLine(lineSize);
    }

    private void recordWidestLine(int lineSize) {
      if (lineSize > maxLineWidth) {
        maxLineWidth = lineSize;
        widestLineNumber = lineCount;
      }
    }

    public int getLineCount() {
      return lineCount;
    }
```

```
    public int getMaxLineWidth() {
      return maxLineWidth;
    }

    public int getWidestLineNumber() {
      return widestLineNumber;
    }

    public LineWidthHistogram getLineWidthHistogram() {
      return lineWidthHistogram;
    }

    public double getMeanLineWidth() {
      return (double)totalChars/lineCount;
    }

    public int getMedianLineWidth() {
      Integer[] sortedwidths = getSortedWidths();
      int cumulativeLineCount = 0;
      for (int width : sortedwidths) {
        cumulativeLineCount += lineCountForWidth(width);
        if (cumulativeLineCount > lineCount/2)
          return width;
      }
      throw new Error (“Cannot get here”);
    }

    private int lineCountForWidth(int width) {
      return lineWidthHistogram.getLinesforWidth(width).size();
    }

    private Integer[] getSortedWidths() {
      Set<Integer> widths = lineWidthHistogram.getWidths();
      Integer[] sortedwidths = (widths.toArray(new Integer[0]));
      Arrays.sort(sortedwidths);
      return sortedwidths;
    }
  }
```

6
Objetos y estructuras de datos

Hay una razón para que las variables sean privadas. No queremos que nadie más dependa de ellas. Queremos poder cambiar su tipo o implementación cuando deseemos. Entonces, ¿por qué tantos programadores añaden automáticamente métodos de establecimiento y recuperación que muestran sus variables privadas como si fueran públicas?

Abstracción de datos

Fíjese en la diferencia entre los listados 6-1 y 6-2. Ambos representan los datos de un punto cartesiano, pero uno muestra su implementación y otro la oculta totalmente.

Listado 6-1
Punto concreto.

```
public class Point {
   public double x;
   public double y;
}
```

Listado 6-2
Punto abstracto.

```
public interface Point {
   double getX();
   double getY();
   void setCartesian(double x, double y);
   double getR();
   double getTheta();
   void set Polar(double r, double theta);
}
```

Lo mejor del Listado 6-2 es que no hay forma de saber si la implementación está en coordenadas rectangulares o polares. ¡Puede que en ninguna! Y aun así la interfaz representa sin lugar a dudas una estructura de datos.

Pero representa algo más que una estructura de datos. Los métodos refuerzan una política de acceso. Puede leer las coordenadas de forma

independiente, pero debe establecerlas de forma conjunta como operación atómica.

El Listado 6-1, por su parte, se implementa claramente en coordenadas rectangulares y nos obliga a manipularlas de forma independiente, lo que muestra la implementación. De hecho, la mostraría igualmente, aunque las variables fueran privadas y usáramos métodos variables de establecimiento y recuperación. Para ocultar la implementación no basta con añadir una capa de funciones entre las variables. Se basa en la abstracción. Una clase no fuerza sus variables a través de métodos de establecimiento y recuperación. Por el contrario, muestra interfaces abstractas que permiten a sus usuarios manipular la esencia de los datos sin necesidad de conocer su implementación.

Fíjese en los listados 6-3 y 6-4. El primero usa términos concretos para indicar el nivel de combustible de un vehículo mientras que el segundo lo hace con la abstracción del porcentaje. En el caso concreto, podemos estar seguros de que se trata de métodos de acceso de variables. En el caso abstracto, desconocemos la forma de los datos.

Listado 6-3
Vehículo concreto.

```
public interface Vehicle {
   double getFuelTankCapacityInGallons();
   double getGallonsOfGasoline();
}
```

Listado 6-4
Vehículo abstracto.

```
public interface Vehicle {
   double getPercentFuelRemaining();
}
```

En ambos casos, la segunda opción es preferible. No queremos mostrar los detalles de los datos, sino expresarlos en términos abstractos. Esto no se consigue simplemente mediante interfaces o métodos de establecimiento y recuperación. Hay que meditar seriamente la forma óptima de representar los datos que contiene un objeto. La peor opción es añadir métodos de

establecimiento y recuperación a ciegas.

Antisimetría de datos y objetos

Estos dos ejemplos ilustran la diferencia entre objetos y estructuras de datos. Los objetos ocultan sus datos tras abstracciones y muestran funciones que operan en dichos datos. La estructura de datos muestra sus datos y carece de funciones con significado. Vuelva a leerlos. Fíjese en la naturaleza complementaria de las dos definiciones. Son virtualmente opuestas. Puede parecer una diferencia menor, pero tiene importantes implicaciones.

Fíjese en el ejemplo del Listado 6-5. La clase `Geometry` opera en las tres clases de formas, que son sencillas estructuras de datos sin comportamiento. Todo el comportamiento se encuentra en la clase Geometry.

Listado 6-5
Forma mediante procedimientos.

```
public class Square {
   public Point topLeft;
   public double side;
}

public class Rectangle {
   public Point topLeft;
   public double height;
   public double width;
}

public class Circle {
   public Point center;
   public double radius;
}

public class Geometry {
   public final double PI = 3.141592653589793;

   public double area(Object shape) throws NoSuchShapeException
   {
     if (shape instanceof Square) {
       Square s = (Square)shape;
       return s.side * s.side;
     }
     else if (shape instanceof Rectangle) {
```

```
        Rectangle r = (Rectangle)shape;
        return r.height * r.width;
      }
      else if (shape instanceof Circle) {
        Circle c = (Circle)shape;
        return PI * c.radius * c.radius;
      }
      throw new NoSuchShapeException();
    }
  }
```

Los programadores orientados a objetos se quejarán de que es un ejemplo de procedimiento, y tienen razón. Imagine qué pasaría si añadimos la función `perimeter()` a `Geometry`. ¡Las clases de formas no se verían afectadas! ¡Y las demás clases que dependieran de las formas tampoco! Por otra parte, si añado una nueva forma, tendría que cambiar todas las funciones de `Geometry`. Vuélvalo a leer. Comprobará que las dos condiciones son diametralmente opuestas.

Fíjese ahora en la solución orientada a objetos del Listado 6-6. Aquí, el método `area()` es polimórfico. No se necesita una clase `Geometry`. Por tanto, si añado una nueva forma, ninguna de las funciones existentes se ven afectadas, pero si añado otra función, habrá que cambiar todas las formas[29].

Listado 6-6
Formas polimórficas.

```
public class Square implements Shape {
   private Point topLeft;
   private double side;

   public double area() {
   return side*side;
}

public class Rectangle implements Shape {
   private Point topLeft;
   private double height;
   private double width;

   public double area() {
     return height * width;
   }
}

public class Circle implements Shape {
   private Point center;
```

```
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
      return PI * radius * radius;
    }
  }
```

De nuevo, vemos la naturaleza complementaria de estas dos definiciones; totalmente contrarias. Esto ilustra la dicotomía fundamental entre objetos y estructuras de datos:

El código por procedimientos (el que usa estructuras de datos) facilita la inclusión de nuevas funciones sin modificar las estructuras de datos existentes. El código orientado a objetos, por su parte, facilita la inclusión de nuevas clases sin cambiar las funciones existentes.

El complemento también es cierto:

El código por procedimientos dificulta la inclusión de nuevas estructuras de datos ya que es necesario cambiar todas las funciones. El código orientado a objetos dificulta la inclusión de nuevas funciones ya que es necesario cambiar todas las clases.

Por tanto, lo que es difícil para la programación orientada a objetos es fácil para los procedimientos, y viceversa.

En cualquier sistema complejo habrá ocasiones en las que queramos añadir nuevos tipos de datos en lugar de nuevas funciones. En dichos casos, los objetos y la programación orientada a objetos es lo más adecuado. Por otra parte, en ocasiones tendremos que añadir nuevas funciones en lugar de tipos de datos, para lo que resulta más adecuado usar código por procedimientos y estructuras de datos.

Los programadores experimentados saben que la idea de que todo es un objeto es un mito. En ocasiones solamente *queremos* sencillas estructuras de datos con procedimientos que operen en las mismas.

La ley de Demeter

Existe una conocida heurística denominada *Ley de Demeter*[30] que afirma que un módulo no debe conocer los entresijos de los objetos que manipula. Como vimos en el apartado anterior, los objetos ocultan sus datos y muestran operaciones, lo que significa que un objeto no debe mostrar su estructura interna a través de métodos de acceso ya que, si lo hace, mostraría, no ocultaría, su estructura interna.

En concreto, la ley de Demeter afirma que un método de una clase `c` sólo debe invocar los métodos de:

- `c`.
- Un objeto creado por `f`.
- Un objeto pasado como argumento a `f`.
- Un objeto en una variable de instancia de `c`.

El método *no* debe invocar métodos de objetos devueltos por ninguna de las funciones permitidas. Es decir, no hable con desconocidos, sólo con amigos.

El siguiente código[31] parece incumplir la Ley de Demeter (entre otras cosas) ya que invoca la función `getScratchDir()` en el valor devuelto de `getOptions()` y después invoca `getAbsolutePath()` en el valor devuelto de `getScratchDir()`.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Choque de trenes

Ese tipo de código suele denominarse choque de trenes ya que se asemeja a un grupo de vagones de tren. Estas cadenas de invocaciones suelen considerarse un estilo descuidado y deben evitarse [G36]. Conviene dividirlas de esta forma:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
```

```
final String outputDir = scratchDir.getAbsolutePath();
```

¿Incumplen estos dos fragmentos de código la Ley de Demeter? Sin duda el módulo contenedor sabe que el objeto `ctxt` contiene opciones, que contienen un directorio `scratch`, que tiene una ruta absoluta. La función sabe demasiado. La función que realiza la invocación sabe cómo desplazarse por numerosos objetos diferentes.

Si incumple o no la Ley de Demeter depende de si `ctxt`, `Options` y `ScratchDir` son objetos o estructuras de datos. Si son objetos, debería ocultarse su estructura interna, no mostrarse, y conocer sus detalles internos sería un claro incumplimiento de la Ley de Demeter. Por otra parte, si `ctxt`, `Options` y `ScratchDir` son simples estructuras de datos, mostrarán su estructura interna con naturalidad y la Ley de Demeter no se aplica.

El uso de funciones de acceso complica el problema. Si el código se hubiera escrito de esta otra forma, probablemente no nos preocuparíamos de si se incumple la Ley de Demeter o no.

```
final String outputDir = ctxt.options.ScratchDir.absolutePath;
```

El problema sería menos confuso si las estructuras de datos tuvieran variables públicas y no funciones, y los objetos tuvieran variables privadas y funciones públicas. Sin embargo, existen estructuras y estándares (como los *bean*) que exigen que incluso una sencilla estructura de datos tenga elementos de acceso y mutación.

Híbridos

Esta confusión genera ocasionalmente desafortunadas estructuras híbridas mitad objeto y mitad estructura de datos. Tienen funciones que realizan tareas significativas y también variables públicas o método públicos de acceso y mutación que hacen que las variables privadas sean públicas, y tientan a otras funciones externas a usar dichas variables de la misma forma que un programa por procedimientos usaría una estructura de datos[32]. Estos

híbridos dificultan la inclusión de nuevas funciones y también de nuevas estructuras de datos. Son lo peor de ambos mundos. Evítelos. Indican un diseño descuidado cuyos autores dudan, o peor todavía, desconocen, si necesitan protegerse de funciones o tipos.

Ocultar la estructura

¿Qué pasaría si `ctxt`, `options` y `scratchDir` fueran objetos con un comportamiento real? Como los objetos deben ocultar su estructura interna, no podríamos desplazarnos por los mismos. Entonces, ¿cómo obtendríamos la ruta absoluta del directorio `scratch`?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

o

```
ctxt.getScratchDirectoryOption().getAbsolutePath()
```

La primera opción provocaría una explosión de métodos en el objeto `ctxt`. La segunda asume que `getScratchDirectoryOption()` devuelve una estructura de datos, no un objeto. Ninguna de las opciones parece correcta. Si `ctxt` es un objeto, deberíamos indicarle que hiciera algo, no preguntar sobre sus detalles internos. Entonces, ¿para qué queremos la ruta absoluta del directorio `scratch`? ¿Cómo vamos a usarla? Fíjese en este código del mismo módulo (muchas líneas después):

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

La mezcla de distintos niveles de detalle [G34][G6] es preocupante. Puntos, guiones, extensiones de archivo y objetos `File` no deben mezclarse de esta forma, junto al código contenedor. Si lo ignoramos, vemos que la intención de obtener la ruta absoluta del directorio `scratch` es crear un archivo de borrador de un nombre concreto.

¿Y si le dijéramos al objeto `ctxt` que hiciera esto?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Parece algo razonable para un objeto. Permite a `ctxt` ocultar sus detalles internos e impide que la función actual incumpla la Ley de Demeter y se

desplace por objetos que no debería conocer.

Objetos de transferencia de datos

La quintaesencia de una estructura de datos es una clase con variables públicas y sin funciones. En ocasiones se denomina Objeto de transferencia de datos (*Data Transfer Object* u OTD). Los OTD son estructuras muy útiles, en especial para comunicarse con bases de datos o analizar mensajes de conexiones, etc. Suelen ser los primeros de una serie de fases de traducción que convierten datos sin procesar en objetos en el código de la aplicación. Más común es la forma de bean mostrada en el Listado 6-7. Los bean tienen variables privadas manipuladas por métodos de establecimiento y recuperación. La cuasi-Encapsulación de bean hace que algunos puristas de la programación orientada a objetos se sientan mejor pero no ofrece ningún otro beneficio.

Listado 6-7
address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String Street, String streetExtra,
         String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }
```

```
    public String getCity() {
      return city;
    }

    public String getState() {
      return getState;
    }

    public String getZip() {
      return zip;
    }
  }
```

Registro activo

Los registros activos son una forma especial de OTD. Son estructuras de datos con variables públicas (o de acceso por bean) pero suelen tener métodos de navegación como `save` y `find`. Por lo general, estos registros activos son traducciones directas de tablas de base de datos u otros orígenes de datos. Desafortunadamente, muchos programadores intentan procesar estas estructuras de datos como si fueran objetos y les añaden métodos de reglas empresariales. Es algo extraño ya que crea un híbrido entre una estructura de datos y un objeto.

La solución, evidentemente, consiste en considerar al registro activo una estructura de datos y crear objetos independientes que contengan las reglas empresariales y que oculten sus datos internos (que probablemente sean instancias del propio registro activo).

Conclusión

Los objetos muestran comportamiento y ocultan datos. Esto facilita la inclusión de nuevos tipos de objetos sin necesidad de cambiar los comportamientos existentes. También dificulta la inclusión de nuevos comportamientos en objetos existentes. Las estructuras de datos muestran datos y carecen de comportamiento significativo. Esto facilita la inclusión de nuevos comportamientos en las estructuras de datos existentes, pero dificulta la inclusión de nuevas estructuras de datos en funciones existentes.

En un sistema, en ocasiones necesitaremos la flexibilidad de añadir nuevos tipos de datos, por lo que preferimos objetos para esa parte del sistema. En otros casos, querremos añadir nuevos comportamientos, para lo que preferimos tipos de datos y procedimientos en esa parte del sistema. Los buenos programadores de *software* entienden estos problemas sin prejuicios y eligen el enfoque más adecuado para cada tarea concreta.

Bibliografía

- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

7
Procesar errores

por Michael Feathers

Le parecerá extraño encontrar una sección de control de errores en un libro sobre código limpio. El control de errores es algo que todos tenemos que hacer al programar. Las entradas pueden ser incorrectas y los dispositivos pueden fallar, y cuando lo hacen, los programadores somos responsables de comprobar que el código hace lo que debe hacer.

No obstante, la conexión con el código limpio debe ser evidente. Muchas bases de código están totalmente dominadas por el control de errores. Cuando digo que están dominadas, no quiero decir que únicamente realicen control de

código, sino que es prácticamente imposible ver lo que el código hace debido a todo ese control de errores. El control de errores es importante, *pero si oscurece la lógica, es incorrecto*.

En este capítulo detallaremos diversas técnicas y consideraciones que puede usar para crear código limpio y robusto, código que procese los errores con elegancia y estilo.

Usar excepciones en lugar de códigos devueltos

En el pasado, muchos lenguajes carecían de excepciones. Las técnicas para procesar e informar de errores eran limitadas. Se definía un indicador de error o se devolvía un código de error que el invocador podía comprobar. El código del Listado 7-1 ilustra estos enfoques.

Listado 7-1
DeviceController.java.

```
public class DeviceController {
  …
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Comprobar el estado del dispositivo
    if (handle != DeviceHandle.INVALID) {
      // Guardar el estado del dispositivo en el campo de registro
      retrieveDeviceRecord(handle);
      // Si no está suspendido, cerrarlo
      if { record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended. Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  …
}
```

El problema de estos enfoques es que confunden al invocador. El invocador debe comprobar inmediatamente los errores después de la

invocación. Desafortunadamente, es algo que se suele olvidar. Por ello, es más recomendable generar una excepción al detectar un error. El código de invocación es más limpio. Su lógica no se oscurece por el control de errores.

El Listado 7-2 muestra el código tras generar una excepción en los métodos que pueden detectar errores.

Listado 7-2
DeviceController.java (con excepciones).

```
public class DeviceController {
  …

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    …
    throw new DeviceShutDownError("Invalid handle for: " - id.toString());
    …
  }
  …
}
```

Comprobará que es mucho más limpio. No es cuestión de estética. El código es mejor porque se solventan dos preocupaciones: el algoritmo para apagar el dispositivo y el control de errores ahora se encuentran separados. Puede ver cada uno de ellos y entenderlos de forma independiente.

Crear primero la instrucción try-catch-finally

Uno de los aspectos más interesantes de las excepciones es que *definen un ámbito* en el programa. Al ejecutar código en la parte `try` de una instrucción `try-catch-finally`, indicamos que la ejecución se puede cancelar en cualquier momento y después retomar en `catch`.

Los bloques `try` son como las transacciones, `catch` debe salir del programa en un estado coherente, independientemente de lo que suceda en `try`. Por este motivo, es aconsejable iniciar con una instrucción `try-catch-finally` el código que genere excepciones. De este modo define lo que debe esperar el usuario del código, independientemente de que se produzca un error en el código ejecutado en la cláusula `try`.

Veamos un ejemplo. Imagine que tiene que crear un código que acceda a un archivo y lea objetos serializados.

Comenzamos con una prueba de unidad que muestra que obtendremos una excepción cuando el archivo no exista:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
   sectionStore.retrieveSection("invalid - file");
}
```

La prueba nos lleva a crear lo siguiente:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
   // se devuelve un resultado ficticio hasta tener una implementación real
   return new ArrayList<RecordedGrip>();
}
```

Nuestra prueba falla ya que no genera una excepción. Tras ello, cambiamos la implementación para que intente acceder a un archivo no válido. Esta operación genera una excepción:

```
public List<RecordedGrip> retrieveSection (String sectionName) {
   try {
     FileInputstream stream = new FileInputStream(sectionName)
   } catch (Exception e) {
     throw new StorageException("retrieval error", e);
   }
   return new ArrayList<RecordedGrip>();
}
```

Ahora la prueba es correcta ya que capturamos la excepción y ya podemos refactorizar. Podemos reducir el tipo de la excepción capturada para que coincida con el tipo generado desde el constructor `FileInputStream`:

`FileNotFoundException`:

```
  public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
      FileInputStream stream = new FileInputStream(sectionName);
      stream.close();
    } catch (FileNotFoundException e) {
      throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
  }
```

Ahora que hemos definido el ámbito con una estructura `try-catch`, podemos usar TDD para diseñar el resto de la lógica necesaria. Dicha lógica se añade entre la creación de `FileInputStream` y el cierre, y podemos pretender que no pasa nada incorrecto.

Intente crear pruebas que fuercen las excepciones, para después añadir al controlador un comportamiento que satisfaga dichas pruebas. De este modo primero creará el ámbito de transacción del bloque `try` y podrá mantener la naturaleza de transacción del ámbito.

Usar excepciones sin comprobar

El debate ha terminado. Durante años, los programadores de Java han debatido las ventajas y los problemas de las excepciones comprobadas. Cuando aparecieron en la primera versión de Java, parecían una gran idea. La firma de todos los métodos enumeraría todas las excepciones que se podían pasar a su invocador. Es más, estas excepciones formaban parte del tipo del método. El código no se compilaría si la firma no coincidía con lo que el código iba a hacer.

En aquel momento, pensábamos que las excepciones comprobadas eran una gran idea y sí, ofrecían ciertas ventajas. Sin embargo, ahora es evidente que no se necesitan para crear *software* robusto. C# carece de excepciones comprobadas y, a pesar de los intentos, C++ tampoco, como sucede en Python o Ruby. Y en todos estos lenguajes se puede crear *software* robusto. Por ello, debemos decidir si las excepciones comprobadas valen su precio.

¿Qué precio? El precio de las excepciones comprobadas es un

incumplimiento del principio abierto/cerrado[33]. Si genera una excepción comprobada desde un método de su código y la cláusula `catch` se encuentra tres niveles por debajo, debe declarar dicha excepción en la firma de todos los métodos comprendidos entre su posición y `catch`. Esto significa que un cambio en un nivel inferior del *software* puede forzar cambios de firma en muchos niveles superiores. Será necesario volver a generar e implementar los módulos cambiados, aunque no cambien los elementos a los que hacen referencia.

Piense en la jerarquía de invocación de un sistema. Las funciones de la parte superior invocan a las funciones situadas debajo, que invocan a otras funciones inferiores y así sucesivamente. Imagine que una de las funciones de nivel inferior se modifica de forma que debe generar una excepción. Si la excepción se comprueba, la firma de la función tendrá que añadir una cláusula `throws`. Pero esto significa que todas las funciones que invoquen nuestra función modificada también tendrán que cambiarse para capturar la nueva excepción o para añadir la correspondiente cláusula `throws` en su firma. Y así indefinidamente. El resultado final es una cascada de cambios que pasan desde los niveles inferiores del *software* hasta los superiores. La encapsulación se rompe ya que todas las funciones en la ruta de `throw` deben conocer detalles de la excepción de nivel inferior. Como el cometido de las excepciones es permitimos procesar errores a distancia, es una lástima que las excepciones comprobadas rompan la encapsulación de esta forma.

Las excepciones comprobadas pueden ser útiles si tiene que crear una biblioteca crítica: tendrá que capturarlas. Pero en el desarrollo de aplicaciones generales, los costes de dependencia superan las ventajas.

Ofrecer contexto junto a las excepciones

Las excepciones que genere deben proporcionar el contexto adecuado para determinar el origen y la ubicación de un error. En Java, puede obtener un rastreo de pila de cualquier excepción; sin embargo, no le indicará el cometido de la función fallida.

Redacte mensajes de error informativos y páselos junto a sus

excepciones. Mencione la operación fallida y el tipo de fallo. Si guarda registros en su aplicación, incluya información suficiente para poder registrar el error en la cláusula `catch`.

Definir clases de excepción de acuerdo a las necesidades del invocador

Existen varias formas de clasificar los errores. Podemos hacerlo por origen (¿provienen de uno u otro componente?) o por tipo (¿son fallos del dispositivo, de la red o errores de programación?). Sin embargo, al definir clases de excepción en una aplicación, debemos preocuparnos principalmente en *cómo se capturan*.

Veamos un pobre ejemplo de clasificación de excepciones. Es una instrucción `try-catch-finally` de la invocación de una biblioteca de terceros. Abarca todas las excepciones que las invocaciones pueden generar:

```
ACMEPort port = new ACMEPort(12);

try {
   port.open();
} catch (DeviceResponseException e) {
   reportPortError(e);
   logger.log{“Device response exception”, e);
} catch (ATM1212UnlockedException e) {
   reportPortError(e);
   logger.log(“Unlock exception”, e);
} catch (GMXError e) {
   reportPortError(e);
   logger.log(“Device response exception”);
} finally {
   …
}
```

Esta instrucción contiene elementos duplicados, algo que no debería sorprendernos. En muchos casos de control de excepciones, el trabajo que realizamos es relativamente estándar independientemente de la causa real. Debemos registrar un error y asegurarnos de poder continuar.

En este caso, como sabemos que el trabajo es el mismo independientemente de la excepción, podemos simplificar el código si incluimos la API invocada y nos aseguramos de que devuelve un tipo de

excepción común:

```
LocalPort port = new LocalPort(12);
try {
   port.open();
} catch (PortDeviceFailure e) {
   reportError(e);
   logger.log(e.getMessage(), e);
} finally {
   …
}
```

Nuestra clase `LocalPort` es un simple envoltorio que captura y traduce excepciones generadas por la clase `ACMEPort`:

```
public class LocalPort {
   private ACMEPort innerPort;

   public LocalPort(int portNumber) {
     innerPort = new ACMEPort(portNumber);
   }

   public void open() {
     try {
       innerPort.open();
     } catch (DeviceResponseException e) {
       throw new PortDeviceFailure(e);
     } catch (ATM1212UnlockedException e) {
       throw new PortDeviceFailure(e);
     } catch (GMXError e) {
       throw new PortDeviceFailure(e);
     }
   }
   …
}
```

Los envoltorios como el definido para `ACMEPort` pueden ser muy útiles. De hecho, es recomendable envolver API de terceros. Al hacerlo, se minimizan las dependencias: puede cambiar a otra biblioteca diferente sin apenas problemas y el envoltorio también facilita imitar invocaciones de terceros cuando se prueba el código. Una última ventaja es que no estamos limitados a las decisiones de diseño de API de un determinado fabricante. Puede definir una API que le resulte cómoda. En el ejemplo anterior, definimos un único tipo de excepción para el fallo de puertos y podemos escribir un código mucho más limpio. A menudo, una única clase de excepción es suficiente para una zona concreta del código. La información enviada con la excepción puede distinguir los errores. Use clases diferentes

sólo para capturar una excepción y permitir el paso de otra distinta.

Definir el flujo normal

Si sigue los consejos de apartados anteriores, realizará una importante separación entre la lógica empresarial y el control de errores. La mayoría de su código parecerá un algoritmo limpio y sin adornos. Sin embargo, el proceso desplaza la detección de errores hacia los bordes del programa. Debe envolver API externas para poder generar sus propias excepciones y definir un controlador por encima del código para poder procesar cálculos cancelados. En muchos casos es el enfoque más acertado, pero en ocasiones conviene no cancelar.

Veamos un ejemplo, un código extraño que suma gastos en una aplicación de facturación:

```
try {
   MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
   m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
   m_total += getMealPerDiem();
}
```

En esta empresa, si las comidas son gastos, pasan a formar parte del total. Si no lo son, los trabajadores reciben una cantidad diaria para la comida. La excepción entorpece la lógica. Sería más adecuado no tener que procesar el caso especial y el código sería mucho más sencillo:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

¿De verdad que el código puede ser tan simple? Pues sí. Podemos cambiar `ExpenseReportDAO` para que siempre devuelva un objeto `MealExpense`. Si no hay gastos de comida, devuelve un objeto `MealExpense` que devuelve la dieta diaria como total:

```
public class PerDiemMealExpenses implements MealExpenses {
   public int getTotal() {
     // devolver la dieta diaria predeterminada
```

```
    }
  }
```

Es lo que se denomina *Patrón de Caso Especial* [Fowler]. Se crea una clase o se configura un objeto que procese un caso especial. Al hacerlo, el código cliente no tiene que procesar comportamientos excepcionales. Dichos comportamientos se encapsulan en un objeto de caso especial.

No devolver Null

Creo que toda descripción del control de errores debe mencionar los elementos proclives a errores. El primero es devolver null. He perdido la cuenta de la cantidad de aplicaciones en que las que línea sí y línea también se comprueba null:

```
  public void registerItem(Item item) {
    if (item != null) {
      ItemRegistry registry = peristentStore.getItemRegistry();
      if (registry != null) {
        Item existing = registry.getItem(item.getID());
        if (existing.getBillingPeriod().hasRetailOwner()) {
          existing.register(item);
        }
      }
    }
  }
```

Si trabaja en una base de código como ésta, puede que no le parezca tan mala, pero lo es. Al devolver `null`, básicamente nos creamos trabajo y generamos problemas para los invocadores. Basta con que falte una comprobación de `null` para que la aplicación pierda el control.

¿Se ha fijado en que no hay una comprobación de `null` en la segunda línea de la instrucción `if` anidada? ¿Qué sucedería en tiempo de ejecución si `persistentStore` fuera `null`? Se generaría `NullPointerException` en tiempo de ejecución y se capturaría `NullPointerException` en el nivel superior o no. En ambos casos es incorrecto. ¿Qué debería hace como respuesta a la generación de `NullPointerException` desde el interior de su aplicación? Se puede afirmar que el problema de este código es la ausencia de una comprobación de `null` pero en realidad el problema es su exceso. Si

siente la tentación de devolver `null` desde un método, pruebe a generar una excepción o a devolver un objeto de caso especial. Si invoca un método que devuelva `null` desde una API de terceros, envuélvalo en un método que genere una excepción o devuelva un objeto de caso especial. En muchos casos, los objetos de caso especial son un remedio sencillo. Imagine que tiene el siguiente código:

```
List<Employee> employees = getEmployees();
if (employees != null) {
   for(Employee e : employees) {
     totalPay += e.getPay();
   }
}
```

Ahora, `getEmployees` puede devolver `null`, ¿pero es necesario? Si cambiamos `getEmployee` para que devuelva una lista vacía, podremos limpiar el código:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
   totalPay += e.getPay();
}
```

Afortunadamente, Java dispone de `Collections.emptyList()` y devuelve una lista inmutable predefinida que podemos usar para este cometido:

```
public List<Employee> getEmployees() {
   if (… there are no employees …)
     return Collections.emptyList();
}
```

Si usa este tipo de código, minimizará la presencia de `NullPointerException` y su código será más limpio.

No pasar Null

Devolver `null` desde métodos es incorrecto, pero es peor pasar `null` a métodos. A menos que trabaje con una API que espere que pase `null`, debe evitarlo siempre que sea posible. Veamos otro ejemplo, un sencillo método que calcula una métrica para dos puntos:

```
public class MetricsCalculator
```

```
{
   public double xProjection(Point p1, Point p2) {
     return (p2.x – p1.x) * 1.5;
   }
   …
}
```

¿Qué sucede cuando alguien pasa `null` como argumento?

```
calculator.xProjection(null, new Point (12, 13));
```

Se genera `NullPointerException`, evidentemente.

¿Cómo solucionarlo? Podríamos crear un nuevo tipo de excepción y generarla:

```
public class MetricsCalculator
{
   public double xProjection(Point p1, Point p2) {
     if (p1==null || p2==null) {
       throw InvalidArgumentException(
         “Invalid argument for MetricsCalculator.xProjection”);
     }
     return (p2.x – p1.x) * 1.5; }
   }
}
```

¿Mejor? Puede que sea mejor que una excepción de puntero nulo, pero recuerde que debe definir un controlador para `InvalidArgumentException`. ¿Qué debe hacer el controlador? ¿Hay alguna forma correcta de hacerlo?

Existe otra alternativa, usar un grupo de afirmaciones:

```
public class MetricsCalculator (
   public double xProjection{Point p1, Point p2) {
     assert p1 != null : “p1 should not be null”;
     assert p2 != null : “p2 should not be null”;
     return (p2.x - pl.x) * 1.5;
   }
}
```

Es documentación correcta pero no soluciona el problema. Si alguien pasa `null`, seguirá produciéndose un error de tiempo de ejecución.

En la mayoría de lenguajes de programación no hay una forma correcta de procesar un `null` pasado por accidente. Como éste es el caso, el enfoque racional es impedir que se pase `null` de forma predeterminada. Si lo hace, puede diseñar código sabiendo que `null` en una lista de argumentos indica un problema y los errores serán menores.

Conclusión

El código limpio es legible pero también debe ser robusto. No son objetivos opuestos. Podemos crear código limpio y robusto si consideramos el control de errores una preocupación diferente, algo que vemos de forma independiente desde nuestra lógica principal. Si somos capaces de lograrlo, razonaremos de forma independiente y podemos aumentar la capacidad de mantenimiento de nuestro código.

Bibliografía

- **[Martin]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

8
Límites

por James Grenning

No es habitual que controlemos todo el *software* de nuestros sistemas. En ocasiones, adquirimos paquetes de terceros o usamos código abierto. En otros casos, dependemos de equipos de nuestra propia empresa para producir

componentes o subsistemas que utilizamos. De algún modo debemos integrar este código externo con el nuestro. En este capítulo veremos prácticas y técnicas para definir con claridad los límites de nuestro *software*.

Utilizar código de terceros

Existe una tensión natural entre el proveedor de una interfaz y el usuario de la misma. Los proveedores de paquetes y estructuras de terceros abogan por una capacidad de aplicación global para poder trabajar en diversos entornos y atraer a un público más amplio. Los usuarios, por su parte, desean una interfaz centrada en sus necesidades concretas. Esta tensión puede provocar problemas en los límites de nuestros sistemas.

Analicemos `java.util.Map` como ejemplo. Como puede apreciar en la siguiente lista. `Map` tiene una amplia interfaz con numerosas prestaciones. Esta potencia y flexibilidad es muy útil, pero también puede ser un problema. Por ejemplo, nuestra aplicación puede generar un `Map` y compartirlo. Nuestra intención puede que sea que ninguno de los receptores del mapa borre sus elementos. Pero en la parte superior de la lista encontramos el método `clear()`. Cualquier usuario del mapa puede borrarlo. O puede que nuestra convención de diseño determine que sólo se puedan almacenar objetos concretos en el mapa, pero `Map` no limita de forma fiable los tipos de objetos que admite. Cualquier usuario puede añadir elementos de cualquier tipo a cualquier mapa.

- `clear() void - Map`
- `containsKey (Object key) boolean - Map`
- `containsValue (Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`

- notifyAll() void - Object
- put(Object key, Object value) Object - Map
- putAll(Map t) void - Map
- remove(Object key) Object - Map
- size() int - Map
- toString() String - Object
- values() Collection - Map
- wait() void - Object
- wait(long timeout) void - Object
- wait(long timeout, int nanos) void - Object

Figura 8.1. Los métodos de `Map`

Si nuestra aplicación necesita un mapa de `Sensor`, comprobará que los sensores se definen de esta forma:

```
Map sensors = new HashMap();
```

Tras ello, cuando otra parte del código necesite acceder a sensor, vemos este código:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

No lo vemos una sola vez, sino repetidamente a lo largo del código. El cliente de este código es responsable de obtener un objeto de `Map` y convertirlo al tipo correcto. Funciona, pero no es código limpio. Además, este código no cuenta su historia como debería. La legibilidad del código se podría mejorar mediante el uso de genéricos, como se indica a continuación:

```
    Map<Sensor> sensors = new HashMap<Sensor>();
…
    Sensor s = sensors.get(sensorId);
```

Sin embargo, esto no soluciona el problema de que `Map<Sensor>` ofrezca más prestaciones de las que necesitamos o deseamos.

Al pasar una instancia de `Map<Sensor>` en el sistema, significa que habrá muchos puntos que corregir si la interfaz de `Map` cambia. Seguramente piense que son cambios improbables, pero recuerde que se han producido al añadir compatibilidad con genéricos en Java 5. Sin duda hemos visto sistemas que impiden el uso de genéricos debido a la gran cantidad de cambios necesarios para compensar el uso liberal de `Map`.

Una forma más limpia de usar `Map` sería la siguiente. A ningún usuario

`Sensor` le importa si se usan genéricos o no. Esa opción se ha convertido (y siempre debería serlo) en un detalle de implementación.

```
public class Sensors {
   private Map sensors = new HashMap();

   public Sensor getById(String id) {
     return (Sensor) sensors.get(id);
   }
   //corte
}
```

La interfaz en el límite (`Map`) está oculta. Ha conseguido evolucionar sin apenas impacto en el resto de la aplicación. El uso de genéricos ya no es un problema ya que la conversión y la administración de tipos se procesa dentro de la clase `Sensors`.

Esta interfaz también se ha ajustado y limitado a las necesidades de la aplicación. Genera código más fácil de entender y con menor probabilidad de errores. La clase `Sensors` puede aplicar las reglas empresariales y de diseño.

No sugerimos que se encapsulen de esta forma todos los usos de Map, sino que no se pase `Map` (ni otras interfaces en el límite) por el sistema. Si usa una interfaz de límite como `Map`, manténgala dentro de la clase o la familia de clases en la que se use. Evite devolverla o aceptarla como argumento de API públicas.

Explorar y aprender límites

El código de terceros nos permite obtener mayor funcionalidad en menos tiempo. ¿Por dónde empezamos cuando queremos utilizar un paquete de terceros? Nuestra labor no es probar el código, pero sí crear pruebas para el código de terceros que utilicemos.

Imagine que no es evidente cómo usar una biblioteca de terceros. Podríamos perder uno o varios días en leer la documentación y decidir cómo usarla. Tras ello, podríamos escribir el código para usar el código de terceros y comprobar si se comporta de la forma esperada. No deberíamos sorprendernos por tener que realizar extensas sesiones de depuración intentando localizar errores en nuestro código o en el suyo.

Aprender el código de terceros es complicado, y también integrarlo. Hacer ambas cosas al mismo tiempo es el doble de complicado. Necesitamos un enfoque diferente. En lugar de experimentar y probar el nuevo material en nuestro código de producción, podríamos crear pruebas que analicen nuestro entendimiento del código de terceros. Jim Newkirk las denomina *pruebas de aprendizaje*[34].

En las pruebas de aprendizaje, invocamos la API de terceros como supuestamente la usaríamos en nuestra aplicación. Básicamente realizamos experimentos controlados para comprobar si la entendemos. Las pruebas se centran en lo que queremos obtener de la API.

Aprender `log4j`

Imagine que desea usar el paquete de Apache `log4j` en lugar de su propio dispositivo de registro personalizado. Lo descarga y abre la página inicial de la documentación. Sin una lectura exhaustiva, crea el primer caso de prueba con la esperanza de que escriba `hello` en la consola.

```
@Test
public void testLogCreate() {
   Logger logger = Logger.getLogger("MyLogger");
   logger.info("hello");
}
```

Al ejecutarlo, el registrador genera un error que nos indica que necesitamos algo denominado `Appender`. Tras investigar, descubrimos que existe un elemento `ConsoleAppender`. Creamos `ConsoleAppender` y comprobamos si hemos conseguido revelar los secretos del registro en la consola.

```
@Test
public void testLogAddAppender() {
   Logger logger = Logger.getLogger ("MyLogger");
   ConsoleAppender appender = new ConsoleAppender();
   logger.addAppender(appender);
   logger.info("hello");
}
```

En esta ocasión descubrimos que `Appender` carece de flujo de salida, algo extraño, ya que parece lógico que lo tuviera. Tras recurrir a Google,

probamos lo siguiente:

```
@Test
public void testLogAddAppender() {
   Logger logger = Logger.getLogger("MyLogger");
   logger.removeAllAppenders();
   logger.addAppender(new ConsoleAppender(
     new PatternLayout("%p %t %m%n"),
     ConsoleAppender.SYSTEM_OUT));
   logger.info("hello");
}
```

Funciona; en la consola aparece un mensaje con la palabra `hello`. Resulta extraño tener que indicarle a `ConsoleAppender` que escriba en la consola.

Al eliminar el argumento `ConsoleAppender.SystemOut`, vemos que `hello` sigue impreso. Pero al eliminar `PatternLayout`, de nuevo vemos la queja de la falta de un flujo de salida. Es un comportamiento muy extraño.

Si nos fijamos en la documentación, vemos que el constructor `ConsoleAppender` predeterminado no está configurado, lo que no parece demasiado obvio ni útil. Parece más bien un error o una incoherencia de `log4j`.

Tras nuevas búsquedas en Google, investigaciones y pruebas, conseguimos el Listado 8-1. Hemos descubierto cómo funciona `log4j` y hemos codificado esos conocimientos en un grupo de sencillas pruebas de unidad.

Listado 8-1
LogTest.java.

```
public class LogTest (
   private Logger logger;

   @Before
   public void initialize() {
     logger = Logger.getLogger("logger");
     logger.removeAllAppenders();
     Logger.getRootLogger().removeAllAppenders();
   }
   @Test
   public void basicLogger() {
     BasicConfigurator.configure();
     logger.info("basicLogger");
   }
```

```
    @Test
    public void addAppenderWithStream() {
      logger.addAppender(new ConsoleAppender(
        new PatternLayout (“%p %t %m%n”),
        ConsoleAppender.SYSTEM_OUT));
      logger.info(“addAppenderWithStream”);
    }

    @Test
    public void addAppenderWithoutStream() {
      logger.addAppender(new ConsoleAppender(
        new PatternLayout(“%p %t %m%n”)));
      logger.info(“addAppenderWithoutStream”);
    }
  }
```

Ahora sabemos cómo inicializar un sencillo registrador de consola y encapsular ese conocimiento en nuestra propia clase de registro para que el resto de la aplicación se aísle de la interfaz de límite `log4j`.

Las pruebas de aprendizaje son algo más que gratuitas

Las pruebas de aprendizaje no cuestan nada. De todas formas, hemos tenido que aprender la API y crear las pruebas fue una forma sencilla y aislada de adquirir esos conocimientos. Las pruebas de aprendizaje fueron experimentos precisos que permitieron aumentar nuestros conocimientos.

Las pruebas no sólo son gratuitas, sino también rentables. Cuando aparezcan nuevas versiones del paquete de terceros, ejecutamos las pruebas de aprendizaje para comprobar si hay diferencias de comportamiento.

Las pruebas de aprendizaje demuestran que los paquetes de terceros que usamos funcionan de la forma esperada. Una vez integrados, no hay garantía de que el código de terceros sea compatible con nuestras necesidades. Los autores originales se verán presionados para cambiar el código y ajustarlo a sus propias necesidades. Corregirán errores y añadirán nuevas funciones. En cada versión surgirán nuevos riesgos. Si el paquete de terceros cambia de una forma incompatible con nuestras pruebas, lo sabremos al instante.

Independientemente de que necesite los conocimientos proporcionados por las pruebas de aprendizaje, un límite claro debe estar respaldado por un conjunto de pruebas que ejerciten la interfaz de la misma forma que hace el

código de producción. Sin estas pruebas de límites para facilitar la transición, podríamos conservar la versión antigua más tiempo del necesario.

Usar código que todavía no existe

Existe otro tipo de límite, que separa lo conocido de lo desconocido. En ocasiones, nuestro conocimiento del código parece desvanecerse. Lo que hay al otro lado del límite es desconocido (al menos por el momento). En ocasiones, decidimos no mirar más allá del límite.

Hace años formé parte de un equipo de desarrollo de *software* para un sistema de comunicación por radio. Había un subsistema, el Transmisor, que apenas conocíamos y cuya interfaz todavía no se había diseñado. Como no queríamos quedarnos parados, comenzamos a trabajar alejándonos de la parte desconocida del código.

Sabíamos perfectamente dónde acababa nuestro mundo y comenzaba el nuevo. Mientras avanzábamos, en ocasiones nos topábamos con este límite. Aunque la ignorancia ocultaba nuestra visión más allá del límite, sabíamos cómo queríamos que fuera la interfaz. Queríamos decirle al transmisor algo como lo siguiente:

> *Ajustar el transmisor en la frecuencia proporcionada y emitir una representación analógica de los datos que provienen de este flujo.*

No sabíamos cómo hacerlo ya que todavía no se había diseñado la API. Por ello decidimos determinar después los detalles.

Para no quedarnos bloqueados, definimos nuestra propia interfaz. Le dimos un nombre sencillo, `Transmitter`. Le asignamos el método `transmit` que aceptaba una frecuencia y un flujo de datos. Es la interfaz que deseábamos haber tenido.

Lo mejor de escribir la interfaz que deseábamos haber tenido era que la controlábamos. Esto hace que el código cliente sea más legible y se ciña a los objetivos previstos.

En la figura 8.1 se aprecia que aislamos las clases

`CommunicationsController` de la API del transmisor (que no controlábamos y estaba por definir). Al usar nuestra propia interfaz específica de la aplicación, el código de `CommunicationsController` era limpio y expresivo. Una vez definida la API del transmisor, creamos `TransmitterAdapter` para reducir las distancias. El adaptador[35] encapsulaba la interacción con la API y ofrecía un único punto en el que evolucionaba.

Figura 8.1. Predicción del transmisor

Este diseño también nos ofrece un sello[36] en el código para realizar pruebas. Con un elemento `FakeTransmitter`, podemos probar las clases `CommunicationsController`. También podemos crear pruebas de límite una vez diseñada la API `Transmitter` para asegurarnos de que la utilizamos correctamente.

Límites limpios

En los límites suceden cosas interesantes. Los cambios es una de ellas. Los diseños de código correctos acomodan los cambios sin necesidad de grandes modificaciones. Cuando usamos código que no controlamos, hay que prestar especial atención a proteger nuestra inversión y asegurarnos de que los cambios futuros no son demasiado costosos. El código en los límites requiere una separación evidente y pruebas que definan expectativas. Debemos evitar que el código conozca los detalles de terceros. Es más aconsejable depender de algo que controlemos que de algo que no controlemos, y menos todavía si nos controla. Los límites de terceros se gestionan gracias a la presencia de

puntos mínimos en el código que hagan referencia a los mismos. Podemos envolverlos como hicimos con `Map` o usar un adaptador para convertir nuestra interfaz perfecta en la interfaz proporcionada. En cualquier caso, el código se lee mejor, promueve el uso coherente e interno en el límite y hay menos puntos de mantenimiento cuando cambie el código de terceros.

Bibliografía

- **[BeckTDD]**: *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison Wesley, 19%.
- **[WELC]**: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

9
Pruebas de unidad

Nuestra profesión ha evolucionado mucho en los últimos 10 años. En 1997 nadie había oído hablar del Desarrollo guiado por pruebas (DGP). Para la mayoría, las pruebas de unidad eran pequeños fragmentos de código

desechable que creábamos para asegurarnos de que nuestros programas funcionaban. Escribíamos clases y métodos, y después código *ad hoc* para probarlos, lo que implicaba algún tipo de programa controlador que nos permitiera interactuar manualmente con el programa que habíamos escrito.

Recuerdo crear un programa de C++ para un sistema incrustado de tiempo real a mediados de la década de 1990. El programa era un sencillo temporizador con la siguiente firma:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

La idea era sencilla; el método `execute` de `Command` se ejecutaba en un nuevo subproceso tras el número especificado de milisegundos. El problema era cómo probarlo. Confeccioné un sencillo programa controlador que escuchaba al teclado. Cada vez que se introducía un carácter, se programaba un comando que escribía el mismo carácter cinco segundos después. Introduje una rítmica melodía en el teclado y esperé a que se reprodujera en pantalla cinco segundos después:

> *«I… want-a-girl… just… like-the-girl-who-marr… ied… dear… old… dad.»*

Incluso tarareé la melodía mientras pulsaba la tecla, y la volví a cantar cuando aparecieron los puntos en la pantalla.

Ésa fue mi prueba. Cuando vi que funcionaba y se lo mostré a mis compañeros, me deshice del código de prueba.

Como he afirmado, nuestra profesión ha evolucionado mucho. Ahora crearía una prueba que garantizara el funcionamiento de hasta el más mínimo detalle del código. Aislaría el código del sistema operativo en lugar de invocar las funciones estándar de temporización. Las imitaría para tener control total sobre el tiempo. Programaría comandos que definieran indicadores Booleanos y avanzaría el tiempo, para observar los indicadores y asegurarme de que pasaran de `false` a `true` al cambiar el tiempo al valor correcto. Cuando superara una serie de pruebas, comprobaría que fueran adecuadas para todo el que tuviera que trabajar con el código. Me aseguraría de comprobar las pruebas y el código en el mismo paquete. Sí, hemos

avanzado mucho, pero nos queda mucho por avanzar. Los movimientos Agile y TDD han animado a muchos programadores a crear pruebas de unidad automatizadas y cada vez son más. Pero en esta alocada carrera por añadir pruebas a nuestra disciplina, muchos programadores han pasado por alto dos de los aspectos más sutiles e importantes de diseñar pruebas de calidad.

Las tres leyes del DGP

Todos sabemos que el DGP nos pide que primero creemos las pruebas de unidad, antes que el código de producción. Pero esa norma es sólo la punta del iceberg. Tenga en cuenta las tres siguientes leyes[37]:

- **Primera ley**: No debe crear código de producción hasta que haya creado una prueba de unidad que falle.
- **Segunda ley**: No debe crear más de una prueba de unidad que baste como fallida, y no compilar se considera un fallo.
- **Tercera ley**: No debe crear más código de producción que el necesario para superar la prueba de fallo actual.

Estas tres leyes generan un ciclo de unos 30 segundos de duración. Las pruebas y el código de producción se crean de forma conjunta, las pruebas unos segundos antes que el código. Si trabajamos de esta forma, crearemos decenas de pruebas al día, cientos al mes y miles al año. Si trabajamos de esta forma, las pruebas abarcarán todos los aspectos de nuestro código de producción. El tamaño de dichas pruebas, que puede ser similar al del código de producción, puede suponer un problema de administración.

Realizar pruebas limpias

Hace unos años me pidieron que dirigiera un equipo que había decidido explícitamente que su código de prueba no debía mantenerse con los mismos estándares de calidad que su código de producción. Podían incumplir las

reglas en sus pruebas de unidad. La premisa era «Rápido y directo». No era necesario que las variables tuvieran nombres adecuados, ni que las funciones de prueba fueran breves y descriptivas. No era necesario que el código de prueba estuviera bien diseñado. Bastaba con que funcionara y abarcara el código de producción.

Puede que algunos lectores empaticen con esta decisión. Puede que en el pasado creara el tipo de pruebas que cree para la clase `Timer`. Supone un gran paso crear ese tipo de pruebas desechables a diseñar una *suite* de pruebas de unidad automatizadas. Por ello, como el equipo que dirigía, puede decidir que pruebas incorrectas sea mejor que no tener pruebas.

Pero el equipo no se daba cuenta que tener pruebas incorrectas era igual o peor que no tener prueba alguna. El problema es que las pruebas deben cambiar de acuerdo a la evolución del código. Cuanto menos limpias sean, más difícil es cambiarlas. Cuando más enrevesado sea el código de prueba, más probabilidades de que dedique más tiempo a añadir nuevas pruebas a la *suite* que el empleado en crear el nuevo código de producción. Al modificar el código de producción, las pruebas antiguas comienzan a fallar y el desastre impide que las pruebas se superen, por lo que acaban por convertirse en un obstáculo interminable.

Entre versiones, aumentó el coste de mantener la *suite* de pruebas de mi equipo. Acabó por convertirse en la principal queja entre los desarrolladores. Cuando los directores preguntaron sobre este aumento, los desarrolladores culparon a las pruebas. Al final, se vieron obligados a descartar la *suite* de pruebas completa.

Pero sin una *suite* de pruebas perdieron la posibilidad de garantizar el funcionamiento esperado de los cambios en el código. Sin una *suite* de pruebas no podían asegurar que los cambios en una parte del sistema no afectaran a otras diferentes. Los defectos aumentaron, lo que propició que temieran realizar cambios. Dejaron de limpiar su código de producción por miedo a que los cambios fueran dañinos. El código de producción comenzó a corromperse. Al final, se quedaron sin pruebas, con un código de producción enmarañado y defectuoso, clientes frustrados y la sensación de que su esfuerzo les había fallado.

En cierto modo tenían razón. Su esfuerzo les había fallado. Pero fue su

decisión de permitir que las pruebas fueran incorrectas lo que provocó el fallo. Si hubieran empleado pruebas limpias, su esfuerzo no habría fallado. Puedo afirmarlo con cierta seguridad porque he participado y dirigido muchos equipos que han tenido éxito gracias a pruebas de unidad limpias.

La moraleja de la historia es sencilla: el código de prueba es tan importante como el de producción. No es un ciudadano de segunda. Requiere concentración, diseño y cuidado. Debe ser tan limpio como el código de producción.

Las pruebas propician posibilidades

Si sus pruebas no son limpias, las perderá. Y sin ellas pierde lo mismo que hace que su código de producción sea flexible. Sí, ha leído bien. Las pruebas de unidad son las que hacen que el código sea flexible y se pueda mantener y reutilizar. La razón es sencilla. Si tiene pruebas, no tendrá miedo a realizar cambios en el código. Sin pruebas, cada cambio es un posible error. Independientemente de la flexibilidad de su arquitectura, de la división del diseño, sin pruebas tendrá miedo a realizar cambios por la posibilidad de añadir errores no detectados.

Pero con las pruebas ese miedo desaparece. Cuanto mayor sea el alcance de sus pruebas, menos miedo tendrá. Podrá modificar el código con total impunidad, aunque su arquitectura no sea la mejor y el diseño sea mediocre. Podrá mejorar la arquitectura y el diseño sin miedo alguno.

Por tanto, disponer de una *suite* automatizada de pruebas de unidad que cubran el código de producción es la clave para mantener limpio el diseño y la arquitectura. Las pruebas proporcionan las posibilidades, ya que permiten el cambio.

Si sus pruebas no son limpias, la capacidad de modificar el código se verá limitada y perderá la posibilidad de mejorar la estructura de dicho código. Cuanto menos limpias sean las pruebas, menos lo será el código. En última instancia perderá las pruebas y el código se corromperá.

Pruebas limpias

¿Qué hace que una prueba sea limpia? Tres elementos: legibilidad, legibilidad y legibilidad. La legibilidad es sin duda más importante en las pruebas de unidad que en el código de producción. ¿Qué hace que una prueba sea legible? Lo mismo que en el código: claridad, simplicidad y densidad de expresión. En una prueba debe decir mucho con el menor número de expresiones posible.

Fíjese en el código de FitNesse del Listado 9-1. Estas tres pruebas son difíciles de entender y sin duda se pueden mejorar. Por un lado, hay mucho código duplicado [G5] en las invocaciones repetidas a `addPage` y `assertSubString`. Sobre todo, este código se carga con detalles que interfieren con la expresividad de la prueba.

Listado 9-1
SerializedPageResponderTest.java.

```
public void testGetPageHieratchyAsXml() throws Exception
{
   crawler.addPage(root, PathParser.parse("PageOne"));
   crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
   crawler.addPage(root, PathParser.parse("PageTwo"));

   request.setResource("root");
   request.addInput("type", "pages");
   Responder responder = new SerializedPageResponder();
   SimpleResponse response =
     (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
   String xml = response.getContent();

   assertEquals("text/xml", response.getContentType());
   assertSubString("<name>PageOne</name>", xml);
   assertSubString("<name>PageTwo</name>", xml);
   assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
   WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
   crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
   crawler.addPage(root, PathParser.parse("PageTwo"));

   PageData data = pageOne.getData();
   WikiPageProperties properties = data.getProperties();
```

```
    WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
    symLinks.set("SymPage", "PageTwo");
    pageOne.commit(data);

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
    assertNotSubString("SymPage", xml);
  }

  public void testGetDataAsHtml() throws Exception
  {
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
  }
```

Fíjese en las invocaciones de `PathParser`. Transforman cadenas en instancias de `PagePath` usadas por las arañas. Esta transformación es totalmente irrelevante para la prueba y únicamente complica su cometido. Los detalles circundantes a la creación del respondedor y la obtención y conversión de la respuesta también sobran. También la forma de crear la URL de solicitud a partir de un recurso y un argumento (contribuí a crear este código, por lo que tengo todo el derecho a criticarlo).

Al final, el código no se ha diseñado de forma legible. El lector se ve rodeado de miles de detalles que debe comprender antes de que las pruebas tengan sentido.

Fíjese ahora en las pruebas mejoradas del Listado 9-2. Hacen exactamente lo mismo, pero se han refactorizado de forma más clara y

descriptiva.

Listado 9-2
SerializedPageResponderTest.java (refactorizado)

```
public void testGetPageHierarchyAsXml() throws Exception {
   makePages("PageOne", "PageOne.ChildOne", "PageTwo");

   submitRequest("root", "type:pages");

   assertResponseIsXML();
   assertResponseContains(
     "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
   WikiPage page = makePage("PageOne");
   makePages("PageOne.ChildOne", "PageTwo");

   addLinkTo(page, "PageTwo", "SymPage");

   submitRequest("root", "type:pages");

   assertResponseIsXML();
   assertResponseContains(
     "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
   assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
   makePageWithContent("TestPageOne", "test page");

   submitRequest("TestPageOne", "type:data");

   assertResponseIsXML();
   assertResponseContains("test page", "<Test");
}
```

El patrón Generar-Operar-Comprobar[38] es evidente en la estructura de las pruebas. Cada una se divide claramente en tres partes. La primera crea los datos de prueba, la segunda opera en dichos datos y la tercera comprueba que la operación devuelva los resultados esperados.

Comprobará que se ha eliminado gran parte de los detalles molestos. Las pruebas son concisas y sólo usan los tipos de datos y funciones que realmente necesitan. Todo el que lea estas pruebas sabrá rápidamente para qué sirven y no se perderá entre detalles irrelevantes.

Lenguaje de pruebas específico del dominio

Las pruebas del Listado 9-2 ilustran la creación de un lenguaje específico del dominio para sus pruebas. En lugar de usar las API que los programadores emplean para manipular el sistema, creamos una serie de funciones y utilidades que usan dichas API y que facilitan la escritura y la lectura de las pruebas. Estas funciones y utilidades se convierten en una API especializada usada por las pruebas. Son un lenguaje de pruebas que los programadores usan personalmente para crear sus pruebas y para ayudar a los que después las lean.

Esta API de pruebas no se diseña con antelación, sino que evoluciona con la refactorización continuada del código de prueba. Al igual que refactorizamos el Listado 9-1 en el Listado 9-2, los programadores disciplinados refactorizan su código de prueba en versiones más sucintas y expresivas.

Un estándar dual

En un sentido, el equipo que mencionamos antes tenía razón. El código de la API de pruebas tiene un conjunto de estándares de ingeniería diferentes al código de producción. También tiene que ser sencillo, sucinto y expresivo, pero no tan eficaz como el código de producción. Después de todo, se ejecuta en un entorno de prueba, no de producción, y cada entorno tiene sus propias necesidades.

Fíjese en la prueba del Listado 9-3. La creé como parte de un prototipo de sistema de control medioambiental. Sin entrar en detalles, se aprecia que esta prueba comprueba que la alarma de baja temperatura, el calentador y el fuelle estén activados cuando la temperatura sea demasiado fría.

Listado 9-3
EnvironmentControllerTest.java

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
```

```
      controller.tic();
      assertTrue(hw.heaterState());
      assertTrue(hw.blowerState());
      assertFalse(hw.coolerState());
      assertFalse(hw.hiTempAlarm());
      assertTrue(hw.loTempAlarm());
    }
```

Aquí hay muchos detalles. Por ejemplo, ¿para qué sirve la función `tic`? De hecho, la ignoraría mientras leemos esta prueba. Intente centrarse en saber si está de acuerdo en que el estado final del sistema tiene que ver con que la temperatura sea demasiado baja.

Al leer la prueba, la vista tiene que cambiar entre el nombre del estado comprobado y el sentido del estado comprobado. Vemos `heaterState` y después la vista salta a `assertTrue`. Vemos `coolerState` y nos fijamos en `assertFalse`. Resulta tedioso y dificulta la lectura de la prueba.

He conseguido mejorar la legibilidad de la prueba transformándola en el Listado 9-4.

Listado 9-4
EnvironmentControllerTest.java (refactorizado)

```
  @Test
    public void turnOnLoTempAlarmAtThreshold() throws Exception {
      wayTooCold();
      assertEquals("HBchL", hw.getState());
    }
```

Evidentemente, he ocultado el detalle de la función `tic` creando una función `wayTooCold`. Pero lo importante es la extraña cadena de `assertEquals`. Las mayúsculas significan activado y las minúsculas desactivado, y las letras siempre aparece en este orden: `{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}`.

Aunque prácticamente sea un incumplimiento de las reglas de asignación mental[39], en este caso parece apropiado. Una vez que conocemos el significado, la vista pasa por la cadena y podemos interpretar los resultados. La lectura de la prueba es casi un placer. Fíjese en el Listado 9-5 y compruebe con qué facilidad entiende las pruebas.

Listado 9-5
EnvironmentControllerTest.java (una selección mayor).

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
  }

  @Test
  public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHL", hw.getState());
  }

  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

La función `getState` se reproduce en el Listado 9-6. No es un código muy eficaz. Para que lo sea, deberíamos haber usado `StringBuffer`.

Listado 9-6
MockControlHardware.java.

```
public String getState() {
  String state = "";
  state += heater ? "H" : "h";
  state += blower ? "B" : "b";
  state += cooler ? "C" : "c";
  state += hiTempAlarm ? "H" : "h";
  state += loTempAlarm ? "L" : "l";
  return state;
}
```

`StringBuffer` es poco atractivo. Incluso en código de producción, intento evitarlo si el coste es mínimo, como podría suceder en el Listado 9-6. Pero esta aplicación es claramente un sistema incrustado en tiempo real y es probable que los recursos del equipo y la memoria estén limitados. Sin

embargo, el entorno de pruebas es improbable que lo esté. Es la naturaleza del estándar dual. Hay cosas que nunca haría en un entorno de producción totalmente válidas para un entorno de prueba. Suelen ser problemas de memoria o eficacia de la CPU, pero nunca problemas de limpieza.

Una afirmación por prueba

Existe una escuela de pensamiento[40] que afirma que todas las funciones de prueba de una prueba JUnit sólo deben tener una instrucción de afirmación. Puede parecer una regla draconiana pero la ventaja se aprecia en el Listado 9-5. Las pruebas llegan a una misma conclusión, que se entiende de forma rápida y sencilla.

¿Pero qué sucede con el Listado 9-2? No parece razonable afirmar que el resultado es XML y que contiene determinadas subcadenas. Sin embargo, podemos dividir la prueba en dos, cada una con una afirmación concreta, como se muestra en el Listado 9-7.

Listado 9-7
SerializedPageResponderTest.java (una sola afirmación).

```
public void testGetPageHierarchyAsXml() throws Exception {
   givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

   whenRequestIsIssued("root", "type:pages");

   thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
   givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

   whenRequestIsIssued("root", "type:pages");

   thenResponseShouldContain(
     "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
   );
}
```

He cambiado los nombres de las funciones para usar la convención *dado-cuando-entonces*[41]. De este modo las pruebas son más fáciles de leer. Desafortunadamente, al dividir las pruebas se genera código duplicado.

Podemos eliminar los duplicados por medio del patrón Método de plantilla[42] e incluir las partes *dado/cuando* en la clase base, y las partes entonces en derivaciones diferentes. O podríamos crear una clase de prueba independiente e incluir las partes *dado* y *cuando* en la función `@Before` y las partes entonces en cada función `@Test`. Pero parece un mecanismo excesivo para un problema tan menor. Al final, opto por las afirmaciones múltiples del Listado 9-2. Considero que la regla de una sola afirmación es una directriz adecuada[43]. Siempre intento crear un lenguaje de pruebas específico del dominio que la complemente, como en el Listado 9-5, pero no rechazo incluir más de una afirmación en una prueba. Creo que lo mejor que podemos decir es que el número de afirmaciones de una prueba debe ser mínimo.

Un solo concepto por prueba

Puede que una regla más indicada sea probar un único concepto en cada función de prueba. No queremos extensas funciones que prueben una cosa diferente tras otra, como sucede en el Listado 9-8. Esta prueba debería dividirse en tres diferentes que probaran tres cosas distintas. Al combinarlas en la misma función se obliga al lector a determinar por qué cada sección se ubica en ese punto y qué prueba dicha sección.

Listado 9-8

```
/**
* Varias pruebas para el método addMonths().
*/
public void testAddMonths() {
   SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

   SerialDate d2 = SerialDate.addMonths(1, d1);
   assertEquals(30, d2.getDayOfMonth());
   assertEquals(6, d2.getMonth());
   assertEquals(2004, d2.getYYYY());

   SerialDate d3 = SerialDate.addMonths(2, d1);
   assertEquals(31, d3.getDayOfMonth());
   assertEquals(7, d3.getMonth());
   assertEquals(2004, d3.getYYYY());
```

```
    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
  }
```

Las tres funciones deberían ser las siguientes:

- Dado el último día de un mes con 31 días (como mayo):
  1. Cuando se añade un mes, si el último día de ese mes es el 30 (como en junio), entonces la fecha debe ser el día 30 de ese mes, no el 31.
  2. Cuando se añaden dos meses a esa fecha, si el último mes tiene 31 días, entonces la fecha debe ser el día 31.
- Dado el último día de un mes con 30 días (como junio):
  1. Cuando se añade, si el último día de ese mes tiene 31 días, entonces la fecha debe ser el 30, no el 31.

Expresado de esta forma, se aprecia que existe una regla general entre las distintas pruebas. Al incrementar el mes, la fecha no puede ser mayor que su último día. Esto implica que al incrementar el mes en el 28 de febrero debe generarse el 28 de marzo. Falta esa prueba y convendría que la escribiéramos.

Así pues, no son las múltiples afirmaciones del Listado 9-8 las causantes del problema, sino el hecho de que se prueba más de un concepto. Probablemente la regla óptima sea minimizar el número de activos por concepto y probar un solo concepto por función de prueba.

F.I.R.S.T.[44]

Las pruebas limpias siguen otras cinco reglas, cuyas iniciales forman las siglas FIRST en inglés:

Rapidez (*Fast*): Las reglas deben ser rápidas y ejecutarse de forma rápida. Si lo hacen lentamente, no las ejecutará con frecuencia. Al no hacerlo, no detectará los problemas con la suficiente antelación como para solucionarlos. No se sentirá con libertad para limpiar el código, que acabará corrompiéndose.

Independencia (*Independent*): Las pruebas no deben depender entre ellas. Una prueba no debe establecer condiciones para la siguiente. Debe poder ejecutar cada prueba de forma independiente y en el orden que desee. Si las pruebas dependen unas de otras, la primera que falle provocará una sucesión de fallos, dificultará el diagnóstico y ocultará efectos posteriores.

Repetición (*Repeatable*): Las pruebas deben poder repetirse en cualquier entorno. Debe poder ejecutarlas en el entorno de producción, en el de calidad y en su portátil de camino a casa en un tren sin red. Si no puede repetir las pruebas en cualquier entorno, siempre tendrá una excusa de su fallo. También verá que no puede ejecutar las pruebas si el entorno no está disponible.

Validación automática (*Self-Validating*): Las pruebas deben tener un resultado booleano: o aciertan o fallan. No debe tener que leer un extenso archivo de registro para saber si una prueba ha acertado, ni comparar manualmente dos archivos de texto distintos para ello. Si las pruebas no se validan automáticamente, el fallo puede ser subjetivo y la ejecución de las pruebas puede requerir una extensa evaluación manual.

Puntualidad (*Timely*): Las pruebas deben crearse en el momento preciso: antes del código de producción que hace que acierten. Si crea las pruebas después del código de producción, puede que resulte difícil probarlo. Puede decidir qué parte del código de producción sea demasiado difícil de probar. No diseñe código de producción que no se pueda probar.

Conclusión

Apenas hemos abordado la superficie de este tema. De hecho, se podría crear un libro entero sobre pruebas limpias. Las pruebas son tan importantes para la salud de un proyecto como el código de producción. Puede que incluso más, ya que conservan y mejoran la flexibilidad, capacidad de mantenimiento y reutilización del código de producción. Por ello, intente que sean limpias. Trabaje para que resulten expresivas y concisas. Invente API de prueba que actúen como lenguaje específico del dominio que le ayude a crear las pruebas.

Si deja que las pruebas se corrompan, sucederá lo mismo con el código de

producción. Mantenga limpias las pruebas.

Bibliografía

- **[RSpec]**: *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesay, David Chelimsky, Pragmatic Bookshelf, 2008.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

10
Clases

con Jeff Langr

Hasta ahora nos hemos centrado en escribir bien líneas y bloques de código. Nos hemos adentrado en la correcta composición de las funciones y en su interrelación. Pero a pesar de la atención dedicada a la expresividad de las instrucciones y las funciones, no tendremos código limpio hasta que nos fijemos en los niveles superiores de su organización. Hablemos sobre clases.

Organización de clases

De acuerdo a la convención estándar de Java, una clase debe comenzar con una lista de variables. Las constantes estáticas públicas, si existen, deben aparecer primero. Tras ello, las variables estáticas privadas y después las variables de instancia privadas. No suele ser necesario usar variables públicas.

Las funciones públicas deben seguir a la lista de variables. Incluimos las utilidades públicas invocadas por una función pública tras la propia función pública. Este sistema cumple la regla descendente y permite que el programa se lea como un artículo de periódico.

Encapsulación

Queremos que nuestras variables y funciones de utilidad sean privadas, pero no es imprescindible. En ocasiones podemos proteger una variable o función de utilidad para que sea accesible para una prueba. Las reglas mandan. Si una regla del mismo paquete tiene que invocar una función o acceder a una variable, hacemos que tenga ámbito `protected` o de paquete. Sin embargo, primero veremos una forma de mantener la privacidad. La relajación de la encapsulación siempre es un último resorte.

Las clases deben ser de tamaño reducido

La primera regla de las clases es que deben ser de tamaño reducido. La segunda regla es que deben ser todavía más reducidas. No, no vamos a repetir el mismo texto en el capítulo sobre las funciones, pero como sucede con las funciones, el tamaño reducido es lo principal a la hora de diseñar una clase. Y la pregunta inmediata es qué nivel de reducción. Con las funciones medimos el tamaño contando líneas físicas. Con las clases usamos otra medida distinta: las responsabilidades[45].

El Listado 10-1 muestra una clase, `SuperDashboard`, que muestra 70

métodos públicos. Muchos programadores estarán de acuerdo en que es un tamaño excesivo. Algunos denominarían a `SuperDashboard` una clase Dios.

Listado 10-1
Demasiadas responsabilidades.

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public String getCustomizerLanguagePath()
  public void setSystemConfigPath(String systemConfigPath)
  public String getSystemConfigDocument()
  public void setSystemConfigDocument(String systemConfigDocument)
  public boolean getGuruState()
  public boolean getNoviceState()
  public boolean getOpenSourceState()
  public void showObject(MetaObject object)
  public void showProgress(String s)
  public boolean isMetadataDirty()
  public void setIsMetadataDirty(boolean isMetadataDirty)
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public void setMouseSelectState(boolean isMouseSelected)
  public boolean isMouseSelected()
  public LanguageManager getLanguageManager()
  public Project getProject()
  public Project getFirstProject()
  public Project getLastProject()
  public String getNewProjectName()
  public void setComponentSizes(Dimension dim)
  public String getCurrentDir()
  public void setCurrentDir(String newDir)
  public void updateStatus(int dotPos, int markPos)
  public Class[] getDataBaseClasses()
  public MetadataFeeder getMetadataFeeder()
  public void addProject(Project project)
  public boolean setCurrentProject(Project project)
  public boolean removeProject(Project project)
  public MetaProjectHeader getProgramMetadata()
  public void resetDashboard()
  public Project loadProject(String fileName, String projectName)
  public void setCanSaveMetadata(boolean canSave)
  public MetaObject getSelectedObject()
  public void deselectObjects()
  public void setProject(Project project)
  public void editorAction(String actionName, ActionEvent event)
  public void setMode(int mode)
  public FileManager getFileManager()
  public void setFileManager(FileManager fileManager)
  public ConfigManager getConfigManager()
  public void setConfigManager(ConfigManager configManager)
  public ClassLoader getClassLoader()
```

```
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
    public MetaObject pasting(
      MetaObject target, MetaObject pasted, MetaProject project)
    public void processMenuItems(MetaObject metaObject)
    public void processMenuSeparators(MetaObject metaObject)
    public void processTabPages(MetaObject metaObject)
    public void processPlacement(MetaObject object)
    public void processCreateLayout(MetaObject object)
    public void updateDisplayLayer(MetaObject object, int layerIndex)
    public void propertyEditedRepaint(MetaObject object)
    public void processDeleteObject(MetaObject object)
    public boolean getAttachedToDesigner()
    public void processProjectChangedState(boolean hasProjectChanged)
    public void processObjectNameChanged(MetaObject object)
    public void runProject()
    public void setAllowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    //… y otros muchos métodos no públicos…
  }
```

¿Y si SuperDashboard sólo incluyera los métodos mostrados en el Listado 10-2?

Listado 10-2
¿Suficientemente reducido?

```
  public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
  }
```

Cinco métodos no es demasiado, ¿verdad? En este caso sí ya que a pesar del reducido número de métodos, `SuperDashboard` también tiene demasiadas responsabilidades.

El nombre de una clase debe describir las responsabilidades que

desempeña. De hecho, el nombre es la primera forma para determinar el tamaño de una clase. Si no podemos derivar un nombre conciso para una clase, seguramente sea demasiado extenso. Cuanto más ambiguo sea el nombre de la clase, más probabilidades hay de que tenga demasiadas responsabilidades. Por ejemplo, los nombres de clase con palabras como `Processor`, `Manager` o `Super` suelen indicar una desafortunada acumulación de responsabilidades.

También debemos ser capaces de escribir una breve descripción de la clase en unas 25 palabras, sin usar las palabras «`si`», «`o`», «`y`» o «`pero`». ¿Cómo describiríamos `SuperDashboard`?: `SuperDashboard` permite acceder al componente con el enfoque y nos permite controlar los números de versión y producto. El primer `y` indica que `SuperDashboard` tiene demasiadas responsabilidades.

El Principio de responsabilidad única

El Principio de responsabilidad única (*Single Responsibility Principle*, SRP) [46] indica que una clase o módulo debe tener uno y sólo un motivo para cambiar. Este principio nos indica la definición de responsabilidad y una directriz para el tamaño de la clase. Las clases sólo deben tener una responsabilidad, un motivo para cambiar. La clase `SuperDashboard` aparentemente reducida del Listado 10-2 tiene dos motivos para cambiar. Primero, controla información de versión que supuestamente debe actualizarse cada vez que se comercialice el *software*. Por otra parte, gestiona componentes de Java Swing (un derivado de JFrame, la representación Swing de una ventana de IGU de nivel superior). Sin duda, querremos cambiar el número de versión si cambiamos el código Swing, pero lo contrario no es necesario: podríamos cambiar la información de versión en función de los cambios de otro código del sistema. La identificación de responsabilidades (los motivos del cambio) nos permite reconocer y mejorar las abstracciones en nuestro código. Podemos extraer los tres métodos de `SuperDashboard` relacionados con la información de versiones en una clase independiente como `Version` (véase el Listado 10-3.) La clase `Version` es una construcción

que se puede reutilizar en otras aplicaciones.

Listado 10-3
Una clase con una única responsabilidad.

```
public class Version {
   public int getMajorVersionNumber()
   public int getMinorVersionNumber()
   public int getBuildNumber()
}
```

SRP es uno de los conceptos más importantes del diseño orientado a objetos y también uno de los más sencillos de entender y cumplir, pero también es uno de los que más se abusa al diseñar clases. Habitualmente nos encontramos clases que hacen demasiadas cosas. ¿Por qué?

Crear *software* que funcione y crear *software* limpio son dos actividades diferentes. Muchos tenemos un cerebro limitado, de modo que nos centramos en que el código funcione más que en su organización y limpieza. Es algo totalmente válido. Mantener objetivos separados es tan importante en nuestras actividades de programación como en nuestros programas.

El problema es que muchos creemos que hemos terminado cuando el programa funciona. No cambiamos al *otro* objetivo de organización y limpieza. Pasamos al siguiente problema en lugar de retroceder y dividir las clases en unidades independientes con una única responsabilidad.

Al mismo tiempo, muchos programadores temen que un elevado número de pequeñas clases con un único propósito dificulten la comprensión del conjunto. Les preocupa que tengan que desplazarse entre las clases para determinar cómo funciona un aspecto concreto.

Sin embargo, un sistema con muchas clases reducidas no tiene más elementos móviles que un sistema con algunas clases enormes. En ambos hay que entender lo mismo. La pregunta es si quiere organizar sus herramientas en cajas con muchos pequeños cajones que contengan componentes bien definidos y etiquetados, o usar varios cajones grandes en los que mezcle todo.

Todos los sistemas tienen una gran lógica y complejidad. El objetivo principal para gestionar dicha complejidad es organizarla para que un programador sepa dónde buscar y comprenda la complejidad directamente afectada en cada momento concreto. Por el contrario, un sistema con clases

multipropósito de mayor tamaño nos obliga a buscar entre numerosos elementos que no siempre necesitamos conocer.

Para reformular los puntos anteriores, diremos que los sistemas deben estar formados por muchas claves reducidas, no por algunas de gran tamaño. Cada clase reducida encapsula una única responsabilidad, tiene un solo motivo para cambiar y colabora con algunas otras para obtener los comportamientos deseados del sistema.

Cohesión

Las clases deben tener un número reducido de variables de instancia. Los métodos de una clase deben manipular una o varias de dichas variables. Por lo general, cuantas más variables manipule un método, más cohesión tendrá con su clase. Una clase en la que cada variable se usa en cada método tiene una cohesión máxima.

Por lo general, no es recomendable ni posible crear este tipo de clases pero queremos que la cohesión de nuestras clases sea elevada. Si lo logramos, significa que los métodos y variables de la clase dependen unos de otros y actúan como un todo lógico.

Fíjese en la implementación de `Stack` en el Listado 10-4. Es una clase muy consistente. De los tres métodos, sólo `size()` no usa ambas variables.

Listado 10-4
Stack.java, una clase consistente.

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
```

```
      if (topOfStack == 0)
        throw new PoppedWhenEmpty();
      int element = elements.get(--topOfStack);
      elements.remove(topOfStack);
      return element;
    }
  }
```

La estrategia de reducir el tamaño de las funciones y de las listas de parámetros suele provocar la proliferación de variables de instancia usadas por un subconjunto de los métodos. Si esto sucede, siempre existe al menos una clase que intenta huir de la clase de mayor tamaño. Debe intentar separar las variables y métodos en dos o más clases para que las nuevas sean más consistentes.

Mantener resultados consistentes en muchas clases de tamaño reducido

La división de grandes funciones en otras más pequeñas aumenta la proliferación de clases. Imagine una gran función con numerosas variables declaradas. Imagine que desea extraer una pequeña parte de esa función en otra independiente. Sin embargo, el código que extrae usa cuatro de las variables declaradas en la función. ¿Debe pasar las cuatro variables como argumentos a la nueva función?

En absoluto. Si ascendemos estas cuatro variables a variables de instancia de la clase, podremos extraer el código sin pasar las variables. Resultaría más sencillo dividir la función en pequeños fragmentos.

Desafortunadamente, eso significaría que nuestras clases perderían cohesión ya que acumularían más y más variables de instancia que sólo existen para que otras funciones las compartan. Pero un momento. Si apenas existen funciones que compartan determinadas variables, ¿no son entonces una clase con derecho propio? Por supuesto. Cuando las clases pierdan cohesión, divídalas.

Por tanto, dividir una gran función en otras más reducidas también nos permite dividir varias clases más reducidas. De este modo mejora la organización del programa y su estructura resulta más transparente. Como ejemplo, usaremos un ejemplo obtenido del libro de Knuth *Literate*

Programming[47]. El Listado 10-5 muestra una traducción a Java del programa `PrintPrimes` de Knuth. Para hacerle justicia, no es el programa que creó sino el resultado generado por su herramienta WEB. Lo usamos aquí por ser un magnífico punto de partida para dividir una función de gran tamaño en varias funciones y clases más reducidas.

Listado 10-5
PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
   public static void main(String[] args) {
     final int M = 1000;
     final int RR = 50;
     final int CC = 4;
     final int WW = 10;
     final int ORDMAX = 30;
     int P[] = new int[M + 1];
     int PAGENUMBER;
     int PAGEOFFSET;
     int ROWOFFSET;
     int C;
     int J;
     int K;
     boolean JPRIME;
     int ORD;
     int SQUARE;
     int N;
     int MULT[] = new int[ORDMAX + 1];

     J = 1;
     K = 1;
     P[1] = 2;
     ORD = 2;
     SQUARE = 9;

     while (K < M) {
       do {
         J = J + 2;
         if (J == SQUARE) {
           ORD = ORD + 1;
           SQUARE = P[ORD] * P[ORD];
           MULT[ORD - 1] = J;
         }
         N = 2;
         JPRIME = true;
         while (N < ORD && JPRIME) {
           while (MULT[N] < J)
```

```
            MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
          N = N + 1;
        }
      } while (!JPRIME);
      K = K + 1;
      P[K] = J;
    }
    {
      PAGENUMBER = 1;
      PAGEOFFSET = 1;
      while (PAGEOFFSET <= M) {
        System.out.println("The First " + M +
            " Prime Numbers --- Page " + PAGENUMBER);
        System.out.println("");
        for   (ROWOFFSET   =   PAGEOFFSET;   ROWOFFSET   <   PAGEOFFSET   +   RR;
ROWOFFSET++) {
          for (C = 0; C < CC;C++)
            if (ROWOFFSET + C * RR <= M)
              System.out.format("%10d", P[ROWOFFSET + C * RR]);
          System.out.println("");
        }
        System.out.println("\f");
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
      }
    }
  }
}
```

Este programa, escrito como una sola función, es un desastre. El sangrado de su estructura es excesivo y hay demasiadas variables extrañas. Como mínimo, la función debería dividirse en otras más pequeñas. Los listados del 10-6 al 10-8 muestran la división del código del Listado 10-5 en clases y funciones de menor tamaño, además de los nombres elegidos para dichas clases, funciones y variables.

Listado 10-6
PrimePrinter.java (refactorizado)

```
package literatePrimes;

public class PrimePrinter (
  public static void main(String[] args) {
    final int NUMBER_OF_PRIME5 = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
```

```
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
             COLUMNS_PER_PAGE,
             "The First " + NUMBER_OF_PRIMES +
             " Prime Numbers");
    tablePrinter.print(primes);
  }
}
```

Listado 10-7
RowColumnPagePrinter.java.

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
             int columnsPerPage,
             String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }

  public void print(int data[]) {
    int pageNumber = 1;
    for (int firstIndexOnPage = 0;
        firstIndexOnPage < data.length;
        firstIndexOnPage += numbersPerPage) {
      int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
             data.length - 1);
      printPageHeader(pageHeader, pageNumber);
      printPage(firstIndexOnPage, lastIndexOnPage, data);
      printStream.println("\f");
      pageNumber++;
    }
  }

  private void printPage (int firstIndexOnPage,
             int lastIndexOnPage,
             int[] data) {
```

```
    int firstIndexOfLastRowOnPage =
      firstIndexOnPage + rowsPerPage - 1;
    for (int firstIndexInRow = firstIndexOnPage;
        firstIndexInRow <= firstIndexOfLastRowOnPage;
        firstIndexInRow++) {
      printRow(firstIndexInRow, lastIndexOnPage, data);
      printStream.println("");
    }
  }

  private void printRow(int firstIndexInRow,
            int lastIndexOnPage,
            int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
      int index = firstIndexInRow + column * rowsPerPage;
      if (index <= lastIndexOnPage)
        printStream.format("%10d", data[index]);
    }
  }

  private void printPageHeader(String pageHeader,
            int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
  }

  public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
  }
}
```

Listado 10-8
PrimeGenerator.java

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirtsPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
```

```
    }

    private static void checkOddNumbersForSubsequentPrimes() {
      int primeIndex = 1;
      for (int candidate = 3;
          primeIndex < primes.length;
          candidate += 2) {
        if (isPrime(candidate))
          primes[primeIndex++] = candidate;
      }
    }

    private static boolean isPrime(int candidate) {
      if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
        multiplesOfPrimeFactors.add(candidate);
        return false;
      }
      return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }

    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
      int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
      int      leastRelevantMultiple      =      nextLargerPrimeFactor      *
nextLargerPrimeFactor;
      return candidate == leastRelevantMultiple;
    }

    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
      for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n))
          return false;
      }
      return true;
    }

    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
      return
        candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }

    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
      int multiple = multiplesOfPrimeFactors.get(n);
      while (multiple < candidate)
        multiple += 2 * primes[n];
      multiplesOfPrimeFactors.set(n, multiple);
      return multiple;
    }
  }
```

Lo primero que apreciará es que ha aumentado la longitud del programa,

de una a casi tres páginas. Este aumento se debe a varios motivos. En primer lugar, el programa refactorizado usa nombres de variable más extensos y descriptivos. Por otra parte, usa declaraciones de funciones y clases como comentarios del código. Por último, usamos espacios en blanco y técnicas de formato para mantener la legibilidad.

El programa se ha dividido en tres responsabilidades principales. La parte principal se incluye en la clase `PrimePrinter`, responsable de controlar el entorno de ejecución. Cambia si se modifica el método de invocación. Por ejemplo, si este programa se convierte en un servicio SOA, es la clase que se verá afectada.

`RowColumnPagePrinter` sabe cómo aplicar formato a una lista de números con una determinada cantidad de filas y columnas. Si es necesario cambiar el formato del resultado, es la clase que se verá afectada.

La clase `PrimeGenerator` sabe cómo generar una lista de números primos. No se creará una instancia como objeto. La clase es sólo un ámbito útil en el que declarar y ocultar sus variables. Esta clase cambia si se modifica el algoritmo para calcular números primos. No hemos reescrito el programa. No hemos empezado de cero y los hemos vuelto a diseñar. En realidad, si se fija atentamente en los dos programas, verá que usan los mismos algoritmos y mecanismos.

El cambio se ha realizado creando una *suite* de pruebas que verifican el comportamiento preciso del primer programa. Tras ello, se aplican numerosos cambios mínimos, de uno en uno. Tras cada cambio, se ejecuta el programa para garantizar que el comportamiento no varía. Paso a paso, el primer programa se limpia y se transforma en el segundo.

Organizar los cambios

En muchos sistemas, el cambio es continuo. Cada cambio supone un riesgo de que el resto del sistema no funcione de la forma esperada. En un sistema limpio organizamos las clases para reducir los riesgos de los cambios.

La clase `Sql` del Listado 10-9 se usa para generar cadenas SQL de forma correcta con los metadatos adecuados. Es un trabajo continuo y, como tal, no

admite funciones SQL como instrucciones `update`. Cuando la clase `Sql` tenga que admitir una instrucción `update`, tendremos que abrirla para realizar modificaciones. El problema de abrir una clase es el riesgo que conlleva. Cualquier modificación puede afectar a otro código de la clase. Debe probarse concienzudamente.

Listado 10-9
Clase que debemos abrir para realizar cambios.

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
  private String valuesList(Object[] fields, final Column[] columns)
  private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

La clase `Sql` debe cambiar al añadir un nuevo tipo de instrucción. También debe cambiar cuando variemos los detalles de un tipo de instrucción concreto; por ejemplo, si tenemos que modificar la funcionalidad `select` para admitir selecciones secundarias. Estos dos motivos de cambio significan que la clase `Sql` incumple SRP.

Podemos detectar este incumplimiento desde un punto de vista organizativo. El método `outline` de `Sql` muestra que hay métodos privados, como `selectWithCriteria`, que parecen relacionarse únicamente con instrucciones `select`.

El comportamiento de métodos privados aplicados a un pequeño subconjunto de una clase puede ser una heurística útil para detectar zonas que mejorar. Sin embargo, la verdadera razón debe ser el cambio del sistema. Si la clase `Sql` se considera totalmente lógica, no debemos preocuparnos por separar las responsabilidades. Si no necesitamos funcionalidad de actualización en el futuro, podemos olvidarnos de `Sql`. Pero si tenemos que

abrir una clase, debemos corregir el diseño.

¿Y si optamos por una solución como la del Listado 10-10? Los métodos públicos de interfaz definidos en `Sql` en el Listado 10-9 se refactorizan en sus propias variantes de la clase `Sql`. Los métodos privados, como `valuesList`, se mueven directamente a las posiciones necesarias. El comportamiento privado se reduce a un par de clases de utilidad: `Where` y `ColumnList`.

Listado 10-10
Un grupo de clases cerradas.

```
Abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}

public class CreateSql extends Sql {
   public CreateSql(String table, Column[] columns)
   @Override public String generate()
}

public class SelectSql extends Sql {
   public SelectSql(String table, Column[] columns)
   @Override public String generate()
}

public class InsertSql extends Sql {
   public SelectSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
   public SelectWithCriteriaSql(
     String table, Column[] columns, Criteria criteria)
   @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
   public SelectWithMatchSql(
     String table, Column[] columns, Column column, String pattern)
   @Override public String generate()
}

public class FindByKeySql extends Sql {
   public FindByKeySql(
     String table, Column[] columns, String keyColumn, String keyValue)
   @Override public String generate()
}
```

```
public class PreparedInsertSql extends Sql {
   public PreparedInsertSql(String table, Column[] columns)
   @Override public String generate()
   private String placeholderList(Column[] columns)
}

public class Where {
   public Where(String criteria)
   public String generate()
}

public class ColumnList {
   public ColumnList(Column[] columns)
   public String generate()
}
```

El código de cada clase se simplifica enormemente. El tiempo necesario para entender las clases se reduce al mínimo. El riesgo de que una función afecte a otra desaparece casi por completo. Desde el punto de vista de las pruebas, resulta más sencillo probar la lógica de esta solución, ya que las clases se aíslan unas de otras.

Además, cuando llegue el momento de añadir las instrucciones `update`, no cambia ninguna de las clases existentes. Añadimos la lógica para generar instrucciones `update` a una nueva subclase de `Sql`, `UpdateSql`. Este cambio no afecta a otro código del sistema.

Nuestra lógica `Sql` reestructurada representa lo mejor de ambos mundos. Cumple con SRP y también con otro principio clave del diseño de clases orientadas a objetos, denominado Principio abierto/cerrado[48]: las clases deben abrirse para su ampliación para cerrarse para su modificación. La nueva clase `Sql` se abre a nuevas funcionalidades mediante la creación de subclases pero podemos realizar estos cambios y mantener cerradas las demás clases. Basta con añadir nuestra clase `UpdateSql`.

Debemos estructurar nuestros sistemas para ensuciarlos lo menos posible cuando los actualicemos con nuevas funciones o cambios. En un sistema ideal, incorporamos nuevas funciones ampliándolo, no modificando el código existente.

Aislarnos de los cambios

Las necesidades cambiarán y también lo hará el código. En la programación orientada a objetos aprendemos que hay clases concretas que contienen detalles de implementación (el código) y clases abstractas que sólo representan conceptos. Una clase cliente que dependa de detalles concretos está en peligro si dichos detalles cambian. Podemos recurrir a interfaces y clases abstractas para aislar el impacto de dichos detalles.

Las dependencias de detalles de concretos crean retos para nuestro sistema. Si tenemos que crear la clase `Portfolio` y ésta depende de una API `TokyoStockExchange` externa para obtener su valor, nuestros casos de prueba se verán afectados por la volatilidad de esta búsqueda. Resulta complicado crear una prueba cuando se obtiene una respuesta diferente cada cinco minutos. En lugar de diseñar `Portfolio` para que dependa directamente de `TokyoStockExchange`, creamos una interfaz, `StockExchange`, que declara un único método:

```
public Interface StockExchange {
   Money currentPrice(String symbol);
}
```

Diseñamos `TokyoStockExchange` para implementar esta interfaz. También nos aseguramos de que el constructor de `Portfolio` adopte como argumento una referencia a StockExchange:

```
public Portfolio {
   private StockExchange exchange;
   public Portfolio(StockExchange exchange) {
     this.exchange = exchange;
   }
   //…
}
```

Ahora la prueba puede crear una implementación de la interfaz `StockExchange` que emule `TokyoStockExchange`. Esta implementación de prueba fijará el valor actual del símbolo que usemos en la prueba.

Si nuestra prueba demuestra la adquisición de cinco acciones de Microsoft para nuestra cartera de valores, diseñe el código de la implementación de prueba para que siempre devuelva 100 dólares por acción de Microsoft. Nuestra implementación de prueba de la interfaz `StockExchange` se reduce a una sencilla búsqueda de tabla. De este modo

podemos crear una prueba que espere un valor de cartera total de 500 dólares:

```
public class PortfolioTest {
   private FixedStockExchangeStub exchange;
   private Portfolio portfolio;

   @Before
   protected void setUp() throws Exception {
     exchange = new FixedStockExchangeStub();
     exchange.fix("MSFT", 100);
     portfolio = new Portfolio(exchange);
   }

   @Test
   public void GivenFiveMSFTTotalShouldBe500() throws Exception {
     portfolio.add(5, "MSFT");
     Assert.assertEquals(500, portfolio.value());
   }
}
```

Si diseccionamos un sistema para poder probarlo de esta forma, resultará más flexible y se podrá reutilizar. La ausencia de conexiones significa que los elementos del sistema se aíslan entre ellos y de otros cambios. Este aislamiento hace que comprendamos mejor los elementos del sistema.

Al minimizar las conexiones de esta forma, nuestras clases cumplen otro principio de diseño: *Dependency Inversion Principle* (DIP) o Principio de inversión de dependencias[49]. Básicamente afirma que nuestras clases deben depender de abstracciones, no de detalles concretos.

En lugar de depender de los detalles de implementación de la clase `TokyoStockExchange`, nuestra clase Portfolio depende de la interfaz `StockExchange`, que representa el concepto abstracto de solicitar el precio actual de una acción. Esta abstracción aísla todos los datos concretos de la obtención de dicho precio, incluyendo de dónde se obtiene.

Bibliografía

- **[RDD]**: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

- **[Knuth92]**: *Literate Programming*, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.

11
Sistemas

por el Dr. Kevin Dean Wampler

«La complejidad es letal. Acaba con los desarrolladores y dificulta la planificación, generación y pruebas de los productos».
—Ray Ozzie, CTO, Microsoft Corporation

Cómo construir una ciudad

¿Podría encargarse de todos los detalles por su cuenta? Seguramente no. Incluso la gestión de una ciudad existente sería demasiado para una sola persona. Y aun así, las ciudades funcionan (en la mayoría de los casos). Funcionan porque tienen equipos que controlan partes concretas de la ciudad, el alcantarillado, la red eléctrica, el tráfico, la seguridad, las normativas

urbanísticas, etc. Algunos se encargan de aspectos generales y otros se centran en los detalles.

Las ciudades también funcionan porque disponen de evolucionados niveles de abstracción y modularidad que permiten a individuos y componentes trabajar de forma eficaz, sin necesidad de entender el trasfondo general.

Aunque los equipos de *software* se suelen organizar de esta forma, los sistemas en los que trabajan no suelen contar con la misma separación de aspectos y niveles de abstracción. En este capítulo veremos cómo mantener la limpieza en niveles superiores de abstracción, en el sistema.

Separar la construcción de un sistema de su uso

En primer lugar, recuerde que la construcción es un proceso muy diferente al uso. Mientras escribo estas líneas, a través de la ventana veo un nuevo hotel en construcción en Chicago. Hoy instalarán una gran grúa. Todos los obreros llevan casco. Dentro de un año habrán acabado el hotel. La grúa desaparecerá. El edificio estará terminado, con su reluciente fachada de cristal y su atractiva decoración. La gente que trabajará en él también será diferente.

> *Los sistemas de software deben separar el proceso de inicio, en el que se crean los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que toma el testigo tras el inicio.*

El proceso de inicio es un aspecto que toda aplicación debe abordar. Es el primero que veremos en este capítulo. La separación de aspectos es una de las técnicas de diseño más antiguas e importantes de nuestra profesión.

Desafortunadamente, muchas aplicaciones no lo hacen. El código del proceso de inicio se mezcla con la lógica de tiempo de ejecución. Veamos un ejemplo típico:

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl (…); //¿Lo bastante predeterminado para la
```

```
mayoría de los casos?
     return service;
  }
```

Es la técnica de *inicialización/evaluación tardía* y tiene sus méritos. No incurrimos en la sobrecarga de la construcción a menos que usemos el objeto realmente, y como resultado el tiempo de inicio se puede acelerar. También evitamos que se devuelva `null`.

Sin embargo, ahora tenemos una dependencia en `MyServiceImpl` y todo lo que su constructor requiere (que he omitido). No podemos compilar sin resolver estas dependencias, aunque nunca usemos un objeto de este tipo en tiempo de ejecución.

Las pruebas también pueden ser un problema. Si `MyServiceImpl` es un objeto pesado, tendremos que asegurarnos de asignar el correspondiente *test double*[50] u objeto simulado al campo de servicio antes de invocar este método en las pruebas de unidad. Como la lógica de la construcción se mezcla con el procesamiento normal de tiempo de ejecución, debemos probar todas las rutas de ejecución (como la prueba `null` y su bloque). Al contar con ambas responsabilidades, el método hace más de una cosa, por lo que se incumple el principio de responsabilidad única.

Lo peor de todo es que no sabemos si `MyServiceImpl` es el objeto correcto en todos los casos. ¿Por qué la clase con este método tiene que conocer el contexto global? ¿Podemos saber realmente cuál es el objeto correcto que usar aquí? ¿Es posible que un mismo tipo sea el correcto para todos los contextos posibles?

Un caso de *inicialización tardía* no es un problema serio. Sin embargo, suele haber muchos casos de este tipo de configuración en las aplicaciones. Por tanto, la estrategia de configuración global (si existe) se disemina por la aplicación, sin apenas modularidad y con una significativa duplicación.

Si somos diligentes sobre el diseño de sistemas robustos y bien formados, no debemos permitir fallos de modularidad. El proceso de inicio de la construcción y conexión de objetos no es una excepción. Debemos modularizar este proceso y asegurarnos de contar con una estrategia global y coherente para resolver las dependencias principales.

Separar Main

Una forma de separar la construcción del uso consiste en trasladar todos los aspectos de la construcción a `main` o a módulos invocados por `main`, y diseñar el resto del sistema suponiendo que todos los objetos se han creado y conectado correctamente (véase la figura 11.1).

El flujo de control es fácil de seguir. La función `main` crea los objetos necesarios para el sistema, los pasa a la aplicación y ésta los utiliza. Verá que las flechas de dependencia atraviesan la barrera entre `main` y la aplicación. Todas van en la misma dirección, alejándose de `main`, lo que significa que la aplicación no tiene conocimiento de `main` ni del proceso de construcción. Simplemente espera que todo se haya construido correctamente.

Figura 11.1. Separación de la construcción en `main()`.

Factorías

En ocasiones, la aplicación tendrá que ser responsable de la creación de un objeto. Por ejemplo, en un sistema de procesamiento de pedidos, la aplicación debe crear las instancias `LineItem` que añadir a `Order`. En este caso, podemos usar el patrón de factoría abstracta[51] para que la aplicación controle cuándo crear `LineItem`, pero mantener los detalles de dicha construcción separados del código de la aplicación (véase la figura 11.2).

De nuevo vemos que todas las dependencias se desplazan desde `main` a la aplicación `OrderProcessing`, lo que significa que la aplicación se desconecta de los detalles de creación de `LineItem`. Esta capacidad se incluye en `LineItemFactoryImplementation`, en el extremo `main` de la línea. Y sin embargo, la aplicación tiene control total sobre cuándo se crean las instancias `LineItem` e incluso puede proporcionar argumentos de constructor específicos de la aplicación.

Figura 11.2. Separación de la construcción con una factoría.

Inyectar dependencias

Un potente mecanismo para separar la construcción del uso es la Inyección de dependencias, la aplicación de Inversión de control (*Inversion of Control* o IoC) a la administración de dependencias[52]. La Inversión de control pasa responsabilidades secundarias de un objeto a otros dedicados a ese cometido, por lo que admite el principio de responsabilidad única. En el contexto de la administración de dependencias, un objeto no debe ser responsable de instanciar dependencias, sino que debe delegar esta responsabilidad en otro mecanismo autorizado, de modo que se invierte el control. Como la configuración es un aspecto global, este mecanismo autorizado suele ser la

rutina `main` o un contenedor de propósito especial.

Las búsquedas JNDI son una implementación parcial de la inyección de dependencias, en las que un objeto solicita a un servidor de directorios un servicio que coincida con un nombre concreto.

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

El objeto invocador no controla el tipo de objeto devuelto (siempre que implemente la interfaz correcta, evidentemente), pero es el que resuelve la dependencia de forma activa.

La verdadera inyección de dependencias va un paso más allá. La clase no hace nada directamente para resolver sus dependencias, es totalmente pasiva. Por el contrario, ofrece métodos de establecimiento o argumentos de constructor (o ambos) que se usan para inyectar las dependencias. En el proceso de construcción, el contenedor de inyección de dependencias crea instancias de los objetos necesarios (normalmente bajo demanda) y usa los argumentos de constructor o métodos de establecimiento proporcionados para conectar las dependencias. Los objetos dependientes empleados suelen especificarse a través de un archivo de configuración o mediante programación en un módulo de construcción de propósito especial.

La estructura Spring proporciona el contenedor de inyección de dependencias más conocido para Java[53]. Los objetos que se van a conectar se definen en un archivo de configuración XML y después se solicitan objetos concretos por nombre en código de Java. Veremos un ejemplo en breve.

¿Y qué sucede con las virtudes de la inicialización tardía? En ocasiones es útil con la inyección de dependencias. Por un lado, muchos contenedores de inyección de dependencias no crean un objeto hasta que es necesario. Por otra parte, muchos de estos contenedores cuentan con mecanismos para invocar factorías o crear `proxies` que se pueden usar para evaluación tardía y optimizaciones similares[54].

Evolucionar

Las ciudades nacen de pueblos, que nacen de asentamientos. Inicialmente, los

caminos son estrechos y prácticamente inexistentes, después se asfaltan y aumentan de tamaño.

Los pequeños edificios y solares vacíos se llenan de otros mayores que acaban convirtiéndose en rascacielos. Al principio no hay servicios, electricidad, agua, alcantarillado o Internet (¡vaya!). Estos servicios se añaden cuando aumenta la densidad de población.

Este crecimiento no es fácil. Cuántas veces mientras conduce por una carretera llena de baches y ve una señal de obras no se ha preguntado por qué no la hicieron más ancha desde un principio.

No se podía haber hecho de otra forma. ¿Quién puede justificar el gasto en una autopista de seis carriles que atraviese un pequeño pueblo como anticipación a un supuesto crecimiento? ¿Quién querría una autopista así en su ciudad?

Conseguir sistemas perfectos a la primera es un mito. Por el contrario, debemos implementar hoy, y refactorizar y ampliar mañana. Es la esencia de la agilidad iterativa e incremental. El desarrollo controlado por pruebas, la refactorización y el código limpio que generan hace que funcione a nivel del código.

¿Pero qué sucede en el nivel del sistema? ¿La arquitectura del sistema no requiere una planificación previa? Sin duda no puede aumentar incrementalmente algo sencillo a algo complejo, ¿o sí?

> *Los sistemas de software son únicos si los comparamos con los sistemas físicos. Sus arquitecturas pueden crecer incrementalmente, si mantenemos la correcta separación de los aspectos.*

La naturaleza efímera de los sistemas de *software* hace que sea posible, como veremos. Primero nos centraremos en una arquitectura que no separa correctamente los aspectos. Las arquitecturas EJB1 y EJB2 originales no separaban correctamente los aspectos y por tanto imponían barreras innecesarias al crecimiento orgánico. Imagine un bean de entidad para una clase `Bank` persistente. Un bean de entidad es una representación en memoria de datos relacionales, es decir, una fila de una tabla.

Primero, debe definir una interfaz local (en proceso) o remota (MVJ

independiente), que los clientes usen. El Listado 1-1 muestra una posible interfaz local:

Listado 11-1
Una interfaz local EJB2 para el EJB Bank.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
   String getStreetAddrl{} throws EJBException;
   String getStreetAddr2{} throws EJBException;
   String getCity() throws EJBException;
   String getState() throws EJBException;
   String getZipCode() throws EJBException;
   void setStreetAddr1(String street1) throws EJBException;
   void setStreetAddr2(String street2) throws EJBException;
   void setCity(String city) throws EJBException;
   void setState(String state) throws EJBException;
   void setZipCode(String zip) throws EJBException;
   Collection getAccounts() throws EJBException;
   void setAccounts(Collection accounts) throws EJBException;
   void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Mostramos diversos atributos de la dirección de `Bank` y una colección de cuentas del banco, cuyos datos se procesarán por un EJB `Account` diferente. El Listado 11-2 muestra la correspondiente clase de implementación del bean `Bank`.

Listado 11-2
Implementación del bean de entidad EJB2.

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
   // Lógica empresarial…
   public abstract String getStreerAddr1();
   public abstract String getStreetAddr2();
   public abstract String getCity();
   public abstract String getState();
   public abstract String getZipCode();
   public abstract void setStreetAddr1(String street1);
```

```
    public abstract void setStreetAddr2(String street2);
    public abstract void setcity(String city);
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
    public abstract Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountPTO accountDTO) {
      InitialContext context = new InitialContext();
      AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
      AccountLocal account = accountHome.create(accountDTO);
      Collection accounts = getAccounts();
      accounts.add(account);
    }
    // Lógica del contenedor EJB
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) {…}
    public void ejbPostCreate(Integer id) {…}
    // El resto tendría que implementarse pero se deja vacío:
    public void setEntityContext(EntityContext ctxt) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
  }
```

No mostramos la correspondiente interfaz `LocalHome`, básicamente una factoría usada para crear objetos, no los métodos de consulta `Bank` que pueda añadir.

Por último, debemos crear uno o varios descriptores de implementación XML que especifiquen los detalles de asignación relacional de objetos en un almacén persistente, el comportamiento deseado de la transacción, limitaciones de seguridad y demás.

La lógica empresarial está directamente conectada al contenedor de la aplicación EJB2. Debe crear subclases de tipos de contenedor y proporcionar los métodos de ciclo vital necesarios para el contenedor. Debido a esta conexión al contenedor pesado, las pruebas de unidad aisladas son complicadas. Es necesario imitar el contenedor, algo difícil, o perder demasiado tiempo en la implementación de EJB y pruebas en un servidor real. La reutilización fuera de la arquitectura EJB2 es imposible, debido a esta estrecha conexión. Por último, incluso la programación orientada a objetos se ve afectada. Un bean no se puede heredar de otro. Fíjese en la lógica para

añadir una nueva cuenta. En bean EJB2 es habitual definir Objetos de transferencia de datos (*Data Transfer Objects* o DTO), estructuras sin comportamiento. Esto suele generar tipos redundantes con los mismos datos y requiere código predefinido para copiar datos entre objetos.

Aspectos transversales

La arquitectura EJB2 se acerca a la verdadera separación de aspectos en determinados aspectos. Por ejemplo, los comportamientos transaccionales, de seguridad y comportamiento deseados se declaran en los descriptores de implementación, independientemente del código fuente. Aspectos como la persistencia suelen cruzar los límites de objeto naturales de un dominio. Por lo general intentará mantener todos sus objetos mediante la misma estrategia, por ejemplo con un determinado DBMS[55] y no archivos planos, usando determinadas convenciones de nomenclatura para tablas y columnas, una semántica transaccional coherente, etc.

En principio, puede razonar su estrategia de persistencia de una forma modular y encapsulada, pero en la práctica tendrá que distribuir el mismo código que implemente la estrategia de persistencia entre varios objetos. Usamos el término transversales para este tipo de aspectos. De nuevo, la estructura de persistencia podría ser modular y la lógica de dominios, aislada, también. El problema es la intersección entre ambos dominios.

De hecho, la forma en que la arquitectura EJB procesa persistencia, seguridad y transacciones es una Programación orientada a aspectos (*Aspect Oriented Programming* o AOP)[56] anticipada, un enfoque de carácter general para restaurar la modularidad en aspectos transversales. En AOP, construcciones modulares denominadas aspectos especifican qué puntos del sistema deben modificar su comportamiento de forma coherente para admitir un determinado aspecto. Esta especificación se realiza mediante un sucinto mecanismo de declaración o programación.

Si usamos la persistencia como ejemplo, podría declarar qué objetos y atributos (o patrones) deben conservarse y después delegar las tareas de persistencia a su estructura de persistencia. Las modificaciones de

comportamiento no son invasivas[57] para el código de destino. Veamos tres aspectos o mecanismos similares en Java.

Proxies de Java

Los `proxies` de Java son útiles en casos sencillos, como envolver invocaciones de métodos en objetos o clases concretas. Sin embargo, los proxies dinámicos proporcionados en el JDK sólo funcionan con interfaces. Para aplicarlos a clases, debe usar una biblioteca de manipulación de código de *bytes*, como CGLIB, ASM o Javassist[58].

El Listado 11-3 muestra la estructura de un proxy JDK para ofrecer asistencia de persistencia a nuestra aplicación Bank; únicamente abarca los métodos para obtener y establecer la lista de cuentas.

Listado 11-3
Ejemplo de proxy del JDK.

```
// Bank.java (eliminando nombres de paquetes…)
import java.utils.*;

// La abstracción de un banco.
public interface Bank {
   Collection<Account> getAccounts();
   void setAccounts(Collection<Accounts> accounts);
}

// BankImpl.java
import java.utils.*;

// "Plain Old Java Object" POJO que implementa la abstracción.
public class BankImpl implements Bank {
   private List<Account> accounts;

   public Collection<Account> getAccounts() {
     return accounts;
   }
   public void setAccounts(Collections<Accounts> accounts) {
     this.accounts = new ArrayList<Accounts>();
     for (Account account: accounts) {
       this.accounts.add(account);
     }
   }
}
```

```
// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// «InvocationHandler» necesario para la API de proxy.
public class BankProxyHandler implements InvocationHandler {
   private Bank bank;

   public BankHandler (Bank bank) {
     this.bank = bank;
   }

   // Método definido en InvocationHandler
   public Object invoke(Object proxy, Method method, Object[] args)
       throws Throwable {
     String methodName = method.getName();
     if (methodName.equals("getAccounts")) {
       bank.setAccounts(getAccountsFromDatabase());
       return bank.getAccounts();
     } else if (methodName.equals("setAccounts")) {
       bank.setAccounts((Collection<Account>) args[0]);
       setAccountsToDatabase(bank.getAccounts());
       return null;
     } else {
       …
     }
   }

   // Muchos detalles:
   protected Collection<Account> getAccountsFromDatabase() {…}
   protected void setAccountsToDatabase(Collection<Account> accounts) {…}
}

//En otra parte…

Bank bank = (Bank) Proxy.newproxyInstance(
   Bank.class.getClassLoader(),
   new Class[] { Bank.class },
   new BankProxyHandler(new BankImpl()));
```

Definimos la interfaz `Bank`, que envolvemos en el proxy y un POJO (*Plain-Old Object* u Objeto sencillo de Java), `BankImpl`, que implementa la lógica empresarial (encontrará más información sobre POJO en un apartado posterior).

La API Proxy requiere un objeto `InvocationHandler` que invocar para implementar las invocaciones de métodos `Bank` realizadas en el proxy. `BankProxyHandler` usa la API de reflexión de Java para asignar las invocaciones de métodos genéricos a los métodos correspondientes de

`BankImpl`, y así sucesivamente.

El código es abundante y complejo, incluso para este sencillo caso[59]. El uso de una de las bibliotecas de manipulación de *bytes* es igualmente complicado. El volumen y la complejidad de este código son dos de los inconvenientes de los `proxies`. Dificultan la creación de código limpio. Además, los proxies no ofrecen un mecanismo para especificar puntos de ejecución globales del sistema, imprescindibles para una verdadera solución AOP[60].

Estructuras AOP Java puras

Afortunadamente, gran parte del código predefinido de proxy se puede procesar de forma automática mediante herramientas. Los proxies se usan internamente en varias estructuras de Java como Spring AOP y JBoss AOP, para implementar aspectos en Java[61]. En Spring, se crea la lógica empresarial en forma de POJO, específicos de su dominio. No dependen de estructuras empresariales (ni de otros dominios). Por tanto, son conceptualmente más sencillos y más fáciles de probar. Su relativa simplicidad garantiza que se implementen correctamente las correspondientes historias y el mantenimiento y evolución del código en historias futuras.

La infraestructura necesaria de la aplicación, incluidos aspectos transversales como persistencia, transacciones, seguridad, almacenamiento en caché y recuperación ante fallos, se incorpora por medio de archivos de configuración declarativos o API. En muchos casos, se especifican aspectos de bibliotecas Spring o JBoss, en los que la estructura controla el uso de proxies de Java o bibliotecas de código de *bytes* de forma transparente al usuario. Estas declaraciones controlan el contenedor de inyección de dependencias, que crea instancias de los principales objetos y las conecta bajo demanda.

El Listado 11-4 muestra un fragmento tipo de un archivo de configuración de Spring V2.5, app.xml[62].

Listado 11-4
Archivo de configuración de Spring 2.X

```
<beans>
  …
  <bean id="appDataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"
  p:driverClassName="com.mysql.jdbc.Driver"
  p:url="jdbc:mysql://localhost:3306/mydb"
  p:username="me"/>

  <bean id="bankDataAccessObject"
  class="com.example.banking.persistence.BankDataAccessObject"
  p:dataSource-ref="appDataSource"/>

  <bean id="bank"
  class="com.example.banking.model.Bank"
  p:dataAccessObject-ref="bankDataAccessObject"/>
  …
</beans>
```

Cada bean es como una parte de una muñeca rusa anidada, con un objeto de domino de un proxy `Bank` (envuelto) por un Objeto de acceso a datos (*Data Accessor Object*, DAO), que también se procesa a través de un proxy por medio de un origen de datos de controlador JDBC (véase la figura 11.3).

Figura 11.3. La "muñeca rusa" de elementos de decoración.

El cliente cree que invoca `getAccounts()` en un objeto `Bank`, pero en realidad se comunica con el objeto `DECORATOR`[63] más externo de un grupo, un objeto que amplía el comportamiento básico del POJO `Bank`. Podríamos añadir otros objetos de decoración para transacciones, almacenamiento en caché y demás.

En la aplicación, bastan unas líneas para solicitar al contenedor de ID los objetos de nivel superior del sistema, como se especifica en el archivo XML.

```
XmlBeanFactory bf =
   new XmlBeanFactory(new ClassPathResource("app.xml", getclass()));
Bank bank = (Bank) bf.getBean("bank");
```

Como apenas se necesitan líneas de código Java específico de Spring, la aplicación se desconecta casi por completo de Spring y desaparecen los problemas de conexión de sistemas como EJB2.

Aunque XML puede ser difícil de leer[64], la directiva especificada en estos archivos de configuración es más sencilla que la complicada lógica de proxy y aspectos oculta a la vista y creada de forma automática. Es una arquitectura tan atractiva que sistemas como Spring modificaron totalmente el estándar EJB para la versión 3. EJB3 sigue el modelo de Spring de aspectos transversales admitidos mediante declaraciones con archivos de configuración XML y/o anotaciones de Java 5.

El Listado 11-5 muestra nuestro objeto Bank reescrito en EJB3[65].

Listado 11-5
Un EJB Bank EJB3.

```
package com.example.banking.model;
import javax.persistence;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private int id;

   @Embeddable // Un objeto en línea en la fila DB de Bank
   public class Address {
     protected String streetAddr1;
     protected String streetAddr2;
     protected String city;
     protected String state;
     protected String zipCode;
   }

   @Embedded
   private Address address;

   @OneToMany (cascade = CascadeType.ALL, fetch = FetchType.EAGER,
          mappedBy="bank")
   private Collection<Account> accounts = new ArrayList<Account>();
```

```
    public int getId() {
      return id;
    }

    public void setID(int id) {
      this.id = id;
    }

    public void addAccount(Account account) {
      account.setBank(this);
      accounts.add(account);
    }

    public Collection<Account> getAccounts() {
      return accounts;
    }

    public void setAccounts(Collection<Account> accounts) {
      this.accounts = accounts;
    }
  }
```

Este código es mucho más limpio que el código EJB2 original. Se conservan algunos detalles de entidades, en las anotaciones. Sin embargo, como no hay información fuera de las anotaciones, el código es limpio y fácil de probar, mantener y demás.

Parte de la información de persistencia de las anotaciones se puede cambiar a descriptores de implementación XML si es necesario, dejando un POJO puro. Si los detalles de asignación de persistencia no cambian con frecuencia, muchos equipos pueden optar por mantener las anotaciones pero con menos obstáculos que si usaran EJB2.

Aspectos de AspectJ

Por último, la herramienta más completa de separación a través de aspectos es el lenguaje AspectJ[66], una extensión de Java que ofrece compatibilidad de primer nivel para aspectos como construcciones de modularidad. Los enfoques puros de Java proporcionados por Spring AOP y JBoss AOP son suficientes en el 80-90 por 100 de los casos en los que los aspectos son útiles. Sin embargo, AspectJ ofrece un conjunto de herramientas avanzadas y completas para la separación de aspectos. El inconveniente de AspectJ es la necesidad de adoptar nuevas herramientas y aprender nuevas construcciones

del lenguaje. Los problemas de adopción se han mitigado parcialmente gracias a la introducción de un formato de anotación de AspectJ, en el que se usan anotaciones de Java 5 para definir aspectos con código puro de Java. Además, la estructura Spring dispone de funciones que facilitan la incorporación de aspectos basados en anotaciones en un equipo con experiencia limitada con AspectJ.

El análisis completo de AspectJ supera los objetivos de este libro. Si necesita más información al respecto, consulte [AspectJ], [Colyer] y [Spring].

Pruebas de unidad de la arquitectura del sistema

La separación a través de enfoques similares a aspectos no se puede menospreciar. Si puede crear la lógica de dominios de su aplicación mediante POJO, sin conexión con los aspectos arquitectónicos a nivel del código, entonces se podrá probar realmente la arquitectura. Puede evolucionar de simple a sofisticado, de acuerdo a las necesidades, adoptando nuevas tecnologías bajo demanda. No es necesario realizar un *Buen diseño por adelantado* (Big Design Up Front[67], BDUF). De hecho, BDUF puede ser negativo ya que impide la adaptación al cambio, debido a la resistencia fisiológica a descartar esfuerzos previos y a la forma en que las decisiones arquitectónicas influyen en la concepción posterior del diseño.

Los arquitectos deben realizar BDUF ya que no resulta factible aplicar cambios arquitectónicos radicales a una estructura física una vez avanzada la construcción[68]. Aunque el *software* se rige por una física propia[69], es económicamente factible realizar cambios radicales si la estructura del *software* separa sus aspectos de forma eficaz.

Esto significa que podemos iniciar un proyecto de *software* con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura. Algunos de los principales sitios Web del mundo han alcanzado una gran disponibilidad y rendimiento por medio de sofisticadas técnicas de almacenamiento en caché, seguridad, virtualización y demás, todo ello de forma eficaz y flexible ya que los diseños mínimamente conectados son adecuadamente simples en cada

nivel de abstracción y ámbito. Evidentemente, no quiere decir que acometamos los proyectos sin timón. Debemos tener expectativas del ámbito general, objetivos y un programa, así como la estructura general del sistema resultante. Sin embargo, debemos mantener la capacidad de cambiar de rumbo en respuesta a las circunstancias.

La arquitectura EJB inicial es una de las API conocidas con un exceso de ingeniería y que compromete la separación de aspectos. Incluso las API bien diseñadas pueden ser excesivas cuando no resultan necesarias. Una API correcta debe desaparecer de la vista en la mayoría de los casos, para que el equipo dedique sus esfuerzos creativos a las historias implementadas. En caso contrario, las limitaciones arquitectónicas impedirán la entrega eficaz de un valor óptimo para el cliente. Para recapitular:

> *Una arquitectura de sistema óptima se compone de dominios de aspectos modularizados, cada uno implementado con POJO. Los distintos dominios se integran mediante aspectos o herramientas similares mínimamente invasivas. Al igual que en el código, en esta arquitectura se pueden realizar pruebas.*

Optimizar la toma de decisiones

La modularidad y separación de aspectos permite la descentralización de la administración y la toma de decisiones. En un sistema suficientemente amplio, ya sea una ciudad o un proyecto de *software*, no debe haber una sola persona que adopte todas las decisiones.

Sabemos que conviene delegar las responsabilidades en las personas más cualificadas. Solemos olvidar que también conviene posponer decisiones hasta el último momento. No es falta de responsabilidad; nos permite tomar decisiones con la mejor información posible. Una decisión prematura siempre es subjetiva. Si decidimos demasiado pronto, tendremos menos información del cliente, reflexión mental sobre el proyecto y experiencia con las opciones de implementación.

La agilidad que proporciona un sistema POJO con aspectos modularizados nos permite adoptar decisiones óptimas a tiempo, basadas en los conocimientos más recientes. Además, se reduce la complejidad de estas decisiones.

Usar estándares cuando añadan un valor demostrable

La construcción de edificios es una maravilla para la vista debido al ritmo empleado (incluso en invierno) y los extraordinarios diseños posibles gracias a la tecnología actual. La construcción es un sector maduro con elementos, métodos y estándares optimizados que han evolucionado bajo presión durante siglos.

Muchos equipos usaron la arquitectura EJB2 por ser un estándar, aunque hubiera bastado con diseños más ligeros y sencillos. He visto equipos obsesionados con estándares de moda y que se olvidaron de implementar el valor para sus clientes.

Los estándares facilitan la reutilización de ideas y componentes, reclutan individuos con experiencia, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares puede tardar demasiado para el sector, y algunos pierden el contacto con las verdaderas necesidades de aquello para los que están dirigidos.

Los sistemas necesitan lenguajes específicos del dominio

La construcción de edificios, como muchos dominios, ha desarrollado un rico lenguaje con vocabularios, frases y patrones[70] que comunican información esencial de forma clara y concisa. En el mundo del *software*, ha renacido el interés por crear Lenguajes específicos del dominio (*Domain-Specific Languages* o DSL)[71], pequeños lenguajes independientes de creación de secuencias de comandos o API de lenguajes estándar que permiten crear

código que se lea de una forma estructurada, como lo escribiría un experto del dominio. Un buen DSL minimiza el vacío de comunicación entre un concepto de dominio y el código que lo implementa, al igual que las prácticas ágiles optimizan la comunicación entre un equipo y los accionistas del proyecto. Si tiene que implementar la lógica de dominios en el mismo lenguaje usado por un experto del dominio, hay menos riesgo de traducir incorrectamente el dominio en la implementación.

Los DSL, si se usan de forma eficaz, aumentan el nivel de abstracción por encima del código y los patrones de diseño. Permiten al desarrollador revelar la intención del código en el nivel de abstracción adecuado.

> *Los lenguajes específicos del dominio permiten expresar como POJO todos los niveles de abstracción y todos los dominios de la aplicación, desde directivas de nivel superior a los detalles más mínimos.*

Conclusión

Los sistemas también deben ser limpios. Una arquitectura invasiva afecta a la lógica de dominios y a la agilidad. Si la lógica de dominios se ve afectada, la calidad se resiente, ya que los errores se ocultan y las historias son más difíciles de implementar. Si la agilidad se ve comprometida, la productividad sufre y las ventajas de TDD se pierden.

En todos los niveles de abstracción, los objetivos deben ser claros. Esto sólo sucede si crea POJO y usa mecanismos similares a aspectos para incorporar otros aspectos de implementación de forma no invasiva.

Independientemente de que diseñe sistemas o módulos individuales, no olvide usar los elementos más sencillos que funcionen.

Bibliografía

- **[Alexander]**: Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York, 1979.

- **[AOSD]**: Puerto de Desarrollo de *software* orientado a aspectos, http://aosd.net.
- **[ASM]**: Página de ASM, http://asm.objectweb.org/.
- **[AspectJ]**: http: //eclipse.org/aspectj.
- **[CGLIB]**: Biblioteca de generación de código, http://cglib.sourceforge.net/.
- **[Colyer]**: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ*, Person Education, Inc., Upper Saddle River, NJ, 2005.
- **[DSL]**: Lenguaje de programación específico del dominio, http://es.wikipedia.org/wiki/Lenguaje_espec%C3%ADfico_del_dominio
- **[Fowler]**: Inversión de contenedores de control y el patrón de inyección de dependencias (http://martinfowler.com/articles/injection.html).
- **[Goetz]**: Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxies*, http://www.ibm.com/developerworks/java/library/j-jtp08305.html.
- **[Javassist]**: Página de Javassist, http://www.csg.is.titech.ac.jp/chiba/javassist/.
- **[JBoss]**: Página de JBoss, http: //jboss.org.
- **[JMock]**: JMock: Una biblioteca de objetos Mock ligeros para Java, http://jmock.org.
- **[Kolence]**: Kenneth W. Kolence, *Software physics and computer performance measurements*, *Proceedings of the ACM annual conference-Volume* 2, Boston, Massachusetts, pp. 1024-1040,1972.
- **[Spring]**: *The Spring Framework*, http://www.springframework.org.
- **[Mezzaros07]**: *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

12
Emergencia

con Jeff Langr

Limpieza a través de diseños emergentes

Imagine que existieran cuatro sencillas reglas para crear diseños de calidad. Imagine que siguiéndolas accediera a la estructura y al diseño de su código y facilitara la aplicación de principios como SRP y DIP. Imagine que estas cuatro reglas facilitaran la emergencia de diseños de calidad.

Muchos consideramos que las cuatro reglas de Kent Beck de diseño sencillo[72] son fundamentales para crear un *software* bien diseñado.

Según Kent, un diseño es sencillo si cumple estas cuatro reglas:

- Ejecuta todas las pruebas.
- No contiene duplicados.
- Expresa la intención del programador.
- Minimiza el número de clases y métodos.

Describiremos estas reglas en orden de importancia.

Primera regla del diseño sencillo: Ejecutar todas las pruebas

En primer lugar, un diseño debe generar un sistema que actúe de la forma prevista. Un sistema puede tener un diseño perfecto sobre el papel pero si no existe una forma sencilla de comprobar que realmente funciona de la forma esperada, el esfuerzo sobre el papel es cuestionable.

Un sistema minuciosamente probado y que supera todas las pruebas en todo momento se denomina sistema testable. Es una afirmación obvia, pero importante. Los sistemas que no se pueden probar no se pueden verificar, y un sistema que no se puede verificar no debe implementarse.

Afortunadamente, crear sistemas testables hace que diseñemos clases de tamaño reducido y un solo cometido. Resulta más sencillo probar clases que cumplen el SRP. Cuantas más pruebas diseñemos, más nos acercaremos a elementos más fáciles de probar. Por lo tanto, hacer que nuestro sistema se pueda probar nos ayuda a crear mejores diseños.

Las conexiones rígidas dificultan la creación de pruebas. Del mismo modo, cuantas más pruebas creemos, más usaremos principios como DIP y

herramientas con inyección de dependencias, interfaces y abstracción para minimizar dichas conexiones. Nuestros diseños mejorarán todavía más.

En especial, seguir una sencilla regla que afirme que debemos realizar pruebas y ejecutarlas continuamente afecta el cumplimiento por parte de nuestro sistema de los principales objetivos de la programación orientada a objetos de baja conexión y elevada cohesión. La creación de pruebas conduce a obtener mejores diseños.

Reglas 2 a 4 del diseño sencillo: Refactorizar

Una vez creadas las pruebas, debemos mantener limpio el código y las clases. Para ello, refactorizamos el código progresivamente. Tras añadir unas líneas, nos detenemos y reflejamos el nuevo diseño. ¿Ha empeorado? En caso afirmativo, lo limpiamos y ejecutamos las pruebas para comprobar que no hay elementos afectados. *La presencia de las pruebas hace que perdamos el miedo a limpiar el código y que resulte dañado*.

En la fase de refactorización, podemos aplicar todos los aspectos del diseño de *software* correcto. Podemos aumentar la cohesión, reducir las conexiones, separar las preocupaciones, modularizar aspectos del sistema, reducir el tamaño de funciones y clases, elegir nombres más adecuados, etc. Aquí también aplicamos las tres últimas reglas del diseño correcto: eliminar duplicados, garantizar la capacidad de expresión y minimizar el número de clases y métodos.

Eliminar duplicados

Los duplicados son los mayores enemigos de un sistema bien diseñado. Suponen un esfuerzo adicional, riesgos añadidos y una complejidad a mayores innecesaria. Los duplicados se manifiestan de diversas formas. Las líneas de código similar pueden modificarse para que parezcan refactorizadas, y hay otras formas de duplicación como la de implementación. Por ejemplo, podríamos tener dos métodos en una clase de colección:

```
int size() {}
boolean isEmpty() {}
```

Podríamos tener implementaciones separadas para cada método. El método `isEmpty` podría controlar un valor booleano y `size` un contador, o podemos eliminar la duplicación y vincular `isEmpty` a la definición de `size`:

```
boolean isEmpty() {
   return 0 == size();
}
```

La creación de un sistema limpio requiere la eliminación de duplicados, aunque sean unas cuantas líneas de código. Fíjese en el siguiente ejemplo:

```
public void scaleToOneDimension {
     float desiredDimension, float imageDimension) {
   if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
     return;
   float scalingFactor = desiredDimension / imageDimension;
   scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

   RenderedOp newImage = ImageUtilities.getScaledImage(
     image, scalingFactor, scalingFactor);
   image.dispose();
   System.gc();
   image = newImage;
}
public synchronized void rotate(int degrees) {
   RenderedOp newImage = ImageUtilities.getRotatedImage(
     image, degrees);
   image.dispose();
   System.gc();
   image = newImage;
}
```

Para mantener limpio este sistema, debemos eliminar la pequeña cantidad de duplicación entre los métodos `scaleToOneDimension` y `rotate`:

```
public void scaleToOneDimension (
     float desiredDimension, float imageDimension) {
   if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
     return;
   float scalingFactor = desiredDimension / imageDimension;
   scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
   replaceImage(ImageUtilities.getScaledImage(
     image, scalingFactor, scalingFactor));
}

public synchronized void rotate (int degrees) {
   replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}

private void replaceImage(RenderedOp newImage) (
```

```
    image.dispose();
    System.gc();
    image = newImage;
  }
```

Al extraer a este reducido nivel, comenzamos a detectar incumplimientos de SRP. Por ello, podríamos cambiar un nuevo método extraído a otra clase. Esto aumenta su visibilidad. Otro miembro del equipo puede ver la necesidad de volver a extraer el nuevo método y usarlo en otro contexto diferente. Esta reutilización mínima puede reducir considerablemente la complejidad del sistema. Saber cómo lograrlo es fundamental para alcanzar la reutilización a gran escala.

El patrón *Método de plantilla*[73] es una técnica muy utilizada para eliminar duplicados de nivel superior. Por ejemplo:

```
public class VacationPolicy {
   public void accrueUSDivisionVacation() {
     // código para calcular las vacaciones en función de las horas trabajadas
     //…
     // código para garantizar que las vacaciones cumplen los mínimos legales
     //…
     // código para aplicar vacation al registro payroll
     //…
   }

   public void accrueEUDivisionVacation() {
     // código para calcular las vacaciones en función de las horas trabajadas
     //…
     // código para garantizar que las vacaciones cumplen los mínimos legales
     //…
     // código para aplicar vacation al registro payroll
     //…
   }
}
```

El código entre `accrueUSDivisionVacation` y `accrueEuropeanDivisionVacation` es prácticamente idéntico, a excepción del cálculo de mínimos legales. Esa parte del algoritmo cambia en función del tipo de empleado. Podemos eliminar la duplicación evidente si aplicamos el patrón de *Método de plantilla*:

```
abstract public class VacationPolicy {
   public void accrueVacation() {
     calculateBaseVacationHours();
     alterForLegalMinimums();
     applyToPayroll();
   }
```

```
    private void calculateBaseVacationHours() { /* … */ };
    abstract protected void alterForLegalMinimums();
    private void applyToPayroll(); { /* … */ };
  }

  public class USVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
      // Lógica específica de EE.UU.
    }
  }

  public class EUVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
      // Lógica específica de la UE.
    }
  }
```

Las subclases ocupan el vacío generado en el algoritmo `accrueVacation` y solamente proporcionan los datos que no están duplicados.

Expresividad

Muchos tenemos experiencia con código enrevesado. Muchos lo hemos creado. Es fácil crear código que entendamos, ya que durante su creación nos centramos en comprender el problema que intentamos resolver. Los encargados de mantener el código no lo comprenderán de la misma forma.

El principal coste de un proyecto de *software* es su mantenimiento a largo plazo. Para minimizar los posibles defectos al realizar cambios, es fundamental que comprendamos el funcionamiento del sistema. Al aumentar la complejidad de los sistemas, el programador necesita más tiempo para entenderlo y aumentan las posibilidades de errores. Por tanto, el código debe expresar con claridad la intención de su autor. Cuando más claro sea el código, menos tiempo perderán otros en intentar comprenderlo. Esto reduce los defectos y el coste de mantenimiento.

Puede expresarse si elige nombres adecuados. El objetivo es ver el nombre de una clase y función, y que sus responsabilidades no nos sorprendan.

También puede expresarse si reduce el tamaño de funciones y clases. Al hacerlo, resulta más sencillo asignarles nombres, crearlas y comprenderlas.

Otra forma de expresarse es usar una nomenclatura estándar. Los patrones de diseño, por ejemplo, se basan en la comunicación y en la capacidad de expresión. Al usar los nombres de patrones estándar, como `COMMAND` o `VISITOR`, en los nombres de las clases que implementan dichos patrones puede describir sucintamente su diseño a otros programadores.

Las pruebas de unidad bien escritas también son expresivas. Uno de los principales objetivos de una prueba es servir de documentación mediante ejemplos. Los que lean las pruebas deben entender con facilidad para qué sirve una clase.

Pero la forma más importante de ser expresivo es la práctica. A menudo, conseguimos que el código funcione y pasamos al siguiente problema sin detenernos en facilitar la lectura del código para otros. No olvide que seguramente sea el próximo que lea el código.

Por tanto, afronte su creación con orgullo. Dedique tiempo a sus funciones y clases. Seleccione nombres mejores, divida las funciones extensas en otras más reducidas y cuide su obra. El cuidado es un recurso precioso.

Clases y métodos mínimos

Incluso conceptos tan básicos como la eliminación de código duplicado, la expresividad del código y SRP pueden exagerarse. En un esfuerzo por reducir el tamaño de clases y métodos, podemos crear demasiadas clases y métodos reducidos. Esta regla también sugiere minimizar la cantidad de funciones y clases.

Una gran cantidad de clases y métodos suele indicar un dogmatismo sin sentido. Imagine un estándar de código que insista en la creación de una interfaz para todas las clases, o a programadores que insisten en qué campos y comportamientos siempre deben separarse en clases de datos y clases de comportamiento. Este dogma debe evitarse y cambiarse por un enfoque más pragmático.

Nuestro objetivo es reducir el tamaño general del sistema además del tamaño de clases y funciones, pero recuerde que esta regla es la de menor

prioridad de las cuatro. Por ello, aunque sea importante reducir la cantidad de clases y funciones, es más importante contar con pruebas, eliminar duplicados y expresarse correctamente.

Conclusión

¿Existen prácticas sencillas que puedan reemplazar a la experiencia? Por supuesto que no. Sin embargo, las prácticas descritas en este capítulo y en el libro son una forma cristalizada de décadas de experiencia de muchos autores. La práctica del diseño correcto anima y permite a los programadores adoptar principios y patrones que en caso contrario tardarían años en aprender.

Bibliografía

- **[XPE]**: *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison Wesley, 1999.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

13
Concurrencia

por Brett L. Schuchert

“Los objetos son abstracciones de procesamiento.
Los subprocesos son abstracciones de programaciones”.
—James O. Coplien[74]

La creación de programas concurrentes limpios es complicada, muy complicada. Es mucho más sencillo crear código que se ejecute en un mismo proceso. También es fácil crear código de subprocesamiento múltiple que parezca correcto en la superficie pero que esté dañado a niveles más profundos. Este código funciona correctamente hasta que el sistema se somete a determinadas presiones.

En este capítulo analizaremos la necesidad de la programación concurrente y sus dificultades. Tras ello, presentaremos diversas recomendaciones para superar dichas dificultades y crear código concurrente limpio. Por último, finalizaremos con los problemas relacionados con la prueba de código concurrente.

La concurrencia limpia es un tema complejo, merecedor de un libro propio. Aquí, intentaremos ofrecer una visión general, que después ampliaremos en el apéndice A. Si simplemente tiene curiosidad por el tema, le bastará con este capítulo. Si necesita entender la concurrencia a un nivel más profundo, consulte también el apéndice.

¿Por qué concurrencia?

La concurrencia es una estrategia de desvinculación. Nos permite desvincular lo que se hace de dónde se hace. En aplicación de un solo proceso, el qué y el cuándo están tan firmemente vinculados que el estado de la aplicación se puede determinar analizando la huella de la pila. Un programador que depure este tipo de sistemas puede definir un punto de interrupción (o varios) y saber el estado de la aplicación en función del punto al que se llegue.

La desvinculación del qué del dónde puede mejorar considerablemente el rendimiento y la estructura de una aplicación. Desde un punto de vista estructural, la aplicación parece una serie de equipos colaboradores y no un gran bucle principal. Esto puede hacer que el sistema sea más fácil de

comprender y ofrece diversas formas de separar las preocupaciones. Pongamos por caso el modelo Servlet estándar de aplicaciones Web. Estos sistemas se ejecutan bajo un contenedor Web o EJB que gestiona parcialmente la concurrencia. Los servlet se ejecutan de forma asíncrona cuando se reciben solicitudes Web. El programador de los servlet no tiene que gestionar todas las solicitudes entrantes. En principio, la ejecución de cada servlet vive en un mundo propio y se desvincula del resto.

Evidentemente, si fuera tan sencillo, no necesitaríamos este capítulo. De hecho, la desvinculación proporcionada por los contenedores Web dista mucho de ser perfecta. Los programadores de servlet deben asegurarse de que sus programas sean correctos. No obstante, las ventajas estructurales del modelo de servlet son significativas.

Pero la estructura no es el único motivo para adoptar la concurrencia. Algunos sistemas tienen limitaciones de tiempo de respuesta y producción que requieren soluciones concurrentes manuales. Imagine un dispositivo para añadir información, con un solo proceso, que obtiene datos de distintos sitios Web y los combina en un resumen diario. Al tener un solo proceso, accede por turnos a cada sitio Web y siempre termina uno antes de comenzar el siguiente. Su recorrido diario debe ejecutarse en menos de 24 horas. Sin embargo, al añadir nuevos sitios Web, el tiempo aumenta hasta necesitarse más de 24 horas para recopilar todos los datos. El único proceso implica una prolongada espera para completar la E/S. Podríamos mejorar el rendimiento con ayuda de un algoritmo de subprocesamiento múltiple que visite más de un sitio Web por vez.

Imagine un sistema que procesa un usuario por vez y sólo requiere un segundo por cada uno. Su capacidad de respuesta es válida para un número reducido de usuarios pero si aumenta, también lo hace el tiempo de respuesta del sistema. Ningún usuario querrá esperar a otros 150. Podríamos mejorar el tiempo de respuesta de este sistema procesando varios usuarios a la vez. Imagine un sistema que interprete grandes conjuntos de datos pero que sólo ofrezca una solución completa tras procesarlos todos. Se podría procesar cada conjunto de datos en un equipo distinto, para poder procesarlos todos en paralelo.

Mitos e imprecisiones

También existen motivos evidentes para adoptar la concurrencia aunque, como indicamos antes, sea complicada. Si no presta la suficiente atención, pueden darse casos desagradables. Veamos los mitos e imprecisiones más habituales:

- *La concurrencia siempre mejora el rendimiento*: En ocasiones lo hace pero sólo cuando se puede compartir tiempo entre varios procesos o procesadores. Ninguna situación es trivial.
- *El diseño no cambia al crear programas concurrentes*: De hecho, el diseño de un algoritmo concurrente puede ser muy distinto al de un sistema de un solo proceso. La desvinculación entre el qué y el cuándo suele tener un efecto importante en la estructura del sistema.
- *No es importante entender los problemas de concurrencia al trabajar con un contenedor Web o EJB*: En realidad, debe saber lo que hace su contenedor y protegerlo de problemas de actualizaciones concurrentes y bloqueo, como veremos después.

Veamos otros aspectos relacionados con la creación de *software* concurrente:

- *La concurrencia genera cierta sobrecarga*, tanto en rendimiento como en la creación de código adicional.
- *La concurrencia correcta es compleja*, incluso para problemas sencillos.
- *Los errores de concurrencia no se suelen repetir*, de modo que se ignoran[75] en lugar de considerarse verdaderos problemas.
- *La concurrencia suele acarrear un cambio fundamental de la estrategia de diseño*.

Desafíos

¿Qué hace que la programación concurrente sea tan complicada? Fíjese en la

siguiente clase:

```
public class X {
   private int lastIdUsed;

   public int getNextId() {
     return ++lastIdUsed;
   }
}
```

Imagine que creamos una instancia `X`, establecemos el campo `lastIdUsed` en 42 y después compartimos la instancia entre dos procesos. Imagine ahora que esos dos procesos invocan el método `getNextId()`; hay tres resultados posibles:

- El primer proceso obtiene el valor 43, el segundo el valor 44 y `lastIdUsed` es 44.
- El primer proceso obtiene el valor 44, el segundo el valor 43 y `lastIdUsed` es 44.
- El primer proceso obtiene el valor 43, el segundo el valor 43 y `lastIdUsed` es 43.

El sorprendente tercer resultado[76] se produce cuando los dos procesos coinciden. Se debe a que pueden adoptar varias rutas posibles en una línea de código de Java y algunas generan resultados incorrectos. ¿Cuántas rutas distintas existen? Para responder, debemos entender lo que hace el compilador justo a tiempo con el código de *bytes* generado, y lo que el modelo de memoria de Java considera atómico.

Una rápida respuesta, con el código de *bytes* generado, es que existen 12 870 rutas de ejecución diferentes[77] para los dos procesos ejecutados en el método `getNextId`. Si el tipo de `lastIdUsed` cambia de `int` a `long`, el número de rutas asciende a 2 704 156. Evidentemente, muchas generan resultados válidos. El problema es que *algunas no lo hacen*.

Principios de defensa de la concurrencia

A continuación le mostramos una serie de principios y técnicas para proteger

a sus sistemas de los problemas del código concurrente.

Principio de responsabilidad única (SRP)

SRP[78] establece que un método, clase o componente sólo debe tener un motivo para cambiar. El diseño de concurrencia es lo bastante complejo como para ser un motivo de cambio con derecho propio y, por tanto, debe separarse del resto del código. Desafortunadamente, es habitual incrustar los detalles de la implementación de concurrencia directamente en otro código de producción.

Tenga en cuenta los siguientes aspectos:

- *El código relacionado con la concurrencia tiene su propio ciclo de desarrollo*, cambios y ajustes.
- *El código relacionado con la concurrencia tiene sus propios desafíos*, diferentes y más complicados, que los del código no relacionado con la concurrencia.
- El número de formas en las que el código incorrecto basado en la concurrencia puede fallar lo complica ya de por sí, sin la carga añadida del código de aplicación circundante.

Recomendación: *Separe el código de concurrencia del resto del código*[79].

Corolario: Limitar el ámbito de los datos

Como hemos visto, dos procesos que modifican el mismo campo u objeto compartido pueden interferir entre ellos y provocar un comportamiento inesperado. Una solución consiste en usar la palabra clave `synchronized` para proteger una sección importante del código que use el objeto compartido, aunque conviene limitar la cantidad de estas secciones. Cuantos más puntos actualicen datos compartidos, es más probable que:

- Se olvide de proteger uno o varios de esos puntos, y se dañe el código que modifica los datos compartidos.
- Se duplique el esfuerzo necesario para garantizar la protección de todos los elementos (incumplimiento de DRY[80]).
- Resulta complicado determinar el origen de los fallos, que por naturaleza son difíciles de detectar.

Recomendación: *Encapsule los datos y limite el acceso a los datos compartidos*.

Corolario: Usar copias de datos

Una forma de evitar datos compartidos es no compartirlos. En algunos casos se pueden copiar objetos y procesarlos como si fueran de sólo lectura. En otros, se pueden copiar objetos, recopilar los resultados de varios procesos en las copias y después combinar los resultados en un mismo proceso. Si existe una forma sencilla de evitar los objetos compartidos, el código resultante tendrá menos problemas. Puede que le preocupe el coste de la creación de objetos adicionales. Merece la pena experimentar y comprobar si es un problema real. No obstante, si el uso de copias de objetos permite al código evitar la sincronización, las ventajas de evitar el bloque compensan la creación adicional y la sobrecarga de la recolección de elementos sin usar.

Corolario: Los procesos deben ser independientes

Pruebe a crear el código de sus procesos de forma que cada uno sea independiente y no comparta datos con otros. Cada uno procesa una solicitud cliente y todos los datos necesarios provienen de un origen sin compartir y se almacenan como variables locales. De este modo, los procesos se comportan como si fueran los únicos del mundo y no existieran requisitos de sincronización. Por ejemplo, las subclases de `HttpServlet` reciben toda su información como parámetros pasados en los métodos `doGet` y `doPost`. Esto hace que cada servlet actúe como si dispusiera de su propio equipo. Mientras

el código del servlet sólo use variables locales, es imposible que cause problemas de sincronización. Evidentemente, muchas aplicaciones que usan `servlet` se topan con recursos compartidos como conexiones de base de datos.

Recomendación: *Intente dividir los datos en subconjuntos independientes que se puedan procesar en procesos independientes, posiblemente en distintos procesadores*.

Conocer las bibliotecas

Java 5 ofrece muchas mejoras para el desarrollo concurrente con respecto a versiones anteriores. Existen diversos aspectos que tener en cuenta a la hora de crear código de procesos en Java 5:

- Usar las colecciones compatibles con procesos proporcionadas.
- Usar la estructura de ejecución de tareas no relacionadas.
- Usar soluciones antibloqueo siempre que sea posible.
- Varias clases de bibliotecas no son compatibles con procesos.

Colecciones compatibles con procesos

En los albores de Java, Doug Lea escribió el conocido libro[81] *Concurrent Programming in Java*. Al mismo tiempo, desarrolló varias colecciones compatibles con procesos, que posteriormente pasaron a formar parte del JDK en el paquete `java.util.concurrent`. Las colecciones de dicho paquete son compatibles con casos de procesos múltiples y tienen un rendimiento adecuado. De hecho, la implementación `ConcurrentHashMap` tiene mejor rendimiento que `HashMap` en la mayoría de los casos. También permite lecturas y escrituras simultáneas, y dispone de métodos que admiten operaciones de composición habituales que en caso contrario serian incompatibles con subprocesos. Si Java 5 es su entorno de desarrollo, comience con `ConcurrentHashMap`.

Existen otras clases añadidas para admitir diseño avanzado de concurrencia. Veamos algunos ejemplos:

`ReentrantLock`	Bloqueo que se puede adquirir en un método y liberar en otro.
`semaphore`	Una implementación del clásico semáforo, un bloqueo con un contador.
`CountDownLatch`	Bloqueo que espera un número de eventos antes de liberar todos los subprocesos retenidos. De este modo todos tienen la misma oportunidad de iniciarse al mismo tiempo.

Recomendación: *Revise las clases de las que disponga. En el caso de Java, debe familiarizarse con java.util.concurrent, java.util.concurrent.atomic y java.util.concurrent.locks.*

Conocer los modelos de ejecución

Existen diversas formas de dividir el comportamiento de una aplicación concurrente. Para describirlos debe conocer ciertas definiciones básicas.

`Recursos vinculados`	Recursos de tamaño o número fijo usados en un entorno concurrente, como por ejemplo conexiones de base de datos y búfer de lectura/escritura de tamaño fijo.
`Exclusión mutua`	Sólo un proceso puede acceder a datos o a un recurso compartido por vez.
`Inanición`	Se impide que un proceso o grupo de procesos continúen demasiado tiempo o indefinidamente. Por ejemplo, si permite primero la ejecución de los procesos más rápidos, los que se ejecutan durante más tiempo pueden perecer de inanición si los primeros no terminan nunca.
`Bloqueo`	Dos o más procesos esperan a que ambos terminen. Cada proceso tiene un recurso y ninguno puede terminar hasta que obtenga el otro recurso.
`Bloqueo`	Procesos bloqueados, intentando realizar su labor pero

`activo`	estorbándose unos a otros. Por motivos de resonancia, los procesos siguen intentando avanzar pero no pueden durante demasiado tiempo, o de forma indefinida.

Tras mostrar estas definiciones, ya podemos describir los distintos modelos de ejecución empleados en la programación concurrente.

Productor-Consumidor[82]

Uno o varios procesos productores crean trabajo y lo añaden a un búfer o a una cola. Uno o varios procesos consumidores adquieren dicho trabajo de la cola y lo completan. La cola entre productores y consumidores es un recurso vinculado, lo que significa que los productores deben esperar a que se libere espacio en la cola antes de escribir y los consumidores deben esperar hasta que haya algo que consumir en la cola. La coordinación entre productores y consumidores a través de la cola hace que unos emitan señales a otros. Los productores escriben en la cola e indican que ya no está vacía. Los consumidores leen de la cola e indican que ya no está llena. Ambos esperan la notificación para poder continuar.

Lectores-Escritores[83]

Cuando un recurso compartido actúa básicamente como fuente de información para lectores pero ocasionalmente se actualiza por parte de escritores, la producción es un problema. El énfasis de la producción puede provocar la inanición y la acumulación de información caducada. Las actualizaciones pueden afectar a la producción. La coordinación de lectores para que no lean algo que un escritor está actualizando y viceversa es complicada. Los escritores tienden a bloquear a los lectores durante periodos prolongados, lo que genera problemas de producción.

El desafío consiste en equilibrar las necesidades de ambos para satisfacer un funcionamiento correcto, proporcionar una producción razonable y evitar la inanición. Una sencilla estrategia hace que los escritores esperen hasta que deje de haber lectores antes de realizar una actualización. Si hay lectores

continuos, los escritores perecen de inanición.

Por otra parte, si hay escritores frecuentes y se les asigna prioridad, la producción se ve afectada. Determinar el equilibrio y evitar problemas de actualización concurrente es el objetivo de este modelo.

La cena de los filósofos[84]

Imagine varios filósofos sentados alrededor de una mesa redonda. A la izquierda de cada uno hay un tenedor. En el centro de la mesa, una gran fuente de espaguetis. Los filósofos pasan el tiempo pensando a menos que tengan hambre. Cuando tienen hambre, utilizan los tenedores situados a ambos lados para comer. No pueden comer a menos que tengan dos tenedores. Si el filósofo situado a la derecha o izquierda de otros ya tiene uno de los tenedores que necesita, tendrá que esperar a que termine de comer y deje los tenedores. Cuando un filósofo termina de comer, vuelve a colocar los tenedores en la mesa hasta que vuelve a tener hambre. Cambie los filósofos por procesos y los tenedores por recursos y tendrá un problema habitual en muchas aplicaciones en las que los procesos compiten por recursos. A menos que se diseñen correctamente, los sistemas que compiten de esta forma experimentan problemas de bloqueo, bloqueo mutuo, producción y degradación de la eficacia. La mayoría de problemas de concurrencia que encontrará serán alguna variante de éstos. Analice los algoritmos y cree soluciones propias para estar preparado cuando surjan problemas de concurrencia.

Recomendación: *Aprenda estos algoritmos básicos y comprenda sus soluciones*.

Dependencias entre métodos sincronizados

Las dependencias entre métodos sincronizados generan sutiles errores en el código concurrente. Java cuenta con `synchronized`, que protege métodos individuales. No obstante, si hay más de un método sincronizado en la misma clase compartida, puede que su sistema sea incorrecto[85].

Recomendación: *Evite usar más de un método en un objeto compartido*.

En ocasiones tendrá que usar más de un método en un objeto compartido. En ese caso, hay tres formas de crear código correcto:

- **Bloqueo basado en clientes**: El cliente debe bloquear al servidor antes de invocar el primer método y asegurarse de que el alcance del bloque incluye el código que invoque el último método.
- **Bloqueo basado en servidores**: Debe crear un método en el servidor que bloquee el servidor, invoque todos los métodos y después anule el bloqueo. El cliente debe invocar el nuevo método.
- **Servidor adaptado**: Cree un intermediario que realice el bloque. Es un ejemplo de bloqueo basado en servidores en el que el servidor original no se puede modificar.

Reducir el tamaño de las secciones sincronizadas

La palabra clave `synchronized` presenta un bloqueo. Todas las secciones de código protegidas por el mismo bloque sólo tendrán un proceso que las ejecute en un momento dado. Los bloqueos son costosos ya que generan retrasos y añaden sobrecarga. Por ello, no conviene colapsar el código con instrucciones `synchronized`. Por otra parte, las secciones críticas[86] deben protegerse, de modo que debemos diseñar nuestro código con el menor número posible de secciones críticas.

Algunos programadores intentan lograrlo ampliando el tamaño de sus secciones críticas. Sin embargo, al ampliar la sincronización más allá de la sección crítica mínima aumentan los problemas y afecta negativamente al rendimiento[87].

Recomendación: *Reduzca al máximo el tamaño de las secciones* `synchronized`.

Crear código de cierre correcto es complicado

Crear un sistema activo y que se ejecute indefinidamente es distinto a crear algo que funcione de forma temporal y después se cierre correctamente. Entre los problemas más habituales destacan los bloqueos[88], con procesos que esperan una señal para continuar que nunca se produce.

Imagine, por ejemplo, un sistema con un proceso principal que genera varios procesos secundarios y que espera a que todos terminen antes de liberar sus recursos y cerrarse. ¿Qué sucede si uno de los procesos secundarios está bloqueado? El principal esperará indefinidamente y el sistema nunca se cerrará.

Imagine ahora un sistema similar al que se le indica que se cierre. El proceso principal indica a todos los secundarios que abandonen sus tareas y terminen. Pero imagine que dos procesos secundarios funcionan como par productor/consumidor y que el productor recibe una señal del principal y se cierra rápidamente. El consumidor espera un mensaje del productor y puede quedar bloqueado en un estado en el que no recibe la señal del principal, lo que también impide que éste finalice.

Son situaciones habituales. Por tanto, si tiene que crear código concurrente con cierres correctos, tendrá que dedicar tiempo a que el cierre se produzca de forma correcta.

Recomendación: *Planifique con antelación el proceso de cierre y pruébelo hasta que funcione. Le llevará más tiempo del que espera. Repase los algoritmos existentes porque será complicado.*

Probar código con procesos

Demostrar que el código es correcto no resulta práctico. Las pruebas no garantizan su corrección. Sin embargo, las pruebas adecuadas pueden minimizar los riesgos, en especial en aplicaciones de un solo proceso. Cuando hay dos o más procesos que usan el mismo código y trabajan con datos compartidos, la situación se vuelve más compleja.

Recomendación: *Cree pruebas que puedan detectar problemas y ejecútelas periódicamente, con distintas configuraciones de programación y del sistema, y cargas. Si las pruebas fallan, identifique el fallo. No lo ignore*

porque las pruebas superen una ejecución posterior.

Hay muchos factores que tener en cuenta. Veamos algunas recomendaciones concretas:

- Considere los fallos como posibles problemas de los procesos.
- Consiga que primero funcione el código sin procesos.
- El código con procesos se debe poder conectar a otros elementos.
- El código con procesos debe ser modificable.
- Ejecute con más procesos que procesadores.
- Ejecute en diferentes plataformas.
- Diseñe el código para probar y forzar fallos.

Considerar los fallos como posibles problemas de los procesos

El código con procesos hace que fallen elementos que no deberían fallar. Muchos desarrolladores desconocen cómo interactúan los procesos con otro tipo de código. Los problemas del código con procesos pueden mostrar sus síntomas una vez cada mil o un millón de ejecuciones.

Los intentos por repetir los sistemas pueden resultar frustrantes, lo que suele provocar que los programadores consideren el fallo como algo aislado. Es recomendable asumir que los fallos aislados no existen. Cuanto más los ignore, mayor será la cantidad de código que se acumule sobre un enfoque defectuoso.

Recomendación: *No ignore los fallos del sistema como algo aislado*.

Conseguir que primero funcione el código sin procesos

Puede parecer evidente pero no está de más recordarlo. Asegúrese de que el código funciona fuera de sus procesos. Por lo general, esto significa crear algunos POJO que los procesos deban invocar. Los POJO no son compatibles con los procesos y por tanto se pueden probar fuera de su entorno. Conviene

incluir en los POJO la mayor cantidad posible del sistema.

Recomendación: *No intente identificar fallos de procesos y que no sean de procesos al mismo tiempo. Asegúrese de que su código funciona fuera de los procesos*.

El código con procesos se debe poder conectar a otros elementos

Cree el código compatible con la concurrencia de forma que se pueda ejecutar en distintas configuraciones:

- Un proceso, varios procesos y variarlo durante la ejecución.
- El código con procesos interactúa con algo que puede ser real o probado.
- Ejecutar con pruebas dobles ejecutadas de forma rápida, lenta y variable.
- Configurar pruebas que ejecutar en diferentes iteraciones.

Recomendación: *El código con procesos debe poder conectar a otros elementos y ejecutar en distintas configuraciones*.

El código con procesos debe ser modificable

La obtención del equilibrio adecuado de procesos suele requerir operaciones de ensayo y error. En las fases iniciales, compruebe el rendimiento del sistema bajo diferentes configuraciones. Permita que se puedan modificar los distintos procesos y también durante la ejecución del sistema. También puede permitir la modificación automática en función de la producción y la utilización del sistema.

Ejecutar con más procesos que procesadores

Cuando el sistema cambia de tarea, se producen reacciones. Para promover el intercambio de tareas, realice la ejecución con más procesos que procesadores o núcleos. Cuanto mayor sea la frecuencia de intercambio de las

tareas, más probabilidades existen de que el código carezca de una sección crítica o se produzcan bloqueos.

Ejecutar en diferentes plataformas

En 2007 diseñamos un curso sobre programación concurrente, principalmente en OS X. La clase se presentó con Windows XP ejecutado en una MV. Se crearon pruebas para ilustrar condiciones de fallo que fallaban con más frecuencia en OS X que en XP.

En todos los casos, el código probado era incorrecto. Esto refuerza el hecho de que cada sistema operativo tiene una política de procesos diferente que afecta a la ejecución del código. El código con procesos múltiples se comporta de forma distinta en cada entorno[89]. Debe ejecutar sus pruebas en todos los entornos de implementación posibles.

Recomendación: *Ejecute el código con procesos en todas las plataformas de destino con frecuencia y en las fases iniciales.*

Diseñar el código para probar y forzar fallos

Es habitual que los fallos del código concurrente se oculten. Las pruebas sencillas no suelen mostrarlos. En realidad, suelen ocultarse durante el procesamiento normal. Pueden aparecer horas, días o semanas después.

La razón de que los problemas de procesos sean infrecuentes, esporádicos y apenas se repitan es que sólo fallan algunas de las miles de rutas posibles que recorren una sección vulnerable. Por tanto, la probabilidad de adoptar una ruta fallida es realmente baja, lo que dificulta la detección y la depuración.

Se preguntará cómo aumentar las posibilidades de capturar estos casos. Puede diseñar el código y forzarle a que se ejecute en diferentes órdenes añadiendo métodos como `Object.wait()`, `Object.sleep()`, `Object.yield()` y `Object.priority()`.

Estos métodos afectan al orden de ejecución y, por tanto, aumentan las posibilidades de detectar un error. Resulta más adecuado que el código

incorrecto falle lo antes posible y con frecuencia. Hay dos opciones de instrumentación de código:

- Manual.
- Automática.

Manual

Puede añadir invocaciones de `wait()`, `sleep()`, `yield()` y `priority()` manualmente a su código, en especial si tiene que probar un fragmento especialmente escabroso. Veamos un ejemplo:

```
public synchronized String nextUrlOrNull() {
   if (hasNext()) {
     String url = urlGenerator.next();
     Thread.yield(); // se añade para pruebas.
     updateHasNext();
     return url;
   }
   return null;
}
```

La invocación de `yield()` cambia la ruta de ejecución adoptada por el código y posiblemente hace que el código falla donde no lo hacía antes. Si el código falla, no se debe a la invocación de yield() añadida[90]. Se debe a que el código es incorrecto y hemos hecho que el fallo sea más evidente. Este enfoque presenta varios problemas:

- Tendrá que buscar manualmente los puntos adecuados donde hacerlo.
- ¿Cómo sabe dónde incluir la invocación y qué tipo de invocación usar?
- La presencia de este código en un entorno de producción ralentiza innecesariamente el código.
- Es un enfoque que puede o no detectar los fallos; de hecho, no las tiene todas consigo.

Lo que necesitamos es una forma de hacerlo durante la fase de pruebas, no de producción. También debemos poder mezclar configuraciones entre ejecuciones, lo que aumenta las probabilidades de detectar los errores.

Evidentemente, si dividimos el sistema POJO que no sepa nada los procesos en clases que controlen los procesos, resultará más sencillo ubicar los puntos en los que instrumentar el código. Es más, podríamos crear diferentes pruebas que invoquen los POJO bajo distintos regímenes de invocaciones a `sleep`, `yield` y demás.

Automática

Puede usar herramientas como la estructura orientada a aspectos, CGLIB o ASM para instrumentar su código mediante programación. Por ejemplo, podría usar una clase con un único método:

```
public class ThreadJigglePoint {
   public static void jiggle() {
   }
}
```

Puede añadir invocaciones en distintos puntos del código:

```
public synchronized String nextUrlOrNull() {
   if(hasNext()) {
     ThreadJigglePoint.jiggle();
     String url = urlGenerator.next();
     ThreadJigglePoint.jiggle();
     updateHasNext();
     ThreadJigglePoint.jiggle();
     return url;
   }
   return null;
}
```

Tras ello, use un sencillo aspecto que seleccione aleatoriamente entre no hacer nada, pausar o generar un resultado.

Imagine que la clase `ThreadJigglePoint` tiene dos implementaciones. La primera implementa `jiggle` para no hacer nada y se usa en producción. La segunda genera un número aleatorio para elegir entre `sleep`, `yield` o nada. Si ejecuta sus pruebas mil veces con `jiggle` de forma aleatoria, puede descubrir algunos fallos. Si la prueba es satisfactoria, al menos puede felicitarse por haber actuado correctamente. Aunque sea un tanto simple, puede resultar una opción razonable en lugar de recurrir a una herramienta más sofisticada. La herramienta ConTest[91], desarrollada por IBM, tiene un funcionamiento

similar pero es más sofisticada.

El objetivo es que los procesos del código se ejecuten en distinto orden en momentos diferentes. La combinación de pruebas bien escritas y ejecuciones aleatorias puede aumentar considerablemente la capacidad de detectar errores.

Recomendación: *Use estas estrategias para detectar errores*.

Conclusión

Es complicado conseguir código concurrente correcto. El código sencillo se puede complicar al añadir varios procesos y datos compartidos. Si tiene que crear código concurrente, tendrá que hacerlo con rigor o se enfrentará a sutiles y esporádicos fallos.

En primer lugar, siga el principio de responsabilidad única. Divida su sistema en varios POJO que separen el código compatible con procesos del resto. Asegúrese de probar únicamente el código compatible con procesos y nada más, por lo que este código debe ser de tamaño reducido y específico.

Conozca los orígenes de los problemas de concurrencia: varios procesos que operen en datos compartidos o usen una agrupación de recursos común. Los casos de límites, como el cierre correcto o la conclusión de la iteración de un bucle, pueden ser especialmente espinosos.

Conozca su biblioteca y los algoritmos fundamentales. Debe comprender cómo las funciones de la biblioteca permiten resolver problemas similares a los de los algoritmos fundamentales.

Aprenda a localizar regiones del código que se puedan bloquear y bloquéelas. No bloquee otras regiones que no lo necesiten. Evite invocar una sección bloqueada desde otra. Para ello debe saber si un elemento está compartido o no. Reduzca la cantidad de objetos compartidos y su ámbito. Cambie los diseños de los objetos con datos compartidos para acomodar clientes en lugar de obligar a los clientes a gestionar el estado compartido.

Los problemas se acumularán. Los que no aparezcan inicialmente suelen considerarse esporádicos y suelen producirse en la fase de carga o de modo aparentemente aleatorio. Por tanto, debe poder ejecutar su código con

procesos en diferentes configuraciones y plataformas de forma repetida y continua. La capacidad de prueba, algo natural si aplica las tres leyes de TDD, implica cierto nivel de conectividad, lo que ofrece la compatibilidad necesaria para ejecutar código en distintas configuraciones.

La probabilidad de detectar errores mejora si se toma el tiempo necesario para instrumentar su código. Puede hacerlo manualmente o mediante tecnologías automatizadas. Hágalo en las fases iniciales. Es aconsejable ejecutar el código basado en procesos durante el mayor tiempo posible antes de pasarlo a producción.

Si adopta un enfoque limpio, aumentarán las probabilidades de hacerlo de forma correcta.

Bibliografía

- **[Lea99]**: *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

14
Refinamiento sucesivo

Caso práctico de un analizador de argumentos de línea de comandos

Este capítulo es un caso práctico de refinamiento sucesivo. Veremos un

módulo que comienza correctamente pero no mantiene dicha corrección. Tras ello, veremos cómo se refactoriza y se limpia. Muchos hemos tenido que analizar argumentos de línea de comando. Si no disponemos de una utilidad para ello, recorremos la matriz de cadenas pasadas a la función principal. Puede encontrar utilidades de calidad pero ninguna hace exactamente lo que necesitamos. Por ello, decidí crear una propia, a la que he denominado `Args`. `Args` es muy fácil de usar. Basta crearla con los argumentos de entrada y una cadena de formato, y después consultar a la instancia de `Args` los valores de los argumentos. Fíjese en el siguiente ejemplo:

Listado 14-1
Uso de Args

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argument error: %s\n", e.errorMessage());
  }
}
```

Comprobará lo sencillo que es. Creamos una instancia de la clase `Args` con dos parámetros. El primero es la cadena de formato o esquema: `"l,p#,d*"`. Define tres argumentos de línea de comandos. El primero, `-l`, es un argumento booleano. El segundo, `-p`, es un argumento entero. El tercero, `-d`, es un argumento de cadena. El segundo parámetro del constructor `Args` es la matriz de argumentos de línea de comandos pasada a `main`. Si el constructor no genera `ArgsException`, la línea de comandos entrante se ha analizado y se puede consultar la instancia `Args`. Se usan métodos como `getBoolean`, `getInteger` y `getString` para acceder a los valores de los argumentos por sus nombres.

Si hay un problema, ya sea en la cadena de formato o en los argumentos de línea de comandos, se genera `ArgsException`. La descripción del error se

puede recuperar del método `errorMessage` de la excepción.

Implementación de Args

El Listado 14-2 es la implementación de la clase `Args`. Examínela con atención. El estilo y la estructura se han trabajado concienzudamente y espero que los imite.

Listado 14-2
Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
   private Map<Character, ArgumentMarshaler> marshalers;
   private Set<Character> argsFound;
   private ListIterator<String> currentArgument;

   public Args(String schema, String[] args) throws ArgsException {
     marshalers = new HashMap<Character, ArgumentMarshaler>();
     argsFound = new HashSet<Character>();

     parseSchema(schema);
     parseArgumentStrings(Arrays.asList(args));
   }

   private void parseSchema(String schema) throws ArgsException {
     for (String element : schema.split(“,”))
       if (element.length() > 0)
         parseSchemaElement(element.trim());
   }

   private void parseSchemaElement(String element) throws ArgsException {
     char elementId = element.charAt(0);
     String elementTail = element.substring(1);
     validateSchemaElementId(element Id);
     if (elementTail.length() == 0)
       marshalers.put(elementId, new BooleanArgumentMarshaler());
     else if (elementTail.equals(“*”))
       marshalers.put(elementId, new StringArgumentMarshaler());
     else if (elementTail.equals(“#”))
       marshalers.put(elementId, new IntegerArgumentMarshaler());
     else if (elementTail.equals(“##”))
       marshalers.put(elementId, new DoubleArgumentMarshaler());
```

```
      else if (elementTail.equals("[*]"))
        marshalers.put(elementId, new StringArrayArgumentMarshaler());
      else
        throw     new     ArgsException(INVALID_ARGUMENT_FORMAT,     elementId,
elementTail);
    }

    private void validateSchemaElementId(char elementId) throws ArgsException
{
      if {!Character.isLetter(elementId))
        throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
    }

    private    void    parseArgumentStrings(List<String>    argsList)    throws
ArgsException
    {
      for          (currentArgument          =          argsList.listIterator();
currentArgument.hasNext();)
      {
        String argString = currentArgument.next();
        if (argString.startsWith("-")) {
          parseArgumentCharacters(argString.substring(1));
        } else {
          currentArgument.previous();
          break;
        }
      }
    }

    private void parseArgumentCharacters(String argChars) throws ArgsException
{
      for (int i = 0; i < argChars.length(); i++)
        parseArgumentCharacter(argChars.charAt(i));
    }

    private void parseArgumentCharacter(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      if (m == null) {
        throw new ArgsException (UNEXPECTED_ARGUMENT, argChar, null);
      } else {
        argsFound.add(argChar);
        try {
          m.set(currentArgument);
        } catch (ArgsException e) {
          e.setErrorArgumentId(argChar);
          throw e;
        }
      }
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public int nextArgument() {
```

```
      return currentArgument.nextIndex();
    }

    public boolean getBoolean(char arg) {
      return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public String getString(char arg) {
      return StringArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public int getInt(char arg) {
      return IntegerArgumentMarshaler.getValue (marshalers.get(arg));
    }

    public double getDouble(char arg) {
      return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public String[] getStringArray(char arg) {
      return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
    }
  }
```

Puede leer el código de arriba a abajo sin necesidad de saltar de un punto a otro ni buscar hacia adelante. Lo que seguramente busque es la definición de `ArgumentMarshaler`, que hemos omitido intencionadamente. Tras leer el código, comprenderá la interfaz `ArgumentMarshaler` y la función de sus variantes. Veamos algunas de ellas (entre los listados 14-3 y 14-6).

Listado 14-3
ArgumentMarshaler.java

```
  public interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
  }
```

Listado 14-4
BooleanArgumentMarshaler.java

```
  public class BooleanArgumentMarshaler implements ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set (Iterator<String> currentArgument) throws ArgsException {
      booleanValue = true;
    }

    public static boolean getValue(ArgumentMarshaler am) {
```

```
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
    }
  }
```

Listado 14-5
StringArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = “”;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return “”;
  }
}
```

Listado 14-6
IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      IntValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }
```

```
    public static int getValue (ArgumentMarshaler am) {
      if (am != null && am instanceof IntegerArgumentMarshaler)
        return ((IntegerArgumentMarshaler) am).intValue;
      else
        return 0;
    }
  }
```

Las otras variantes de `ArgumentMarshaler` simplemente repiten este patrón en matrices `double` y `String` y sólo complicarían el capítulo. Puede consultarlas como ejercicio. Otro fragmento que puede resultar complicado es la definición de las constantes de código de error, incluidas en la clase `ArgsException` (véase el Listado 14-7).

Listado 14-7
ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) { super(message); }

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode,
            char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
```

```
    }

    public String getErrorParameter() {
      return errorParameter;
    }

    public void setErrorParameter(String errorParameter) {
      this.errorParameter = errorParameter;
    }

    public ErrorCode getErrorCode() {
      return errorCode;
    }

    public void setErrorCode(ErrorCode errorCode) {
      this.errorCode = errorCode;
    }

    public String errorMessage() {
      switch (errorCode) {
        case OK:
          return “TILT: Should not get here.”;
        case UNEXPECTED_ARGUMENT:
          return String.format(“Argument -%c unexpected.”, errorArgumentId);
        case MISSING_STRING:
          return String.format(“Could not find string parameter for -%c.”,
                errorArgumentId);
        case INVALID INTEGER:
          return String.format(“Argument -%c expects an integer but was ‘%s’.”,
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
          return String.format(“Could not find integer parameter for -%c.”,
                errorArgumentId);
        case INVALID_DOUBLE:
          return String.format(“Argument -%c expects a double but was ‘%s’.”,
                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
          return String.format(“Could not find double parameter for -%c.”,
                errorArgumentId);
        case INVALID_ARGUMENT_NAME:
          return String.format(“‘%c’ is not a valid argument name.”,
                errorArgumentId);
        case INVALID_ARGUMENT_FORMAT:
          return String.format(“‘%s’ is not a valid argument format.”,
                errorParameter);
      }
      return “”;
    }

    public enum ErrorCode {
      OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
      MISSING_STRING,
      MISSING_INTEGER, INVALID_INTEGER,
      MISSING_DOUBLE, INVALID_DOUBLE }
  }
```

Es sorprendente la cantidad de código necesario para detallar este sencillo concepto. Uno de los motivos es el uso de un lenguaje especialmente profuso. Java, al ser un lenguaje de tipos estáticos, requiere muchas palabras para satisfacer el sistema de tipos. En lenguajes como Ruby, Python o Smalltalk, este programa es mucho más reducido[92].

Vuelva a leer el código. Fíjese especialmente en los nombres de los elementos, el tamaño de las funciones y el formato. Si tiene experiencia como programador, partes del estilo o la estructura no le convencerán, pero espero que, desde un punto de vista global, considere que el programa está bien escrito y tiene una estructura limpia.

Por ejemplo, debería ser evidente cómo añadir un nuevo tipo de argumento, como una fecha o un número complejo, y que dicha inclusión apenas requeriría código. En definitiva, bastaría con una nueva variante de `ArgumentMarshaler`, una nueva función `getXXX` y una nueva instrucción `case` en la función `parseSchemaElement`. También habría un nuevo código `ArgsException.ErrorCode` y un nuevo mensaje de error.

Cómo se ha realizado

No diseñé este programa de principio a fin en su forma actual y, sobre todo, no espero que pueda crear programas limpios y elegantes a la primera. Si algo hemos aprendido en las dos últimas décadas es que la programación es un arte más que una ciencia. Para escribir código limpio, primero debe crear código imperfecto y después limpiarlo. No debería sorprenderle. Ya lo aprendimos en el colegio cuando los profesores (normalmente en vano) nos obligaban a crear borradores de nuestras redacciones. El proceso, nos decían, era escribir un primer borrador, después otro, y después otros muchos hasta lograr una versión definitiva. Para escribir redacciones limpias, el refinamiento debía ser continuado.

Muchos programadores noveles (como sucede con los alumnos) no siguen este consejo. Creen que el objetivo principal es que el programa funcione. Una vez que lo consiguen, pasan a la siguiente tarea, y conservan el estado funcional del programa, sea cual sea. Los programadores

experimentados saben que esto es un suicidio profesional.

Args: El primer borrador

El Listado 14-8 muestra una versión inicial de la clase `Args`. Funciona, pero es un desastre.

Listado 14-8
Args.java (primer borrador)

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs =
    new HashMap<Character, String>();
  private Map<Character, Integer> intArgs =
    new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK,        MISSING_STRING,        MISSING_INTEGER,        INVALID_INTEGER,
UNEXPECTED_ARGUMENT}

  public Args(String schema. String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
```

```
      }
      return valid;
    }

    private boolean parseSchema() throws ParseException {
      for (String element : schema.split(“,”)) {
        if (element.length() > 0) {
          String trimmedElement = element.trim();
          parseSchemaElement(trimmedElement);
        }
      }
      return true;
    }

    private void parseSchemaElement(String element) throws ParseException {
      char elementId = element.charAt(0);
      String elemenTail = element.substring(1);
      validateSchemaElementId(elementId);
      if (isBooleanSchemaElement(elementTail));
        parseBooleanSchemaElement(elementId);
      else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
      else if (isIntegerSchemaElement(elementTail)) {
        parseIntegerSchemaElement(elementId);
      } else {
        throw new ParseException(
          String.format(“Argument: %c has invalid format: %s.”,
               elementId, elementTail), 0);
      }
    }

    private void validateSchemaElementId(char elementId) throws ParseException
{
      if (!Character.isLetter(elementId)) {
        throw new ParseException(
          “Bad character:” + elementId + “in Args format: ” + schema, 0);
      }
    }

    private void parseBooleanSchemaElement(char elementId) {
      booleanArgs.put(elementId, false);
    }

    private void parseIntegerSchemaElement(char elementId) {
      intArgs.put(elementId, 0);
    }

    private void parseStringSchemaElement(char elementId) {
      stringArgs.put(elementId, “”);
    }

    private boolean isStringSchemaElement(String elementTail) {
      return elementTail.equals(“*”);
    }
```

```
    private boolean isBooleanSchemaElement(String elementTail) {
      return elementTail.length() == 0;
    }

    private boolean isIntegerSchemaElement(String elementTail) {
      return elementTail.equals("#");
    }

    private boolean parseArguments() throws ArgsException {
      for    (currentArgument    =    0;    currentArgument    <    args.length;
currentArgument++)
      {
        String arg = args[currentArgument];
        parseArgument(arg);
      }
      return true;
    }

    private void parseArgument(String arg) throws ArgsException {
      if (arg.startsWith("-"))
        parseElements(arg);
    }

    private void parseElements(String arg) throws ArgsException {
      for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
    }

    private void parseElement(char argChar) throws ArgsException {
      if (setArgument(argChar))
        argsFound.add(argChar);
      else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
      }
    }

    private boolean setArgument(char argChar) throws ArgsException {
      if (isBooleanArg(argChar))
        setBooleanArg(argChar, true);
      else if (isStringArg(argChar))
        setStringArg(argChar);
      else if (isIntArg(argChar))
        setIntArg(argChar);
      else
        return false;

      return true;
    }

    private boolean isIntArg(char argChar) {
      return intArgs.containsKey(argChar);
    }
```

```
private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.put(argChar, new Integer(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}

private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private boolean isStringArg(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
  return "-[" + schema + “]”;
  else
  return “”;
}
```

```
    public String errorMessage() throws Exception {
      switch (errorCode) {
        case OK:
          throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
          return unexpectedArgumentMessage();
        case MISSING_STRING:
          return String.format("Could not find string parameter for -%c.",
               errorArgumentId);
        case INVALID_INTEGER:
          return  String.format("Argument  -  %c  expects  an  integer  but  was
'%s'.",
               errorArgumentId, errorParameter);
        case MISSING_INTEGER:
          return String.format("Could not find integer parameter for -%c.",
               errorArgumentId);
      }
      return "";
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer("Arguments(s) -");
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(" unexpected.");

      return message.toString();
    }

    private boolean falseIfNull(Boolean b) {
      return b != null && b;
    }

    private int zeroIfNull(Integer i) {
      return i == null ? 0 : i;
    }

    private String blankIfNull(String s) {
      return s = null ? "" : s;
    }

    public String getString(char arg) {
      return blankIfNull(stringArgs.get(arg));
    }

    public int getInt(char arg) {
      return zeroIfNull(intArgs.get(arg));
    }

    public boolean getBoolean(char arg) {
      return falseIfNull(booleanArgs.get(arg));
    }

    public boolean has(char arg) (
```

```
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }

    private class ArgsException extends Exception {
    }
  }
```

Espero que su reacción inicial ante tal cantidad de código es alegrarse por no haberlo conservado tal cual. Si ha sido su reacción, recuerde que será la que tengan otros que lean un borrador de su código.

En realidad, primer borrador es lo mejor que se puede decir sobre este código. Evidentemente es un trabajo en progreso. La cantidad de variables de instancia es apabullante. Cadenas extrañas como «`TILT`», `HashSet` y `TreeSet`, y los bloques `try-catch-catch` aumentan el desastre.

No era mi intención crear este desastre. En realidad, intentaba mantener cierta organización, como demuestra la elección de nombres de funciones y variables, y la estructura del programa. Pero es evidente que el problema se me fue de las manos.

El desastre aumentó gradualmente. Las versiones anteriores no fueron tan malas. Por ejemplo, el Listado 14-9 muestra una versión inicial en la que sólo funcionaban los argumentos booleanos.

Listado 14-9
Args.java (sólo argumentos booleanos)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
```

```
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
  }

  private boolean parseSchema() {
    for (String element : schema.split(“,”)) {
      parseSchemaElement(element);
    }
    return true;
  }

  private void parseSchemaElement(String element) {
    if (element.length() == 1) {
      parseBooleanSchemaElement(element);
    }
  }

  private void parseBooleanSchemaElement(String element) {
    char c = element.charAt(0);
    if (Character.isLetter(c)) {
      booleanArgs.put(c, false);
    }
  }

  private boolean parseArguments() {
    for (String arg : args)
      parseArgument(arg);
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith(“-”))
      parseElement(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (isBoolean(argChar)) {
      numberOfArguments++;
      setBooleanArg(argChar, true);
```

```
      } else
        unexpectedArguments.add(argChar);
    }

    private void setBooleanArg(char argChar, boolean value) {
      booleanArgs.put(argChar, value);
    }

    private boolean isBoolean(char argChar) {
      return booleanArgs.containsKey(argChar);
    }

    public int cardinality() {
      return numberOfArguments;
    }

    public String usage() {
      if (schema.length() > 0)
        return "-["+schema+"]";
      else
        return "";
    }

    public String errorMessage() {
      if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
      } else
        return "";
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer("Argument(s) -");
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(" unexpected.");

      return message.toString();
    }

    public boolean getBoolean(char arg) {
      return booleanArgs.get(arg);
    }
  }
```

Aunque hay motivos para quejarse del código, no es tan malo. Es compacto y sencillo, y fácil de entender. Sin embargo, en este código se aprecia la semilla del desastre posterior y resulta evidente porqué.

La versión posterior sólo tiene dos tipos de argumentos más que ésta: `String` e `integer`. La inclusión de sólo dos tipos más tiene un tremendo impacto negativo en el código. Lo convierte de algo que sería razonablemente

mantenible en algo que seguramente esté plagado de errores.

Añadí los dos tipos de argumento de forma incremental. Primero, el argumento `String`, que genera lo siguiente:

Listado 14-10
Args.java (booleano y String)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs =
    new HashMap<Character, String>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgument = '\0';

  enum ErrorCode { OK, MISSING_STRING }

  private ErrorCode errorCode = ErrorCode.OK;

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(","")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
```

```
    }

    private void parseSchemaElement(String element) throws ParseException {
      char elementId = element.charAt(0);
      String elementTail = element.substring(1);
      validateSchemaElementId(elementId);
      if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
      else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
    }

    private void validateSchemaElementId(char elementId) throws ParseException
{
      if (!Character.isLetter(elementId)) {
        throw new ParseException(
          "Bad character:" + elementId + "in Args format: " + schema, 0);
      }
    }

    private void parseStringSchemaElement(char elementId) {
      stringArgs.put(elementId, "");
    }

    private boolean isStringSchemaElement(String elementTail) {
      return elementTail.equals("*");
    }

    private boolean isBooleanSchemaElement(String elementTail) {
      return elementTail.length() == 0;
    }

    private void parseBooleanSchemaElement(char elementId) {
      booleanArgs.put(elementId, false);
    }

    private boolean parseArguments() {
      for    (currentArgument    =    0;    currentArgument    <    args.length;
currentArgument++)
      {
        String arg = args[currentArgument];
        parseArgument(arg);
      }
      return true;
    }

    private void parseArgument(String arg) {
      if (arg.startsWith("-"))
        parseElements(arg);
    }

    private void parseElements(String arg) {
      for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
    }
```

```
    private void parseElement(char argChar) {
      if (setArgument(argChar))
        argsFound.add(argChar);
      else {
        unexpectedArguments.add(argChar);
        valid = false;
      }
    }

    private boolean setArgument(char argChar) {
      boolean set = true;
      if (isBoolean(argChar))
        setBooleanArg(argChar, true);
      else if (isString(argChar))
        setStringArg (argChar, “”);
      else
        set = false;

      return set;
    }

    private void setStringArg(char argChar, String s) {
      currentArgument++;
      try {
        stringArgs.put(argChar, args[currentArgument]);
      } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgument = argChar;
        errorCode = ErrorCode.MISSING_STRING;
      }
    }

    private boolean isString(char argChar) {
      return stringArgs.containsKey(argChar);
    }

    private void setBooleanArg(char argChar, boolean value) {
      booleanArgs.put(argChar, value);
    }

    private boolean isBoolean(char argChar) {
      return booleanArgs.containsKey(argChar);
    }

    public int cardinality() {
      return argsFound.size();
    }

    public String usage() {
      if (schema.length() > 0)
        return “-[” + schema + “]”;
      else
        return “”;
    }
```

```
    public String errorMessage() throws Exception {
      if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
      } else
        switch (errorCode) {
          case MISSING_STRING:
            return String.format (“Could not find string parameter for -%c.”,
                errorArgument);
          case OK:
            throw new Exception(“TILT: Should not get here.”);
        }
      return “”;
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer(“Argument(s) -”);
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(“ unexpected.”);

      return message.toString();
    }

    public boolean getBoolean(char arg) {
      return falseIfNull(booleanArgs.get(arg));
    }

    private boolean falseIfNull(Boolean b) {
      return b == null ? false : b;
    }

    public String getString(char arg) {
      return blankIfNull(stringArgs.get(arg));
    }

    private String blankIfNull(String s) {
      return s == null ? “” : s;
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }
  }
```

Comprobará que empieza a desbocarse. No es terrible pero el desastre se está gestando. Basta con incluir el tipo de argumento `integer` para que resulte fatídico.

Entonces me detuve

Todavía tenía que añadir otros dos tipos de argumentos y sabía que empeorarían las cosas. Si los forzaba, seguramente funcionarían pero provocaría un desastre demasiado complicado de arreglar. Si la estructura del código tenía que poder mantenerse, era el momento de corregirla.

Por ello dejé de añadir elementos y comencé la refactorización. Tras añadir los argumentos `String` e `integer`, sabía que cada uno necesitaría nuevo código en tres puntos principales. En primer lugar, cada tipo de argumento necesita una forma de analizar su elemento de esquema para poder seleccionar el `HashMap` de ese tipo. Tras ello, sería necesario analizar cada tipo de argumento en las cadenas de línea de comandos y convertirlos en su tipo correcto. Por último, cada tipo de argumento necesitaría un método `getXXX` para poder devolverlo al invocador como su tipo correcto.

Muchos tipos diferentes y todos con métodos similares, lo que en realidad era una clase. Y de este modo nació el concepto de `ArgumentMarshaler`.

Sobre el incrementalismo

Una de las mejores formas de acabar con un programa es realizar cambios masivos con la intención de mejorarlo. Algunos programas nunca se recuperan de estas mejoras. El problema es lo complicado que resulta conseguir que el programa funcione de la misma forma que antes de la mejora.

Para evitarlo, recurro a la disciplina TDD (*Test-Driven Development* o Desarrollo guiado por pruebas). Una de las doctrinas centrales de este enfoque es mantener la ejecución del sistema en todo momento. Es decir, con TDD no puedo realizar cambios que afecten al funcionamiento del sistema. Todos los cambios deben mantenerlo como antes de los cambios. Para lograrlo, necesito una serie de pruebas automatizadas que ejecutar rápidamente y que verifiquen que el comportamiento del sistema no ha variado. Para la clase `Args`, creé una serie de pruebas de unidad y aceptación. Las pruebas de unidad se crearon en Java y se administraron con JUnit. Las

pruebas de aceptación se crearon como páginas wiki en FitNesse. Podría haber ejecutado estas pruebas en cualquier momento y, si eran satisfactorias, sabría que el sistema funcionaba de la forma especificada.

Así pues, comencé a realizar pequeños cambios. Cada uno desplazaba la estructura del sistema hacia el concepto `ArgumentMarshaler`, y cada cambio mantenía el funcionamiento del sistema. El primer cambio realizado fue añadir el esqueleto de `ArgumentMarshaller` al final del desastre anterior (véase el Listado 14-11).

Listado 14-11
ArgumentMarshaller añadido a Args.java

```
private class ArgumentMarshaler {
  private boolean booleanValue = false;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() { return booleanValue; }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Evidentemente, esto no afectaría a nada, por lo que realicé la modificación más sencilla posible que afectara a la mínima cantidad de código. Cambié `HashMap` para que los argumentos Boolean aceptaran `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
  new HashMap<Character, ArgumentMarshaler>();
```

Esto afectaba a varias instrucciones que corregí rápidamente.

```
…
  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
  }
…
```

```
    private void setBooleanArg(char argChar, boolean value) {
      booleanArgs.get(argChar).setBoolean(value);
    }
…
    public boolean getBoolean(char arg) {
      return falseIfNull (booleanArgs.get(arg).getBoolean());
    }
```

Estos cambios se aplican a las zonas que mencionamos antes: `parse`, `set` y `get` para el tipo de argumento. Desafortunadamente, aunque sean cambios menores, algunas de las pruebas comenzaron a fallar. Si se fija atentamente en `getBoolean`, comprobará que se puede invocar con `y` pero no existe un argumento `y`, por lo que `booleanArgs.get('y')` devolverá `null` y la función generará `NullPointerException`. La función `falseIfNull` se usa como protección ante este hecho pero el cambio aplicado hace que la función sea irrelevante.

El incrementalismo exigía que esto funcionara antes de realizar otros cambios. La solución no era demasiado complicada; bastaba con cambiar la comprobación de `null`. Ya no era necesario comprobar `null` en boolean, sino en `ArgumentMarshaller`.

Primero, eliminé la invocación de `falseIfNull` en la función `getBoolean`. Ya no servía de nada, por lo que eliminé directamente la función. Las pruebas seguían fallando igual, lo que suponía que no había nuevos errores.

```
  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
  }
```

Tras ello, dividí la función en dos líneas y añadí `ArgumentMarshaller` a una variable propia: `argumentMarshaller`. No me preocupaba el extenso nombre de la variable; era redundante y estorbaba a la función, por lo que lo reduje a `am` [N5].

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am.getBoolean();
  }
```

Y tras ello añadí la lógica de detección de `null`.

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
```

```
    return am != null && am.getBoolean();
  }
```

Argumentos de cadena

La inclusión de los argumentos `String` fue similar a la de los argumentos `boolean`. Tuve que cambiar `HashMap` y conseguir que funcionaran `parse`, `set` y `get`. No deberían producirse sorpresas posteriores a excepción de que la implementación completa se incluía en la clase `ArgumentMarshaller` en lugar de distribuirla en variantes.

```
    private Map<Character, ArgumentMarshaler> stringArgs =
      new HashMap<Character, ArgumentMarshaler>();
  …
    private void parseStringSchemaElement(char elementId) {
      stringArgs.put(elementId, new StringArgumentMarshaler());
    }
  …
    private void setStringArg(char argChar) throws ArgsException {
      currentArgument++;
      try {
        stringArgs.get(argChar).setString(args[currentArgument]);
      } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }
  …
    public String getString (char arg) {
      Args.ArgumentMarshaler am = stringArgs.get(arg);
      return am == null ? "" : am.getString();
    }
  …
    private class ArgumentMarshaler {
      private boolean booleanValue = false;
      private String stringValue;

      public void setBoolean(boolean value) {
        booleanValue = value;
      }

      public boolean getBoolean() {
        return booleanValue;
      }

      public void setString(String s) {
```

```
        stringValue = s;
      }

      public String getString() {
        return stringValue == null ? "" : stringValue;
      }
    }
```

De nuevo, estos cambios se realizaron individualmente para conservar las pruebas, aunque fallaran. Si una prueba fallaba, me aseguraba de que fuera correcta antes de continuar con el siguiente cambio.

Ya debería reconocer mi intención. Tras incluir el comportamiento de señalización en la clase base `ArgumentMarshaler`, comencé a transferirlo a las variantes, para de esta forma mantener el funcionamiento mientras cambiaba gradualmente la forma del programa.

El siguiente paso consistía en transferir la funcionalidad del argumento `int` a `ArgumentMarshaler`. De nuevo, no hubo sorpresas.

```
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  …
    private void parseIntegerSchemaElement(char elementId) {
      intArgs.put(elementId, new IntegerArgumentMarshaler());
    }
  …
    private void setIntArg(char argChar) throws ArgsException {
      currentArgument++;
      String parameter = null;
      try {
        parameter = args[currentArgument];
        intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
      } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }
  …
    public int getInt(char arg) {
      Args.ArgumentMarshaler am = intArgs.get(arg);
      return am == null ? 0 : am.getInteger();
    }
```

```
…
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? “” : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }
  }
```

Tras transferir la señalización a `ArgumentMarshaler`, comencé a transferir la funcionalidad a las variantes. El primer paso fue pasar la función `setBoolean` a `BooleanArgumentMarshaller` y garantizar su correcta invocación. Para ello creé un método `set` abstracto.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
  private String stringValue;
  private int integerValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
```

```
      return stringValue == null ? “” : stringValue;
    }

    public void set Integer(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);
  }
```

Tras ello, implementé el método `set` en `BooleanArgumentMarshaller`.

```
  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
      booleanValue = true;
    }
  }
```

Y por último cambié la invocación de `setBoolean` por la de `set`.

```
  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set(“true”);
  }
```

Las pruebas seguían siendo satisfactorias. Como este cambio hacía que `set` se implementara en `BooleanArgumentMarshaler`, eliminé el método `setBoolean` de la clase base `ArgumentMarshaler`.

La función abstracta `set` acepta un argumento `String` pero la implementación de `BooleanArgumentMarshaler` no lo usa. He incluido el argumento porque sabía que `StringArgumentMarshaler` e `IntegerArgumentMarshaler` lo utilizarían.

Tras ello, el objetivo era implementar el método `get` en `BooleanArgumentMarshaler`. La implementación de funciones get siempre es escabrosa ya que el tipo devuelto tiene que ser `Object` y en este caso debe convertirse a `Boolean`.

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
  }
```

Para compilarlo, añadí la función `get` a `ArgumentMarshaler`.

```
  private abstract class ArgumentMarshaler {
    …
```

```
    public Object get() {
      return null;
    }
  }
```

Se compila y las pruebas fallan. Para que vuelvan a funcionar, basta con convertir `get` en abstracto e implementarlo en `BooleanArgumentMarshaler`.

```
  private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    …

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set (String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

De nuevo, las pruebas son satisfactorias. Ahora tanto `get` como `set` se implementan en `BooleanArgumentMarshaler`. Esto me permite eliminar la antigua función `getBoolean` de `ArgumentMarshaler`, cambiar la variable protegida `booleanValue` a `BooleanArgumentMarshaler` y convertirla en privada.

Repetí el mismo patrón de cambios con las cadenas. Implementé `set` y `get`, eliminé las funciones sin usar y desplacé las variables.

```
    private void setStringArg(char argChar) throws ArgsException {
      currentArgument++;
      try {
        stringArgs.get(argChar).set(args[currentArgument]);
      } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

  …
    public String getString(char arg) {
      Args.ArgumentMarshaler am = stringArgs.get(arg);
      return am == null ? “” : (String) am.get();
    }
```

```
…

  private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {

    public void set(String s){

    }

    public Object get() {
      return null;
    }
  }
}
```

Por último, repetí el proceso con los enteros. Resulta más complicado ya que los enteros deben analizarse y la operación de análisis puede generar una

excepción, pero el resultado es más indicado ya que el concepto de `NumberFormatException` se oculta totalmente en `IntegerArgumentMarshaler`.

```
  private boolean isIntArg(char argChar) { return intArgs.containsKey(argChar);
}

     private void setIntArg(char argChar) throws ArgsException {
       currentArgument++;
       String parameter = null;
       try {
         parameter = args[currentArgument];
         intArgs.get(argChar).set(parameter);
       } catch (ArrayIndexOutOfBoundsException e) {
         valid = false;
         errorArgumentId = argChar;
         errorCode = ErrorCode.MISSING_INTEGER;
         throw new ArgsException();
       } catch (ArgsException e) {
         valid = false;
         errorArgumentId = argChar;
         errorParameter = parameter;
         errorCode = ErrorCode.INVALID_INTEGER;
         throw e;
       }
     }
  …
     private void setBooleanArg(char argChar) {
       try {
         booleanArgs.get(argChar).set("true");
       } catch (ArgsException e) {
       }
     }
  …
     public int getInt(char arg) {
       Args.ArgumentMarshaler am = intArgs.get(arg);
       return am == null ? 0 : (Integer) am.get();
     }
  …
     private abstract class ArgumentMarshaler {
       public abstract void set(String s) throws ArgsException;
       public abstract Object get();
     }
  …
     private class IntegerArgumentMarshaler extends ArgumentMarshaler {
       private int intValue = 0;

       public void set(String s) throws ArgsException {
         try {
           intValue = Integer.parseInt(s);
         } catch (NumberFormatException s) {
           throw new ArgsException();
         }
```

```
    }

    public Object get() {
      return intValue;
    }
  }
```

Evidentemente, las pruebas seguían funcionando. Tras ello, me deshice de las distintas asignaciones de la parte superior del algoritmo, lo que hace que el sistema sea mucho más genérico. Sin embargo, no las puede eliminar ya que afectaría a la integridad del sistema. En su lugar, añadí un nuevo `Map` para `ArgumentMarshaler` y, tras ello, cambié uno a uno los métodos para que usaran la nueva asignación en lugar de las originales.

```
public class Args {
…
   private Map<Character, ArgumentMarshaler> booleanArgs =
     new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> stringArgs =
     new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> intArgs =
     new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> marshalers =
     new HashMap<Character, ArgumentMarshaler>();
…
   private void parseBooleanSchemaElement(char elementId) {
     ArgumentMarshaler m = new BooleanArgumentMarshaler();
     booleanArgs.put(elementId, m);
     marshalers.put(elementId, m);
   }

   private void parseIntegerSchemaElement(char elementId) {
     ArgumentMarshaler m = new IntegerArgumentMarshaler();
     intArgs.put(elementId, m);
     marshalers.put(elementId, m);
   }

   private void parseStringSchemaElement(char elementId) {
     ArgumentMarshaler m = new StringArgumentMarshaler();
     stringArgs.put(elementId, m);
     marshalers.put(elementId, m);
   }
```

Las pruebas seguían funcionando. Tras ello, cambié `isBooleanArg` de esto:

```
private boolean isBooleanArg(char argChar) {
   return booleanArgs.containsKey(argChar);
}
```

a este otro:

```
private boolean isBooleanArg(char argChar) {
   ArgumentMarshaler m = marshalers.get(argChar);
   return m instanceof BooleanArgumentMarshaler;
}
```

Las pruebas funcionaban, por lo que apliqué el mismo cambio en `isIntArg` e `isStringArg`.

```
private boolean isIntArg(char argChar) {
   ArgumentMarshaler m = marshalers.get(argChar);
   return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(char argChar) {
   ArgumentMarshaler m = marshalers.get(argChar);
   return m instanceof StringArgumentMarshaler;
}
```

Las pruebas eran correctas, por lo que eliminé las invocaciones duplicadas de `marshalers.get`:

```
private boolean setArgument(char argChar) throws ArgsException {
   ArgumentMarshaler m = marshalers.get(argChar);
   if (isBooleanArg(m))
     setBooleanArg(argChar);
   else if (isStringArg(m))
     setStringArg(argChar);
   else if (isIntArg(m))
     setIntArg(argChar);
   else
     return false;

   return true;
}

private boolean isIntArg (ArgumentMarshaler m) {
   return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg (ArgumentMarshaler m) {
   return m instanceof StringArgumentMarshaler;
}

private boolean isBooleanArg (ArgumentMarshaler m) {
   return m instanceof BooleanArgumentMarshaler;
}
```

Los tres argumentos `isxxxArg` ya no tenían sentido, de modo que los reubiqué:

```
private boolean setArgument(char argChar) throws ArgsException {
   ArgumentMarshaler m = marshalers.get(argChar);
   if (m instanceof BooleanArgumentMarshaler)
     setBooleanArg(argChar);
```

```
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(argChar);
    else
      return false;

    return true;
  }
```

Tras ello, empecé a usar la asignación `marshalers` en las funciones `set`, dividiendo el uso de las otras tres asignaciones. Comencé por los elementos `boolean`.

```
    private boolean setArgument(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(argChar);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(argChar);
      else
        return false;
      return true;
    }
…
    private void setBooleanArg(ArgumentMarshaler m) {
      try {
        m.set("true"); // era: booleanArgs.get(argChar).set("true");
      } catch (ArgsException e) {
      }
    }
```

Las pruebas seguían siendo correctas de modo que repetí la operación con las cadenas y los enteros. De esta manera se puede integrar parte del desagradable código de gestión de excepciones en la función `setArgument`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
```

```
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

Ya podía eliminar las tres asignaciones antiguas. Primero, debía cambiar la función `getBoolean` de:

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
  }
```

a:

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }
```

Este último cambio puede parecer sorprendente. ¿Por qué de repente decidí enfrentarme a `ClassCastException`? Por tener una serie de pruebas de

unidad y otra serie independiente de pruebas de aceptación creadas en FitNesse. Las pruebas de FitNesse garantizan que si se invoca `getBoolean` en un argumento no Booleano, se obtiene `false`. No sucede lo mismo con las pruebas de unidad. Hasta el momento, sólo había ejecutado las pruebas de unidad[93].

Este último cambio me permitió extraer otro uso de la asignación boolean:

```
private void parseBooleanSchemaElement(char elementId) {
   ArgumentMarshaler m = new BooleanArgumentMarshaler();
   booleanArgs.put(elementId, m);
   marshalers.put(elementId, m);
}
```

Y ahora ya podemos eliminar la asignación boolean.

```
public class Args {
…
   private Map<Character, ArgumentMarshaler> booleanArgs =
   new HashMap<Character, ArgmentMarshaler>();
   private Map<Character, ArgumentMarshaler> stringArgs =
   new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> intArgs =
   new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> marshalers =
   new HashMap<Character, ArgumentMarshaler>();
…
```

Tras ello, cambié los argumentos `String` e `Integer` de la misma forma y limpié los valores `boolean`.

```
private void parseBooleanSchemaElement(char elementId) {
   marshalers.put(elementId, new BooleanArgumentMarshaler());
}

private void parseIntegerSchemaElement(char elementId) {
   marshalers.put(elementId, new IntegerArgumentMarshaler());
}

private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
}
…
public String getString(char arg) {
   Args.ArgumentMarshaler am = marshalers.get(arg);
   try {
     return am null ? “” : (String) am.get();
   } catch (ClassCastException e) {
     return “”;
   }
}
```

```
public int getInt(char arg) {
   Args.ArgumentMarshaler am = marshalers.get(arg);
   try {
     return am == null ? 0 : (Integer) am.get();
   } catch (Exception e) {
     return 0;
   }
}
…
public class Args {
…
   private Map<Character, ArgumentMarshaler> stringArgs =
   new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> intArgs =
   new HashMap<Character, ArgumentMarshaler>();
   private Map<Character, ArgumentMarshaler> marshalers =
   new HashMap<Character, ArgumentMarshaler>();
…
```

Seguidamente, dispuse en línea los tres métodos `parse` ya que no servían para mucho:

```
private void parseSchemaElement(String element) throws ParseException {
   char elementId = element.charAt(0);
   String elementTail = element.substring(1);
   validateSchemaElementId(elementId);
   if (isBooleanSchemaElement(elementTail))
     marshalers.put(elementId, new BooleanArgumentMarshaler());
   else if (isStringSchemaElement(elementTail))
     marshalers.put(elementId, new StringArgumentMarshaler());
   else if (isIntegerSchemaElement(elementTail)) {
     marshalers.put(elementId, new IntegerArgumentMarshaler());
   } else {
     throw new ParseException(String.format(
     "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
   }
}
```

Es el momento de ver la estructura completa. El Listado 14-12 muestra la clase `Args` actual.

Listado 14-12
Args.java (tras la primera refactorización)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
   private String schema;
```

```
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
      new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    private enum ErrorCode {
      OK,        MISSING_STRING,        MISSING_INTEGER,        INVALID_INTEGER,
UNEXPECTED_ARGUMENT}

    public Args(String schema, String[] args) throws ParseException {
      this.schema = schema;
      this.args = args;
      valid = parse();
    }

    private boolean parse() throws ParseException {
      if (schema.length() == 0 && args.length == 0)
        return true;
      parseSchema();
      try {
        parseArguments();
      } catch (ArgsException e) {
      }
      return valid;
    }

    private boolean parseSchema() throws ParseException {
      for (String element : schema.split(","")) {
        if (element.length() > 0) {
          String trimmedElement = element.trim();
          parseSchemaElement(trimmedElement);
        }
      }
      return true;
    }

    private void parseSchemaElement(String element) throws ParseException {
      char elementId = element.charAt(0);
      String elementTail = element.substring(1);
      validateSchemaElementId(elementId);
      if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
      else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentWarshaler());
      else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
      } else {
        throw new ParseException(String.format(
```

```
        “Argument: %c has invalid format: %s.”, elementId, elementTail), 0);
      }
    }

    private void validateSchemaElementId(char elementId) throws ParseException
{
      if (!Character.isLetter(elementId)) {
        throw new ParseException(
        “Bad character:” + elementId + “in Args format: ” + schema, 0);
      }
    }

    private boolean isStringSchemaElement(String elementTail) {
      return elementTail.equals(“*”);
    }

    private boolean isBooleanSchemaElement(String elementTail) {
      return elementTail.length() == 0;
    }

    private boolean isIntegerSchemaElement(String elementTail) {
      return elementTail.equals(“#”);
    }

    private boolean parseArguments() throws ArgsException {
      for (currentArgument=0; currentArgument<args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
      }
      return true;
    }

    private void parseArgument(String arg) throws ArgsException {
      if (arg.startsWith(“-”))
        parseElements(arg);
    }

    private void parseElements(String arg) throws ArgsException {
      for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
    }

    private void parseElement(char argChar) throws ArgsException {
      if (setArgument(argChar))
        argsFound.add(argChar);
      else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
      }
    }

    private boolean setArgument(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      try {
```

```
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set (args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true");
    } catch (ArgsException e) {
    }
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
```

```
    if (schema.length() > 0)
      return = “-[” + schema + “]”;
    else
      return “”;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception(“TILT: Should not get here.”);
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format(“Could not find string parameter for -%c.”,
              errorArgumentId);
      case INVALID_INTEGER:
        return String.format(“Argument -%c expects an integer but was ‘%s’.”,
              errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format(“Could not find integer parameter for -%c.”,
              errorArgumentId);
    }
    return “”;
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer(“Argument(s) -”);
    for {char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(“ unexpected.”);

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get (arg);
    try {
      return am == null ? “” : (String) am.get();
    } catch (ClassCastException e) {
      return “”;
    }
  }
```

```
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }

  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = “”;

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
```

```
        } catch (NumberFormatException e) {
          throw new ArgsException();
        }
      }

      public Object get() {
        return intValue;
      }
    }
  }
```

Tras todo este esfuerzo, es un tanto decepcionante. La estructura ha mejorado pero todavía hay demasiadas variables en la parte superior; se mantiene un terrible caso de tipos en `setArgument`; y todas las funciones `set`. Sin mencionar el procesamiento de errores. Todavía nos queda mucho trabajo por hacer.

Mi intención es eliminar el caso de tipos de `setArgument` [G23] y que sólo incluya una invocación a `ArgumentMarshaler.set`. Para ello, debo desplazar `setIntArg`, `setStringArg` y `setBooleanArg` a las correspondientes variantes de `ArgumentMarshaler`. Pero hay un problema.

Si se fija atentamente en `setIntArg`, comprobará que usa dos variables de instancia: `args` y `currentArg`. Para desplazar `setIntArg` hasta `BooleanArgumentMarshaler`, tengo que pasar `args` y `currentArgs` como argumentos de función. Muy desagradable [F1]. Resultaría más indicado pasar un argumento y no dos. Afortunadamente, la solución es sencilla. Podemos convertir la matriz `args` en `list` y pasar `Iterator` a las funciones `set`. Para el siguiente cambio necesité diez pasos, y superar todas las pruebas tras cada uno. Pero sólo mostraremos el resultado. Debería determinar la mayoría de estos pequeños pasos.

```
public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;
```

```
    private enum ErrorCode {
      OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT
}

    public Args(String schema. String[] args) throws ParseException {
      this.schema = schema;
      argsList = Arrays.asList(args);
      valid = parse();
    }

    private boolean parse() throws ParseException {
      if (schema.length() == 0 && argsList.size() == 0)
        return true;
      parseSchema();
      try {
        parseArguments();
      } catch (ArgsException e) {
      }
      return valid;
    }
  …
    private boolean parseArguments() throws ArgsException {
      for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
      }

      return true;
    }
  …
    private void setIntArg(ArgumentMarshaler m) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        m.set(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
      }
    }

    private void setStringArg(ArgumentMarshaler m) throws ArgsException {
      try {
        m.set (currentArgument.next());
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }
```

Son pequeños cambios que conservan el funcionamiento de las pruebas. Ahora podemos empezar a desplazar las funciones `set` a las correspondientes variantes. Primero, debemos realizar el siguiente cambio en `setArgument`:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

Es un cambio importante ya que queremos eliminar totalmente la cadena `if-else`. Por tanto, debemos excluir la condición de error.

Ya podemos empezar a desplazar las funciones `set`. La función `setBooleanArg` es trivial, de modo que la prepararemos en primer lugar. El objetivo es cambiar la función `setBooleanArg` para redirigirla a `BooleanArgumentMarshaler`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m, currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);

  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
```

```
    return true;
  }
  ---
  private void setBooleanArg (ArgumentMarshaler m,
                Iterator<String> currentArgument)
                 throws ArgsException {
    try {
      m.set("true");
    catch (ArgsException e) {
    }
  }
```

¿No acabamos de incluir el procesamiento de excepciones? Añadir elementos para después excluirlos es habitual en los procesos de refactorización. Los pasos reducidos y la necesidad de que las pruebas sigan siendo correctas implican que los elementos cambien de posición. La refactorización es como resolver el cubo de Rubik. Se necesitan muchos pasos pequeños para lograr un objetivo mayor. Cada paso habilita el siguiente.

Se preguntará por qué pasamos `iterator` si `setBooleanArg` no lo necesita. Pues porque `setIntArg` y `setStringArg` sí. Y como el objetivo es implementar las tres funciones a través de un método abstracto en `ArgumentMarshaller`, es necesario pasarlo a `setBooleanArg`.

Ahora `setBooleanArg` no sirve de nada. Si hubiera una función `set` en `ArgumentMarshaler`, podríamos invocarla directamente. Es el momento de crear dicha función. El primer paso consiste en añadir el nuevo método abstracto a `ArgumentMarshaler`.

```
  private abstract class ArgumentMarshaler {
    public abstract void set(Iterator<String> currentArgument)
                throws ArgsException;
    public abstract void set (String s) throws ArgsException;
    public abstract Object get();
  }
```

Evidentemente, esto afecta a todas las variantes, de modo que implementamos el nuevo método en cada una.

```
  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      booleanValue = true;
    }

    public void set(String s) {
```

```
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = “”;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

Y ahora ya podemos eliminar `setBooleanArg`:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
```

```
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

Las pruebas siguen siendo satisfactorias y la función `set` se implementa en Boolean `ArgumentMarshaler`. Podemos repetir la operación con las cadenas y los enteros.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
      m.set(currentArgument);
      else if (m instanceof IntegerArgumentMarshaler)
      m.set(currentArgument);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
  ---
  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = “”;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

    public void set(String s){
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
```

```
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        set(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
      }
    }

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Y el golpe de gracia: se elimina el caso de tipos.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;

    try {
      m.set(currentArgument);
      return true;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
  }
```

Ya podemos deshacernos de las funciones de `IntegerArgumentMarshaler` y limpiar el resto.

```
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0

    public void set (Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
```

```
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

También podemos convertir `ArgumentMarshaler` en una interfaz.

```
  private interface ArgumentMarshaler {
    void set (Iterator<String> currentArgument) throws ArgsException;
    Object get();
  }
```

Veamos ahora lo sencillo que resulta añadir un nuevo tipo de argumento a la estructura. Apenas necesitaremos cambios y los que apliquemos tendrán que ser aislados. En primer lugar, añadimos un nuevo caso de prueba para comprobar que el argumento `double` funciona correctamente:

```
  public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x","42.3"});
    assertTrue(args.isValid());
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
  }
```

Limpiamos el código de análisis de esquemas y añadimos la detección `##` para el tipo de argumento `double`.

```
  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
```

```
    else
      throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
```

Seguidamente, creamos la clase `DoubleArgumentMarshaler`.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
}
```

Esto nos obliga a añadir un nuevo código de error (`ErrorCode`).

```
private enum ErrorCode {
   OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
   MISSING_DOUBLE, INVALID_DOUBLE}
```

Y necesitamos una función `getDouble`.

```
public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am = null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
}
```

Y todas las pruebas son correctas. Ha sido sencillo. A continuación comprobamos que el procesamiento de errores funciona correctamente. El siguiente caso de prueba comprueba que se declare un error si se proporciona

una cadena que no se puede analizar a un argumento `##`.

```
public void testInvalidDouble() throws Exception {
   Args args = new Args("x##", new String[] {"-x", "Forty two"});
   assertFalse(args.isValid());
   assertEquals(0, args.cardinality());
   assertFalse(args.has('x'));
   assertEquals(0, args.getInt('x'));
   assertEquals("Argument -x expects a double but was 'Forty two'.",
          args.errorMessage());
}
---
public String errorMessage() throws Exception {
   switch (errorCode) {
     case OK:
       throw new Exception("TILT: Should not get here.");
     case UNEXPECTED_ARGUMENT:
       return unexpectedArgumentMessage();
     case MISSING_STRING:
       return String.format("Could not find string parameter for -%c.",
              errorArgumentId);
     case INVALID_INTEGER:
       return String.format("Argument -%c expects an integer but was '%s'.",
              errorArgumentId, errorParameter);
     case MISSING_INTEGER:
       return String.format("Could not find integer parameter for -%c.",
              errorArgumentId);
     case INVALID_DOUBLE:
       return String.format("Argument -%c expects a double but was '%s'.",
              errorArgumentId, errorParameter);
     case MISSING_DOUBLE:
       return String.format("Could not find double parameter for -%c",
              errorArgumentId);
   }
   return "";
}
```

Y las pruebas son satisfactorias. La siguiente prueba garantiza que se detecte correctamente la ausencia de un argumento `double`.

```
public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[]{"-x"});
    assertFalse (args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
             args.errorMessage());
}
```

Es correcto. La incluimos para que el ejemplo resulte más completo.

El código de excepciones no es atractivo y no pertenece realmente a la

clase `Args`. También generamos `ParseException`, que no nos pertenece. Por ello, combinamos todas las excepciones en una única clase `ArgsException` y la incluimos en su propio módulo.

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    public ArgsException() {}

    public ArgsException(String message) { super(message); }

    public enum ErrorCode {
      OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
      MISSING_DOUBLE, INVALID_DOUBLE }
}
…
public class Args {
   …
   private char errorArgumentId = '\0';
   private String errorParameter = "TILT";
   private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
   private List<String> argsList;

   public Args(String schema, String[] args) throws ArgsException {
     this.schema = schema;
     argsList = Arrays.asList(args);
     valid = parse();
   }

   private boolean parse() throws ArgsException {
     if (schema.length() == 0 && argsList.size() == 0)
       return true;
     parseSchema();
     try {
       parseArguments();
     } catch (ArgsException e) {
     }
     return valid;
   }

   private boolean parseSchema() throws ArgsException {
     …
   }

   private void parseSchemaElement(String element) throws ArgsException {
     …
     else
       throw new ArgsException(
         String.format("Argument: %c has invalid format: %s.",
              elementId, elementTail));
   }
```

```
  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(
        "Bad character:" + elementId + "in Args format: " + schema);
    }
  }

  …

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
    argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

…

  private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set (Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
```

```
      }
    }

    public Object get() {
      return intValue;
    }
  }

  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
        throw new ArgsException);
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
}
```

Muy bien. Ahora, `Args` solamente genera `ArgsException`. Al desplazar `ArgsException` a un módulo propio, podemos añadir a dicho módulo gran parte del código de error y extraerlo del módulo `Args`. Es una posición natural y evidente para incluir todo el código y nos permitirá limpiar posteriormente el módulo `Args`.

Ya hemos separado el código de excepciones y de error del módulo `Args` (véanse los listados del 14-13 al 14-16). Para ello realizamos una serie de 30 pasos mínimos y las pruebas fueron satisfactorias entre todos ellos.

Listado 14-13
ArgsTest.java.

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;
```

```
public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args(“”, new String[0]);
    assertEquals(0, args.cardinality());
  }

  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args(“”, new String[]{“-x”});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
      e.getErrorCode());
      assertEquals(‘x’, e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args(“”, new String[]{“-x”, “-y”});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
      e.getErrorCode());
      assertEquals(‘x’, e.getErrorArgumentId());
    }

  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args(“*”, new String[]{});
      fail(“Args constructor should have thrown exception”);
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
             e.getErrorCode());
      assertEquals(‘*’, e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
      new Args(“f~”, new String[]{});
      fail(“Args constructor should have throws exception”);
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
      assertEquals(‘f’, e.getErrorArgumentId());
    }
  }

  public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args(“x”, new String []{“-x”});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean(‘x’));
```

```
    }

    public void testSimpleStringPresent() throws Exception {
      Args args = new Args("x*", new String[]{"-x", "param"});
      assertEquals(1, args.cardinality());
      assertTrue(args.has('x'));
      assertEquals("param", args.getString('x'));
    }

    public void testMissingStringArgument() throws Exception {
      try {
        new Args("x*", new String[]{"-x"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
      }
    }

    public void testSpacesInFormat() throws Exception {
      Args args = new Args("x, y", new String[]{"-xy"});
      assertEquals(2, args.cardinality());
      assertTrue(args.has('x'));
      assertTrue(args.has('y'));
    }

    public void testSimpleIntPresent() throws Exception {
      Args args = new Args("x#", new String[]{"-x", "42"});
      assertEquals(1, args.cardinality());
      assertTrue(args.has('x'));
      assertEquals(42, args.getInt('x'));
    }

    public void testInvalidInteger() throws Exception {
      try {
        new Args("x#", new String[] {"-x", "Forty two"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_INTEGER,
e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
      }
    }

    public void testMissingInteger() throws Exception {
      try {
        new Args("x#", new String[]{"-x"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_INTEGER,
e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
      }
    }
```

```
    public void testSimpleDoublePresent() throws Exception {
      Args args = new Args("x##", new String[]("-x", "42.3"});
      assertEquals(1, args.cardinality());
      assertTrue(args.has('x'));
      assertEquals(42.3, args.getDouble('x'), .001);
    }

    public void testInvalidDouble() throws Exception {
      try {
        new Args("x##", new String []{"-x", "Forty two"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
      }
    }

    public void testMissingDouble() throws Exception {
      try {
        new Args("x##", new String[]{"-x"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
      }
    }
  }
```

Listado 14-14
ArgsExceptionTest.java.

```
  public class ArgsExceptionTest extends TestCase {
    public void testUnexpectedMessage() throws Exception {
      ArgsException e =
        new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
              'x', null);
      assertEquals("Argument -x unexpected.", e.errorMessage());
    }

    public void testMissingStringMessage() throws Exception {
      ArgsException e = new
ArgsException(ArgsException.ErrorCode.MISSING_STRING,
              'x', null);
      assertEquals("Could not find string parameter for –x.",
e.errorMessage());
    }

    public void testInvalidIntegerMessage() throws Exception {
      ArgsException e =
        new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
```

```
                ‘x’, “Forty two”);
      assertEquals(“Argument –x expects an integer but was ‘Forty two’.”,
                e.errorMessage());
     }

     public void testMissingIntegerMessage() throws Exception {
       ArgsException e =
         new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, ‘x’, null);
       assertEquals(“Could    not    find    integer    parameter    for    -x.”,
e.errorMessage());
     }

     public void testInvalidDoubleMessage() throws Exception {
       ArgsException                  e                  =                  new
ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                ‘x’, “Forty two”);
       assertEquals(“Argument -x expects a double but was ‘Forty two’.”,
                e.errorMessage());
     }

     public void testMissingDoubleMessage() throws Exception {
       ArgsException                  e                  =                  new
ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                ‘x’, null);
       assertEquals(“Could    not    find    double    parameter    for    -x.”,
e.errorMessage());
     }
   }
```

Listado 14-15
ArgsException.java.

```
   public class ArgsException extends Exception {
     private char errorArgumentId = ‘\0’;
     private String errorParameter = “TILT”;
     private ErrorCode errorCode = ErrorCode.OK;

     public ArgsException() {}

     public ArgsException(String message) {super(message);}

     public ArgsException(ErrorCode errorCode) {
       this.errorCode = errorCode;
     }

     public ArgsException(ErrorCode errorCode, String errorParameter) {
       this.errorCode = errorCode;
       this.errorParameter = errorParameter;
     }

     public ArgsException(ErrorCode errorCode, char errorArgumentId,
       String errorParameter) {
       this.errorCode = errorCode;
```

```
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
              errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
              errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
              errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
              errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
              errorArgumentId);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
```

```
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Listado 14-16
Args.java.

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
  }

  private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(“,”)) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals(“*”))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals(“#”))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals(“##”))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
             elementId, elementTail);
  }
```

```
    private void validateSchemaElementId(char elementId) throws ArgsException
{
      if (!Character.isLetter(elementId)) {
        throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
               elementId, null);
      }
    }

    private void parseArguments() throws ArgsException {
      for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
      }
    }

    private void parseArgument(String arg) throws ArgsException {
      if (arg.startsWith("-"))
        parseElements(arg);
    }

    private void parseElements(String arg) throws ArgsException {
      for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
    }

    private void parseElement(char argChar) throws ArgsException {
      if (setArgument(argChar))
        argsFound.add(argChar);
      else {
        throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
               argChar, null);
      }
    }

    private boolean setArgument(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      if (m == null)
        return false;
      try {
        m.set(currentArgument);
        return true;
      } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
      }
    }

    public int cardinality() {
      return argsFound.size();
    }

    public String usage() {
      if (schema.length() > 0)
        return "-[" + schema + "]";
```

```
      else
        return “”;
    }

    public boolean getBoolean(char arg) {
      ArgumentMarshaler am = marshalers.get(arg);
      boolean b = false;
      try {
        b = am != null && (Boolean) am.get();
      } catch (ClassCastException e) {
        b = false;
      }
      return b;
    }

    public String getString(char arg) {
      ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? “” : (String) am.get();
      } catch (ClassCastException e) {
        return “”;
      }
    }

    public int getInt(char arg) {
      ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? 0 : (Integer) am.get();
      } catch (Exception e) {
        return 0;
      }
    }

    public double getDouble(char arg) {
      ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? 0 : (Double) am.get();
      } catch (Exception e) {
        return 0.0;
      }
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }
  }
```

La mayoría de los cambios realizados en la clase `Args` han sido eliminaciones. Gran parte del código se extrajo de `Args` y se añadió a `ArgsException`. Perfecto. También cambiamos todos los elementos `ArgumentMarshaller` a sus propios archivos. Mejor todavía.

El diseño de *software* correcto se basa gran parte en las particiones, en crear zonas adecuadas para incluir distintos tipos de código. Esta separación hace que el código sea más fácil de entender y mantener.

Especialmente interesante es el método `errorMessage` de `ArgsException`. Incumple claramente el SRP al incluir el formato de mensajes de error en `Args`. `Args` debe centrarse en el procesamiento de argumentos, no en el formato de los mensajes de error. Sin embargo, ¿realmente tiene sentido incluir el código de formato de mensajes de error en `ArgsException`?

Francamente es un compromiso. Los usuarios que no deseen los mensajes de error proporcionados por `ArgsException` tendrán que crear los suyos propios, pero la utilidad de mensajes de error ya preparados es evidente.

Ya debería haberse dado cuenta de la distancia recorrida con respecto a la solución mostrada al inicio del capítulo. Las transformaciones finales puede examinarlas por su cuenta.

Conclusión

No basta con que el código funcione. El código que funciona suele ser incorrecto. Los programadores que se conforman con código funcional no se comportan de forma profesional. Puede que teman que no tienen tiempo para mejorar la estructura y el diseño del código, pero discrepo. No hay nada que afecte más negativamente a un proyecto de desarrollo que el código incorrecto. Los plazos incorrectos se pueden rehacer y los requisitos equivocados se pueden volver a definir. La dinámica incorrecta de un equipo se puede reparar pero el código incorrecto se corrompe y se convierte en una carga que arrastra al equipo completo. He visto equipos dominados por el desastre que han generado y que han dominado su destino.

Evidentemente, el código incorrecto se puede limpiar pero resulta muy costoso. Cuando el código se corrompe los módulos se insinúan unos a otros y generan multitud de dependencias ocultas y entrelazadas. La localización y división de dependencias antiguas es una tarea larga y complicada. Por otra parte, resulta relativamente sencillo mantener código limpio. Si comete un error en un módulo, es más fácil limpiarlo directamente. Mejor todavía, si

cometió un error hace cinco minutos, es muy fácil limpiarlo ahora.

Por tanto, la solución consiste en mantener el código limpio y sencillo siempre que se pueda y no dejar que llegue a corromperse.

15
Aspectos internos de JUnit

JUnit es una de las estructuras de Java más conocidas. De concepción sencilla, definición precisa y documentación elegante. ¿Y su código? En este capítulo analizaremos un ejemplo extraído de la estructura JUnit.

La estructura JUnit

JUnit ha tenido muchos autores, comenzando por Kent Beck y Eric Gamma en un vuelo a Atlanta. Kent quería aprender Java y Eric quería saber más sobre la estructura de pruebas Smalltalk de Kent. “¿Hay algo más natural que

dos fanáticos enciendan sus portátiles y empiecen a escribir código?”[94] Tras tres horas de trabajo de altura, habían creado los fundamentos de JUnit.

El módulo que analizaremos es un inteligente fragmento de código que permite identificar errores de comparación de cadenas. El nombre del módulo es `ComparisonCompactor`. Dadas dos cadenas diferentes, como `ABCDE` y `ABXDE`, muestra la diferencia entre ambas generando una cadena como `<…B[X]D…>`.

Podríamos explicarlo más, pero los casos de prueba son mejores. Fíjese en el Listado 15-1 para comprender los requisitos de este módulo. Analice la estructura de las pruebas. ¿Podrían ser más simples o más evidentes?

Listado 15-1
ComparisonCompactorTest.java.

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

   public void testMessage() {
     String failure= new ComparisonCompactor(0, “b”, “c”).compact(“a”);
     assertTrue(“a expected:<[b]> but was:<[c]>”.equals(failure));
   }

   public void testStartSame() {
     String failure= new ComparisonCompactor(1, “ba”, “bc”).compact(null);
     assertEquals(“expected:<b[a]> but was:<b[c]>”, failure);
   }

   public void testEndSame() {
     String failure= new ComparisonCompactor(1, “ab”, “cb”).compact(null);
     assertEquals(“expected:<[a]b> but was:<[c]b>”, failure);
   }

   public void testSame() {
     String failure= new ComparisonCompactor(1, “ab”, “ab”).compact(null);
     assertEquals(“expected:<ab> but was:<ab>”, failure);
   }

   public void testNoContextStartAndEndSame() {
     String failure= new ComparisonCompactor(0, “abc”, “adc”).compact(null);
     assertEquals(“expected:<…[b]…> but was:<…[d]…>”, failure);
   }
```

```
public void testStartAndEndContext() {
  String failure= new ComparisonCompactor(1, “abc”, “adc”).compact(null);
  assertEquals(“expected:<a[b]c> but was:<a[d]c>”, failure);
}

public void testStartAndEndContextWithEllipses() {
  String failure=
    new ComparisonCompactor(1, “abcde”, “abfde”).compact(null);
  assertEquals(“expected:<…b[c]d…> but was:<…b[f]d…>”, failure);
}

public void testComparisonErrorStartSameComplete() {
  String failure= new ComparisonCompactor(2, “ab”, “abc”).compact(null);
  assertEquals(“expected:<ab[]> but was:<ab[c]>”, failure);
}

public void testComparisonErrorEndSameComplete() {
  String failure= new ComparisonCompactor(0, “bc”, “abc”).compact(null);
  assertEquals(“expected:<[]…> but was:<[a]…>”, failure);
}

public void testComparisonErrorEndSameCompleteContext() {
  String failure= new ComparisonCompactor(2, “bc”, “abc”).compact(null);
  assertEquals(“expected:<[]bc> but was:<[a]bc>”, failure);
}

public void testComparisonErrorOverlapingMatches() {
  String failure= new ComparisonCompactor(0, “abc”, “abbc”).compact(null);
  assertEquals(“expected:<…[]…> but was:<…[b]…>”, failure);
}

public void testComparisonErrorOverlapingMatchesContext() {
  String failure= new ComparisonCompactor(2, “abc”, “abbc”).compact(null);
  assertEquals(“expected:<ab[]c> but was:<ab[b]c>”, failure);
}
public void testComparisonErrorOverlapingMatches2() {
  String failure= new ComparisonCompactor(0, “abcdde”,
    “abcde”).compact(null);
  assertEquals(“expected:<…[d]…> but was:<…[]…>”, failure);
}

public void testComparisonErrorOverlapingMatches2Context() {
  String failure=
    new ComparisonCompactor(2, “abcdde”, “abcde”).compact(null);
  assertEquals(“expected:<…cd[d]e> but was:<…cd[]e>”, failure);
}

public void testComparisonErrorWithActualNull() {
  String failure= new ComparisonCompactor(0, “a”, null).compact(null);
  assertEquals(“expected:<a> but was:<null>”, failure);
}

public void testComparisonErrorWithActualNullContext() {
  String failure= new ComparisonCompactor(2, “a”, null).compact(null);
```

```
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testBug609972() {
    String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
    assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
  }
}
```

Realicé un análisis de alcance de código en `ComparisonCompactor` con estas pruebas. El código se cubre en un 100 por 100. Cada línea, cada instrucción `if` y cada bucle `for` se ejecuta con las pruebas. De este modo sé que el código funciona y sus autores me merecen el mayor de los respetos.

El código `ComparisonCompactor` se reproduce en el Listado 15-2. Examínelo. Creo que lo encontrará bien distribuido, razonablemente expresivo y estructuralmente sencillo. Cuando termine, lo diseccionaremos.

Listado 15-2
ComparisonCompactor.java (Original).

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "…";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;

  public ComparisonCompactor(int contextLength,
            String expected,
            String actual) {
    fContextLength = contextLength;
    fExpected = expected;
```

```
    fActual = actual;
  }

  public String compact(String message) {
    if (fExpected == null || fActual == null || areStringsEqual())
      return Assert.format(message, fExpected, fActual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(fExpected);
    String actual = compactString(fActual);
    return Assert.format(message, expected, actual);
  }

  private String compactString(String source) {
    String result = DELTA_START +
             source.substring(fPrefix, source.length() -
             fSuffix + 1) + DELTA_END;
    if (fPrefix > 0)
      result = computeCommonPrefix() + result;
    if (fSuffix > 0)
      result = result + computeCommonSuffix();
    return result;
  }

  private void findCommonPrefix() {
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
      if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
        break;
    }
  }

  private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (;
        actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
        actualSuffix--, expectedSuffix--) {
      if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
        break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
  }

  private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : “”) +
          fExpected.substring(Math.max(0, fPrefix - fContextLength),
             fPrefix);
  }

  private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
             fExpected.length());
```

```
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
            (fExpected.length() - fSuffix + 1 < fExpected.length() -
            fContextLength ? ELLIPSIS : “”);
  }

  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
  }
}
```

Puede que tenga varias quejas sobre el módulo. Incluye expresiones extensas y extraños elementos `+1`. Pero en general, está bastante bien. Después de todo, podría haber sido como el Listado 15-3.

Listado 15-3
ComparisonCompator.java (defactorizado)

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
```

```
    }

    private String compactString(String s) {
      String result =
        "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
      if (pfx > 0)
        result = (pfx > ctxt ? "…" : "") +
          s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
      if (sfx > 0) {
        int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
        result = result + (s1.substring(s1.length() - sfx + 1, end) +
          (s1.length() - sfx + 1 < s1.length() - ctxt ? "…" : ""));
      }
      return result;
    }

  }
```

Aunque los autores hicieron un buen trabajo con este módulo, la *Regla del Boy Scout*[95] muestra que podrían haberlo dejado más limpio de lo que se encontró. ¿Cómo podemos mejorar el código original del Listado 15-2? Lo primero que no necesitamos es el prefijo `f` de las variables miembro [N6]. Los entornos actuales hacen que este tipo de código de ámbito sea redundante, por lo que eliminaremos todas las `f`.

```
  private int contextLength;
  private String expected;
  private String actual;
  private int prefix;
  private int suffix;
```

Tras ello, tenemos una condicional sin encapsular al inicio de la función compact [G28].

```
  public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())
      return Assert.format(message, expected, actual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
  }
```

Es necesario encapsular esta condicional para que nuestra intención sea más clara. Por tanto, extraemos un método que la explique.

```
  public String compact(String message) {
    if (shouldNotCompact())
      return Assert.format(message, expected, actual);
```

```
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
  }

  private boolean shouldNotCompact() {
     return expected == null || actual == null || areStringsEqual();
  }
```

En la función `compact`, `this.expected` y `this.actual` no son demasiado relevantes. Sucede al cambiar el nombre de `fExpected` por `expected`. ¿Por qué esta función tiene variables con los mismos nombres que las variables miembro? ¿No representan cosas diferentes?[N4]. Los nombres deben ser exclusivos.

```
  String compactExpected = compactString(expected);
  String compactActual = compactString(actual);
```

Los negativos son más difíciles de entender que los positivos [G29]. Por ello, invertimos esa instrucción `if` para cambiar el sentido de la condicional.

```
  public String compact(String message) {
    if (canBeCompacted()) {
      findCommonPrefix();
      findCommonSuffix();
      String compactExpected = compactString(expected);
      String compactActual = compactString(actual);
      return Assert.format(message, compactExpected, compactActual);
    } else {
      return Assert.format(message, expected, actual);
    }
  }

  private boolean canBeCompacted() {
    return expected != null && actual != null && !areStringsEqual();
  }
```

El nombre de la función es extraño [N7]. Aunque compacta las cadenas, puede que lo haga si `canBeCompacted` devuelve `false`. Al asignar el nombre `compact` a esta función se oculta el efecto secundario de la comprobación de errores. Además, la función devuelve un mensaje con formato, no sólo las cadenas compactadas. Por tanto, el nombre de la función debería ser `formatCompactedComparison`. De esta forma, se lee mejor junto al argumento de la función:

```
public String formatCompactedComparison(String message) {
```

El cuerpo de la instrucción `if` es donde se realiza la verdadera compactación de las cadenas. Debemos extraerlo como método con el nombre `compactExpectedAndActual`. Sin embargo, queremos que la función `formatCompactedComparison` realice todo el formato. La función `compact…` sólo debe realizar la compactación [G30], de modo que la dividimos de esta forma:

```
…
   private String compactExpected;
   private String compactActual;
…

   public String formatCompactedComparison(String message) {
     if (canBeCompacted()) {
       compactExpectedAndActual();
       return Assert.format(message, compactExpected, compactActual);
     } else {
       return Assert.format(message, expected, actual);
     }
   }

   private void compactExpectedAndActual() {
     findCommonPrefix();
     findCommonSuffix();
     compactExpected = compactString(expected);
     compactActual = compactString(actual);
   }
```

Para ello, hemos tenido que ascender `compactExpected` y `compactActual` a variables miembro. No me gusta la forma en que las dos últimas líneas de la nueva función devuelven variables pero las dos primeras no lo hacen. No utilizan convenciones coherentes [G11]. Debemos cambiar `findCommonPrefix` y `findCommonSuffix` para que devuelvan los valores de prefijo y sufijo.

```
private void compactExpectedAndActual() {
   prefixIndex = findCommonPrefix();
   suffixIndex = findCommonSuffix();
   compactExpected = compactString(expected);
   compactActual = compactString(actual);
}

private int findCommonPrefix() {
   int prefixIndex = 0;
   int end = Math.min(expected.length(), actual.length());
   for (; prefixIndex < end; prefixIndex++) {
```

```
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
    }
    return prefixIndex;
  }

  private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
    }
    return expected.length() - expectedSuffix;
  }
```

También debemos cambiar los nombres de las variables miembro para que sean más preciosas [N1], ya que en el fondo son índices.

Al examinar `findCommonSuffix` vemos una conexión temporal oculta [G31]; depende de que `prefixIndex` se calcule por `findCommonPrefix`. Si estas dos funcione se invocan de forma desordenada, la sesión de depuración posterior sería complicada. Por ello, para mostrar esta combinación temporal, haremos que `findCommonSuffix` acepte `prefixIndex` como argumento.

```
  private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }

  private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
    }
    return expected.length() - expectedSuffix;
  }
```

No me convence del todo. El hecho de pasar `prefixIndex` como argumento es un tanto arbitrario [G32]. Permite establecer el orden pero no explica la necesidad del mismo. Otro programador podría deshacer esta operación ya que no se indica en ningún momento para qué sirve el parámetro. Adoptemos un enfoque diferente.

```
private void compactExpectedAndActual() {
   findCommonPrefixAndSuffix();
   compactExpected = compactString(expected);
   compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
   findCommonPrefix();
   int expectedSuffix = expected.length() - 1;
   int actualSuffix = actual.length() - 1;
   for (;
       actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--
       ) {
     if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
       break;
   }
   suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
   prefixIndex = 0;
   int end = Math.min(expected.length(), actual.length());
   for (; prefixIndex < end; prefixIndex++)
     if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
       break;
}
```

Devolvemos `findCommonPrefix` y `findCommonSuffix` a su posición original, cambiamos el nombre de `findCommonSuffix` por `findCommonPrefixAndSuffix` y hacemos que invoque `findCommonPrefix` antes de hacer nada más. De ese modo se establece la naturaleza temporal de ambas funciones de forma más evidente que antes. Además, se muestra el mínimo atractivo de `findCommonPrefixAndSuffix`, que limpiaremos a continuación:

```
private void findCommonPrefixAndSuffix() {
   findCommonPrefix();
   int suffixLength = 1;
   for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
     if (charFromEnd(expected, suffixLength) !=
         charFromEnd(actual, suffixLength))
       break;
   }
   suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
   return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
```

```
    return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
  }
```

Mucho mejor. Muestra que `suffixIndex` es en realidad la longitud del sufijo y que su nombre no es correcto. Lo mismo sucede con `prefixIndex`, aunque en ese caso índice y longitud son sinónimos. Incluso así, es más coherente usar `length`. El problema es que la variable `suffixIndex` no es de base cero, sino de base `1` y no es una verdadera longitud. Éste es el motivo de la abundancia de `+1` en `computeCommonSuffix` [G33]. Lo corregimos. En el Listado 15-4 puede ver el resultado.

Listado 15-4
ComparisonCompactor.java (versión intermedia).

```
  public class ComparisonCompactor {
  …
    private int suffixLength;
  …
    private void findCommonPrefixAndSuffix() {
      findCommonPrefix();
      suffixLength = 0;
      for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
        if (charFromEnd(expected, suffixLength) !=
            charFromEnd(actual, suffixLength))
          break;
      }
    }

    private char charFromEnd(String s, int i) {
      return s.charAt(s.length() - i -1);
    }

    private boolean suffixOverlapsPrefix(int suffixLength) {
      return actual.length() - suffixLength <= prefixLength ||
        expected.length() - suffixLength <= prefixLength;
    }

  …
    private String compactString(String source) {
      String result =
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +
        DELTA_END;
      if (prefixLength > 0)
        result = computeCommonPrefix() + result;
      if (suffixLength > 0)
```

```
        result = result + computeCommonSuffix();
      return result;
    }

  …
    private String computeCommonSuffix() {
      int end = Math.min(expected.length() - suffixLength +
        contextLength, expected.length()
      );
      return
        expected.substring(expected.length() - suffixLength, end) +
        (expected.length() - suffixLength <
          expected.length() - contextLength ?
            ELLIPSIS : “”);
    }
```

Cambiamos `+1` en computeCommonSuffix por un `-1` en charFromEnd, donde tiene sentido, y dos operadores `<=` en `suffixOverlapsPrefix`, totalmente correctos. De este modo podemos cambiar el nombre de `suffixIndex` por `suffixLength`, lo que mejora considerablemente la legibilidad del código.

Pero hay un problema. Al comenzar a eliminar los `+1`, me fijé en la siguiente línea de `compactstring`:

```
  if (suffixLength > 0)
```

Búsquela en el Listado 15-4. Como ahora `suffixLength` es una unidad menos que antes, debemos cambiar el operador `>` por `>=`. Pero eso no tiene sentido. Ahora sí. Significa que no tenía sentido antes y que seguramente fuera un error. Bueno, no del todo. Tras un análisis detallado, vemos que ahora la instrucción `if` impide que se añada un sufijo de longitud cero. Antes de realizar el cambio, la instrucción `if` no funcionaba ya que `suffixIndex` nunca podía ser menos de uno.

Esto cuestiona ambas instrucciones `if` en `compactString`. Parece como si se pudieran eliminar. Por ello, las comentamos y ejecutamos las pruebas. Satisfactorias. Reestructuremos `compactString` para eliminar las instrucciones `if` sobrantes y simplificar la función [G9].

```
  private String compactString(String source) {
    return
      computeCommonPrefix() +
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
```

```
      DELTA_END +
      computeCommonSuffix();
  }
```

Mucho mejor. Ahora vemos que la función `compactString` simplemente combina los fragmentos. Probablemente lo podríamos limpiar más, en pequeñas operaciones, pero en lugar de desarrollar el resto de los cambios, mostraremos el resultado final en el Listado 15-5.

Listado 15-5
ComparisonCompactor.java (versión definitiva).

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "…";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
    int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual);
  }

  private boolean shouldBeCompacted() {
    return !shouldNotBeCompacted();
  }

  private boolean shouldNotBeCompacted() {
    return expected == null ||
```

```
        actual == null ||
        expected.equals(actual);
    }

    private void findCommonPrefixAndSuffix() {
      findCommonPrefix();
      suffixLength = 0;
      for (; !suffixOverlapsPrefix(); suffixLength++) {
        if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength)
        )
          break;
      }
    }

    private char charFromEnd(String s, int i) {
      return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix() {
      return actual.length() - suffixLength <= prefixLength ||
        expected.length() - suffixLength <= prefixLength;
    }

    private void findCommonPrefix() {
      prefixLength = 0;
      int end = Math.min(expected.length(), actual.length());
      for (; prefixLength < end; prefixLength++)
        if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
          break;
    }

    private String compact(String s) {
      return new StringBuilder()
        .append(startingEllipsis())
        .append(startingContext())
        .append(DELTA_START)
        .append(delta(s))
        .append(DELTA_END)
        .append(endingContext())
        .append(endingEllipsis())
        .toString();
    }

    private String startingEllipsis() {
      return prefixLength > contextLength ? ELLIPSIS : “”;
    }

    private String startingContext() {
      int contextStart = Math.max(0, prefixLength - contextLength);
      int contextEnd = prefixLength;
      return expected.substring(contextStart, contextEnd);
    }

    private String delta(String s) {
```

```
      int deltaStart = prefixLength;
      int deltaEnd = s.length() - suffixLength;
      return s.substring(deltaStart, deltaEnd);
    }

    private String endingContext() {
      int contextStart = expected.length() - suffixLength;
      int contextEnd =
        Math.min(contextStart + contextLength, expected.length());
      return expected.substring(contextStart, contextEnd);
    }

    private String endingEllipsis() {
      return (suffixLength > contextLength ? ELLIPSIS : “”);
    }
  }
```

Bastante atractivo. El módulo se separa en un grupo de funciones de análisis y otro grupo de funciones de síntesis. Se ordenan topológicamente para que la definición de cada función aparezca donde realmente se usa. Primero se muestran las funciones de análisis y después las de síntesis. Si se fija atentamente, verá que he invertido algunas de las decisiones adoptadas inicialmente. Por ejemplo, he añadido algunos métodos extraídos a `formatCompactedComparison` y he modificado el sentido de la expresión `shouldNotBeCompacted`. Es algo habitual. A menudo, un cambio de refactorización lleva a otro que a su vez lleva a deshacer el primero. La refactorización es un proceso iterativo de ensayo y error, e inevitablemente converge en algo que consideramos digno de un profesional.

Conclusión

Hemos cumplido la Regla del Boy Scout. Hemos dejado este módulo más limpio de como lo encontramos. No es que no estuviera limpio originalmente, ya que el trabajo de sus autores es excelente, pero cualquier módulo se puede mejorar y es nuestra responsabilidad dejar el código más limpio de lo que lo encontramos.

16
Refactorización de SerialDate

Si visita `http://www.jfree.org/jcommon/index.php`, encontrará la biblioteca JCommon. En su interior incluye el paquete `org.jfree.date` y, dentro de éste, la clase `SerialDate`. Vamos a analizar esta clase.

El autor de `SerialDate` es David Gilbert. David es un programador experimentado y competente. Como veremos, muestra un elevado grado de profesionalidad y disciplina en su código. En lo que a éste respecta, se puede

considerar de calidad. Y voy a despedazarlo.

No es un acto de malicia, ni tampoco me creo mejor que David y con el derecho de juzgar su código. De hecho, si leyera algún código que he creado, seguramente tendría que objetar muchos aspectos del mismo. No es un acto de arrogancia. Lo que voy a hacer no es más que una revisión profesional, algo con lo que todos deberíamos sentirnos cómodos y algo que deberíamos agradecer si alguien lo hace. A través de las críticas es como podemos aprender, como hacen médicos, pilotos o abogados. Y nosotros, como programadores, también tenemos que aprender a hacerlo.

Otra cosa más sobre David Gilbert: es más que un buen programador. David ha tenido el valor y la buena voluntad de ofrecer este código al público gratuitamente, para que cualquiera pueda usarlo y examinarlo. ¡Bien hecho!

`SerialDate` (véase el Listado B-1) es una clase que representa una fecha en Java. ¿Para qué se necesita una clase que represente una fecha si Java ya cuenta con `java.util.Date` y `java.util.Calendar`, entre otras? El autor creó esta clase como respuesta a un problema que yo también he padecido. El comentario de su Javadoc inicial (línea 67) lo explica. Podríamos cuestionar su intención, pero yo también he sufrido este problema y se agradece una clase sobre fechas en lugar de horas.

Primero, conseguir que funcione

Hay varias pruebas de unidad en la clase `SerialDateTests` (véase el Listado B-2). Todas son satisfactorias. Desafortunadamente, un rápido examen demuestra que no comprueban todos los aspectos [T1]. Por ejemplo, al realizar una búsqueda de usos en el método `MonthCodeToQuarter` (línea 334) se indica que no se usa [F4]. Por lo tanto, las pruebas de unidad no lo comprueban. Por ello, recurrí a Clover para ver el alcance de las pruebas de unidad. Clover indicó que las pruebas sólo ejecutan 91 de las 185 instrucciones ejecutables de `SerialDate` (aproximadamente el 50 por 100) [T2]. El mapa de alcance muestra grandes fragmentos de código sin ejecutar desperdigados por la clase.

Mi objetivo era comprender la clase y refactorizarla, algo que no podía

lograr sin una cobertura mayor de las pruebas. Por ello diseñé mi propia *suite* de pruebas de unidad independientes (véase el Listado B-4).

Si se fija en las pruebas, comprobará que muchas están comentadas, ya que no se superaron. Representan un comportamiento que considero debería incluirse en `SerialDate`. Por tanto, al refactorizar `SerialDate`, intentaré que estas pruebas funcionen.

Incluso con algunas de las pruebas comentadas, el informe de Clover indica que ahora ejecutan 170 (el 92 por ciento) de las 185 instrucciones ejecutables. Un gran resultado que creo que puedo mejorar.

Las primeras pruebas comentadas (líneas 23-63) son un tanto pretenciosas. El programa no fue diseñado para superar estas pruebas, pero el comportamiento me parecía evidente [G2]. Desconozco por qué se ha creado el método `testWeekdayCodeToString` pero ya que está ahí, parece obvio que no debe distinguir entre mayúsculas y minúsculas. El diseño de las pruebas fue sencillo [T3] y más todavía que fueran satisfactorias; simplemente cambié las líneas 259 y 263 para usar `equalsIgnoreCase`.

Comenté las pruebas de las líneas 32 y 45 ya que no estaba seguro de si las abreviaturas `tues` y `thurs` se admitían o no. Las pruebas de las líneas 153 y 154 no se superaron, aunque deberían haberlo hecho [G2]. Podemos corregirlas, junto a las pruebas de las líneas 163 a la 213, si realizamos los siguientes cambios en la función `stringToMonthCode`.

```
457  if ((result < 1) || (result > 12)) { result = -1;
458        for (int i = 0; i < monthNames.length; i++) {
459          if (s.equalsIgnoreCase(shortMonthNames[i])) {
460            result = i + 1;
461            break;
462          }
463          if (s.equalsIgnoreCase(monthNames[i])) {
464            result = i + 1;
465            break;
466          }
467        }
468  }
```

La prueba comentada de la línea 318 descubre un error en el método `getFollowingDayOfWeek` (línea 672). El 25 de diciembre de 2004 fue sábado y el siguiente sábado fue el 1 de enero de 2005. Sin embargo, al ejecutar la

prueba, vemos que `getFollowingDayOfWeek` devuelve el 25 de diciembre como siguiente sábado después del 25 de diciembre, un error evidente [G3], [T1]. Vemos el problema en la línea 685. Es un error de condición de límite típico [T5]. Debería ser lo siguiente:

```
685 if (baseDOW >= targetWeekday) {
```

Conviene destacar que esta función sufrió una reparación anterior. El historial de cambios (línea 43) muestra que se corrigieron los errores en `getPreviousDayOfWeek`, `getFollowingDayOfWeek` y `getNearestDayOfWeek` [T6].

La prueba de unidad `testGetNearestDayOfWeek` (línea 329), que prueba el método `getNearestDayOfWeek` (línea 705), inicialmente no era tan extensa y completa. Añadí multitud de casos de prueba ya que los iniciales no se superaban [T6]. Puede ver el patrón de fallos si se fija en los casos de prueba comentados. El patrón es revelador [T7]. Muestra que el algoritmo falla si el día más próximo es de una fecha futura. Evidentemente se trata de algún tipo de error de condición de límite [T5].

El patrón de alcance de las pruebas generado por Clover también es interesante [T8]. La línea 719 nunca se ejecuta, lo que significa que la instrucción `if` de la línea 718 siempre es `false`, pero si nos fijamos en el código, indica que debe ser `true`. La variable `adjust` siempre es negativa y no puede ser mayor o igual a 4, por lo que el algoritmo es incorrecto.

A continuación se muestra el algoritmo correcto:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
   adjust -= 7;

return SerialDate.addDays (adjust, base);
```

Por último, las pruebas de la líneas 417 y 429 se pueden superar si se genera `IllegalArgumentException` en lugar de devolver una cadena de error desde `weekInMonthToString` y `relativeToString`. Con estos cambios, todas las pruebas de unidad se superan y creo que ahora `SerialDate` funciona. Llega el momento de hacer que sea correcta.

Hacer que sea correcta

Describiremos `SerialDate` de arriba a abajo para mejorarla en nuestro recorrido. Aunque no lo veamos en este análisis, ejecutaré todas las pruebas de unidad de `JCommon`, incluida mi prueba de unidad mejorada para `SerialDate`, con todos los cambios efectuados. Por ello, tenga la seguridad de que todos los cambios que vea funcionan para `JCommon`.

En la línea 1 vemos abundantes comentarios sobre información de licencia, derechos de autor, autores e historial de cambios. Asumo que hay ciertos aspectos legales que mostrar, por lo que los derechos de autor y las licencias deben conservarse. Por otra parte, el historial de cambios es una rémora de la década de 1960. Ahora tenemos herramientas de control de código fuente que se encargan de ello. Hay que eliminar este historial [C1].

La lista de importación que comienza en la línea 61 se puede reducir por medio de `java.text.*` y `java.util.*`. [J1]

No me convence el formato HTML del Javadoc (línea 67). Un archivo fuente con más de un lenguaje me parece un problema. Este comentario tiene *cuatro* lenguajes: Java, español, Javadoc y html [G1]. Con tantos lenguajes se hace difícil mantener la coherencia. Por ejemplo, la ubicación de las líneas 71 y 72 se pierde al generar el Javadoc y además, ¿quién quiere ver `<ul>` y `<li>` en el código fuente? Una estrategia más acertada consiste en rodear el comentario con `<pre>` para que el formato del código fuente se conserve en el Javadoc[96].

La línea 86 es la declaración de la clase. ¿Por qué se le asigna el nombre `SerialDate`? ¿Qué sentido tiene la palabra `serial`? ¿Es porque la clase se deriva de `Serializable`? Parece improbable.

Basta de adivinanzas. Sé por qué (o al menos eso creo) se usa la palabra `serial`. La clave se encuentra en las constantes `SERIAL_LOWER_BOUND` y `SERIAL_UPPER_BOUND` de las líneas 98 y 101. Y una clave todavía mejor es el comentario de la línea 830. El nombre de la clase es `SerialDate` ya que se implementa con un número de serie, que parece ser el número de días desde el 30 de diciembre de 1899.

Pero esto supone un problema. Por un lado, el término «número de serie» no es realmente correcto. Puede ser un detalle menor pero la representación es más un desplazamiento relativo que un número de serie. El término «número de serie» tiene que ver más con marcadores de identificación de productos que con fechas. Por ello, no lo considero especialmente descriptivo [N1]. Un término más descriptivo sería «ordinal».

El segundo problema es más significativo. El nombre `SerialDate` implica una implementación. Esta clase es abstracta. No es necesario que implique nada sobre la implementación; de hecho, es aconsejable ocultarla. Por ello, creo que el nombre se encuentra en un nivel de abstracción incorrecto [N2]. En mi opinión, el nombre de esta clase debería ser simplemente `Date`.

Desafortunadamente, hay demasiadas clases con el nombre `Date` en la biblioteca de Java, de modo que no es el más adecuado. Como esta clase trabaja con días y no horas, podríamos usar `Day`, pero ya se usa en otros muchos puntos. Al final, opté por `DayDate` como mejor opción.

A partir de ahora, usaremos `DayDate`. Recuerde que los listados que va a leer siguen usando `SerialDate`.

Entiendo porque `DayDate` se hereda de `Comparable` y `Serializable`. ¿Pero de `MonthConstants`? La clase `MonthConstants` (véase el Listado B-3) es una serie de constantes finales estáticas que definen los meses. Heredar de clases con constantes es un viejo truco que los programadores de Java usan para evitar expresiones como `MonthConstants.January`, pero es una mala idea [J2]. `MonthConstants` debería ser una enumeración.

```
public abstract class DayDate implements Comparable,
             Serializable {
   public static enum Month {
     JANUARY(1),
     FEBRUARY(2),
     MARCH(3),
     APRIL(4),
     MAY(5),
     JUNE(6),
     JULY(7),
     AUGUST(8),
     SEPTEMBER(9),
     OCTOBER(10),
     NOVEMBER(11),
     DECEMBER(12);
```

```
    Month(int index) {
      this.index = index;
    }

    public static Month make(int monthIndex) {
      for (Month m : Month.values()) {
        if (m.index == monthIndex)
          return m;
      }
      throw new IllegalArgumentException("Invalid month index " + monthIndex);
    }
    public final int index;
  }
```

Al cambiar `MonthConstants` por esta enumeración se modifica la clase `DayDate` y todos sus usuarios. Tardé una hora en realizar todos los cambios. Sin embargo, las funciones que antes aceptaban un valor `int` para el mes, ahora aceptan un enumerador `Month`. Esto significa que podemos deshacernos del método `isValidMonthCode` (línea 326) y de la comprobación de errores del código de los meses como en `monthCodeToQuarter` (línea 356) [G5]. Tras ello, en la línea 91, tenemos `serialVersionUID`. Esta variable se usa para controlar el señalizador. Si la cambiamos, con lo que todos los elementos `DayDate` escritos con una versión antigua del *software* serán ilegibles y se generará `InvalidClassException`. Si no declara la variable `serialVersionUID`, el compilador genera una automáticamente y será diferente cada vez que modifique el módulo. Ya sé que todos los documentos recomiendan el control manual de esta variable, pero creo que el control automático de la señalización es más seguro [G4]. Después de todo, prefiero depurar una `InvalidClassException` que el extraño comportamiento que se produciría si me olvido de cambiar `serialVersionUID`. Por ello, eliminaré la variable, al menos por ahora[97].

Creo que el comentario de la línea 93 es redundante. Los comentarios redundantes sólo sirven para acumular mentiras y desinformación [C2]. Por ello los eliminaré.

Los comentarios de las líneas 97 y 100 hablan sobre números de serie, que ya hemos mencionado antes [C1]. Las variables que describen son la primera y última fecha posible que DayDate puede describir. Podríamos hacer que fuera más claro [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Desconozco por qué `EARLIEST_DATE_ORDINAL` es 2 en lugar de 0. El comentario de la línea 829 sugiere que tiene que ver con la forma de representar fechas en Microsoft Excel. Hay información mucho más completa en una variante de `DayDate`: `SpreadsheetDate` (véase el Listado B-5). El comentario de la línea 71 describe este problema.

El problema parece relacionado con la implementación de `SpreadsheetDate` y no con `DayDate`. Mi conclusión es que `EARLIEST_DATE_ORDINAL` y `LATEST_DATE_ORDINAL` no pertenecen a `DayDate` y deberían cambiarse a `SpreadsheetDate` [G6].

De hecho, una búsqueda en el código demuestra que estas variables sólo se usan en `SpreadsheetDate`. Ni en `DayDate`, ni en otras clases de la estructura JCommon. Por lo tanto, las cambio por `SpreadsheetDate`.

Las siguientes variables, `MINIMUM_YEAR_SUPPORTED` y `MAXIMUM_YEAR_SUPPORTED` (líneas 104 y 107), constituyen un dilema. Parece evidente que si `DayDate` es una clase abstracta que no dice nada sobre implementación, no debería informarnos de un año mínimo o máximo. De nuevo, siento la necesidad de cambiar las variables a `SpreadsheetDate` [G6]. Pero una búsqueda rápida de los usuarios de estas variables muestra que otra clase las utiliza: `RelativeDayOfWeekRule` (véase el Listado B-6), Vemos dicho uso en las líneas 177 y 178, en la función `getDate`, donde se usan para comprobar que el argumento de `getDate` sea un año válido. El dilema es que un usuario de una clase abstracta necesita información sobre su implementación.

Tendremos que proporcionar esta información sin contaminar `DayDate`. Por lo general, obtendríamos la información de implementación de una instancia de una variante. Sin embargo, la función `getDate` no recibe una instancia de `DayDate`, aunque sí la devuelve, lo que significa que debe crearla en alguna parte. La solución está en las líneas 187-205. La instancia `DayDate` se crea por medio de una de estas tres funciones: `getPreviousDayOfWeek`, `getNearestDayOfWeek` o `getFollowingDayOfWeek`. Si nos fijamos en el

listado `DayDate`, vemos que estas funciones (líneas 638-724) devuelven una fecha creada por `addDays` (línea 571), que invoca `createInstance` (línea 808), que crea `SpreadsheetDate` [G7].

No es recomendable que las clases base conozcan sus variantes. Para corregirlo, debemos usar el patrón de *factoría abstracta*[98] y crear `DayDateFactory`. Esta factoría creará las instancias de `DayDate` que necesitamos y también responderá a preguntas sobre la implementación, como las fechas máxima y mínima.

```
  public abstract class DayDateFactory {
    private static DayDateFactory factory = new SpreadsheetDateFactory();
    public static void set Instance(DayDateFactory factory) {
      DayDateFactory.factory = factory;
    }

    protected abstract DayDate _makeDate(int ordinal);
    protected  abstract  DayDate  _makeDate(int  day,  DayDate.Month  month,  int
year);
    protected abstract DayDate _makeDate(int day, int month, int year);
    protected abstract DayDate _makeDate(java.util.Date date);
    protected abstract int _getMinimumYear();
    protected abstract int _getMaximumYear();

    public static DayDate makeDate(int ordinal) {
      return factory._makeDate(ordinal);
    }

    public static DayDate makeDate(int day, DayDate.Month month, int year) {
      return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(int day, int month, int year) {
      return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(java.util.Date date) {
      return factory._makeDate(date);
    }

    public static int getMinimumYear() {
      return factory._getMinimumYear();
    }

    public static int getMaximumYear() {
      return factory._getMaximumYear();
    }
  }
```

Esta clase de factoría sustituye los métodos `createInstance` por métodos

makeDate, lo que mejora ligeramente los nombres [N1]. De forma predeterminada es SpreadsheetDateFactory pero se puede cambiar por otra factoría. Los métodos estáticos delegados en métodos abstractos usan una combinación de los *patrones de instancia única*[99], *decorador*[100] y *factoría abstracta* que considero muy útil. SpreadsheetDateFactory tiene este aspecto:

```
public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, DayDate.Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

Como puede apreciar, hemos enviado las variables MINIMUM_YEAR_SUPPORTED y MAXIMUM_YEAR_SUPPORTED a SpreadsheetDate, donde pertenecen [G6].

El siguiente problema de DayDate son las constantes de días, comenzando en la línea 109. Deberían ser otra enumeración [J3]. Ya hemos visto este patrón, de modo que no lo repetiremos. Se incluye en los listados definitivos.

Seguidamente, vemos una serie de tablas que comienzan en LAST_DAY_OF_MONTH (línea 140). El primer problema con estas tablas es que

los comentarios que las describen son redundantes [C3]. Basta con sus nombres, de modo que eliminamos los comentarios.

No hay motivos para que la tabla no sea privada [G8], ya que existe una función estática `lastDayOfMonth` que proporciona los mismos datos.

La siguiente tabla, `AGGREGATE_DAYS_TO_END_OF_MONTH`, es más misteriosa, ya que no se usa en ninguna parte de la estructura JCommon [G9], de modo que la elimino.

Lo mismo sucede con `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

La siguiente tabla, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, sólo se usa en `SpreadsheetDate` (líneas 434 y 473), lo que me hace dudar si transferirla a `SpreadsheetDate`. La razón de no cambiarla es que la tabla no es específica de ninguna implementación concreta [G6]. Por otra parte, sólo existe la implementación `SpreadsheetDate`, de modo que la tabla debe acercarse a donde se vaya a usar [G10],

Para zanjar la duda y ser coherentes [G11], deberíamos privatizar la tabla y mostrarla a través de una función como `julianDateOfLastDayOfMonth`. Pero nadie parece que la necesita. Es más, la tabla se puede cambiar a `DayDate` si una nueva implementación de `DayDate` la necesita. Así que la cambiamos.

Lo mismo sucede con `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Tras ello, vemos tres grupos de constantes que se pueden convertir en enumeraciones (líneas 162-205). La primera selecciona una semana de un mes. La transformo en la enumeración `WeekInMonth`.

```
public enum WeekInMonth {
   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
   public final int index;

   WeekInMonth(int index) {
     this.index = index;
   }
}
```

El segundo grupo de constantes (líneas 177-187) es más complicado. Las constantes `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` e `INCLUDE_BOTH` se usan para describir si las fechas finales de un intervalo deben incluirse en el mismo. Matemáticamente, se describe como intervalo

abierto, intervalo a medio abrir e intervalo cerrado. Creo que resulta más claro con la nomenclatura matemática [N3], de modo que lo cambio por la enumeración `DateInterval` con los enumeradores `CLOSED, CLOSED_LEFT, CLOSED_RIGHT` y `OPEN`.

El tercer grupo de constantes (líneas 18-205) describen si la búsqueda de un día concreto de la semana devuelve la última instancia, la siguiente o la más próxima. Decidir un nombre adecuado es complicado. Al final, opté por `WeekdayRange` con los enumeradores `LAST, NEXT` y `NEAREST`.

Puede que no esté de acuerdo con los nombres elegidos. Para mí tienen sentido. Lo importante es que ahora son más fáciles de cambiar [J3]. Ya no se pasan como enteros, sino como símbolos. Puedo usar la función de cambio de nombre de mi IDE para cambiar los nombres o los tipos sin preocuparme de haberme olvidado de un `-1` o un `2` en alguna parte del código o de que la declaración de un argumento `int` no estén bien descrita.

El campo de descripción de la línea 208 no parece que se use en ninguna parte. Lo elimino junto a sus elementos de acceso y mutación [G9]. También elimino el constructor predeterminado de la línea 213 [G12]. El compilador se encargará de generarlo.

Podemos ignorar el método `isValidWeekdayCode` (líneas 216-238) ya que lo eliminamos al crear la enumeración `Day`.

Llegamos al método `stringToWeekdayCode` (líneas 242-270). Los Javadoc que no suponen demasiado para la firma del método sobran [C3], [G12]. El único valor de este Javadoc es la descripción del valor devuelto `-1`. Sin embargo, como cambiamos a la enumeración `Day`, el comentario es en realidad incorrecto [C2]. Ahora el método genera `IllegalArgumentException`. Por ello, eliminamos el Javadoc.

También elimino las palabras clave `final` de argumentos y declaraciones de variables, ya que no parecen servir de mucho [G12]. La eliminación de `final` no goza de gran aceptación. Por ejemplo, Robert Simmons[101] recomienda «… diseminar `final` por la totalidad del código». No estoy de acuerdo. Creo que existen casos para usar `final`, por ejemplo como constante ocasional, pero en general, esta palabra clave apenas añade valor y suele ser

un estorbo. Puede que lo piense porque el tipo de errores que puede capturar `final` ya se capturan en las pruebas de unidad que he creado.

Las instrucciones `if` duplicadas [G5] del bucle `for` (líneas 259 y 263) son irrelevantes, de modo que las conecté en una única instrucción `if` con el operador `||`. También usé la enumeración `Day` para dirigir el bucle `for` y realicé otros cambios estéticos.

Este método no pertenece realmente a `DayDate`. En realidad es la función de análisis de `Day`. Por lo tanto, lo cambié a la enumeración `Day`, lo que hizo que aumentara considerablemente de tamaño. Como el concepto de `Day` no depende de `DayDate`, extraje la enumeración `Day` de la clase `DayDate` a un archivo propio [G13].

También cambié la siguiente función, `weekdayCodeToString` (líneas 272-286) a la enumeración `Day` y le asigné el nombre `toString`.

```
  public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);

    public final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

    Day(int day) {
      index = day;
    }

    public static Day make(int index) throws IllegalArgumentException {
      for (Day d : Day.values())
        if (d.index == index)
          return d;
      throw new IllegalArgumentException(
        String.format("Illegal day index: %d.", index));
    }

    public static Day parse(String s) throws IllegalArgumentException {
      String[] shortWeekdayNames =
        dateSymbols.getShortWeekdays();
      String[] weekDayNames =
        dateSymbols.getWeekdays();

      s = s.trim();
      for (Day day : Day.values()) {
```

```
        if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
            s.equalsIgnoreCase(weekDayNames[day.index])) {
          return day;
        }
      }
      throw new IllegalArgumentException(
      String.format("%s is not a valid weekday string", s));
    }

    public String toString() {
      return dateSymbols.getWeekdays()[index];
    }
  }
```

Hay dos funciones `getMonths` (líneas 288-316). La primera invoca la segunda. La segunda solamente se invoca desde la primera. Por ello, las he combinado en una y las he simplificado considerablemente [G9], [G12], [F4]. Por último, he cambiado el nombre por otro más descriptivo [N1].

```
  public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
  }
```

La función `isValidMonthCode` (líneas 326-346) es ahora irrelevante gracias a la enumeración `Month`, de modo que la elimino [G9].

La función `monthCodeToQuarter` (líneas 356-375) parece sufrir *envidia de las características*[102] [G14] y seguramente pertenezca a la enumeración `Month` como método `quarter`, motivo por el que la sustituyo.

```
  public int quarter() {
    return 1 + (index-1)/3;
  }
```

De este modo, la enumeración `Month` tiene tamaño suficiente como para estar en una clase propia. La extraigo de DayDate para mantener la coherencia con la enumeración `Day` [G11], [G13].

Los dos siguientes métodos tienen el nombre `monthCodeToString` (líneas 377-426). Vemos de nuevo que uno invoca al otro con un indicador. No es recomendable pasar un indicador como argumento de una función, en especial si dicho indicador sólo selecciona el formato del resultado [G15]. Por tanto, cambio de nombre, simplifico y reestructuro estas funciones y las incluyo en la enumeración `Month` [N1], [N3], [C3], [G14].

```
  public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
```

```
  }

  public String toShortString() {
     return dateFormatSymbols.getShortMonths()[index – 1];
  }
```

El siguiente método es `stringToMonthCode` (líneas 428-472). Lo cambio de nombre, lo paso a la enumeración `Month` y lo simplifico [N1], [N3], [C3], [G14], [G12].

```
  public static Month parse(String s) {
     s = s.trim();
     for (Month m : Month.values())
       if (m.matches(s))
         return m;
     try {
       return make(Integer.parseInt(s));
     }
     catch (NumberFormatException e) {}
       throw new IllegalArgumentException("Invalid month " + s);
  }

  private boolean matches(String a) {
     return s.equalsIgnoreCase(toString()) ||
       s.equalsIgnoreCase(toShortString());
  }
```

El método `isLeapYear` (líneas 495-517) se puede modificar para que sea más expresivo [G16].

```
  public static boolean isLeapYear(int year) {
     boolean fourth = year % 4 == 0;
     boolean hundredth = year % 100 == 0;
     boolean fourHundredth = year % 400 == 0;
     return fourth && (!hundredth || fourHundredth);
  }
```

La siguiente función, `leapYearCount` (líneas 519-536) no pertenece realmente a `DayDate`. Nadie la invoca, excepto los dos métodos de `SpreadsheetDate`, de modo que la desplazo hacia abajo [G6].

La función `lastDayOfMonth` (líneas 538-560) usa la matriz `LAST_DAY_OF_MONTH`, que en realidad pertenece a la enumeración `Month` [G17], por lo que la cambio de ubicación. También simplifico la función y aumento su expresividad [G16].

```
  public static int lastDayOfMonth(Month month, int year) {
     if (month == Month.FEBRUARY && isLeapYear(year))
       return month.lastDay() + 1;
     else
```

```
    return month.lastDay();
}
```

Ahora empieza a ponerse interesante. La siguiente función es `addDays` (líneas 562-576). En primer lugar, como opera en las variables de `DayDate`, no debería ser estática [G18]. La cambio por un método de instancia. Por otra parte, invoca la función `toSerial`, cuyo nombre deberíamos cambiar por `toOrdinal` [N1]. Por último, el método se puede simplificar.

```
public DayDate addDays(int days) {
   return DayDateFactory.makeDate(toOrdinal() + days);
}
```

Lo mismo sucede con `addMonths` (líneas 578-602). Debería ser un método de instancia [G18]. El algoritmo es un tanto complicado, de modo que recurro a la explicación de variables temporales[103] [G19] para que sea más transparente. También cambio el nombre del método `getYYY` por `getYear` [N1].

```
public DayDate addMonths(int months) {
   int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
   int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
   int resultYear = resultMonthAsOrdinal / 12;
   Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
   int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
   int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
   return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

La función addYears (líneas 604-626) es similar al resto.

```
public DayDate plusYears(int years) {
   int resultYear = getYear() + years;
   int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
   int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
   return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

Hay algo que me preocupa sobre el cambio de estos métodos de estáticos a métodos de instancia. ¿La expresión `date.addDays(5)` aclara que el objeto `date` no cambia y que se devuelve una nueva instancia de `DayDate` o se supone, equivocadamente, que se añaden cinco días al objeto `date`? Pensará que no es un gran problema, pero un fragmento de código como el siguiente puede ser muy engañoso [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // desplazar la fecha una semana
```

Un lector de este código podría aceptar que `addDays` cambia el objeto `date`, de modo que necesitamos un nombre que acabe con la ambigüedad [N4]: `plusDays` y `plusMonths`. Creo que la intención del método se captura correctamente por medio de

```
DayDate date = oldDate.plusDays(5);
```

mientras que el siguiente no transmite con fluidez al lector que el objeto `date` ha cambiado:

```
date.plusDays(5);
```

Los algoritmos son cada vez más interesantes. `getPreviousDayOfWeek` (líneas 628-660) funciona pero es complicado. Tras meditar en lo que sucedía [G21], pude simplificarlo y aplicar la explicación de variables temporales [G19] para aclarar su significado. También lo cambié de método estático a método de instancia [G18] y me deshice del método de instancia duplicado [G5] (líneas 997-1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
   int offsetToTarget = targetDayOfWeek.index = getDayOfWeek().index;
   if (offsetToTarget >= 0)
     offsetToTarget - 7;
   return plusDays(offsetToTarget);
}
```

Sucede exactamente lo mismo con `getFollowingDayOfWeek` (líneas 662-693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
   int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
   if (offsetToTarget <= 0)
     offsetToTarget += 7;
   return plusDays(offsetToTarget);
}
```

La siguiente función es `getNearestDayOfWeek` (líneas 695-726), que corregimos en un apartado anterior. Pero esos cambios no son coherentes con el patrón actual de las dos últimas funciones [G11]. Por ello, recurro a la explicación de *variables temporales* [G19] para aclarar el algoritmo.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
   int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
   int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
   int offsetToPreviousTarget = offsetToFutureTarget - 7;

   if (offsetToFutureTarget > 3)
```

```
      return plusDays(offsetToPreviousTarget);
    else
      return plusDays(offsetToFutureTarget);
  }
```

El método `getEndOfCurrentMonth` (líneas 728-740) es un tanto extraño ya que es un método de instancia que envidia [G14] a su propia clase aceptado un argumento `DayDate`. Lo convierto en un verdadero método de instancia y clarifico algunos de los nombres.

```
  public DayDate getEndOfMonth() {
    Month month = getMonth();
    int year = getYear();
    int lastDay = lastDayOfMonth(month, year);
    return DayDateFactory.makeDate(lastDay, month, year);
  }
```

La refactorización de `weekInMonthToString` (líneas 742-761) resultó ser muy interesante. Mediante las herramientas de refactorización de mi IDE, primero cambié el método a la enumeración `WeekInMonth` creada antes y después cambié el nombre por `toString`. Tras ello, lo convertí en método de instancia. Todas las pruebas fueron correctas (¿adivina hacia dónde nos dirigimos?).

Seguidamente, eliminé el método. Fallaron cinco afirmaciones (líneas 411-415 del Listado B-4). Cambié estas líneas para usar los nombres de los enumeradores (`FIRST`, `SECOND`, etc.). Las pruebas fueron correctas. ¿Ve por qué? ¿Puede ver también por qué son necesarios estos pasos? La herramienta de refactorización se encargó de que los invocadores anteriores de `weekInMonthToString` invocaran ahora `toString` en el enumerador `weekInMonth` ya que todos los enumeradores implementan `toString` para devolver sus nombres…

Desafortunadamente, me pasé de listo. A pesar de la elegancia de la cadena de refactorización, comprobé que los únicos usuarios de esta función eran las pruebas que acababa de modificar, de modo que las eliminé. Así pues, tras determinar que sólo las pruebas invocaban `relativeToString` (líneas 765-781), eliminé directamente la función y sus pruebas.

Hemos llegado a los métodos abstractos de esta clase abstracta. Y el primero es `toSerial` (líneas 838-844). En un apartado anterior cambié el nombre por `toOrdinal`. Al verlo en este contexto, decidí que el cambio de

nombre debería ser por `getOrdinalDay`. El siguiente método abstracto es `toDate` (líneas 838-844). Convierte `DayDate` en `java.util.Date`. ¿Por qué es abstracto? Si analizamos su implementación en `SpreadsheetDate` (líneas 198-207 del Listado B-5), vemos que no depende de la implementación de esa clase [G6]. Por tanto, lo desplazo hacia arriba.

Los métodos `getYYYY`, `getMonth` y `getDayOfMonth` son evidentemente abstractos. Sin embargo, `getDayOfWeek` debería ascender desde `SpreadSheetDate` ya que no depende de nada de lo que encontremos en `DayDate` [G6]. ¿O sí?

Si se fija atentamente (línea 247 del Listado B-5), verá que el algoritmo depende implícitamente del origen del día ordinal (es decir, el día de la semana del día 0). Por ello, aunque esta función carezca de dependencias físicas que no se puedan cambiar a `DayDate`, cuenta con una dependencia lógica. Este tipo de dependencias lógicas me molestan [G22]. Si algo lógico depende de la implementación, también debería haber algo físico. Además, me parece que el propio algoritmo podría ser genérico y que debería depender en menor medida de la implementación [G6]. Por tanto, creé un método abstracto en `DayDate` con el nombre `getDayOfWeekForOrdinalZero` y lo implementé en `SpreadsheetDate` para devolver `Day.SATURDAY`. Tras ello, envié el método `getDayOfWeek` a `DayDate` y lo cambié para que invocara `getOrdinalDay` y `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
   Day startingDay = getDayOfWeekForOrdinalZero();
   int startingOffset = startingDay.index - Day.SUNDAY.index;
   return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Fíjese en el comentario de las líneas 895-899. ¿Necesitamos realmente esta repetición? Como de costumbre, eliminé este comentario junto a los demás.

El siguiente método es `compare` (líneas 902-913). De nuevo, es incorrectamente abstracto [G6], por lo que cambio la implementación a `DayDate`. Además, el nombre no es descriptivo [N1]. En realidad, este método devuelve la diferencia en días desde el argumento, por lo que cambié el nombre por `daysSince`. Tampoco existían pruebas para este método, de

modo que las creé.

Las seis siguientes funciones (líneas 915-980) son métodos abstractos que deben implementarse en `DayDate`, por lo que las extraje de `SpreadsheetDate`.

La última función, `isInRange` (líneas 982-995), también debe extraerse y refactorizarse. La instrucción `switch` no es agradable [G23] y se puede modificar si enviamos los casos a la enumeración `DateInterval`.

```
public enum DateInterval {
  OPEN {
    public boolean isIn(int d, int left, int right) {
      return d > left && d < right;
    }
  },
  CLOSED_LEFT {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d < right;
    }
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };

  public abstract boolean isIn(int d, int left, int right);
}
```

```
public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
  int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
  return interval.isIn(getOrdinalDay(), left, right);
}
```

Con esto llegamos al final de `DayDate`. Realizaremos una nueva pasada por la clase completa para comprobar cómo fluye. Primero, el comentario inicial está desfasado, de modo que lo reduzco y lo mejoro [C2].

Tras ello, desplazo las enumeraciones restantes a sus propios archivos [G12].

Seguidamente, desplazo la variable estática (`dateFormatSymbols`) y tres métodos estáticos (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) a una nueva clase con el nombre `DateUtil` [G6].

Cambio los métodos abstractos a una posición superior, donde pertenecen [G24].

Cambio `Month.make` por `Month.fromInt` [N1] y repito la operación con las demás enumeraciones. También creo un método de acceso `toInt()` para todas las enumeraciones y convierto en privado el campo `index`.

Se produce una interesante duplicación [G5] en `plusYears` y `plusMonths` que conseguí eliminar extrayendo un nuevo método con el nombre `correctLastDayOfMonth`, lo que aclaraba el significado de los tres métodos.

Me deshice del número mágico 1 [G25] y lo sustituí por `Month.JANUARY.toInt()` o `Day.SUNDAY.toInt()`, según el caso. Me detuve en limpiar los algoritmos de `SpreadsheetDate`. El resultado final se puede comprobar en los listados B.7 a B.16.

El alcance del código en `DayDate` se *ha reducido* al 84.9 por 100, no porque se pruebe una cantidad menor de funcionalidad, sino porque la clase se ha reducido tanto que las líneas sin alcance tienen un peso mayor. Ahora, en `DayDate` las pruebas se aplican a 45 de las 53 instrucciones ejecutables. Las líneas sin alcance son tan triviales que no merece la pena probarlas.

Conclusión

Otra vez hemos aplicado la Regla del Boy Scout. Hemos entregado el código más limpio de lo que lo recibimos. Nos ha llevado tiempo, pero ha merecido la pena. El alcance de las pruebas ha aumentado, hemos corregido algunos errores y hemos aclarado y reducido el tamaño del código. La próxima persona que lo lea seguramente lo encontrará más fácil de leer. Y probablemente esa persona sea capaz de limpiarlo algo más de lo que hemos hecho nosotros.

Bibliografía

- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented*

Software, Gamma et al., Addison-Wesley, 1996.

- **[Simmons04]**: *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.
- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.
- **[Beck97]**: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

17
Síntomas y heurística

En su magnífico libro *Refactoring*[104], Martin Fowler identifica diversos síntomas de código (*Smells*). La lista que mostramos a continuación incluye muchos de los síntomas de Martin y otros propios. También contiene otras perlas y heurística que suelo emplear en mi trabajo.

Para compilar esta lista he examinado diversos programas y los he refactorizado. Al aplicar un cambio, me preguntaba el por qué y anotaba el motivo. El resultado es una extensa lista de aspectos que no me «huelen» bien cuando leo código.

La lista se debe leer de arriba a abajo, y también se puede usar como referencia.

Comentarios

C1: Información inapropiada

No es apropiado que un comentario contenga información que se pueda almacenar en otro tipo de sistema como un sistema de control de código fuente, de seguimiento de problemas o de mantenimiento de registros. Los historiales de cambios, por ejemplo, abarrotan los archivos de código con abundante texto sin interés alguno. Por lo general, metadatos como autores, fechas de modificación, números SPR y similares no deben aparecer en los comentarios. Los comentarios deben reservarse para notas técnicas sobre el código y el diseño.

C2: Comentario obsoleto

Un comentario anticuado, irrelevante e incorrecto es obsoleto. Los comentarios envejecen rápidamente. Es recomendable no escribir un comentario que vaya a quedar obsoleto. Si detecta un comentario obsoleto, conviene actualizarlo o eliminarlo lo antes posible. Los comentarios obsoletos tienden a alejarse del código que describían. Se convierten en islas de irrelevancia y desorientación en el código.

C3: Comentario redundante

Un comentario es redundante si describe algo que ya se define correctamente

por sí mismo. Por ejemplo:

```
i++; // incrementar i
```

Otro ejemplo es un Javadoc que no dice más (o incluso menos) que la firma de una función:

```
/**
* @param sellRequest
* @return
* @throws ManagedComponentException
*/
public SellResponse beginSellItem(SellRequest sellRequest)
   throws ManagedComponentException
```

Los comentarios deben comunicar lo que el código no pueda expresar por sí mismo.

C4: Comentario mal escrito

Un comentario que merezca la pena escribir merece la pena ser leído. Si piensa escribir un comentario, asegúrese de que es el mejor que puede crear. Elija las palabras con atención. Use gramática y puntuación correctas. No divague. No afirme lo evidente. Sea breve.

C5: Código comentado

Me molesta ver grandes fragmentos de código comentado. ¿Quién sabe qué antigüedad tienen? ¿Quién sabe si tiene sentido o no? Pero nadie lo borra porque piensa que alguien más lo necesita.

Ese código se estanca y se corrompe, y cada día que pasa es menos relevante. Invoca funciones que ya no existen. Usa variables cuyos nombres han cambiado. Se rige por convenciones obsoletas. Contamina los módulos en los que aparece y distrae a los usuarios que lo leen. El código comentado es una aberración.

Cuando vea código comentado, elimínelo. No se preocupe, el sistema de control de código fuente lo recordará. Si alguien lo necesita, puede consultar una versión anterior. No sufra el código comentado para sobrevivir.

Entorno

E1: La generación requiere más de un paso

La generación de un proyecto debería ser una operación sencilla. No debería tener que comprobar demasiados elementos del control de código fuente. No debería necesitar una secuencia de antiguos comandos ni secuencias de comandos dependientes del contexto para generar cada elemento. No debería tener que buscar los distintos archivos JAR, XML y similares necesarios para el sistema. Debería finalizar el sistema con un sencillo comando y después ejecutar otro igual de sencillo para generarlo.

```
svn get mySystem
cd mySystem
ant all
```

E2: Las pruebas requieren más de un paso

Debería poder ejecutar todas las pruebas de unidad con un solo comando. En el mejor de los casos, debería poder ejecutarlas pulsando un botón de su IDE. En el peor, debería poder ejecutar un único comando en una línea de comandos. La capacidad de ejecutar todas las pruebas es tan importante que debe ser algo rápido, sencillo y obvio.

Funciones

F1: Demasiados argumentos

Las funciones deben tener un número reducido de argumentos. Lo mejor es que no tengan, seguido de uno, dos y tres argumentos. Más de tres ya es cuestionable y debería evitarse (véase el capítulo 3).

F2: Argumentos de salida

Los argumentos de salida son ilógicos. El lector espera que los argumentos sean entradas, no salidas. Si su función tiene que cambiar el estado de algo, haga que cambie el estado del objeto en el que se invoca (véase el capítulo 3).

F3: Argumentos de indicador

Los argumentos booleanos declaran abiertamente que la función hace más de una cosa. Resultan confusos y deben eliminarse (véase el capítulo 3).

F4: Función muerta

Los métodos que nunca se invocan deben descartarse. La presencia de código muerto es innecesaria. No tema eliminar la función. Su sistema de control de código fuente la recordará.

General

G1: Varios lenguajes en un archivo de código

Los modernos entornos de programación actuales permiten incluir varios lenguajes diferentes en el mismo archivo de código. Por ejemplo, un archivo de Java puede contener fragmentos de XML, HTML, YAML, JavaDoc, JavaScript, y similares. Además de HTML, un archivo JSP podría incluir Java, sintaxis de biblioteca de etiquetas, comentarios en español, Javadoc, XML, JavaScript, etc. Resulta confuso en el mejor de los casos y un desastre en el peor.

Lo ideal sería que el archivo de código incluyera un solo lenguaje pero, en realidad, seguramente tendremos que usar más de uno. Debemos intentar minimizar la cantidad y el alcance de los lenguajes adicionales en nuestros

archivos de código.

G2: Comportamiento evidente no implementado

De acuerdo al *Principio de la Mínima Sorpresa*[105], una función o clase debe implementar los comportamientos que otro programador esperaría. Por ejemplo, imagine una función que traduce el nombre de un día en una *enumeración* que represente dicho día.

```
Day day = DayDate.StringToDay(String dayName);
```

Esperaríamos que la cadena «`Monday`» se tradujera en Day.MONDAY. También esperaríamos la traducción de las abreviaturas habituales y que la función ignorara mayúsculas y minúsculas.

Cuando un comportamiento obvio no se implementa, los lectores y usuarios del código ya no dependen de su intuición sobre los nombres de las funciones. Pierden su confianza en el autor original y se ven obligados a leer los detalles del código.

G3: Comportamiento incorrecto en los límites

Parece evidente afirmar que el código debe comportarse de forma correcta. El problema es que no nos damos cuenta de lo complicado que es dicho comportamiento correcto. Los programadores suelen crear funciones que esperan que funcionen y confían en su intuición más que en comprobar que el código funciona en todos los casos de límites.

No existe sustituto para la meticulosidad. Las condiciones de límite, los casos extremos, las excepciones, representan algo que puede confundir a un algoritmo elegante e intuitivo. *No dependa de su intuición*. Busque todas las condiciones de límite y cree pruebas para cada una.

G4: Medidas de seguridad canceladas

Chernobyl se derritió porque el director de la central ignoró todos y cada uno de los mecanismos de seguridad. Impedían que se realizara un experimento.

El resultado fue que el experimento no salió bien y el mundo fue testigo de la primera gran catástrofe nuclear para la población.

Anular las medidas de seguridad es un riesgo. Puede que sea necesario ejercer el control manual sobre `serialVersionUID` pero siempre es arriesgado. La desactivación de determinadas advertencias del compilador (o de todas) puede ayudarle a conseguir la generación, pero corre el riesgo de sufrir interminables sesiones de depuración. Desactivar las pruebas que fallan y convencerse de que conseguirá que después sean satisfactorias es tan erróneo como pensar que sus tarjetas de crédito son dinero gratuito.

G5: Duplicación

Una de las reglas más importantes del libro y que debe tomarse muy en serio. La práctica totalidad de los autores que escriben sobre diseño de *software* mencionan esta regla. Dave Thomas y Andy Hunt la denominaron principio DRY[106] (*Don't Repeat Yourself*, No repetirse). Kent Beck la convirtió en uno de los principios fundamentales de la programación Extreme y la denominó «Una sola vez». Ron Jeffries sitúa esta regla en segunda posición, por debajo de la consecución satisfactoria de todas las pruebas.

Siempre que vea duplicados en el código, indican una oportunidad de abstracción fallida. La duplicación podría convertirse en una subrutina o en otra clase. Al incluir la duplicación en una abstracción, aumenta el vocabulario del lenguaje del diseño. Otros programadores pueden usar sus creaciones abstractas. El código se vuelve más rápido y menos proclive a errores ya que ha aumentado el nivel de abstracción.

El caso más evidente de duplicación es la presencia de fragmentos de código idéntico que parecen pegados repetidamente por el programador, sin sentido. Conviene reemplazarlos por métodos simples.

Una forma más sutil es la cadena `switch/case` o `if/else` que aparece repetidamente en diversos módulos y que siempre prueba las mismas condiciones. Conviene reemplazar estas cadenas por polimorfismo.

Y más sutiles todavía son los módulos con algoritmos similares pero que no comparten las mismas líneas de código. Sigue siendo duplicación y debe

corregirse por medio del patrón de *método de plantilla*[107] o *estrategia*[108].

En realidad, la mayoría de patrones de diseño aparecidos en los últimos 15 años son formas de eliminar la duplicación. Las Formas normales de Codd también son una estrategia para eliminar la duplicación en esquemas de base de datos. Incluso la programación orientada a objetos es una estrategia para organizar módulos y eliminar la duplicación. No debería sorprenderle, ya que se trata de programación estructurada. Creo que el objetivo es evidente: localice los elementos duplicados y elimínelos siempre que pueda.

G6: Código en un nivel de abstracción incorrecto

Es importante crear abstracciones que separen conceptos generales de nivel superior de conceptos detallados de nivel inferior. Para ello, en ocasiones creamos clases abstractas que contengan los conceptos de nivel superior y variantes los de nivel inferior. Si lo hacemos, debemos asegurarnos de que la separación sea completa. Todos los conceptos de nivel inferior deben estar en las variantes y los de nivel superior en la clase base.

Por ejemplo, constantes, variables o funciones de utilidad que solamente pertenezcan a la implementación detallada no deben aparecer en la clase base. La clase base no debe saber nada al respecto de estos elementos.

Esta regla también se aplica a archivos fuente, componentes y módulos. El diseño correcto de *software* requiere la separación de conceptos en distintos niveles y su inclusión en contenedores diferentes. En ocasiones, dichos contenedores son clases base o variantes, y en otros casos son archivos fuente, módulos o componentes. Independientemente del caso, la separación debe ser completa. No queremos que conceptos de nivel inferior y superior se mezclen.

Fíjese en este código:

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

La función `percentFull` se encuentra en el nivel de abstracción

equivocado. Aunque hay implementaciones de `Stack` en las que el concepto de amplitud es razonable, otras no pueden conocer su nivel de amplitud. Por tanto, la función debería incluirse en una interfaz derivada como `BoundedStack`.

Pensará que la implementación podría devolver cero si la pila no tuviera límites. El problema es que no existen pilas totalmente sin límites. No se puede evitar `OutOfMemoryException` mediante la comprobación de

```
stack.percentFull() < 50.0.
```

La implementación de esta función para que devuelva 0 sería una mentira.

La moraleja es que no puede mentir o escapar de una abstracción mal ubicada. El aislamiento de abstracciones es una de las operaciones más complicadas para los desarrolladores de *software* y no se puede corregir cuando se realiza de forma incorrecta.

G7: Clases base que dependen de sus variantes

El motivo más habitual para dividir conceptos en clases base y derivadas es para que los conceptos de nivel superior de la clase base sean independientes de los de nivel inferior de las derivadas. Por ello, cuando vemos clases base que mencionan los nombres de sus variantes, se intuye un problema. Por lo general, las clases base no deben saber nada sobre su derivadas.

Evidentemente, hay excepciones. En ocasiones, el número de variantes es fijo y la clase base tiene código que elegir entre las variantes. Es muy habitual en implementaciones de equipos con estado finito. Sin embargo, en ese caso las variantes y la clase base están íntimamente unidas y siempre se implementan en el mismo archivo jar. En el caso general, deben implementarse en archivos independientes.

Al implementar variantes y clases base en archivos diferentes y garantizar que los archivos de la clase base desconocen el contenido de los archivos de las variantes podemos implementar nuestros sistemas en componentes discretos e independientes. Al modificar dichos componentes, se pueden volver a implementar sin necesidad de implementar de nuevo los componentes base. De este modo se reduce significativamente el impacto del

cambio y se facilita el mantenimiento de los sistemas.

G8: Exceso de información

Los módulos bien definidos tienen interfaces reducidas que nos permiten hacer mucho con poco. Los módulos definidos de forma incorrecta tienen interfaces más amplias que nos obligan a usar distintos gestos para realizar operaciones sencillas. Una interfaz bien definida no ofrece demasiadas funciones y las conexiones son reducidas. Una interfaz definida de forma incorrecta ofrece multitud de funciones que invocar y, por tanto, las conexiones son elevadas. Los buenos programadores de *software* aprenden a limitar la parte de sus clases y módulos que muestran en sus interfaces. Cuantos menos métodos tenga una clase, mejor. Cuantas menos variables conozca una función, mejor. Cuantas menos variables de instancia tenga una clase, mejor.

Oculte sus datos. Oculte sus funciones de utilidad. Oculte sus constantes y elementos temporales. No cree clases con multitud de métodos y variables de instancia. No cree multitud de variables y funciones protegidas para sus subclases. Concéntrese en crear interfaces concisas y de tamaño reducido. Limite la información para reducir las conexiones.

G9: Código muerto

El código muerto es el que no se ejecuta. Se encuentra en el cuerpo de una instrucción `if` que comprueba una condición que no sucede. Se encuentra en el bloque `catch` de una instrucción `try` que carece de `throws`. Se encuentra en pequeños métodos de utilidad que nunca se invocan o en condiciones `switch/case` inexistentes.

El problema del código muerto es que con el tiempo empieza a oler. Cuanto más antiguo es, más profundo el hedor que despide. Se debe a que el código muerto no se actualiza al cambiar los diseños. Sigue *compilándose* pero no se rige por nuevas convenciones o reglas. Se creó en un momento en el que el sistema era *diferente*. Debe tener un entierro digno. Bórrelo del

sistema.

G10: Separación vertical

Variables y funciones deben definirse cerca de donde se utilicen. Las variables locales deben declararse por encima de su primer uso y deben tener un reducido ámbito vertical. No deben declararse a cientos de líneas de distancia de su uso.

Las funciones privadas deben definirse justo debajo de su primer uso. Pertenecen al ámbito de la clase completa pero conviene limitar la distancia vertical entre las invocaciones y las definiciones. Para localizar una función privada debe bastar con buscar debajo de su primer uso.

G11: Incoherencia

Si hace algo de una forma concreta, aplique la misma técnica a operaciones similares. Esto entronca con el principio de mínima sorpresa. Preste atención a las convenciones que elija y, una vez elegidas, asegúrese de mantenerlas. Si en una función concreta usa la variable `response` para almacenar `HttpServletResponse`, use el mismo nombre de variable en las demás funciones que usen objetos `HttpServletResponse`. Si asigna el nombre `processVerificationRequest` a un método, use un nombre similar, como `processDeletionRequest`, para los métodos que procesen otros tipos de solicitudes.

Este tipo de coherencia, si se aplica repetidamente, facilita la lectura y modificación del código.

G12: Desorden

¿Para qué sirve un constructor predeterminado sin implementación? Únicamente desordena el código y lo inunda de elementos sin sentido. Variables sin usar, funciones que nunca se invocan, comentarios que no añaden información, etc. Todos estos elementos sobran y deben eliminarse.

Mantenga limpios sus archivos, bien organizados y sin elementos sobrantes.

G13: Conexiones artificiales

Los elementos que no dependen unos de otros no deben conectarse de forma artificial. Por ejemplo, las *enumeraciones* generales no deben incluirse en clases más específicas ya que esto obliga a la aplicación a saber más sobre dichas clases. Lo mismo sucede con funciones `static` de propósito general declaradas en clases específicas.

Por lo general, una conexión artificial es la que se establece entre dos módulos sin un propósito directo. Es el resultado de incluir una variable, constante o función en una ubicación temporalmente útil pero inadecuada. Es un síntoma de falta de atención.

Piense en dónde debe declarar sus funciones, constantes y variables. No las deje en el punto más cómodo.

G14: Envidia de las características

Uno de los síntomas de Martin Fowler[109]. Los métodos de una clase deben interesarse por las variables y funciones de la clase a la que pertenecen, no por las variables y funciones de otras clases. Cuando un método usa elementos de acceso y mutación de otro objeto para manipular los datos de éste, envidia el ámbito de la clase de dicho objeto. Desea formar parte de la otra clase para tener acceso directo a las variables que manipula.

Por ejemplo:

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

El método `calculateWeeklyPay` se acerca al objeto `HourlyEmployee` para

obtener los datos sobre los que opera. El método `calculateWeeklyPay` envidia el ámbito de `HourlyEmployee`. Su deseo es formar parte de `HourlyEmployee`.

Es recomendable suprimir la envidia de características ya que muestra los detalles internos de una clase a otra. Sin embargo, en ocasiones es un mal necesario. Fíjese en lo siguiente:

```
public class HourlyEmployeeReport {
   private HourlyEmployee employee;

   public HourlyEmployeeReport(HourlyEmployee e) {
     this.employee = e;
   }

   String reportHours() {
     return String.format(
       "Name: %s\tHours:%d.%1d\n",
       employee.getName(),
       employee.getTenthsWorked()/10,
       employee.getTenthsWorked()%10);
   }
}
```

Evidentemente, el método `reportHours` envidia la clase `HourlyEmployee`. Por otra parte, no queremos que `HourlyEmployee` tenga que conocer el formato del informe. Al incluir la cadena de formato en la clase `HourlyEmployee` incumpliríamos varios de los principios del diseño orientado a objetos[110]. Conectaría `HourlyEmployee` al formato del informe y lo mostraría en los cambios de dicho formato.

G15: Argumentos de selector

No hay nada más abominable que un argumento `false` aislado al final de la invocación de una función. ¿Qué significa? ¿Qué cambiaría si fuera `true`? No sólo el propósito de un argumento de selector es difícil de recordar, sino que cada argumento de selector combina varias funciones en una. Los argumentos de selector son una forma indolente de evitar dividir una función de gran tamaño en otras menores. Fíjese en lo siguiente:

```
public int calculateWeeklyPay(boolean overtime) {
   int tenthRate = getTenthRate();
   int tenthsWorked = getTenthsWorked();
```

```
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
    int overtimePay = (int)Math.round(overTime*overtimeRate);
    return straightPay + overtimePay;
  }
```

Esta función se invoca con `true` si las horas extras se pagan como hora y media, y con `false` si se pagan como una hora normal. Ya es bastante malo tener que recordar lo que significa `calculateWeeklyPay(false)` cada vez que aparezca. Pero lo peor de esta función es que el autor ha perdido la oportunidad de crear lo siguiente:

```
  public int straightPay() {
    return getTenthsWorked() * getTenthRate();
  }
  public int overTimePay() {
    int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
    int overTimePay = overTimeBonus(overTimeTenths);
    return straightPay() + overTimePay;
  }
  private int overTimeBonus(int overTimeTenths) {
    double bonus = 0.5 * getTenthRate() * overTimeTenths;
    return (int) Math.round(bonus);
  }
```

Evidentemente, los selectores no deben ser `boolean`. Pueden ser enumeraciones, enteros u otro tipo de argumento que se use para seleccionar el comportamiento de la función. Es más recomendable tener varias funciones que pasar código a una función para seleccionar el comportamiento.

G16: Intención desconocida

Queremos que el código sea lo más expresivo posible. Expresiones extensas, notación Húngara y números mágicos distorsionan la intención del autor. Por ejemplo, veamos la función `overTimePay` cómo podría haber aparecido:

```
  public int m_otCalc() {
    return iThsWkd * iThsRte +
      (int) Math.round(0.5 * iThsRte *
        Math.max(0/ iThsWkd - 400)
      );
  }
```

Aunque parezca reducida y densa, también es prácticamente impenetrable. Es recomendable dedicar tiempo a lograr que la intención de nuestro código sea aparente para nuestros lectores.

G17: Responsabilidad desubicada

Una de las principales decisiones de un programador de *software* es dónde ubicar el código. Por ejemplo, dónde incluir la constante `PI`. ¿En la clase `Math`? ¿Pertenece a la clase `Trigonometry`? ¿O a la clase `Circle`?

El principio de mínima sorpresa vuelve a aparecer. El código debe ubicarse donde el lector espera encontrarlo. La constante `PI` debe incluirse junto a la declaración de las funciones trigonométricas. La constante `OVERTIME_RATE` debe declararse en la clase `HourlyPayCalculator`.

En ocasiones presumimos de dónde añadimos una determinada funcionalidad. Incluimos una función porque nos resulta cómodo pero no porque sea intuitivo para el lector. Por ejemplo, puede que tengamos que imprimir un informe con el total de horas que ha trabajado un empleado. Podríamos sumar las horas en el código que imprime el informe o intentar mantener un total en el código que acepte horarios de trabajo.

Una forma de tomar esta decisión consiste en analizar el nombre de las funciones. Imagine que el módulo del informe tiene la función `getTotalHours`. Imagine también que el módulo que acepta horarios de trabajo tiene la función `saveTimeCard`. ¿Cuál de las dos, por nombre, implica que calcula el total? La respuesta es evidente.

Existen motivos de rendimiento para calcular el total como horarios de trabajo y no como informe impreso. Es correcto, pero el nombre de las funciones debería reflejarlo. Por ejemplo, debería haber una función `computeRunningTotalOfHours` en el módulo de horarios.

G18: Elementos estáticos incorrectos

`Math.max (double a, double b)` es un método estático correcto. No opera

en una única instancia; de hecho, sería un error tener que usar `new Math().max(a,b)` o incluso `a.max(b)`. Todos los datos que usa `max` provienen de sus dos argumentos, no de un objeto. Además, es prácticamente imposible que queramos que `Math.max` sea polimórfico. Sin embargo, en ocasiones creamos funciones estáticas que no deben serlo. Fíjese en este ejemplo:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

De nuevo, parece una función estática razonable. No opera en un objeto concreto y recibe todos los datos de sus argumentos. Sin embargo, existe la posibilidad de que queramos que sea polimórfica. Puede que queramos implementar distintos algoritmos para calcular el precio de la hora, como por ejemplo. `OvertimeHourlyPayCalculator` y `StraightTimeHourlyPayCalculator`. En este caso, la función no debe ser estática. Debería ser una función miembro no estática de `Employee`.

Por lo general, debe decantarse por métodos no estáticos. En caso de duda, convierta la función en no estática. Si realmente quiere que una función sea estática, asegúrese de que nunca querrá que sea polimórfica.

G19: Usar variables explicativas

Kent Beck escribió sobre este tema en su magnífico libro *Smalltalk Best Practice Patterns*[111] y, más recientemente en *Implementation Patterns*[112]. Una de las técnicas más completas para que un programa sea legible consiste en dividir los cálculos en valores intermedios almacenados en variables con nombres descriptivos. Fíjese en este ejemplo de FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
   String key = match.group(1);
   String value = match.group(2);
   headers.put(key.toLowercase(), value);
}
```

El simple uso de variables explicativas ilustra con claridad que el primer grupo comparado es la *clave* y el segundo es el *valor*.

Es complicado excederse en esta técnica. Por lo general, es mejor tener más variables explicativas que menos. Es sorprendente que un módulo opaco

se vuelva más transparente con tan sólo dividir los cálculos en valores intermedios con los nombres adecuados.

G20: Los nombres de función deben indicar lo que hacen

Fíjese en este código:

```
Date newDate = date.add(5);
```

¿Intuye que se añaden cinco días a la fecha o son semanas u horas? ¿La instancia `date` cambia y la función simplemente devuelve un nuevo objeto `Date` sin cambiar el antiguo? *Por la invocación no podemos saber qué hace la función*.

Si la función añade cinco días a la fecha y después la cambia, el nombre debería ser `addDaysTo` o `increaseByDays`. Si, por otra parte, la función devuelve una nueva fecha con cinco días más pero no cambia la instancia `date`, el nombre debería ser `daysLater` o `daysSince`.

Si tiene que fijarse en la implementación (o documentación) de la función para saber qué hace, tendrá que elegir un nombre más apropiado o modificar la funcionalidad para que se pueda incluir en funciones con nombres más acertados.

G21: Comprender el algoritmo

Se crea gran cantidad de código extraño porque los autores no se esfuerzan en comprender el algoritmo. Consiguen que algo funcione combinando instrucciones `if` e indicadores sin pararse a pensar en qué sucede realmente.

La programación es una tarea de exploración. *Creemos* que conocemos el algoritmo adecuado para algo pero después lo modificamos y variamos hasta conseguir que *funcione*. ¿Cómo sabemos que *funciona*? Porque supera los casos de prueba que pensamos.

No es un enfoque equivocado. De hecho, suele ser la única forma de conseguir que una función haga lo que pensamos que debe hacer. Sin

embargo, no basta con conseguir que *funcione*.

Antes de creer que hemos terminado con una función, asegúrese de entender su funcionamiento. No basta con que supere todas las pruebas. Tiene que *estar seguro*[113] de que la solución es la correcta.

Por lo general, la forma óptima de saberlo consiste en refactorizar la función en algo tan limpio y expresivo que su funcionamiento sea *evidente*.

G22: Convertir dependencias lógicas en físicas

Si un módulo depende de otro, dicha dependencia debe ser física, no sólo lógica. El módulo dependiente no debe asumir aspectos (es decir, dependencias lógicas) sobre el módulo del que depende. Por el contrario, debe solicitar de forma explícita al módulo toda la información de la que depende.

Por ejemplo, imagine que tiene que crear una función que imprima un informe de las horas trabajadas por cada empleado. La clase `HourlyReporter` recopila los datos y los pasa a `HourlyReportFormatter` para imprimirlos (véase el Listado 17-1).

Listado 17-1
HourlyReporter.java.

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
```

```
    }

    private void printAndClearItemList() {
      formatter.format(page);
      page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
      LineItem item = new LineItem();
      item.name = e.getName();
      item.hours = e.getTenthsWorked() / 10;
      item.tenths = e.getTenthsWorked() % 10;
      page.add(item);
    }

    public class LineItem {
      public String name;
      public int hours;
      public int tenths;
    }
  }
```

Este código tiene una dependencia lógica que no se ha convertido en física. ¿La detecta? Es la constante `PAGE_SIZE`. ¿Para qué necesita `HourlyReporter` saber el tamaño de la página? El tamaño de la página debe ser responsabilidad de `HourlyReportFormatter`. La declaración de `PAGE_SIZE` en `HourlyReporter` representa una responsabilidad desubicada [G17] que hace que `HourlyReporter` asuma que conoce el tamaño que debe tener la página. Esta presunción es una dependencia lógica. `HourlyReporter` depende de que `HourlyReportFormatter` pueda procesar tamaños de página de hasta 55. Si alguna implementación de `HourlyReportFormatter` no puede asumir esos tamaños, se producirá un error. Podemos convertir en física esta dependencia si creamos un nuevo método en `HourlyReportFormatter` con el nombre `getMaxPageSize()`. Tras ello, `HourlyReporter` invoca esta función en lugar de usar la constante `PAGE_SIZE`.

G23: Polimorfismo antes que If/Else o Switch/Case

Puede parecer una sugerencia extraña dado el tema descrito en el capítulo 6. En este capítulo, afirmo que las instrucciones `switch` son adecuadas en partes del sistema en las que se añadan más funciones nuevas que tipos nuevos.

Por un lado, la mayoría usamos instrucciones `switch` por ser una solución de fuerza bruta evidente, no por ser la solución perfecta. Por tanto, esta heurística nos recuerda que debemos considerar el uso de polimorfismo antes de usar `switch`.

Por otra parte, los casos en que las funciones son más volátiles que los tipos son escasos. Por tanto, toda instrucción `switch` es sospechosa.

Suelo aplicar la siguiente regla de una instrucción `switch`: *No puede haber más de una instrucción `switch` por cada tipo de selección. Los casos de esa instrucción `switch` deben crear objetos polimórficos que ocupen el lugar de otras instrucciones `switch` similares en el resto del sistema*.

G24: Seguir las convenciones estándar

Todos los equipos deben seguir un estándar de diseño de código basado en normas comunes de la industria. Este estándar debe especificar aspectos como dónde declarar variables de instancia, cómo asignar nombres a clases, métodos y variables, dónde añadir llaves, etc. El equipo no debe necesitar un documento que describa estas convenciones ya que su código proporciona los ejemplos.

Todos los miembros del equipo deben seguir estas convenciones, lo que significa que no importa dónde añada cada uno las llaves mientras todos estén de acuerdo en dónde añadirlas.

Si desea saber qué convenciones aplico, puede verlas en el código refactorizado de los listados B.7 a B.14 del apéndice B.

G25: Sustituir números mágicos por constantes con nombre

Es probablemente una de las reglas más antiguas del desarrollo de *software*. Recuerdo haberla leído a finales de la década de 1960 en manuales de COBOL, FORTRAN y PL/1. Por lo general, no es recomendable incluir números sin procesar en el código; debe ocultarlos tras constantes con

nombres correctos. Por ejemplo, el número 86 400 debe ocultarse tras la constante `SECONDS_PER_DAY`. Si va a imprimir 55 líneas por página, la constante 55 debe ocultarse tras la constante `LINES_PER_PAGE`.

Algunas constantes son tan fáciles de reconocer que no siempre necesitan una constante con nombre tras la que ocultarse mientras se usen junto a código explicativo. Por ejemplo:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

¿Necesitamos realmente las constantes `FEET_PER_MILE`, `WORK_HOURS_PER_DAY` y `TWO` en los ejemplos anteriores? El último caso es absurdo. Existen ciertas fórmulas en las que las constantes se escriben mejor como números sin procesar. Puede cuestionar el caso de `WORK_HOURS_PER_DAY` ya que las leyes o las convenciones pueden cambiar. Por otra parte, esa fórmula se lee perfectamente si se incluye el 8 por lo que no es necesario añadir 17 más. En el caso de `FEET_PER_MILE`, el número 5280 es una constante tan conocida y exclusiva que los lectores la reconocerán aunque se muestre de forma independiente en una página sin contexto alguno.

Constantes como 3.141592653589793 también son conocidas y reconocibles. Sin embargo, la probabilidad de errores es alta y no conviene mostrarlas tal cual. Siempre que alguien ve 3.1415927535890793, sabe que es π, y no se molestan en examinarlo (¿ha visto el error de un dígito?). Tampoco queremos que la gente use 3.14, 3.14159, 3.142, y similares. Por lo tanto, es una suerte contar con `Math.PI`.

El término número mágico no sólo se aplica a números, sino a todo símbolo que tenga un valor que no sea descriptivo por sí mismo. Por ejemplo:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

En esta afirmación hay dos números mágicos. El primero es obviamente 7777, aunque no significa que no sea obvio. El segundo es «`John Doe`» y su cometido tampoco está claro.

"`John Doe`" es el nombre del empleado #7777 en una conocida base de datos de pruebas creada por nuestro equipo. Todo el mundo sabe que al

conectarse a la base de datos, ya cuenta con varios empleados con sus valores y atributos. Además, «`John Doe`» representa el único empleado por horas de la base de datos. Por tanto, la prueba debería ser la siguiente:

```
assertEquals(
   HOURLY_EMPLOYEE_ID,
   Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: Precisión

Esperar que la primera coincidencia de una consulta sea la única es una ingenuidad. El uso de números de coma flotante para representar divisas es casi un delito. Evitar bloqueos y/o la administración de transacciones por creer que las actualizaciones concurrentes no son posibles es pura indolencia. Declarar una variable como `ArrayList` cuando se necesita `List` es un exceso de restricciones. Crear todas las variables como `protected` de forma predeterminada es falta de restricciones.

Al adoptar una decisión en el código, debe hacerlo de forma precisa. Debe saber por qué la adopta y cómo afrontará las excepciones. No sea indolente sobre la precisión de sus decisiones. Si decide invocar una función que pueda devolver `null`, asegúrese de comprobar `null`. Si consulta el que considera el único registro de una base de datos, asegúrese de que el código comprueba que no haya otros. Si tiene que trabajar con divisas, use enteros[114] y aplique el redondeo correcto. Si existe la posibilidad de una actualización concurrente, asegúrese de implementar algún tipo de mecanismo de bloqueo.

En el código, la ambigüedad y las imprecisiones son el resultado de desacuerdos o de indolencia. En cualquier caso, elimínelas.

G27: Estructura sobre convención

Aplique las decisiones de diseño con estructura y no convenciones. Las convenciones de nomenclatura son correctas pero resultan inferiores a estructuras que refuerzan la compatibilidad. Por ejemplo, los casos `switch` con enumeraciones de nombres correctos son inferiores a clases base con

métodos abstractos. No estamos obligados a implementar siempre la instrucción `switch/case` de la misma forma, pero las clases base hacen que las clases concretas implementen métodos abstractos.

G28: Encapsular condicionales

La lógica booleana es difícil de entender sin necesidad de verla en el contexto de una instrucción `if` o `while`. Extraiga funciones que expliquen el cometido de la condicional. Por ejemplo:

```
if (shouldBeDeleted(timer))
```

es preferible a

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Evitar condicionales negativas

Las condicionales negativas son más difíciles de entender que las positivas. Por ello, siempre que sea posible, debe expresar las condiciones como positivas. Por ejemplo:

```
if (buffer.shouldCompact())
```

es preferible a

```
if (!buffer.shouldNotCompact())
```

G30: Las funciones sólo deben hacer una cosa

Es tentador crear funciones con varias secciones que realicen una serie de operaciones. Este tipo de funciones hacen más de *una cosa* y deben convertirse en funciones de menor tamaño, cada una para *una cosa*. Por ejemplo:

```
public void pay() {
   for (Employee e : employees) {
     if (e.isPayday()) {
       Money pay = e .calculatePay();
       e.deliverPay(pay);
     }
   }
```

```
}
```

Este fragmento de código realiza tres operaciones. Itera por todos los empleados, comprueba si cada uno debe recibir su paga y después paga al empleado. Se podría reescribir de esta forma:

```
public void pay() {
   for (Employee e : employees)
     payifNecessary(e);
}

private void payifNecessary(Employee e) {
   if (e.isPayday())
     calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
   Money pay = e.calculatePay();
   e.deliverPay(pay);
}
```

Cada una de estas funciones hace una sola cosa (véase el capítulo 3).

G31: Conexiones temporales ocultas

Las conexiones temporales suelen ser necesarias pero no debe ocultar la conexión. Estructure los argumentos de sus funciones de modo que el orden de invocación sea evidente. Fíjese en lo siguiente:

```
public class MoogDiver {
   Gradient gradient;
   List<Spline> splines;

   public void dive(String reason) {
     saturateGradient();
     reticulateSplines();
     diveForMoog(reason);
   }
   …
}
```

El orden de las tres funciones es importante. Debe saturar el degradado antes de poder entrelazar las tiras, para después continuar. Desafortunadamente, el código no aplica esta conexión temporal. Otro programador podría invocar `reticulateSplines` antes de `saturateGradient`, lo que generaría `UnsaturatedGradientException`. Una solución más acertada sería:

```
public class MoogDiver {
   Gradient gradient;
   List<Spline> splines;

   public void dive(String reason) {
     Gradient gradient = saturateGradient();
     List<Spline> splines = reticulateSplines(gradient);
     diveForMoog(splines, reason);
   }
   …
}
```

De este modo se muestra la conexión temporal generando una especie de embudo. Cada función genera un resultado que la siguiente necesita de modo que no se pueden invocar en otro orden.

Puede argumentar que esto aumenta la complejidad de las funciones y tiene razón, pero ese incremento de complejidad sintáctica muestra la verdadera complejidad temporal de la situación.

Observará que he mantenido las variables de instancia. Imagino que son necesarias para los métodos privados de las clases. Incluso así, conservo los argumentos para que la conexión temporal sea explícita.

G32: Evitar la arbitrariedad

Argumente la estructura de su código y asegúrese de que la estructura del código comunica dicho argumento. Si la estructura parece arbitraria, otros se verán con derecho a modificarla.

Si la estructura parece coherente en la totalidad del sistema, otros la usarán y conservarán la convención. Por ejemplo, recientemente repasaba cambios realizados en FitNesse y descubrí lo siguiente:

```
public class AliasLinkWidget extends ParentWidget
{
   public static class VariableExpandingWidgetRoot {
     …

   …
}
```

El problema es que `VariableExpandingWidgetRoot` no debía estar en el ámbito de `AliasLinkWidget`. Es más, otras clases sin relación usaban `AliasLinkWidget.VariableExpandingWidgetRoot` y no tenían por qué saber

nada de `AliasLinkWidget`. Puede que el programador añadiera `VariableExpandingWidgetRoot` a `AliasWidget` por comodidad o que realmente pensara que debía formar parte del ámbito de `AliasWidget`. Independientemente del motivo, el resultado será arbitrario. Las clases públicas que no son utilidades de otra clase no deben incluirse en el ámbito de otra clase. La convención es convertirlas en públicas en el nivel superior de su paquete.

G33: Encapsular condiciones de límite

Las condiciones de límite son difíciles de controlar. Aísle su procesamiento y no permita que se transfieran al resto del código. No necesitamos legiones de `+1` y `-1` por todas partes. Fíjese en este ejemplo de FIT:

```
if (level + 1 < tags.length)
{
   parts = new Parse(body, tags, level + 1, offset + endTag);
   body = null;
}
```

`level+1` aparece dos veces. Es una condición de límite que debe encapsularse en una variable con un nombre como `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length) {
   parts = new Parse(body, tags, nextLevel, offset + endTag);
   body = null;
}
```

G34: Las funciones sólo deben descender un nivel de abstracción

Las instrucciones de una función deben crearse en el mismo nivel de abstracción, un nivel por debajo de la operación descrita por el nombre de la función. Puede que sea la heurística más difícil de interpretar y aplicar. Aunque la idea es simple, como humanos nos cuesta mezclar niveles de abstracción. Fíjese en el siguiente código de FitNesse:

```
public String render() throws Exception
{
```

```
    StringBuffer html = new StringBuffer("<hr");
    if(size > 0)
      html.append(" size=\"").append(size + 1).append("\"");
    html.append(">");

    return html.toString();
  }
```

Si lo analiza, verá lo que sucede. Esta función crea la etiqueta HTML que traza una regla horizontal por la página. La altura de la regla se especifica en la variable `size`.

Fíjese otra vez en el código. Este método mezcla al menos dos niveles de abstracción. El primero es la noción de que una regla horizontal tiene un tamaño. El segundo es la sintaxis de la propia etiqueta `HR`. El código proviene del módulo `HruleWidget` de FitNesse. Este módulo detecta una fila de cuatro o más guiones y la convierte en la correspondiente etiqueta `HR`. Cuantos más guiones haya, mayor será el tamaño.

A continuación le muestro la refactorización del código. He cambiado el nombre del campo `size` para reflejar su verdadero cometido. Contenía el número de guiones adicionales.

```
  public String render() throws Exception
  {
    HtmlTag hr = new HtmlTag("hr");
    if (extraDashes > 0)
      hr.addAttribute("size", hrSize(extraDashes));
    return hr.html();
  }

  private String hrSize(int height)
  {
    int hrSize = height + 1;
    return String.format("%d", hrSize);
  }
```

Este cambio separa correctamente los dos niveles de abstracción. La función `render` simplemente crea una etiqueta `HR` sin tener que saber nada sobre su sintaxis HTML. El módulo `HtmlTag` se encarga de los problemas sintácticos.

De hecho, al realizar este cambio detecté un sutil error. El código original no incluía la barra final en la etiqueta `HR`, como haría el estándar XHTML (es decir, generaba `<hr>` en lugar de `<hr/>`). El módulo `HtmlTag` se había modificado hace tiempo para ajustarlo a XHTML.

La separación de niveles de abstracción es una de las tareas más importantes de la refactorización, y también una de las más complejas. Por ejemplo, fíjese en el siguiente código. Fue mi primer intento de separar los niveles de abstracción del método `HruleWidget.render`.

```
public String render() throws Exception
{
   HtmlTag hr = new HtmlTag("hr");
   if (size > 0) {
     hr.addAttribute ("size", ""+(size+1));
   }
   return hr.html();
}
```

Mi objetivo, en esta fase, es crear la separación necesaria y conseguir superar las pruebas. El objetivo lo alcancé fácilmente pero el resultado fue una función con niveles de abstracción mezclados. En este caso, fueron obra de la etiqueta HR y de la interpretación y el formato de la variable size. Esto indica que al dividir una función en líneas de abstracción, suelen aparecer nuevas líneas de abstracción ocultas por la estructura anterior.

G35: Mantener los datos configurables en los niveles superiores

Si tiene una constante como un valor predeterminado o de configuración que se conoce y se espera en un nivel superior de abstracción, no debe sepultarla en una función de nivel inferior. Muéstrela como argumento para esa función de nivel inferior invocado desde la función de nivel superior. Fíjese en este ejemplo de FitNesse:

```
   public static void main(String[] args) throws Exception
   {
     Arguments arguments = parseCommandLine(args);
     …
   }

public class Arguments
{
   public static final String DEFAULT_PATH = ".";
   public static final String DEFAULT_ROOT = "FitNesseRoot";
   public static final int DEFAULT_PORT = 80;
   public static final int DEFAULT_VERSION_DAYS = 14;
   …
}
```

Los argumentos de línea de comandos se analizan en la primera línea ejecutable de FitNesse. Los valores predeterminados de dichos argumentos se especifican al inicio de la clase `Argument`. No tiene que buscar instrucciones como la siguiente en los niveles inferiores del sistema:

```
if (arguments.port == 0) // usar 80 de forma predeterminada
```

Las constantes de configuración se encuentran en un nivel superior y son fáciles de cambiar. Se pasan al resto de la aplicación. Los niveles inferiores de la aplicación no poseen los valores de estas constantes.

G36: Evitar desplazamientos transitivos

Por lo general, no es recomendable que un módulo sepa demasiado sobre sus colaboradores. En concreto, si A colabora con B y B con C, no queremos que los módulos que usan A sepan nada sobre C (por ejemplo, o queremos `a.getB().getC().doSomething();`).

Es lo que en ocasiones se denomina Ley de Demeter. Los programadores pragmáticos lo denominan *Crear código silencioso*[115]. En cualquier caso, se trata de garantizar que los módulos sólo tienen conocimiento de sus colaboradores inmediatos y no del mapa de navegación completo del sistema. Si varios módulos usan alguna variante de la instrucción `a.getB().getC()`, sería complicado cambiar el diseño y la arquitectura para intercalar Q entre B y C. Tendría que localizar todas las instancias de `a.getB().getC()` y convertirlas a `a.getB().getQ().getC()`. Es la forma en que las arquitecturas se vuelven rígidas. Demasiados módulos saben demasiado sobre la arquitectura. Por el contrario, queremos que nuestros colaboradores intermedios ofrezcan todos los servicios que necesitamos. No debemos deambular por el gráfico de objetos del sistema en busca del método que necesitamos invocar. Bastaría con poder usar:

```
myCollaborator.doSomething().
```

Java

J1: Evitar extensas listas de importación mediante el uso de comodines

Si usa dos o más clases de un paquete, importe el paquete completo con

```
import package.*;
```

Las listas extensas de importaciones intimidan al lector. No queremos colapsar la parte superior de los módulos con 80 líneas de importaciones, sino que sean una instrucción concisa de los paquetes con los que colaboramos.

Las importaciones específicas son dependencias rígidas, mientras que las importaciones de comodín no. Si importa una clase concreta, esa clase *debe* existir, pero si importa un paquete con un comodín, no es necesario que existan clases concretas. La instrucción de importación simplemente añade el paquete a la ruta de búsqueda al localizar los nombres. Por tanto, no se genera una verdadera dependencia en estas importaciones y permiten aligerar las conexiones de nuestros módulos. En ocasiones, la lista extensa de importaciones puede resultar útil. Por ejemplo, si tiene que trabajar con código de legado y desea saber para qué clases crear elementos ficticios, puede examinar la lista de importaciones concretas para determinar los verdaderos nombres cualificados de todas esas clases y después añadirlos. No obstante, este uso de las importaciones concretas no es habitual. Es más, muchos IDE modernos le permiten convertir las importaciones con comodines en una lista de importaciones concretas con un solo comando. Por tanto, incluso en el caso anterior, es recomendable usar comodines. Las importaciones de comodín pueden probar conflictos de nombres y ambigüedades. Dos clases con el mismo nombre pero en paquetes diferentes tienen que importarse de forma concreta o al menos cualificarse de forma específica cuando se usen. Puede resultar molesto pero no es habitual que el uso de importaciones de comodín sea más indicado que el de importaciones concretas.

J2: No heredar constantes

Lo he visto muchas veces y siempre me molesta. Un programador añade

constantes a una interfaz y después accede a las mismas heredando dicha interfaz. Fíjese en el siguiente código:

```
public class HourlyEmployee extends Employee {
   private int tenthsWorked;
   private double hourlyRate;
   public Money calculatePay() {
     int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
     int overTime = tenthsWorked - straightTime;
     return new Money(
       hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
     );
   }
   …
}
```

¿De dónde salen las constantes `TENTHS_PER_WEEK` y `OVERTIME_RATE`? Puede que provengan de la clase `Employee`; comprobémoslo:

```
public abstract class Employee implements PayrollConstants {
   public abstract boolean isPayday();
   public abstract Money calculatePay();
   public abstract void deliverPay(Money pay);
}
```

No, de ahí no. ¿Entonces de dónde? Fíjese atentamente en la clase `Employee`. Implementa `PayrollConstants`.

```
public interface PayrollConstants {
   public static final int TENTHS_PER_WEEK = 400;
   public static final double OVERTIME_RATE = 1.5;
}
```

Es horrible. Las constantes se ocultan en la parte superior de la jerarquía de herencia. No use la herencia para burlar las reglas de ámbito del lenguaje. Use una importación estática:

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
   private int tenthsWorked;
   private double hourlyRate;

   public Money calculatePay() {
     int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
     int overTime = tenthsWorked - straightTime;
     return new Money(
       hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
     );
   }
   …
}
```

J3: Constantes frente a enumeraciones

Ahora que se han añadido enumeraciones al lenguaje (Java 5), ¡úselas! No recurra al viejo truco de `public static final int`. El significado de `int` se puede perder. El de `enum` no, ya que pertenece a una enumeración con nombre.

Es más, analice atentamente la sintaxis de las *enumeraciones*. Pueden tener métodos y campos, lo que las convierte en potentes herramientas que ofrecen mayor expresividad y flexibilidad que los `int`. Fíjese en esta variante del código:

```
public class HourlyEmployee extends Employee {
   private int tenthsWorked;
   HourlyPayGrade grade;

   public Money calculatePay() {
     int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
     int overTime = tenthsWorked - straightTime;
     return new Money(
       grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
     );
   }
   …
}

public enum HourlyPayGrade {
   APPRENTICE {
     public double rate() {
       return 1.0;
     }
   },
   LEUTENANT_JOURNEYMAN (
     public double rate() {
       return 1.2;
     }
   },
   JOURNEYMAN {
     public double rate() {
       return 1.5;
     }
   },
   MASTER {
     public double rate() {
       return 2.0;
     }
   };

   public abstract double rate();
```

```
}
```

Nombres

N1: Elegir nombres descriptivos

No sea demasiado rápido a la hora de elegir un nombre. Asegúrese de que sea descriptivo. Recuerde que los significados suelen variar cuando el código evoluciona, de modo que debe revisar frecuentemente la corrección de los nombres elegidos.

No es una recomendación de sensaciones. En *software*, los nombres constituyen el 90 por 100 de su legibilidad. Dedique tiempo a seleccionarlos con atención y mantenga su relevancia. Los nombres son demasiado importantes como para tratarlos mal.

Fíjese en el siguiente código. ¿Para qué sirve? Si le muestro el mismo código con nombres bien elegidos, tendrá sentido, pero con este formato no es más que una masa de símbolos y números mágicos.

```
public int x() {
   int q = 0;
   int z = 0;
   for (int kk = 0; kk < 10; kk++) {
     if (l[z] == 10)
     {
       q += 10 + (l[z + 1] + l[z + 2]);
       z += 1;
     }
     else if (l[z] + l[z + 1] == 10)
     {
       q += 10 + l[z + 2];
       z += 2;
     } else {
       q += l[z] + l[z + 1];
       z += 2;
     }
   }
   return q;
}
```

A continuación, el código como debería haberse escrito. Este fragmento es en realidad menos completo que el anterior, pero detectará inmediatamente

lo que intenta hacer y es probable que pudiera crear las funciones que faltan en función de ese significado que intuye. Los números mágicos ya no lo son y la estructura del algoritmo es descriptiva y atractiva:

```
public int score() {
   int score = 0;
   int frame = 0;
   for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
     if (isStrike(frame)) {
       score += 10 + nextTwoBallsForStrike(frame);
       frame += 1;
     } else if (isSpare(frame)) {
       score += 10 + nextBallForSpare(frame);
       frame += 2;
     } else {
       score += twoBallsInFrame(frame);
       frame += 2;
     }
   }
   return score;
}
```

Los nombres bien elegidos inundan la estructura del código con descripciones. Dicha inundación define las expectativas del lector sobre el cometido de otras funciones del módulo.

Puede inferir la implementación de `isStrike()` si se fija en el código anterior. Cuando lea el método `isStrike`, será prácticamente lo que esperaba[116].

```
private boolean isStrike(int frame) {
   return rolls[frame] = 10;
}
```

N2: Elegir nombres en el nivel correcto de abstracción

No elija nombres que comuniquen implementación; seleccione nombres que reflejen el nivel de abstracción de la clase o la función con la que trabaje. Es complicado. De nuevo, nos cuesta mezclar niveles de abstracción. Siempre que realice una pasada por su código, es probable que encuentre una variable con nombre en un nivel demasiado bajo. Cambie esos nombres cuando los vea. Para que el código sea legible se necesita una mejora continua. Fíjese en la siguiente interfaz `Modem`:

```
public interface Modem {
```

```
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
  }
```

Inicialmente parece correcta. Las funciones parecen las adecuadas. De hecho lo son para muchas aplicaciones, pero piense ahora en una aplicación en la que algunos módems no se conecten mediante marcación telefónica, sino mediante cables (como los usados para conexiones domésticas a Internet). Puede que algunos se conecten enviando un número de puerto a un concentrador a través de una conexión USB. Es evidente que la noción de números de teléfono se encuentra en un nivel de abstracción equivocado. Una estrategia de nomenclatura más adecuada para este caso sería la siguiente:

```
  public interface Modem {
    boolean connect(String connectionLocator);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedLocator();
  }
```

Ahora los nombres no se limitan a números de teléfono. Se pueden usar para números de teléfono o para otros tipos de estrategia de conexión.

N3: Usar nomenclatura estándar siempre que sea posible

Los nombres son más fáciles de entender si se basan en una convención o un uso existente. Por ejemplo, si emplea el patrón `DECORATOR`, debería usar la palabra `Decorator` en los nombres de las clases. Por ejemplo, `AutoHangupModemDecorator` podría ser el nombre de una clase que permite a un módem colgar automáticamente al final de una sesión. Los patrones son un tipo de estándar. En Java, por ejemplo, las funciones que convierten objetos en representaciones de cadena suelen tener el nombre `toString`. Es mejor seguir estas convenciones que inventar otras propias.

Los equipos suelen inventar su propio sistema estándar de nombres para un proyecto concreto. Eric Evans lo denomina *lenguaje omnipresente* del

proyecto[117]. Su código debe usar los términos de este lenguaje. En definitiva, cuantos más nombres con significado especial y relevante para su proyecto utilice, más fácil será para los lectores saber de qué trata el código.

N4: Nombres inequívocos

Seleccione nombres que ilustren de forma inequívoca el funcionamiento de funciones y variables. Fíjese en este ejemplo de FitNesse:

```
private String doRename() throws Exception
{
  if (refactorReferences)
    renameReferences();
  renamePage();

  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

El nombre de esta función no indica qué hace, al menos en términos amplios y sin concretar. Además, se refuerza por la presencia de la función `renamePage` dentro de la función `doRename`. ¿Qué indican los nombres sobre la diferencia entre ambas funciones? Nada. Un nombre más acertado para la función sería `renamePageAndOptionallyAllReferences`. Puede parecerle extenso, y lo es, pero sólo se invoca desde un punto del módulo, de modo que su valor descriptivo supera su longitud.

N5: Usar nombres extensos para ámbitos extensos

La longitud de un nombre debe estar relacionada con la de su ámbito. Puede usar nombres de variables breves para ámbitos diminutos pero en ámbitos mayores debe emplear nombres extensos.

Los nombres de variables como `i` y `j` son correctos si su ámbito tiene cinco líneas de longitud. Fíjese en el siguiente fragmento del conocido juego de los bolos:

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
```

```
      g.roll(pins);
  }
```

Es totalmente claro y se complicaría si la variable `i` se cambiara por algo como `rollCount`. Por otra parte, las variables y funciones con nombres breves pierden su significado en las grandes distancias. Por tanto, cuanto mayor sea el ámbito del nombre, más extenso y preciso tendrá que ser el nombre.

N6: Evitar codificaciones

Los nombres no deben codificarse con información de tipos o ámbitos. Prefijos como `m_` o `f` no sirven de nada en los entornos actuales. Además, codificaciones de proyecto y/o subsistema como `vis_` (para un sistema de imágenes visuales) distraen la atención y son redundantes. Los entornos actuales proporcionan toda esa información sin tener que modificar los nombres. Aleje sus nombres de la contaminación húngara.

N7: Los nombres deben describir efectos secundarios

Los nombres deben describir todo lo que haga una función, variable o clase. No oculte efectos secundarios con un nombre. No utilice un simple verbo para describir una función que realiza algo más que una simple acción. Fíjese en este código de TestNG:

```
  public ObjectOutputStream getOos() throws IOException {
    if (m_oos == null) {
      m_oos = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oos;
  }
```

Esta función hace algo más que obtener `oos`; lo crea si todavía no se ha creado. Por lo tanto, un nombre más acertado sería `createOrReturnOos`.

Pruebas (Test)

T1: Pruebas insuficientes

¿Cuántas pruebas debe incluir una *suite* de pruebas? Desafortunadamente, muchos programadores dirían que las que parezcan suficientes. Una *suite* de pruebas debe probar todo lo que pueda fallar. Las pruebas son insuficientes mientras haya condiciones que no se hayan examinado o cálculos que no se hayan validado.

T2: Usar una herramienta de cobertura

Las herramientas de cobertura indican vacíos en su estrategia de pruebas. Facilitan la detección de módulos, clases y funciones insuficientemente probadas. Muchos IDE le ofrecen un indicador visual y marcan en verde las líneas cubiertas y en rojo las no cubiertas. De este modo es más rápido y sencillo detectar instrucciones `if` o `catch` cuyos cuerpos no se han comprobado.

T3: No ignorar pruebas triviales

Son fáciles de redactar y su valor documental es mayor que el coste de crearlas.

T4: Una prueba ignorada es una pregunta sobre una ambigüedad

En ocasiones dudamos de un detalle de comportamiento porque los requisitos no son claros. Podemos expresar nuestra duda sobre los requisitos en forma de prueba comentada o como prueba anotada con `@Ignore`. La decisión depende de si la ambigüedad es sobre algo que se compila o no.

T5: Probar condiciones de límite

Preste especial atención a las pruebas de condiciones de límite. Solemos acertar con la parte central de un algoritmo pero malinterpretar los límites.

T6: Probar de forma exhaustiva junto a los errores

Los errores suelen congregarse. Si detecta un error en una función, es recomendable probarla de forma exhaustiva. Seguramente no sea el único error.

T7: Los patrones de fallo son reveladores

En ocasiones diagnosticamos un problema detectando patrones de fallo en los casos de prueba. Es otro argumento para crear casos de prueba lo más completos posibles. Los casos de prueba completos, si se ordenan de forma razonable, revelan patrones.

Como ejemplo, imagine que ha detectado que todas las pruebas con un entero mayor de cinco caracteres fallan. O que fallan todas las pruebas que pasan un número negativo al segundo argumento de una función. En ocasiones, con ver el patrón de rojos y verdes de un informe de pruebas basta para hacer saltar la chispa y llegar a una solución. En el capítulo 16 encontrará un interesante ejemplo en el caso de `SerialDate`.

T8: Los patrones de cobertura de pruebas pueden ser reveladores

El análisis del código que se ejecuta o no en las pruebas superadas suele indicar porqué fallan las pruebas no superadas.

T9: Las pruebas deben ser rápidas

Una prueba lenta no se ejecuta. Cuando las cosas se ponen feas, las pruebas lentas se eliminan de la *suite*. Por lo tanto, intente que sus pruebas sean

rápidas.

Conclusión

Esta lista de heurística y síntomas no se podría considerar completa. De hecho, dudo de que alguna vez exista alguna. Pero puede que ese no sea el objetivo, ya que lo que implica esta lista es un sistema de valores.

El sistema de valores ha sido el objetivo y la base de este libro. El código limpio no se crea siguiendo una serie de reglas. No se convertirá en un maestro del *software* aprendiendo una lista de heurísticas. La profesionalidad y la maestría provienen de los valores que impulsan las disciplinas.

Bibliografía

- **[Refactoring]**: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.
- **[PRAG]**: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **[GOF]**: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.
- **[Beck97]**: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.
- **[Beck07]**: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.
- **[PPP]**: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- **[DDD]**: *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

Apéndice A
Concurrencia II

por Brett L. Schuchert

Este apéndice complementa y amplía el capítulo 13 sobre concurrencia. Se ha escrito como una serie de temas independientes que puede leer en el orden que desee. Algunas secciones están duplicadas para facilitar dicha lectura.

Ejemplo cliente/servidor

Imagine una sencilla aplicación cliente/servidor. Un servidor espera a que un cliente se conecte. Un cliente se conecta y envía una solicitud.

El servidor

A continuación le mostramos una versión simplificada de una aplicación de servidor. El código completo de este ejemplo se recoge en el Listado A-3.

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
   try {
     Socket socket = serverSocket.accept();
     process(socket);
   } catch (Exception e) {
     handle(e);
   }
```

```
}
```

Esta sencilla aplicación espera una conexión, procesa un mensaje entrante y vuelve a esperar a la siguiente solicitud cliente. El código cliente para conectarse al servidor es el siguiente:

```
private void connectSendReceive(int i) {
   try {
     Socket socket = new Socket ("localhost", PORT);
     MessageUtils.sendMessage(socket, Integer.toString(i));
     MessageUtils.getMessage(socket);
     socket.close();
   } catch (Exception e) {
     e.printStackTrace();
   }
}
```

¿Cómo se comporta esta combinación de cliente y servidor? ¿Cómo podemos describir formalmente ese rendimiento? La siguiente prueba afirma que el rendimiento es aceptable:

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
   Thread[] threads = createThreads();
   startAllThreadsw(threads);
   waitForAllThreadsToFinish(threads);
}
```

Se omite la configuración para que el ejemplo sea sencillo (véase "`ClientTest.java`" más adelante). Esta prueba afirma que debe completarse en 10 000 milisegundos.

Es un ejemplo clásico de validación del rendimiento de un sistema. Este sistema debe completar una serie de solicitudes cliente en 10 segundos. Mientras el servidor pueda procesar cada solicitud cliente a tiempo, la prueba será satisfactoria.

¿Qué sucede si la prueba falla? Aparte de desarrollar algún tipo de bucle de consulta de eventos, no hay mucho que hacer en un único proceso para aumentar la velocidad de este código. ¿Se solucionaría el problema con varios procesos? Puede, pero necesitamos saber cómo se consume el tiempo. Hay dos posibilidades:

- **E/S:** Con un socket, conectándose a la base de datos, esperando al intercambio de memoria virtual, etc.
- **Procesador:** Cálculos numéricos, procesamiento de expresiones

regulares, recolección de elementos sin usar, etc.

Los sistemas suelen tener uno de cada, pero para una operación concreta suele haber uno dominante. Si el código se vincula al procesador, mayor cantidad de *hardware* de procesamiento puede mejorar el rendimiento y hacer que se supere la prueba, pero no hay tantos ciclos de CPU disponibles, de modo que añadir procesos a un problema vinculado al procesador no hará que aumente la velocidad.

Por otra parte, si el proceso está vinculado a E/S, la concurrencia puede aumentar la eficacia. Cuando una parte del sistema espera a E/S, otra puede usar ese tiempo de espera para procesar algo distinto, maximizando el uso eficaz de la CPU disponible.

Añadir subprocesos

Imagine que la prueba de rendimiento falla. ¿Cómo podemos mejorar la producción para que la prueba de rendimiento sea satisfactoria? Si el método `process` del servidor está vinculado a la E/S, existe una forma de conseguir que el servidor use subprocesos (basta con cambiar processMessage):

```
void process (final Socket socket) {
  if (socket == null)
    return;

  Runnable clientHandler = new Runnable() {
    public void run() {
      try {
        String message = MessageUtils.getMessage(socket);
        MessageUtils.sendMessage(socket, “Processed: ” + message);
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  };

  Thread clientConnection = new Thread(clientHandler);
  clientConnection.start();
}
```

Asuma que este cambio hace que la prueba se supere[118]; el código es completo, ¿correcto?

Observaciones del servidor

El servidor actualizado completa satisfactoriamente la prueba en algo más de un segundo. Desafortunadamente, la solución genera ciertos problemas.

¿Cuántos subprocesos podría crear nuestro servidor? El código no define límites de modo que podríamos alcanzar el impuesto por la Máquina virtual de Java (MVJ), suficiente en muchos sistemas sencillos. ¿Pero y si el sistema tiene que asumir multitud de usuarios de una red pública? Si se conectan demasiados usuarios al mismo tiempo, el sistema podría colapsarse.

Pero dejemos temporalmente este problema de comportamiento. La solución mostrada tiene problemas de limpieza y estructura. ¿Cuántas responsabilidades tiene el código del servidor?

- Administración de conexiones.
- Procesamiento de clientes.
- Política de subprocesos.
- Política de cierre del servidor.

Desafortunadamente, todas estas responsabilidades se encuentran en la función `process`. Además, el código cruza varios niveles diferentes de abstracción. Por tanto, a pesar de la reducida función `process`, es necesario dividirlo.

Existen varios motivos para cambiar el servidor; por tanto, incumple el principio de responsabilidad única. Para mantener la limpieza de un sistema concurrente, la administración de subprocesos debe limitarse a una serie de puntos controlados. Es más, el código que gestione los subprocesos únicamente debe encargarse de la gestión de subprocesos. ¿Por qué? Si no existe otro motivo, el control de problemas de concurrencia ya es lo suficientemente complicado como para generar simultáneamente otros problemas no relacionados con la concurrencia.

Si creamos una lista independiente para cada una de las responsabilidades anteriores, incluyendo la administración de subprocesos, al cambiar la estrategia de administración de subprocesos, el cambio tiene un menor impacto sobre el código y no contamina a otras responsabilidades. De este

modo también es más sencillo probar las demás responsabilidades sin necesidad de preocuparse de los subprocesos. Veamos la versión actualizada que se encarga de ello:

```
public void run() {
   while (keepProcessing) {
     try {
       ClientConnection clientConnection = connectionManager.awaitClient();
       ClientRequestProcessor requestProcessor
         = new ClientRequestProcessor(clientConnection);
       clientScheduler.schedule(requestProcessor);
     } catch (Exception e) {
       e.printStackTrace();
     }

   }
   connectionManager.shutdown();

}
```

Ahora centra en el mismo punto todos los aspectos relacionados con los subprocesos: `clientScheduler`. Si hay problemas de concurrencia, bastará con examinar un punto concreto:

```
public interface ClientScheduler {
   void schedule(ClientRequestProcessor requestProcessor);
}
```

La política actual es fácil de implementar:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
   public void schedule(final ClientRequestProcessor requestProcessor) {
     Runnable runnable = new Runnable() {
       public void run() {
         requestProcessor.process();
       }
     };

     Thread thread = new Thread(runnable);
     thread.start();
   }
}
```

Tras aislar la administración de subprocesos, resulta más sencillo cambiar el control de los mismos. Por ejemplo, para cambiar a la estructura `Executor` de Java 5 es necesario crear una nueva clase y conectarla (véase el Listado A-1).

Listado A-1

ExecutorClientScheduler.java.

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
   Executor executor;

   public ExecutorClientScheduler(int availableThreads) {
     executor = Executors.newFixedThreadPool(availableThreads);
   }

   public void schedule(final ClientRequestProcessor requestProcessor) {
     Runnable runnable = new Runnable() {
       public void run() {
         requestProcessor.process();
       }
     };
     executor.execute(runnable);
   }
}
```

Conclusión

En este ejemplo concreto, la presencia de la concurrencia ilustra una forma de mejorar la producción de un sistema y otra de validar dicha producción a través de una estructura de pruebas. Al centrar el código de concurrencia en un número reducido de clases, aplicamos el Principio de responsabilidad única. En el caso de la programación concurrente, resulta especialmente importante debido a su complejidad.

Posibles rutas de ejecución

Repase el método `incrementValue`, un método de Java de una línea sin bucles ni ramificaciones:

```
public class IdGenerator {
   int lastIdUsed;

   public int incrementValue() {
     return ++lastIdUsed;
   }
}
```

Ignore el desbordamiento de enteros e imagine que solamente un

subproceso accede a una instancia de `IdGenerator`. En este caso existe una sola ruta de ejecución y un único resultado garantizado:

- El valor devuelto es igual al valor de `lastIdUsed`, y ambos son una unidad mayores que antes de invocar el método.

¿Qué sucede si usamos dos subprocesos y no cambiamos el método? ¿Cuáles son los posibles resultados si cada subproceso invoca `incrementValue` una vez? ¿Cuántas rutas de ejecución posibles hay? Primero, los resultados (imagine que el valor inicial de `lastIdUsed` es 93):

- El primer subproceso obtiene el valor 94, el segundo el valor 95 y `lastIdUsed` es 95.
- El primer subproceso obtiene el valor 95, el segundo el valor 94 y `lastIdUsed` es 95.
- El primer subproceso obtiene el valor 94, el segundo el valor 94 y `lastIdUsed` es 94.

El resultado final, aunque sorprendente, es posible. Para ver los distintos resultados, debemos comprender las diferentes rutas de ejecución posibles y cómo las ejecuta la MVJ.

Número de rutas

Para calcular el número de rutas de ejecución posibles, comenzaremos con el código de *bytes* generado. La única línea de Java (`return ++lastIdUsed;`) se convierte en ocho instrucciones de código de *bytes*. Los dos subprocesos pueden intercambiar la ejecución de estas ocho instrucciones del mismo modo que mezclamos las cartas de una baraja[119]. Incluso con sólo ocho cartas en cada mano, el número de posibles resultados es sorprendente.

Para este sencillo caso de N instrucciones en una secuencia, sin bucles ni condicionales y T subprocesos, el número total de posibles rutas de ejecución es igual a:

$$\frac{(NT)!}{N!^T}$$

Calcular las órdenes posibles

```
Extraído de un correo electrónico de Uncle Bob a Brett:

Con N pasos y T subprocesos hay T * N pasos totales. Antes de cada paso
hay un conmutador de contexto que elige entre los subprocesos. Por tanto,
cada ruta se representa como una cadena de dígitos que denota los cambios
de contexto. Dados los pasos A y B y los subprocesos 1 y 2, las seis rutas
posibles son 1122, 1212, 1221, 2112, 2121 y 2211. O, en términos de pasos,
A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 y A2B2A1B1. Para tres
subprocesos, la secuencia sería 112233, 112323, 113223, 113232, 112233,
121233, 121323, 121332, 123132, 123123…
Una característica de estas cadenas es que siempre debe haber N instancias
de cada T. Por tanto, la cadena 111111 no es válida ya que tiene seis
instancias de 1 y ninguna de 2 y 3.
Por tanto, necesitamos las permutaciones de N 1, N 2… y N T. En realidad
son las permutaciones de N * T tomando cada vez N * T, que es (N * T)!,
pero sin los duplicados. Por tanto, el truco consiste en contar los
duplicados y restarlos de (N * T)!.
Dados dos pasos y dos subprocesos, ¿cuántos duplicados hay? Cada cadena de
cuatro dígitos tiene dos 1 y dos 2. Estos pares se pueden intercambiar sin
modificar el sentido de la cadena. Podríamos intercambiar los 1 o los 2, o
ninguno. Por tanto hay cuatro isomorfas por cada cadena, lo que significa
que hay tres duplicados, de modo que tres de las cuatro opciones son
duplicados; por otra parte, una de las cuatro permutaciones no son
duplicados. 4! * .25 = 6. Este razonamiento parece funcionar.
¿Cuántos duplicados hay? Si N = 2 y T = 2, podría intercambiar los 1, los
2, o ambos. En el caso de N = 2 y T = 3, podría intercambiar los 1, los 2,
los 3, 1 y 2, 1 y 3, o 2 y 3. El intercambio son las permutaciones de N.
Imagine que hay P permutaciones de N. El número de formas diferentes de
organizar dichas permutaciones es P**T.
Por tanto el número de isomorfas posibles es N!**T. Y el número de rutas
es (T*N)!/(N!**T). De nuevo, en nuestro caso T = 2, N = 2 obtenemos 6
(24/4).
Para N = 2 y T = 3 obtenemos 720/8 = 90.
Para N = 3 y T = 3 obtenemos 9!/6^3 = 1680.
```

En nuestro sencillo caso de una sola línea de código Java, que equivale a ocho líneas de código de *bytes* y a dos subprocesos, el número total de posibles rutas de ejecución es 12 870. Si el tipo de `lastIdUsed` es `long`, cada lectura y escritura se convierte en dos operaciones y no una, y el número de posibilidades asciende a 2 704 156.

¿Qué sucede si realizamos un cambio en este método?

```
public synchronized void incrementValue() {
   ++lastIdUsed;
}
```

El número de posibles rutas de ejecución es dos para dos subprocesos y N! para el caso general.

Un examen más profundo

¿Qué piensa del sorprendente resultado de dos subprocesos que invoquen el método una vez (antes de añadir `synchronized`) y obtengan el mismo resultado numérico? ¿Cómo es posible? Vayamos por partes.

¿Qué es una operación atómica? Podemos definir una operación atómica como toda operación ininterrumpible. Por ejemplo, en el siguiente código, la línea 5, donde se asigna `0` a `lastId`, es atómica ya que de acuerdo al modelo de memoria de Java, la asignación a un valor de 32 bits es ininterrumpible.

```
01:     public class Example {
02:          int lastId;
03:
04:          public void resetId() {
05:            value = 0;
06:          }
07:
08:          public int getNextId() {
09:            ++value;
10:          }
11:     }
```

¿Qué sucede si cambiamos el tipo de `lastId` de `int` a `long`? ¿Sigue siendo atómica la línea 5? No de acuerdo a la especificación de la MVJ. Podría ser atómica en un procesador concreto, pero según la especificación de la MVJ, la asignación a un valor de 64 bits requiere dos asignaciones de 32 bits. Esto significa que entre la primera y la segunda podría irrumpir otro subproceso y cambiar uno de los valores.

¿Y qué sucede con el operador de preincremento, `++`, de la línea 9? Este operador se puede interrumpir, de modo que no es atómico. Para entenderlo, repasemos el código de *bytes* de ambos métodos.

Antes de continuar, hay tres definiciones importantes:

- **Marco:** La invocación de un método requiere un marco, el cual incluye la dirección de devolución, los parámetros pasados al método y las variables locales definidas en el mismo. Es una técnica estándar empleada para definir una pila de invocaciones, que se usa en muchos lenguajes modernos para permitir la invocación de funciones y métodos básicos, además de invocaciones recursivas.
- **Variable local:** Las variables definidas en el ámbito del método. Todos los métodos no estáticos tienen al menos una variable, `this`, que representa el objeto actual, el objeto que ha recibido el último mensaje (en el subproceso actual) que ha propiciado la invocación del método.
- **Pila de operandos:** Muchas instrucciones de la MVJ aceptan parámetros. La pila de operandos es donde se incluyen dichos parámetros. La pila es una estructura de datos LIFO (*Last-In, First-Out* o Último en entrar, primero en salir) estándar.

Veamos el código de *bytes* generado para `resetId()`.

Nemónico	**Descripción**	**Pila de operandos posterior**
`ALOAD 0`	Cargar la variable 0ª en la pila de operandos. ¿Qué es la variable 0ª? Es `this.`, el objeto actual. Al invocar el método, el receptor del mensaje, una instancia de `Example`, se envía a la matriz de variables locales del marco creado para la invocación de métodos. Siempre es la primera variable que se añade a todos los métodos de instancia.	`this`
`ICONST_0`	Incluir el valor constante 0 en la pila de operandos.	`this, 0`
`PUTFIELD lastId`	Almacenar el valor superior de la pila (0) en el valor de campo del objeto denominado por la referencia de objeto una posición alejada de la parte superior de la pila, `this`.	`<Vacío>`

Estas tres instrucciones son atómicas ya que a pesar de que el subproceso que las ejecuta podría verse interrumpido por cualquiera de ellas, la información para la instrucción `PUTFIELD` (el valor constante `0` de la parte superior de la pila y la referencia a éste una posición inferior, junto con el valor del campo) no se ve alterada por ningún otro subproceso. Por tanto, al producirse la asignación, sabemos que el valor 0 se almacena en el valor del campo. La operación es atómica. Todos los operandos procesan información local del método, de modo que no hay interferencias entre subprocesos.

Si estas instrucciones se ejecutan en diez subprocesos, hay 4.38679733629e+24 ordenaciones posibles. Sin embargo, sólo hay un resultado posible, de modo que las distintas ordenaciones son irrelevantes. Y además, se garantiza el mismo resultado para valores `long` en este caso. ¿Por qué? Los diez subprocesos asignan un valor constante. Aunque se entremezclen, el resultado final será el mismo. Habrá problemas con la operación `++` en el método `getNextId`. Imagine que `lastId` contiene `42` al inicio de este método. Veamos el código de *bytes* de este nuevo método:

Nemónico	**Descripción**	**Pila de operandos posterior**
`ALOAD 0`	Cargar `this` en la pila de operandos.	`this`
`DUP`	Copiar la parte superior de la pila. Ahora tenemos dos copias de `this` en la pila de operandos.	`this, this`
`GETFIELD lastId`	Recuperar el valor del campo `lastId` del objeto al que se apunta en la parte superior de la pila (`this`) y volver a almacenar el valor en la pila.	`this, 42`
`ICONST_1`	Desplazar la constante entera 1 en la pila.	`this, 42, 1`
`IADD`	Suma entera de los dos valores superiores de la pila de operandos y volver a almacenar el resultado en la pila.	`this, 43`
`DUP_X1`	Duplicar el valor 43 y añadirlo delante de `this`.	`43, this, 43`
`PUTFIELD`	Almacenar el valor superior de la pila de operandos,	`43`

`value`	43, en el valor de campo del objeto actual, representado por el siguiente valor superior de la pila de operandos, `this`.	
`IRETURN`	Devolver el valor superior (y único) de la pila de operandos.	`<Vacío>`

Imagine que el primer subproceso completa las tres primeras instrucciones, hasta `GETFIELD` incluida y después se interrumpe. Aparece un segundo subproceso y ejecuta el método completo, incrementando `lastId` en uno; devuelve 43. Tras ello, el primer subproceso retoma desde donde se detuvo; 42 sigue en la pila de operandos por ser el valor de `lastId` cuando ejecutó `GETFIELD`. Suma uno para obtener 43 y almacena el resultado.

El valor 43 también se devuelve al primer subproceso. Como resultado, uno de los incrementos se pierde ya que el primer subproceso interfiere con el segundo después de que éste haya interrumpido al primero.

Al convertir el método `getNextId()` en `synchronized` se corrige este problema.

Conclusión

No se necesita un conocimiento extenso del código de *bytes* para entender cómo unos subprocesos interrumpen a otros. Si consigue entender este ejemplo, demostrará la posibilidad de varios subprocesos entrelazados, un conocimiento suficiente.

Dicho esto, lo que este sencillo ejemplo revela es la necesidad de entender el modelo de memoria para saber qué se permite y qué no. Equivocadamente se piensa que el operador `++` (pre o postincremento) es atómico, y evidentemente no lo es. Esto significa que tiene que saber:

- Dónde están los objetos y valores compartidos.
- El código que provoca problemas de lectura/actualización concurrente.
- Cómo evitar que se produzcan dichos problemas.

Conocer su biblioteca

La estructura Executor

Como mostramos en `ExecutorClientScheduler.java`, la estructura `Executor` de Java 5 permite la ejecución sofisticada por medio de agrupaciones de subprocesos. Es una clase del paquete `java.util.concurrent`. Si va a crear subprocesos y no usa una agrupación de subprocesos o utiliza una creada a mano, considere el uso de Executor. Hace que el código sea más limpio, más fácil de entender y de menor tamaño.

La estructura `Executor` agrupa subprocesos, los cambia automáticamente de tamaño y los vuelve a crear si es necesario. También admite *futuros*, una construcción de programación concurrente habitual. La estructura `Executor` funciona con clases que implementan `Runnable` y también con clases que implementan la interfaz `Callable`. `Callable` se parece a `Runnable`, pero puede devolver un resultado, una necesidad habitual en soluciones de múltiples subprocesos.

Un *futuro* resulta muy útil cuando el código tiene que ejecutar varias operaciones independientes y esperar a que terminen:

```
public String processRequest(String message) throws Exception {
   Callable<String> makeExternalCall = new Callable<String>() {
     public String call() throws Exception {
       String result = “”;
       // realizar solicitud externa
       return result;
     }
   };

   Future<String> result = executorService.submit(makeExternalCall);
   String partialResult = doSomeLocalProcessing();
   return result.get() + partialResult;
}
```

En este ejemplo, el método comienza a ejecutar el objeto `makeExternalCall`, prosigue con otro procesamiento y la última línea invoca `result.get()`, que se bloquea hasta que el futuro termina.

Soluciones no bloqueantes

La MV Java 5 aprovecha el diseño de los procesadores modernos que admiten actualizaciones fiables y no bloqueantes. Imagine una clase que usa sincronización (y por tanto bloqueo) para proporcionar la actualización compatible con subprocesos de un valor:

```
public class ObjectWithValue {
   private int value;
   public void synchronized incrementValue() { ++value; }
   public int getValue() { return value; }
}
```

Java 5 dispone de varias clases nuevas para este tipo de situaciones, como por ejemplo `AtomicBoolean`, `AtomicInteger` y `AtomicReference`. Podemos modificar el código anterior para usar un enfoque no bloqueante:

```
public class ObjectWithValue {
   private AtomicInteger value = new AtomicInteger(0);

   public void incrementValue() {
     value.incrementAndGet();
   }
   public int getValue() {
     return value.get();
   }
}
```

Aunque use un objeto en lugar de una primitiva y envíe mensajes como `incrementAndGet()` en lugar de `++`, el rendimiento de esta clase supera en la mayoría de los casos al de la versión anterior. En algunos casos será ligeramente más rápido pero los casos en los que es más lento son prácticamente inexistentes.

¿Cómo es posible? Los procesadores modernos disponen de una operación denominada CAS (Compare and Swap, Comparar e intercambiar). Es una operación similar al bloqueo optimista de una base de datos, mientras que la versión sincronizada es similar al bloqueo pesimista.

La palabra clave `synchronized` siempre adquiere un bloqueo, incluso cuando un segundo subproceso no intenta actualizar el mismo valor. Aunque el rendimiento de los bloqueos intrínsecos ha mejorado con respecto a versiones anteriores, sigue siendo muy costoso.

La versión no bloqueante asume inicialmente que varios subprocesos no

modifican el mismo valor con la suficiente periodicidad como para generar un problema. Por el contrario, detecta de forma eficaz si se produce dicha situación y la reintenta hasta que la actualización es satisfactoria. Esta detección suele ser menos costosa que la adquisición de un bloqueo, incluso en situaciones de contención moderada o alta.

¿Cómo lo hace la MV? La operación CAS es atómica. Por tanto, la operación CAS tiene este aspecto:

```
int variableBeingSet;

void simulateNonBlockingSet (int newValue) {
   int currentValue;
   do {
     currentValue = variableBeingSet
   } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
   if(variableBeingSet == currentValue) {
     variableBeingSet = newValue;
     return currentValue;
   }

   return variableBeingSet;
}
```

Cuando un método intenta actualizar una variable compartida, la operación CAS comprueba que la variable establecida sigue teniendo el último valor conocido. En caso afirmativo, se cambia la variable. En caso contrario, la variable no se establece ya que otro subproceso ha conseguido acceder.

El método que realiza el intento (mediante la operación CAS) ve que el cambio no se ha realizado y lo intenta de nuevo.

Clases incompatibles con subprocesos

Existen clases que no son compatibles con subprocesos, como las siguientes:

- `SimpleDateFormat`.
- Conexiones de base de datos.
- Contenedores de `java.util`.

- Servlet.

Algunas clases de colección tienen métodos concretos compatibles con subprocesos. Sin embargo, cualquier operación que invoque más de un método no lo es. Por ejemplo, si no quiere reemplazar algo en `HashTable` porque ya existe, podría crear el siguiente código:

```
if(!hashTable.containsKey(someKey)) {
   hashTable.put(someKey, new SomeValue());
}
```

Cada uno de los métodos es compatible con subprocesos. Sin embargo, otro subproceso podría añadir un valor entre las invocaciones de `containsKey` y `put`. Existen varias formas de solucionar este problema:

- Bloquear primero `HashTable` y comprobar que los demás usuarios de `HashTable` hagan lo mismo; bloqueo basado en clientes:

```
synchronized(map) {
   if(!map.containsKey(key))
     map.put(key,value);
}
```

- Envolver `HashTable` en su propio objeto y usar dos API distintas; bloqueo basado en servidores con un adaptador:

```
public class WrappedHashtable<K, V> {
   private Map<K, V> map = new Hashtable<K, V>();

   public synchronized void putIfAbsent(K key, V value) {
     if (map.containsKey(key))
       map.put(key, value);
   }
}
```

- Usar colecciones compatibles con subprocesos:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,
String>();
map.putIfAbsent(key, value);
```

Las colecciones de `java.util.concurrent` incluyen operaciones como `putIfAbsent()` para acomodar este tipo de operaciones.

Las dependencias entre métodos pueden afectar al código concurrente

El siguiente ejemplo es una forma de añadir dependencias entre métodos:

```
public class IntegerIterator implements Iterator<Integer>
   private Integer nextValue = 0;

   public synchronized boolean hasNext() {
     return nextValue < 100000;
   }
   public synchronized Integer next() {
     if (nextValue == 100000)
       throw new IteratorPastEndException();
     return nextValue++;
   }
   public synchronized Integer getNextValue() {
     return nextValue;
   }
}
```

Veamos otro código que usa `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
   int nextValue = iterator.next();
   // hacer algo con nextValue
}
```

Si un subproceso ejecuta este código no habrá problemas. ¿Qué sucede si dos subprocesos intentan compartir una misma instancia de `IntegerIterator` para procesar el valor que reciba cada uno pero cada elemento de la lista sólo se procesa una vez? En la mayoría de los casos, no hay consecuencias negativas; los subprocesos comparten la lista, procesan los elementos devueltos por el iterador y se detienen cuando éste termina. Sin embargo, existe la posibilidad de que al final de la iteración los dos subprocesos interfieran entre ellos y provoquen que uno supere el final del iterador y se genere una excepción.

El problema es el siguiente: El subproceso 1 pregunta `hasNext()`, que devuelve `true`. El subproceso 1 se evita y el subproceso 2 realiza la misma pregunta, que sigue siendo `true`. Tras ello, el subproceso 2 invoca `next()`, que devuelve un valor, como era de esperar, pero como efecto secundario

hace que `hasNext()` devuelva `false`.

Se vuelve a iniciar el subproceso 1, pensando que `hasNext()` sigue siendo `true`, y después invoca `next()`. Aunque los métodos concretos están sincronizados, el cliente usa dos métodos.

Es un problema real y un ejemplo que puede surgir en código concurrente. En este caso concreto, el problema es especialmente sutil ya que la única ocasión en la que produce un fallo es durante la iteración final del iterador. Si los subprocesos se dividen de forma correcta, puede que uno supere el final del iterador. Es el tipo de error que surge en un sistema que lleva tiempo en producción, y es difícil de detectar. Tiene tres opciones:

- Tolerar el fallo.
- Solucionar el problema cambiando el cliente: bloqueo basado en el cliente.
- Solucionar el problema cambiando el servidor, lo que también provoca que cambie el cliente: bloqueo basado en el servidor.

Tolerar el fallo

En ocasiones, los sistemas se configuran para que un fallo no produzca daños. Por ejemplo, el cliente anterior podía capturar la excepción y limpiarla, aunque sería un tanto torpe. Es como limpiar fugas de memoria reiniciando a medianoche.

Bloqueo basado en el cliente

Para que `IntegerIterator` funcione correctamente con varios subprocesos, cambie el cliente (y los demás) como se indica a continuación:

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
   int nextValue;
   synchronized (iterator) {
     if (!iterator.hasNext())
       break;
     nextValue = iterator.next();
```

```
    }
    doSometingWith(nextValue);
  }
```

Cada cliente añade un bloqueo a través de la palabra clave `synchronized`. Esta duplicación incumple el principio DRY, pero puede ser necesaria si el código usa agrupaciones de terceros no compatibles con subprocesos.

La estrategia es arriesgada ya que todos los programadores que usen el servidor deben acordarse de bloquearlo antes de usarlo y de desbloquearlo cuando terminen. Hace muchos años, trabajé en un sistema que usaba el bloqueo basado en el cliente en un recurso compartido. El recurso se usaba en cientos de puntos distintos del código. Un pobre programador se olvidó de bloquear el recurso en uno de esos puntos.

Era un sistema de varios terminales con *software* de contabilidad para el sindicato de transportistas. Local 705. El ordenador se encontraba en una sala de temperatura controlada de un piso elevado, a unas 50 millas al norte de la sede de Local 705. En la sede, decenas de trabajadores introducían datos en las terminales, conectadas al ordenador mediante líneas telefónicas dedicadas y módem semidúplex de 600bps (esto fue hace *mucho*, mucho tiempo).

Una vez al día, una de las terminales se bloqueaba, sin razón aparente. El bloqueo no tenía preferencia alguna por una terminal o una hora concreta. Es como si alguien echara a suertes la terminal que bloquear y la hora del bloqueo. En ocasiones, se bloqueaba más de una terminal. En ocasiones, podían pasar varios días sin bloqueos.

Inicialmente, se optó por reiniciar como solución, pero era complicado coordinar los reinicios. Tenemos que avisar a la sede y esperar a que todos terminaran lo que estuvieran haciendo en todas las terminales. Tras ello, se apagaba el sistema y se reiniciaba. Si alguien estaba haciendo algo importante para lo que necesitaba una o dos horas, la terminal bloqueada tenía que seguir bloqueada.

Tras varias semanas de depuración, descubrimos que la causa era un contador de búfer circular desincronizado con su puntero. Este búfer controlaba la salida a la terminal. El valor del puntero indicaba que el búfer estaba vacío pero el contador mostraba que estaba lleno. Como estaba vacío, no había nada que mostrar; pero como también estaba lleno, no se podía añadir nada al búfer que mostrar en la pantalla.

Sabíamos qué era lo que bloqueaba las terminales pero no qué provocaba la desincronización del búfer circular, por lo que añadimos un truco para resolver el problema. Se podían leer los conmutadores del panel frontal en el ordenador (esto fue hace mucho, mucho, mucho tiempo). Diseñamos una función de trampa que detectaba si uno de los conmutadores se había generado y después buscábamos un búfer circular que estuviera tanto lleno como vacío. Si lo encontrábamos, lo variábamos. ¡*Voilá*! La terminal bloqueada volvía a funcionar. De este modo no era necesario reiniciar el sistema si una terminal se bloqueaba. La sede nos llamaba y nos decía que había un bloqueo, nos acercábamos hasta la sala de ordenadores y pulsábamos un conmutador.

En ocasiones ellos trabajan los fines de semana pero nosotros no. Por ello, añadimos una función al programador que comprobaba los búfer circulares una vez por minuto y restablecía los que estuvieran tanto llenos como vacíos. De este modo se descongestionaban las pantallas antes de que la dirección llegara al teléfono.

Necesitamos varias semanas de análisis de código de lenguaje de ensamblado antes de localizar al culpable. Habíamos calculado que la frecuencia de los bloqueos se debía a un uso desprotegido del búfer circular, así que sólo *era* necesario determinar el uso fallido. Desafortunadamente, esto fue hace mucho tiempo y no disponíamos de herramientas de búsqueda, referencias cruzadas ni de otras técnicas automáticas de ayuda. Teníamos que escudriñar los listados. En aquel frío invierno de 1971 en Chicago aprendí que los bloqueos basados en el cliente son verdaderamente terribles.

Bloqueo basado en el servidor

La duplicación se puede eliminar si modificamos `IntegerIterator` de esta forma:

```
public class IntegerIteratorServerLocked {
   private Integer nextValue = 0;
   public synchronized Integer getNextOrNull() {
     if (nextValue < 100000)
       return nextValue++;
     else
       return null;
```

```
    }
  }
```

Y también cambia el código cliente:

```
  while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
      break;
    // hacer algo con nextValue
  }
```

En este caso, en realidad cambiamos la API de la clase para que sea compatible con el subproceso[120]. El cliente debe realizar una comprobación de `null` en lugar de comprobar `hasNext()`.

Por lo general, el bloqueo basado en el servidor es preferible por estos motivos:

- Reduce el código repetido: El bloqueo basado en el servidor hace que el cliente bloquee correctamente el servidor. Al incluir el código de bloqueo en el servidor, se libera a los clientes para usar el objeto y no tener que preocuparse de crear código de bloqueo adicional.
- Permite un mejor rendimiento: Puede intercambiar un servidor compatible con subprocesos por otro incompatible en caso de desarrollo de un solo subproceso, lo que evita la sobrecarga.
- Reduce las posibilidades de error: Sólo se necesita un programador que se olvide del bloqueo.
- Aplica una única política: La política se aplica solamente al servidor, no a todos los clientes.
- Reduce el ámbito de las variables compartidas: El cliente las desconoce y tampoco sabe cómo se bloquean. Todo se oculta en el servidor. Cuando se produce un fallo, su origen se busca en menos puntos.

¿Y si no es el propietario del código de servidor?

- Usar un adaptador para cambiar la API y añadir bloqueo

```
  public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
```

```
    if(iterator.hasNext())
      return iterator.next();
    return null;
  }
}
```

- Mejor todavía, usar colecciones compatibles con subprocesos con interfaces ampliadas.

Aumentar la producción

Imagine que desea leer el contenido de una serie de páginas de una lista de URL en la red. Al leer cada página, la analizamos para acumular estadísticas. Después de leer todas, imprimimos un informe de resumen.

La siguiente clase devuelve el contenido de una página, dada una URL.

```
public class PageReader {
  //…
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);

    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

La siguiente clase es el iterador que proporciona el contenido de las páginas en función de un iterador de URL:

```
public class PageIterator {
  private PageReader reader;
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
    if (urls.hasNext())
      getPageFor(urls.next());
```

```
      else
        return null;
    }

    public String getPageFor(String url) {
      return reader.getPageFor(ur1);
    }
  }
```

Se puede compartir una instancia de `PageIterator` entre varios subprocesos distintos, cada uno con su propia instancia de `PageReader` para leer las páginas que obtiene del iterador.

Hemos reducido el tamaño del bloque `synchronized`. Simplemente contiene la sección crítica de `PageIterator`. Siempre conviene sincronizar lo menos posible.

Cálculo de producción de un solo subproceso

Vayamos con los cálculos. Imagine lo siguiente, de acuerdo al argumento anterior:

- Tiempo de E/S para recuperar una página (de media): 1 segundo.
- Tiempo de procesamiento para analizar la página (de media): .5 segundos.
- E/S requiere 0 por 100 de la CPU mientras que el procesamiento requiere 100 por 100.

Si se procesan *N* páginas en un mismo subproceso, el tiempo de ejecución total es de 1.5 segundos * *N*. En la figura A.1 puede ver una instantánea de 13 páginas, aproximadamente 19.5 segundos.

Figura A.1. Un único subproceso

Cálculo de producción con varios subprocesos

Si se pueden recuperar páginas en cualquier orden y procesarlas de forma independiente, entonces es posible usar varios subprocesos para aumentar la producción. ¿Qué sucede si usamos tres subprocesos? ¿Cuántas páginas podemos obtener en el mismo tiempo?

Como se aprecia en la figura A.2, la solución con varios procesos permite que el análisis de las páginas vinculado al proceso se solape con la lectura de las mismas, vinculada a E/S. En un mundo ideal, significaría que el procesador se utiliza totalmente. Cada lectura de página por segundo se solapa con dos análisis. Por tanto, podemos procesar dos páginas por segundo, lo que triplica la producción de la solución con un solo proceso.

Figura A.2. Tres subprocesos concurrentes.

Bloqueo mutuo

Imagine una aplicación Web con dos agrupaciones de recursos compartidos de tamaño finito:

- Una agrupación de conexiones de base de datos para tareas locales de almacenamiento de procesos.

- Una agrupación de conexiones MQ a un repositorio principal.

Imagine que hay dos operaciones en la aplicación: crear y actualizar:

- Crear: Adquirir una conexión al repositorio principal y la base de datos. Comunicarse con el repositorio principal y después almacenar el trabajo local en la base de datos de procesos.
- Actualizar: Adquirir una conexión a la base de datos y después al repositorio principal. Leer el trabajo de la base de datos y enviarlo al repositorio principal.

¿Qué sucede con los usuarios que superan el tamaño de la agrupación? Imagine que el tamaño de cada agrupación es **10**.

- 10 usuarios intentan usar crear, de modo que se adquieren diez conexiones de base de datos y cada subproceso se interrumpe después de esta adquisición pero antes de adquirir una conexión al repositorio principal.
- 10 usuarios intentan usar actualizar, de modo que se adquieren las diez conexiones al repositorio principal y cada subproceso se interrumpe después de adquirir el repositorio principal pero antes de adquirir una conexión a la base de datos.
- Ahora los 10 subprocesos crear deben esperar a adquirir una conexión al repositorio principal pero los 10 subprocesos actualizar deben esperar a adquirir una conexión a la base de datos.
- Bloqueo mutuo. El sistema no se recupera nunca.

Puede parecerle una situación improbable pero ¿quién desea un sistema que se colapsa cada semana? ¿Quién quiere depurar un sistema con síntomas tan difíciles de reproducir? Es el tipo de problema que tarda semanas en resolverse.

Una solución habitual consiste en añadir instrucciones de depuración para determinar qué sucede. Evidentemente, estas instrucciones cambian tanto el código que el bloqueo mutuo se genera en otras situaciones y tarda meses en

volver a producirse[121].

Para solucionar realmente el problema del bloqueo absoluto, debemos entender sus causas. Para que se produzca, deben darse cuatro condiciones:

- Exclusión mutua.
- Bloqueo y espera.
- No expropiación.
- Espera circular.

Exclusión mutua

La exclusión mutua se produce cuando varios subprocesos deben usar los mismos recursos y dichos recursos

- No se pueden usar en varios subprocesos al mismo tiempo.
- Son de número limitado.

Un ejemplo típico de este tipo de recurso es una conexión de base de datos, un archivo abierto para escritura, un bloqueo de registro o un semáforo.

Bloqueo y espera

Cuando un subproceso adquiere un recurso, no lo libera hasta adquirir los demás recursos que necesita y terminar su trabajo.

No expropiación

Un subproceso no puede adueñarse de los recursos de otro. Cuando un subproceso obtiene un recurso, la única forma de que otro lo consiga es que el primero lo libere.

Espera circular

También se denomina abrazo mortal. Imagine dos subprocesos, T1 y T2, y dos recursos, R1 y R2. T1 tiene R1, T2 tiene R2. T1 también necesita R2 y T2 también necesita R1. Es similar al diagrama de la figura A.3:

Figura A.3.

Estas cuatro condiciones deben cumplirse para que se produzca un bloqueo mutuo. Si se incumple alguna de ellas, no se producirá.

Evitar la exclusión mutua

Una estrategia para evitar el bloqueo mutuo es impedir la condición de exclusión mutua, por medio de lo siguiente:

- Usar recursos que permitan un uso simultáneo, como por ejemplo,

  `AtomicInteger`.

- Incrementar el número de recursos para que sea igual o mayor que el número de subprocesos implicados.
- Comprobar que todos los recursos están libres antes de adquirir ninguno.

Desafortunadamente, la mayoría de recursos son limitados y no permiten un uso simultáneo, y es habitual que la identidad del segundo recurso se base en los resultados de operar sobre el primero, pero no se desanime, todavía

quedan tres condiciones.

Evitar bloqueo y espera

También puede eliminar el bloqueo mutuo si rechaza la espera. Compruebe cada uno de los recursos antes de obtenerlos y libere todos los recursos y comience de nuevo si detecta uno que esté ocupado. Este enfoque genera algunos problemas:

- Inanición: Un subproceso no consigue adquirir los recursos que necesita (puede que tenga una combinación exclusiva de recursos que casi nunca esté disponible).
- Bloqueo activo: Varios subprocesos pueden actuar al unísono, adquirir un recurso y liberarlo, de forma repetida. Es especialmente probable en algoritmos de programación de CPU simples (como dispositivos incrustados o algoritmos de equilibrio de subprocesos escritos a mano).

En ambos casos, se puede reducir la producción. El primero reduce la utilización de la CPU, mientras que el segundo genera una elevada utilización de la CPU sin sentido.

Aunque esta estrategia parezca ineficaz, es mejor que nada. Como ventaja, siempre se puede implementar si todo lo demás falla.

Evitar la expropiación

Otra estrategia para evitar el bloqueo mutuo consiste en permitir que todos los subprocesos se apropien de los recursos de otros. Suele realizarse a través de un sencillo mecanismo de solicitudes. Cuando un subproceso descubre que hay un recurso ocupado, le solicita al propietario que lo libere. Si el propietario también espera a otro recurso, lo libera y comienza de nuevo.

Es similar al enfoque anterior, pero, como ventaja, un subproceso puede esperar a un recurso, lo que reduce el número de reinicios. Sin embargo, la gestión de todas estas solicitudes puede resultar complicada.

Evitar la espera circular

Es el enfoque más habitual para impedir el bloqueo mutuo. En la mayoría de sistemas, basta con una sencilla convención acordada entre ambas partes.

En el ejemplo anterior del subproceso 1 que quiere tanto el recurso 1 como el 2, y el subproceso 2 que desea tanto el recurso 2 como el 1, al forzar a ambos subprocesos a que asignen los recursos en el mismo orden se imposibilita la espera circular.

En general, si todos los subprocesos pueden acordar un orden global de los recursos y si todos asignan los recursos en ese orden, el bloqueo mutuo es imposible. Pero como todas las estrategias, también se pueden producir problemas:

- El orden de adquisición puede no corresponderse al orden de uso; por tanto, un recurso adquirido al inicio puede que no se use hasta el final. Esto puede bloquear recursos más tiempo de lo estrictamente necesario.
- En ocasiones no se puede imponer un orden de adquisición de recursos. Si el ID del segundo recurso proviene de una operación realizada en el primero, ese orden no es factible.

Por tanto, existen varias formas de evitar el bloqueo mutuo. Algunas provocan inanición, mientras que otras usan la CPU en exceso y reducen la capacidad de respuesta. ¡*TANSTAAFL*![122]

El aislamiento de la parte relacionada con subprocesos de su solución para permitir ajustes y experimentación es una forma de aprender a determinar las estrategias óptimas.

Probar código con múltiples subprocesos

¿Cómo se puede crear una prueba que demuestre que el siguiente código no es correcto?

```
01:    public class ClassWithThreadingProblem {
02:         int nextId;
03:
```

```
04:         public int takeNextId() {
05:           return nextId++;
06:         }
07:    }
```

Veamos la descripción de una prueba que lo demuestre:

- Recordar el valor actual de `nextId`.
- Crear dos subprocesos y que cada uno invoque `takeNextId()` una vez.
- Comprobar que el valor de `nextId` es dos más que el inicial.
- Ejecutar hasta demostrar que `nextId` sólo se ha incrementado en uno y no en dos.

En el Listado A-2 se reproduce la prueba:

Listado A-2
ClassWithThreadingProblemTest.java.

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:      @Test
09:      public void twoThreadsShouldFailEventually() throws Exception {
10:        final ClassWithThreadingProblem classWithThreadingProblem
             = new ClassWithThreadingProblem();
11:
12:        Runnable runnable = new Runnable() {
13:          public void run() {
14:            ClassWithThreadingProblem.takeNextId();
15:          }
16:        };
17:
18:        for (int i = 0; i < 50000; ++i) {
19:          int startingId = classWithThreadingProblem.lastId;
20:          int expectedResult = 2 + startingId;
21:
22:          Thread t1 = new Thread(runnable);
```

```
23:         Thread t2 = new Thread(runnable);
24:         t1.start();
25:         t2.start();
26:         t1.join();
27:         t2.join();
28:
29:         int endingId = classWithThreadingProblem.lastId;
30:
31:         if (endingId != expectedResult)
32:           return;
33:       }
34:
35:       fail("Should have exposed a threading issue but it did not.");
36:     }
37: }
```

Línea	Descripción
10	Crear una sola instancia de `ClassWithThreadingProblem`. Debemos usar la palabra clave final ya que se usa después en una clase interna anónima.
12-16	Crear una clase interna anónima que use la instancia de `ClassWithThreadingProblem`.
18	Ejecutar este código hasta demostrar que falla, pero no tanto como para que la prueba tarde demasiado. Es un acto de equilibrio; no queremos esperar demasiado para demostrar el fallo. Elegir la cantidad de ejecuciones es complicado, aunque como veremos después, esta cifra se puede reducir considerablemente.
19	Recordar el valor inicial, la prueba intenta demostrar que el código de `ClassWithThreadingProblem` es incorrecto. Si se supera la prueba, lo habrá demostrado. Si la prueba falla, habrá sido incapaz de demostrarlo.
20	Esperamos que el valor final sea dos más que el actual.
22-23	Crear dos subprocesos que usen el objeto creado en las líneas 12-16. De este modo contamos con dos posibles subprocesos que intentan usar nuestra instancia de `ClassWithThreadingProblem` y ambos interfieren entre sí.
24-25	Hacer que los dos subprocesos se puedan ejecutar.

26-27	Esperar a que terminen los dos subprocesos antes de comprobar los resultados.
29	Registrar el valor final.
31-32	¿Es diferente `endingId` a lo que esperábamos? En caso afirmativo, se finaliza la prueba; hemos demostrado que el código es incorrecto. En caso negativo, volver a intentarlo.
35	Si hemos llegado hasta aquí, la prueba no ha podido demostrar que el código de producción era incorrecto en una cantidad de tiempo razonable; el código ha fallado. O no es incorrecto o no hemos realizado suficientes iteraciones para que se produzca la condición de fallo.

Esta prueba establece las condiciones de un problema de actualización concurrente. Sin embargo, el problema es tan infrecuente que la mayoría de las veces la prueba no lo detecta.

En realidad, para detectar el problema debemos establecer el número de iteraciones en más de un millón. Incluso con esa cantidad, en diez ejecuciones de un bucle de 1 000 000, el problema sólo apareció una vez, lo que significa que debemos aumentar las iteraciones para obtener fallos fiables. ¿Cuánto estamos dispuestos a esperar?

Aunque ajustáramos la prueba para obtener fallos fiables en un equipo, seguramente tendríamos que ajustarla con otros valores para ilustrar el fallo en otro equipo, sistema operativo o versión de la MVJ.

Y es un problema *sencillo*. Si no podemos demostrarlo, ¿qué pasará cuando detectemos problemas realmente complejos?

¿Qué enfoques debemos adoptar para demostrar este sencillo fallo? Y, sobre todo, ¿cómo podemos crear pruebas que demuestren fallos en un código más complejo? ¿Cómo podremos saber si el código tiene fallos cuando ni siquiera sabemos dónde buscar?

Veamos algunas sugerencias:

- Pruebas Monte Carlo: Crear pruebas flexibles que se puedan ajustar. Después, ejecutarlas repetidamente, por ejemplo, en un servidor de prueba, y cambiar los valores de ajuste aleatoriamente. Si las pruebas

fallan, el código es incorrecto. Diseñe las pruebas en las fases iniciales para que un servidor de integración continua las ejecute lo antes posible. Registre las condiciones de fallo de las pruebas.

- Ejecutar la prueba en todas las plataformas de desarrollo: De forma repetida y continuada. Cuanto más tiempo se ejecuten las pruebas sin fallos, más probable es que
  - El código de producción sea correcto o
  - Las pruebas no sean adecuadas para revelar los problemas.
- Ejecutar las pruebas en un equipo con distintas cargas: Si puede simular cargas similares a las del entorno de producción, hágalo.

Sin embargo, aunque realice todos estos pasos, no es probable que detecte problemas de subprocesos en el código. Los problemas más complicados son los que sólo se producen una vez cada mil millones de oportunidades. Son el azote de los sistemas complejos.

Herramientas para probar código basado en subprocesos

IBM ha creado la herramienta ConTest[123]. Lo que hace es instrumentar las clases para aumentar las probabilidades de que falle el código sin subprocesos.

No tenemos relación directa con IBM ni con el equipo que ha desarrollado ConTest. Un colega nos la descubrió. Tras varios minutos de usarla, notamos una gran mejoría en la detección de errores.

A continuación, le indicamos cómo usar ConTest:

- Crear pruebas y código de producción, asegurándonos que haya pruebas diseñadas específicamente para simular varios usuarios con diferentes cargas, como mencionamos antes.
- Instrumentar el código de pruebas y producción con ConTest.
- Ejecutar las pruebas.

Al instrumentar el código con ConTest, la tasa de éxito pasó de un fallo por cada millón de iteraciones a un fallo en 30 iteraciones. Los valores de bucle de las distintas ejecuciones de la prueba tras la instrumentación son los siguientes: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Evidentemente, las clases instrumentadas fallaban antes y con mayor fiabilidad.

Conclusión

En este capítulo hemos realizado un breve recorrido por el vasto y complejo territorio de la programación concurrente. Apenas hemos mostrado la superficie. Nos hemos centrado en disciplinas para mantener la limpieza del código concurrente, pero hay mucho más que aprender si tiene pensado diseñar sistemas concurrentes. Le recomendamos que empiece por el libro de Doug Lea *Concurrent Programming in Java: Design Principles and Patterns*[124].

En este capítulo hemos presentado la actualización concurrente y las disciplinas de sincronización y bloqueo para evitarla. Hemos visto cómo los subprocesos pueden mejorar la producción de un sistema vinculado a E/S y las técnicas limpias para lograr dichas mejoras. Hemos descrito el bloqueo mutuo y las disciplinas para evitarlo de forma limpia.

Por último, hemos analizado estrategias para mostrar problemas de concurrencia mediante la instrumentación del código.

Ejemplos de código completos

Cliente/Servidor sin subprocesos

Listado A-3
Server.java

```
package com.objectmentor.clientserver.nonthreaded;
```

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
   ServerSocket serverSocket;
   volatile boolean keepProcessing = true;

   public Server(int port, int millisecondsTimeout) throws IOException {
     serverSocket = new ServerSocket(port);
     serverSocket.setSoTimeout(millisecondsTimeout);
   }

   public void run() {
     System.out.printf("Server Starting\n");

     while (keepProcessing) {
       try {
         System.out.printf("accepting client\n");
         Socket socket = serverSocket.accept();
         System.out.printf("got client\n");
         process(socket);
       } catch (Exception e) {
         handle(e);
       }
     }
   }

   private void handle(Exception e) {
     if (!(e instanceof SocketException)) {
       e.printStackTrace();
     }
   }

   public void stopProcessing() {
     keepProcessing = false;
     closeIgnoringException(serverSocket);
   }

   void process(Socket socket) {
     if (socket == null)
       return;

     try {
       System.out.printf("Server: getting message\n");
       String message = MessageUtils.getMessage(socket);
       System.out.printf("Server: got message: %s\n", message);
       Thread.sleep(1000);
       System.out.printf("Server: sending reply: %s\n", message);
       MessageUtils.sendMessage(socket, "Processed: " + message);
       System.out.printf("Server: sent\n");
```

```
      closeIgnoringException(socket);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void closeIgnoringException(Socket socket) {
    if (socket != null)
      try {
        socket.close();
      } catch (IOException ignore) {
    }
  }

  private void closeIgnoringException(ServerSocket serverSocket) {
    if (serverSocket != null)
      try {
        serverSocket.close();
      } catch (IOException ignore) {
      }
  }
}
```

Listado A-4
ClientTest.java.

```
package com.objectmentor.clientserver.nonthreaded;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
  ServerSocket serverSocket;
  volatile boolean keepProcessing = true;
  public Server(int port, int millisecondsTimeout) throws IOException {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(millisecondsTimeout);
  }

  public void run() {
    System.out.printf("Server Starting\n");

    while (keepProcessing) {
      try {
        System.out.printf("accepting client\n");
        Socket socket = serverSocket.accept();
        System.out.printf("got client\n");
        process(socket);
      } catch (Exception e) {
```

```
        handle(e);
      }
    }
  }

  private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
      e.printStackTrace();
    }
  }

  public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
  }

  void process(Socket socket) {
    if (socket == null)
      return;

    try {
      System.out.printf("Server: getting message\n");
      String message = MessageUtils.getMessage(socket);
      System.out.printf("Server: got message: %s\n", message);
      Thread.sleep(1000);
      System.out.printf("Server: sending reply: %s\n", message);
      MessageUtils.sendMessage(socket, "Processed: " + message);
      System.out.printf("Server: sent\n");
      closeIgnoringException(socket);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void closeIgnoringException(Socket socket) {
    if (socket != null)
      try {
        socket.close();
      } catch (IOException ignore) {
      }
  }

  private void closeIgnoringException(ServerSocket serverSocket) {
    if (serverSocket != null)
      try {
        serverSocket.close();
      } catch (IOException ignore) {
      }
  }
}
```

Listado A-5

MessageUtils.java.

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
   public static void sendMessage(Socket socket, String message)
        throws IOException {
     OutputStream stream = socket.getOutputStream();
     ObjectOutputStream oos = new ObjectOutputStream(stream);
     oos.writeUTF(message);
     oos.flush();
   }

   public static String getMessage(Socket socket) throws IOException {
     InputStream stream = socket.getInputStream();
     ObjectInputStream ois = new ObjectInputStream(stream);
     return ois.readUTF();
   }
}
```

Cliente/Servidor con subprocesos

Para cambiar el servidor para que use subprocesos basta con cambiar el mensaje `process` (las nuevas líneas se muestran en negrita para destacarlas):

```
void process(final Socket socket) {
   if (socket == null)
     return;

   Runnable clientHandler = new Runnable() {
     public void run() {
       try {
         System.out.printf("Server: getting message\n");
         String message = MessageUtils.getMessage(socket);
         System.out.printf("Server: got message: %s\n", message);
         Thread.sleep(1000);
         System.out.printf("Server: sending reply: %s\n", message);
         MessageUtils.sendMessage(socket, "Processed: " + message);
         System.out.printf("Server: sent\n");
         closeIgnoringException(socket);
       } catch (Exception e) {
         e.printStackTrace();
       }
     }
```

```
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Apéndice B
org.jfree.date.SerialDate

Listado B-1
SerialDate.Java

```
/*=============================================================
* JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
*=============================================================
*
* (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
*
* Información del proyecto: http://www.jfree.org/jcommon/index.html
*
* Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
* bajo las condiciones de la Licencia pública general GNU publicada por
* la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
* otra versión posterior (de su elección).
*
* Esta biblioteca se distribuye con la intención de que sea útil, pero
* SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
* e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia
* pública general GNU si necesita más información al respecto.
*
* Debería haber recibido una copia de la Licencia pública general GNU
* junto a esta biblioteca; en caso contrario, contacte con la Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
* EE.UU.
*
```

```
24  * [Java es una marca comercial o marca comercial registrada de Sun
25  * Microsystems, Inc. en Estados Unidos y otros países.]
26  *
27  * ------------------
28  * SerialDate.java
29  * ------------------
30  * (C) Copyright 2001-2005, de Object Refinery Limited.
31  *
32  * Autor original: David Gilbert (para Object Refinery Limited);
33  * Colaborador(es): -;
34  *
35  * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
36  *
37  * Cambios (11-Oct-2001)
38  * -------------------------------------
39  * 11-Oct-2001: Reorganización de la clase y cambio a un nuevo paquete
40  * com.jrefinery.date (DG);
41  * 05-Nov-2001: Se añade un método getDescription() y se elimina la clase
42  * NotableDate (DG);
43  * 12-Nov-2001: IBD requiere el método setDescription(), una vez eliminada
    la clase
44  * NotableDate (DG); Se cambian getPreviousDayOfWeek(),
45  * getFollowingDayOfWeek() y getNearestDayOfWeek() para corregir
46  * errores (DG);
47  * 05-Dic-2001: Error corregido en la clase SpreadsheetDate (DG);
48  * 29-May-2002: Se transfieren las constantes de mes a una interfaz
    independiente
49  * (MonthConstants) (DG);
50  * 27-Ago-2002: Error corregido en el método addMonths(), gracias a Nálevka
    Petr (DG);
51  * 03-Oct-2002: Errores indicados por Checkstyle (DG) corregidos;
52  * 13-Mar-2003: Implementación de Serializable (DG);
53  * 29-May-2003: Error corregido en el método addMonths (DG);
54  * 04-Sep-2003: Implementación de Comparable. Actualización de los javadoc
    isInRange (DG);
55  * 05-Ene-2005: Error corregido en el método addYears() (1096282) (DG);
56  *
57  */
58
59  package org.jfree.date;
60
61  import java.io.Serializable;
62  import java.text.DateFormatSymbols;
63  import java.text.SimpleDateFormat;
64  import java.util.Calendar;
65  import java.util.GregorianCalendar;
```

```
66
67  /**
68  * Clase abstracta que define nuestros requisitos para manipular fechas,
68  * sin limitación a una determinada implementación.
70  * <P>
71  * Requisito 1: coincidir al menos con el procesamiento de fechas en Excel;
72  * Requisito 2: la clase es inmutable;
73  * <P>
74  * ¿Por qué no usar java.util.Date? Lo haremos, cuando tenga sentido. En
    ocasiones,
75  * java.util.Date puede ser demasiado precisa; representa un instante en el
    tiempo,
76  * con una precisión de 1/1000 de segundo (y la fecha depende de la
77  * zona horaria). En ocasiones sólo querremos representar un día concreto
    (como el 21
78  * de enero de 2015) sin preocuparnos de la hora del día, la
79  * zona horaria u otros aspectos. Para eso hemos definido DayDate.
80  * <P>
81  * Puede invocar getInstance() para obtener una subclase concreta de
    SerialDate,
82  * sin preocuparse de su implementación exacta
83  *
84  * @author David Gilbert
85  */
86  public abstract class SerialDate implements Comparable,
87                 Serializable,
88                 MonthConstants {
89
90       /** Para serialización. */
91       private static final long serialVersionUID = -2937160404674236 37L;
92
93       /** Símbolos de formato de fecha. */
94       public static final DateFormatSymbols
95           DATE_FORMAT_SYMBOLS = new
    SimpleDateFormat().getDateFormatSymbols();
96
97       /** Número de serie para el 1 de enero de 1900. */
98       public static final int SERIAL_LOWER_BOUND = 2;
99
100      /** Número de serie para el 31 de diciembre de 9999. */
101      public static final int SERIAL_UPPER_BOUND = 2958465;
102
103      /** Valor de año más bajo admitido por este formato de fecha. */
104      public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106      /** Valor de año más alto admitido por este formato de fecha. */
```

```
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Constante útil para lunes; equivale a java.util.Calendar.MONDAY.
    */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113     * Constante útil para martes; equivale a java.util.Calendar.TUESDAY.
114     * /
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118     * Constante útil para miércoles; equivale a
119     * java.util.Calendar.WEDNESDAY.
120     */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124     * Constante útil para jueves; equivale a java.util.Calendar.THURSDAY.
125     */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Constante útil para viernes; equivale a
    java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132     * Constante útil para sábado; equivale a java.util.Calendar.SATURDAY.
133     */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Constante útil para domingo; equivale a
    java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** Número de días de cada mes en años no bisiestos. */
140     static final int[] LAST_DAY_OF_MONTH =
141     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** Número de días en un año (no bisiesto) hasta el final de cada
    mes. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145       {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** Número de días en un año hasta el final del mes anterior. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
```

```
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151       /** Número de días en un año bisiesto hasta el final de cada mes. */
152       static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155       /**
156       * Número de días en un año bisiesto hasta el final del mes anterior.
157       */
158       static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160           {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162       /** Una constante útil para hacer referencia a la primera semana del
    mes. */
163       public static final int FIRST_WEEK_IN_MONTH = 1;
164
165       /** Una constante útil para hacer referencia a la segunda semana del
    mes. */
166       public static final int SECOND_WEEK_IN_MONTH = 2;
167
168       /** Una constante útil para hacer referencia a la tercera semana del
    mes. */
169       public static final int THIRD_WEEK_IN_MONTH = 3;
170
171       /** Una constante útil para hacer referencia a la cuarta semana del
    mes. */
172       public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174       /** Una constante útil para hacer referencia a la última semana del
    mes. */
175       public static final int LAST_WEEK_IN_MONTH = 0;
176
177       /** Constante de intervalo. */
178       public static final int INCLUDE_NONE = 0;
179
180       /** Constante de intervalo. */
181       public static final int INCLUDE_FIRST = 1;
182
183       /** Constante de intervalo. */
184       public static final int INCLUDE_SECOND = 2;
185
186       /** Constante de intervalo. */
187       public static final int INCLUDE_BOTH = 3;
188
189       /**
```

```
190      * Constante útil para especificar un día de la semana con respecto a
    una fecha
191      * fija.
192      */
193      public static final int PRECEDING = -1;
194
195      /**
196      * Constante útil para especificar un día de la semana con respecto a
    una fecha
197      * fija.
198      */
199      public static final int NEAREST = 0;
200
201      /**
202      * Constante útil para especificar un día de la semana con respecto a
    una fecha
203      * fija.
204      */
205      public static final int FOLLOWING = 1;
206
207      /** Una descripción para la fecha. */
208      private String description;
209
210      /**
211      * Constructor predeterminado.
212      */
213      protected SerialDate() {
214      }
215
216      /**
217      * Devuelve <code>true</code> si el código entero proporcionado
    representa un
218      * día de la semana válido y <code>false</code> en caso contrario.
219      *
220      * @param code el código del que se comprueba la validez.
221      *
222      * @return <code>true</code> si el código entero proporcionado
    representa un
223      * día de la semana válido y <code>false</code> en caso contrario.
224      */
225      public static boolean isValidWeekdayCode(final int code) {
227        switch(code) {
228          case SUNDAY:
229          case MONDAY:
230          case TUESDAY:
231          case WEDNESDAY:
```

```
232         case THURSDAY:
233         case FRIDAY:
234         case SATURDAY:
235           return true;
236         default:
237           return false;
238         }
239
240       }
241
242       /**
243       * Convierte la cadena proporcionada en un día de la semana.
244       *
245       * @param s una cadena que representa el día de la semana.
246       *
247       * @return <code>-1</code> si la cadena no se puede convertir o el día
    de
248       * la semana en caso contrario.
249       */
250       public static int stringToWeekdayCode(String s) {
251
252         final String[] shortWeekdayNames
253           = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256         int result = -1;
257         s = s.trim();
258         for (int i = 0; i < weekDayNames.length; i++) {
259           if (s.equals(shortWeekdayNames[i])) {
260             result = i;
261             break;
262           }
263           if (s.equals(weekDayNames[i])) {
264             result = i;
265             break;
266           }
267         }
268         return result;
269
270       }
271
272       /**
273       * Devuelve una representación en cadena del día de la semana
    proporcionado.
274       * <P>
275       * Necesitamos un enfoque mejor.
```

```
276       *
277       * @param weekday el día de la semana.
278       *
279       * @return una cadena que representa el día de la semana
    proporcionado.
280       */
281       public static String weekdayCodeToString(final int weekday) {
282
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285
286       }
287
288       /**
289       * Devuelve una matriz de nombres de mes.
290       *
291       * @return una matriz de nombres de mes.
292       */
293       public static String[] getMonths() {
294
295         return getMonths(false);
296
297       }
298
299       /**
300       * Devuelve una matriz de nombres de mes.
301       *
302       * @param shortened un indicador para indicar que deben devolverse los
    nombres
303       * de mes en formato reducido.
304       *
305       * @return una matriz de nombres de mes.
306       */
307       public static String[] getMonths(final boolean shortened) {
308
309         if (shortened) {
310           return DATE_FORMAT_SYMBOLS.getShortMonths();
311         }
312         else {
313           return DATE_FORMAT_SYMBOLS.getMonths();
314         }
315
316       }
317
318       /**
319       * Devuelve true si el código entero proporcionado representa un mes
```

```
válido.
     *
     * @param code el código del que se comprueba la validez.
     *
     * return <code>true</code> si el código entero proporcionado
representa un
     * mes válido.
     */
     public static boolean isValidMonthCode(final int code) {

       switch(code) {
         case JANUARY:
         case FEBRUARY:
         case MARCH:
         case APRIL:
         case MAY:
         case JUNE:
         case JULY:
         case AUGUST:
         case SEPTEMBER:
         case OCTOBER:
         case NOVEMBER:
         case DECEMBER:
           return true;
         default:
           return false;
         }

     }

     /**
     * Devuelve el trimestre del mes especificado.
     *
     * @param code el código del mes (1-12).
     *
     * @return el trimestre al que pertenece el mes.
     * @throws java.lang.IllegalArgumentException
     */
     public static int monthCodeToQuarter(final int code) {

       switch(code) {
         case JANUARY:
         case FEBRUARY:
         case MARCH: return 1;
         case APRIL:
```

```
363          case MAY:
364          case JUNE: return 2;
365          case JULY:
366          case AUGUST:
367          case SEPTEMBER: return 3;
368          case OCTOBER:
369          case NOVEMBER:
370          case DECEMBER: return 4;
371          default: throw new IllegalArgumentException(
372            “SerialDate.monthCodeToQuarter: invalid month code.”);
373        }
374
375      }
376
377      /**
378      * Devuelve una cadena que representa el mes proporcionado.
379      * <P>
380      * La cadena devuelta es la forma extensa del nombre del mes obtenido
     de la
381      * configuración regional.
382      *
383      * @param month el mes.
384      *
385      * @return una cadena que representa el mes proporcionado
386      */
387      public static String monthCodeToString(final int month) {
388
389        return monthCodeToString(month, false);
390
391      }
392
393      /**
394      * Devuelve una cadena que representa el mes proporcionado.
395      * <P>
396      * La cadena devuelta es la forma extensa o reducida del nombre del
     mes
397      * obtenido de la configuración regional.
398      *
399      * @param month el mes.
400      * @param shortened si <code>true</code> devuelve la abreviatura del
401      * mes.
402      *
403      * @return una cadena que representa el mes proporcionado.
404      * @throws java.lang.IllegalArgumentException
405      */
406      public static String monthCodeToString(final int month,
```

```
                final boolean shortened) {

        // comprobar argumentos…
        if (!isValidMonthCode(month)) {
          throw new IllegalArgumentException(
            “SerialDate.monthCodeToString: month outside valid range.”);
        }

        final String[] months;

        if (shortened) {
          months = DATE_FORMAT_SYMBOLS.getShortMonths();
        }
        else {
          months = DATE_FORMAT_SYMBOLS.getMonths();
        }

        return months[month - 1];

      }

      /**
      * Convierte una cadena en el código del mes.
      * <P>
      * Este método devuelve una de las constantes JANUARY, FEBRUARY, …,
      * DECEMBER correspondientes a la cadena. Si la cadena no se
      * reconoce, este método devuelve -1.
      *
      * @param s la cadena que analizar.
      *
      * @return <code>-1</code> si la cadena no se puede analizar, o el mes del
      * año en caso contrario.
      */
      public static int stringToMonthCode(String s) {

        final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
        final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();

        int result = -1;
        s = s.trim();

        // primero intentar analizar la cadena como entero (1-12)…
        try {
          result = Integer.parseInt(s);
```

```
451         }
452         catch (NumberFormatException e) {
453           // suprimir
454         }
455
456         // buscar por los nombres de los meses…
457         if ((result < 1) || (result > 12)) {
458           for (int i = 0; i < monthNames.length; i++) {
459             if (s.equals(shortMonthNames[i])) {
460               result = i + 1;
461               break;
462             }
463             if (s.equals(monthNames[i])) {
464               result = i + 1;
465               break;
466             }
467           }
468         }
469
470         return result;
471
472       }
473
474       /**
475       * Devuelve true si el código entero proporcionado representa una
    semana
476       * del mes válida y false en caso contrario
477       *
478       * @param code el código del que se comprueba la validez.
479       * @return <code>true</code> si el código entero proporcionado
    representa una
480       * semana del mes válida.
481       */
482       public static boolean isValidWeekInMonthCode(final int code) {
483
484         switch(code) {
485           case FIRST_WEEK_IN_MONTH:
486           case SECOND_WEEK_IN_MONTH:
487           case THIRD_WEEK_IN_MONTH:
488           case FOURTH_WEEK_IN_MONTH:
489           case LAST_WEEK_IN_MONTH: return true;
490           default: return false;
491         }
492
493       }
```

```
494
495       /**
496       * Determina si el año especificado es bisiesto o no.
497       *
498       * @param yyyy el año (entre 1900 y 9999).
499       *
500       * @return <code>true</code> si el año especificado es bisiesto.
501       */
502       public static boolean isLeapYear(final int yyyy) {
503
504         if ((yyyy % 4) != 0) {
505           return false;
506         }
507         else if ((yyyy % 400) == 0) {
508           return true;
509         }
510         else if ((yyyy % 100) == 0) {
511           return false;
512         }
513         else {
514           return true;
515         }
516
517       }
518
519       /**
520       * Devuelve el número de años bisiestos desde 1900 hasta el año
    especificado
521       * INCLUSIVE.
522       * <P>
523       * 1900 no es un año bisiesto.
524       *
525       * @param yyyy el año (entre 1900 y 9999).
526       *
527       * @return el número de años bisiestos desde 1900 hasta el año
    especificado.
528       */
529       public static int leapYearCount(final int yyyy) {
530
531         final int leap4 = (yyyy - 1896) / 4;
532         final int leap100 = (yyyy - 1800) / 100;
533         final int leap400 = (yyyy - 1600) / 400;
534         return leap4 - leap100 + leap400;
535
536       }
537
```

```
538       /**
539       * Devuelve el número del último día del mes, teniendo en cuenta los
540       * años bisiestos.
541       *
542       * @param month el mes.
543       * @param yyyy el año (entre 1900 y 9999).
544       *
545       * @return el número del último día del mes.
546       */
547       public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551           return result;
552         }
553         else if (isLeapYear(yyyy)) {
554           return result + 1;
555         }
556         else {
557           return result;
558         }
559
560       }
561
562       /**
563       * Crea una nueva fecha añadiendo el número especificado de días a la
     fecha
564       * base.
565       *
566       * @param days el número de días que añadir (puede ser negativo).
567       * @param base la fecha base.
568       *
569       * @return una nueva fecha.
570       */
571       public static SerialDate addDays(final int days, final SerialDate
     base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576       }
577
578       /**
579       * Crea una nueva fecha añadiendo el número especificado de meses a la
     fecha
580       * base.
```

```
581      * <P>
582      * Si la fecha base es próxima al final del mes, el día del resultado
583      * se puede ajustar ligeramente: 31 Mayo + 1 mes = 30 Junio.
584      *
585      * @param months el número de meses que añadir (puede ser negativo).
586      * @param base la fecha base.
587      *
588      * @return una nueva fecha.
589      */
590      public static SerialDate addMonths(final int months,
591                final SerialDate base) {
592
593        final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                / 12;
595        final int mm = (12 * base.getYYYY() + base.getMonth() + months – 1)
596                % 12 + 1;
597        final int dd = Math.min(
598          base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599        );
600        return SerialDate.createInstance(dd, mm, yy);
601
602      }
603
604      /**
605      * Crea una nueva fecha añadiendo el número especificado de años a la
     fecha
606      * base.
607      *
608      * @param years el número de años que añadir (puede ser negativo).
609      * @param base la fecha base.
610      *
611      * @return Una nueva fecha.
612      */
613      public static SerialDate addYears(final int years, final SerialDate
     base) {
614
615        final int baseY = base.getYYYY();
616        final int baseM = base.getMonth();
617        final int baseD = base.getDayOfMonth();
618
619        final int targetY = baseY + years;
620        final int targetD = Math.min(
621          baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622        );
623
624        return SerialDate.createInstance(targetD, baseM, targetY);
```

```
625
626      }
627
628      /**
629      * Devuelve la última fecha correspondiente al día de la semana
    especificado y
630      * ANTERIOR a la fecha base.
631      *
632      * @param targetWeekday un código para el día de la semana de destino.
633      * @param base la fecha base.
634      *
635      * @return la última fecha correspondiente al día de la semana
    especificado y
636      * ANTERIOR a la fecha base.
637      */
638      public static SerialDate getPreviousDayOfWeek(final int
    targetWeekday,
639                 final SerialDate base) {
640
641        // comprobar argumentos…
642        if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643          throw new IllegalArgumentException(
644            “Invalid day-of-the-week code.”
645          );
646        }
647
648        // buscar la fecha…
649        final int adjust;
650        final int baseDOW = base.getDayOfWeek();
651        if (baseDOW > targetWeekday) {
652          adjust = Math.min(0, targetWeekday - baseDOW);
653        }
654        else {
655          adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656        }
657
658        return SerialDate.addDays(adjust, base);
659
660      }
661
662      /**
663      * Devuelve la primera fecha que coincide con el día de la semana
    especificado
664      * y POSTERIOR a la fecha base.
665      *
666      * @param targetWeekday un código para el día de la semana de destino.
```

```
667      * @param base la fecha base.
668      *
669      * @return la primera fecha que coincide con el día de la semana
    especificado
670      * y POSTERIOR a la fecha base.
671      */
672      public static SerialDate getFollowingDayOfWeek(final int
    targetWeekday,
673                final SerialDate base) {
674
675        // comprobar argumentos…
676        if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677          throw new IllegalArgumentException(
678            “Invalid day-of-the-week code.”
679          );
680        }
681
682        // buscar la fecha…
683        final int adjust;
684        final int baseDOW = base.getDayOfWeek();
685        if (baseDOW > targetWeekday) {
686          adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687        }
688        else {
689          adjust = Math.max(0, targetWeekday - baseDOW);
690        }
691
692        return SerialDate.addDays(adjust, base);
693      }
694
695      /**
696      * Devuelve la fecha que coincide con el día de la semana especificado
    y más
697      * PRÓXIMA a la fecha base.
698      *
699      * @param targetDOW un código para el día de la semana de destino.
700      * @param base la fecha base.
701      *
702      * @return la fecha que coincide con el día de la semana especificado
    y más
703      * PRÓXIMA a la fecha base.
704      */
705      public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                final SerialDate base) {
707
708        // comprobar argumentos…
```

```
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710           throw new IllegalArgumentException(
711             “Invalid day-of-the-week code.”
712           );
713         }
714
715         // buscar la fecha…
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719           adjust = 7 - adjust;
720         }
721         if (adjust <= -4) {
722           adjust = 7 + adjust;
723         }
724         return SerialDate.addDays(adjust, base);
725
726       }
727
728       /**
729       * Avanzar la fecha hasta el último día del mes.
730       *
731       * @param base la fecha base.
732       *
733       * @return una nueva fecha de serie.
734       */
735       public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737           base.getMonth(), base.getYYYY()
738         );
739         return SerialDate.createInstance(last, base.getMonth(),
    base.getYYYY());
740       }
741
742       /**
743       * Devuelve una cadena correspondiente al código de la semana del mes.
744       * <P>
745       * Necesitamos un enfoque mejor.
746       *
747       * @param count un código entero que representa la semana del mes.
748       *
749       * @return una cadena correspondiente al código de la semana del mes.
750       */
751       public static String weekInMonthToString(final int count) {
752
```

```
753         switch (count) {
754           case SerialDate.FIRST_WEEK_IN_MONTH : return “First”;
755           case SerialDate.SECOND_WEEK_IN_MONTH : return “Second”;
756           case SerialDate.THIRD_WEEK_IN_MONTH : return “Third”;
757           case SerialDate.FOURTH_WEEK_IN_MONTH : return “Fourth”;
758           case SerialDate.LAST_WEEK_IN_MONTH : return “Last”;
759           default :
760             return “SerialDate.weekInMonthToString(): invalid code.”;
761         }
762
763       }
764
765       /**
766       * Devuelve una cadena que representa el valor ‘relativo’
    proporcionado.
767       * <P>
768       * Necesitamos un enfoque mejor.
769       *
770       * @param relative una constante que representa el valor ‘relativo’.
771       *
772       * @return una cadena que representa el valor ‘relativo’
    proporcionado.
773       */
774       public static String relativeToString(final int relative) {
775
776         switch (relative) {
777           case SerialDate.PRECEDING : return “Preceding”;
778           case SerialDate.NEAREST : return “Nearest”;
779           case SerialDate.FOLLOWING : return “Following”;
780           default : return “ERROR : Relative To String”;
781         }
782
783       }
784
785       /**
786       * Método de factoría que devuelve una instancia de una subclase
    concreta de
787       * {@link SerialDate}.
788       *
789       * @param day el día (1-31).
790       * @param month el mes (1-12).
791       * @param yyyy el año (entre 1900 y 9999).
792       *
793       * @return Una instancia de {@link SerialDate}
794       */
795       public static SerialDate createInstance(final int day, final int
```

```
    month,
796                 final int yyyy) {
797        return new SpreadsheetDate(day, month, yyyy);
798      }
799
800      /**
801      * Método de factoría que devuelve una instancia de una subclase
    concreta de
802      * {@link SerialDate}.
803      *
804      * @param serial numero de serie del día (1 de enero de 1900 = 2).
805      *
806      * @return una instancia de SerialDate.
807      */
808      public static SerialDate createInstance(final int serial) {
809        return new SpreadsheetDate(serial);
810      }
811
812      /**
813      * Método de factoría que devuelve una instancia de una subclase de
    SerialDate.
814      *
815      * @param date Un objeto de fecha de Java.
816      *
817      * @return una instancia de SerialDate.
818      */
818      public static SerialDate createInstance(final java.util.Date date) {
820
821        final GregorianCalendar calendar = new GregorianCalendar();
822        calendar.setTime(date);
823        return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                 calendar.get(Calendar.MONTH) + 1,
825                 calendar.get(Calendar.YEAR));
826
827      }
828
829      /**
830      * Devuelve el número de serie de la fecha, siendo el 1 de enero de
    1900 = 2 (se
831      * corresponde, casi totalmente, al sistema de numeración empleado en
    Microsoft
832      * Excel para Windows y Lotus 1-2-3).
833      *
834      * @return el número de serie de la fecha.
835      */
836      public abstract int toSerial();
```

```
837
838       /**
839       * Devuelve java.util.Date. Como java.util.Date tiene mayor precisión
     que
840       * SerialDate, debemos definir una convención para “la hora del día”.
841       *
842       * @return this como <code>java.util.Date</code>.
843       */
844       public abstract java.util.Date toDate();
845
846       /**
847       * Devuelve una descripción de la fecha.
848       *
849       * @return una descripción de la fecha.
850       */
851       public String getDescription() {
852         return this.description;
853       }
854
855       /**
856       * Establece la descripción de la fecha.
857       *
858       * @param description la nueva descripción de la fecha.
859       */
860       public void setDescription(final String description) {
861         this.description = description;
862       }
863
864       /**
865       * Convierte la fecha en una cadena.
866       *
867       * @return una representación en cadena de la fecha.
868       */
869       public String toString() {
870         return getDayOfMonth() + “-” +
     SerialDate.monthCodeToString(getMonth())
871                 + “-” + getYYYY();
872       }
873
874       /**
875       * Devuelve el año (con un intervalo válido de 1900 a 9999).
876       *
877       * @return el año.
878       */
879       public abstract int getYYYY();
880
```

```
881       /**
882       * Devuelve el mes (Enero = 1, Febrero = 2, Marzo = 3).
883       *
884       * @return el mes del año.
885       */
886       public abstract int getMonth();
887
888       /**
889       * Devuelve el día del mes.
890       *
891       * @return el día del mes.
892       */
893       public abstract int getDayOfMonth();
894
895       /**
896       * Devuelve el día de la semana.
897       *
898       * @return el día de la semana.
899       */
900       public abstract int getDayOfWeek();
901
902       /**
903       * Devuelve la diferencia (en días) entre esta fecha y la
904       * 'otra' fecha especificada.
905       * <P>
906       * El resultado es positivo si esta fecha es posterior a la 'otra' y
907       * negativo si es anterior.
908       *
909       * @param other la fecha con la que se compara.
910       *
911       * @return la diferencia entre esta fecha y la otra.
912       */
913       public abstract int compare(SerialDate other);
914
915       /**
916       * Devuelve true si esta SerialDate representa la misma fecha que la
917       * SerialDate especificada.
918       *
919       * @param other la fecha con la que se compara.
920       *
921       * @return <code>true</code> si esta SerialDate representa la misma
    fecha que
922       * la SerialDate especificada.
923       */
924       public abstract boolean isOn(SerialDate other);
```

```
925
926       /**
927       * Devuelve true si esta SerialDate representa una fecha anterior en
928       * comparación a la SerialDate especificada.
929       *
930       * @param other la fecha con la que se compara.
931       *
932       * @return <code>true</code> si esta SerialDate representa una fecha
     anterior
933       * en comparación a la SerialDate especificada.
934       */
935       public abstract boolean isBefore(SerialDate other);
936
937       /**
938       * Devuelve true si esta SerialDate representa la misma fecha que la
939       * SerialDate especificada.
940       *
941       * @param other la fecha con la que se compara.
942       *
943       * @return <code>true</code> si esta SerialDate representa la misma
     fecha
944       * que la SerialDate especificada.
945       */
946       public abstract boolean isOnOrBefore(SerialDate other);
947
948       /**
949       * Devuelve true si esta SerialDate representa la misma fecha que la
950       * SerialDate especificada.
951       *
952       * @param other la fecha con la que se compara.
953       *
954       * @return <code>true</code> si esta SerialDate representa la misma
     fecha
955       * que la SerialDate especificada.
956       */
957       public abstract boolean isAfter(SerialDate other);
958
959       /**
960       * Devuelve true si esta SerialDate representa la misma fecha que la
961       * SerialDate especificada.
962       *
963       * @param other la fecha con la que se compara.
964       *
965       * @return <code>true</code> si esta SerialDate representa la misma
     fecha
966       * que la SerialDate especificada.
```

```
967       */
968       public abstract boolean isOnOrAfter(SerialDate other);
969
970       /**
971       * Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
972       * rango especificado (INCLUSIVE). El orden de fecha de d1 y d2 no es
973       * importante.
974       *
975       * @param d1 fecha límite del rango.
976       * @param d2 la otra fecha límite del rango.
977       *
978       * @return Un valor booleano.
979       */
980       public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982       /**
983       * Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
984       * rango especificado (el invocador especifica si los puntos finales
     se
985       * incluyen o no). El orden de fecha de d1 y d2 no es importante.
986       *
987       * @param d1 fecha límite del rango.
988       * @param d2 la otra fecha límite del rango.
989       * @param include un código que controla si las fechas inicial y final
990       * se incluyen o no en el rango.
991       *
992       * @return Un valor booleano.
993       */
994       public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                 int include);
996
997       /**
998       * Devuelve la última fecha que coincide con el día de la semana
     especificado y
999       * que es ANTERIOR a esta fecha.
1000      *
1001      * @param targetDOW un código para el día de la semana de destino.
1002      *
1003      * @return la última fecha que coincide con el día de la semana
     especificado y
1004      * que es ANTERIOR a esta fecha.
1005      */
1006      public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007        return getPreviousDayOfWeek(targetDOW, this);
1008      }
1009
```

```
1010      /**
1011      * Devuelve la primera fecha que coincide con el día de la semana
     especificado
1012      * y que es POSTERIOR a esta fecha.
1013      *
1014      * @param targetDOW un código para el día de la semana de destino.
1015      *
1016      * @return la primera fecha que coincide con el día de la semana
     especificado
1017      * que es POSTERIOR a esta fecha.
1018      */
1019      public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020        return getFollowingDayOfWeek(targetDOW, this);
1021      }
1022
1023      /**
1024      * Devuelve la fecha más próxima que coincide con el día de la semana
     especificado.
1025      *
1026      * @param targetDOW un código para el día de la semana de destino.
1027      *
1028      * @return la fecha más próxima que coincide con el día de la semana
     especificado.
1029      */
1030      public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031        return getNearestDayOfWeek(targetDOW, this);
1032      }
1033
1034 }
```

Listado B-2
SerialDateTest.java

```
1   /* ===============================================================
2   * JCommon : biblioteca gratuita de clases de propósito general para
    Java(tm)
3   * ===============================================================
4   *
5   * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
6   *
7   * Información del projecto: http://www.jfree.org/jcommon/index.html
8   *
9   * Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
10  * bajo las condiciones de la Licencia pública general GNU publicada por
11  * la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
```

```
12  * otra versión posterior (de su elección).
13  *
14  * Esta biblioteca se distribuye con la intención de que sea útil, pero
15  * SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
16  * e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general
    GNU
17  * si necesita más información al respecto.
18  *
19  * Debería haber recibido una copia de la Licencia pública general GNU
20  * junto a esta biblioteca; en caso contrario, contacte con la Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22  * EE.UU.
23  *
24  * [Java es una marca comercial o marca comercial registrada de Sun
25  * Microsystems, Inc. en Estados Unidos y otros países.]
26  *
27  * -------------------------
28  * SerialDateTests.java
29  * -------------------------
30  * (C) Copyright 2001-2005, por Object Refinery Limited.
31
32  * Autor original: David Gilbert (por Object Refinery Limited);
33  * Colaborador(es): -;
34
35  * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36
37  * Cambios
38  * -----------
39  * 15-Nov-2001: Version 1 (DG);
40  * 25-Jun-2002: Se elimina la importación innecesaria (DG);
41  * 24-Oct-2002: Errores indicados Checkstyle corregidos (DG);
42  * 13-Mar-2003: Se añade prueba de serialización (DG);
43  * 05-Jan-2005: Se añade prueba para el informe de errores 1096282 (DG);
44  *
45  */
46
47  package org.jfree.date.junit;
48
49  import java.io.ByteArrayInputStream;
50  import java.io.ByteArrayOutputStream;
51  import java.io.ObjectInput;
52  import java.io.ObjectInputStream;
53  import java.io.ObjectOutput;
54  import java.io.ObjectOutputStream;
55
```

```
56 import junit.framework.Test;
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
63 /**
64 * Pruebas JUnit para la clase {@link SerialDate}.
65 */
66 public class SerialDateTests extends TestCase {
67
68      /** Fecha que representa 9 de noviembre.
69      private SerialDate nov9Y2001;
70
71      /**
72      * Crea un nuevo caso de prueba.
73      *
74      * @param name el nombre.
75      */
76      public SerialDateTests(final String name) {
77        super(name);
78      }
79
80      /**
81      * Devuelve una suite de pruebas para el ejecutor de pruebas JUnit.
82      *
83      * @return La suite de pruebas.
84      */
85      public static Test suite() {
86        return new TestSuite(SerialDateTests.class);
87      }
88
89      /**
90      * Problema.
91      */
92      protected void setUp() {
93        this.nov9Y2001 = SerialDate.createInstance(9,
   MonthConstants.NOVEMBER, 2001);
94      }
95
96      /**
97      * 9 Nov 2001 más dos meses debe ser 9 Ene 2002.
98      */
99      public void testAddMonthsTo9Nov2001() {
```

```
100        final SerialDate jan9Y2002 = SerialDate.addMonths(2,
    this.nov9Y2001);
101        final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102        assertEquals(answer, jan9Y2002);
103      }
104
105      /**
106      * Caso de prueba de un error, ya corregido.
107      */
108      public void testAddMonthsTo5Oct2003() {
109        final SerialDate d1 = SerialDate.createInstance(5,
    MonthConstants.OCTOBER, 2003);
110        final SerialDate d2 = SerialDate.addMonths(2, d1);
111        assertEquals(d2, SerialDate.createInstance(5,
    MonthConstants.DECEMBER, 2003));
112      }
113
114      /**
115      * Caso de prueba de un error, ya corregido.
116      */
117      public void testAddMonthsTo1Jan2003() {
118        final SerialDate d1 = SerialDate.createInstance(1,
    MonthConstants.JANUARY, 2003);
119        final SerialDate d2 = SerialDate.addMonths(0, d1);
120        assertEquals(d2, d1);
121      }
122
123      /**
124      * El lunes anterior al viernes 9 de noviembre de 2001 debe ser el 5 de
    noviembre.
125      */
126      public void testMondayPrecedingFriday9Nov2001() {
127        SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128          SerialDate.MONDAY, this.nov9Y2001
129        );
130        assertEquals(5, mondayBefore.getDayOfMonth());
131      }
132
133      /**
134      * El lunes posterior al viernes 9 de noviembre de 2001 debe ser el 12
    de noviembre.
135      */
136      public void testMondayFollowingFriday9Nov2001() {
137        SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138          SerialDate.MONDAY, this.nov9Y2001
139        );
```

```
140       assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142
143     /**
144     * El lunes más próximo al viernes 9 de noviembre de 2001 debe ser el
    12 de noviembre.
145     */
146     public void testMondayNearestFriday9Nov2001() {
147       SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148         SerialDate.MONDAY, this.nov9Y2001
149       );
150       assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152
153     /**
154     * El lunes más próximo al 22 de enero de 1970 cae en el 19.
155     */
156     public void testMondayNearest22Jan1970() {
157       SerialDate jan22Y1970 = SerialDate.createInstance(22,
    MonthConstants.JANUARY, 1970);
158       SerialDate mondayNearest =
    SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
159       assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161
162     /**
163     * El problema es que la conversión de días en cadenas devuelva el
    resultado
164     * correcto. En realidad este resultado depende de la configuración
    regional.
165     */
166     public void testWeekdayCodeToString() {
167
168       final String test =
    SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169       assertEquals("Saturday", test);
170
171     }
172
173     /**
174     * Probar la conversión de una cadena en día de la semana. Esta prueba
    falla si
175     * la configuración regional predeterminada no usa nombres de días en
    inglés
176     */
177     public void testStringToWeekday() {
178
```

```
179       int weekday = SerialDate.stringToWeekdayCode(“Wednesday”);
180       assertEquals(SerialDate.WEDNESDAY, weekday);
181
182       weekday = SerialDate.stringToWeekdayCode(“ Wednesday ”);
183       assertEquals(SerialDate.WEDNESDAY, weekday);
184
185       weekday = SerialDate.stringToWeekdayCode(“Wed”);
186       assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191     * Probar la conversión de una cadena en mes. Esta prueba falla si la
192     * configuración regional predeterminada no usa nombres de días en
    inglés
193     */
194     public void testStringToMonthCode() {
195
196       int m = SerialDate.stringToMonthCode(“January”);
197       assertEquals(MonthConstants.JANUARY, m);
198
199       m = SerialDate.stringToMonthCode(“ January ”);
200       assertEquals(MonthConstants.JANUARY, m);
201
202       m = SerialDate.stringToMonthCode(“Jan”);
203       assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208     * Probar la conversión de un código de mes en cadena.
209     */
210     public void testMonthCodeToStringCode() {
211
212       final String test =
    SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213       assertEquals(“December”, test);
214
215     }
216
217     /**
218     * 1900 no es un año bisiesto.
219     */
220     public void testIsNotLeapYear1900() {
221       assertTrue(!SerialDate.isLeapYear(1900));
222     }
```

```
223
224     /**
225     * 2000 es un año bisiesto.
226     */
227     public void testIsLeapYear2000() {
228       assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232     * El número de años bisiestos desde 1900 y hasta 1899 incluido es 0.
233     */
234     public void testLeapYearCount1899() {
235       assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239     * El número de años bisiestos desde 1900 y hasta 1903 incluido es 0.
240     */
241     public void testLeapYearCount1903() {
242       assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246     * El número de años bisiestos desde 1900 y hasta 1904 incluido es 1.
247     */
248     public void testLeapYearCount1904() {
249       assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253     * El número de años bisiestos desde 1900 y hasta 1999 incluido es 24.
254     */
255     public void testLeapYearCount1999() {
256       assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260     * El número de años bisiestos desde 1900 y hasta 2000 incluido es 25.
261     */
262     public void testLeapYearCount2000() {
263       assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267     * Serializar una instancia, restaurarla y comprobar la igualdad.
```

```
268     */
269     public void testSerialization() {
270
271       SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272       SerialDate d2 = null;
273
274       try {
275         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276         ObjectOutput out = new ObjectOutputStream(buffer);
277         out.writeObject(d1);
278         out.close();
279
280         ObjectInput in = new ObjectInputStream(
new               ByteArrayInputStream(buffer.toByteArray()));
281         d2 = (SerialDate) in.readObject();
282         in.close();
283       }
284       catch (Exception e) {
285         System.out.println(e.toString());
286       }
287       assertEquals(d1, d2);
288
289     }
290
291     /**
292     * Prueba para el informe de error 1096282 (ya corregido).
293     */
294     public void test1096282() {
295       SerialDate d = SerialDate.createInstance(29, 2, 2004);
296       d = SerialDate.addYears(1, d);
297       SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298       assertTrue(d.isOn(expected));
299     }
300
301     /**
302     * Diversas pruebas para el método addMonths().
303     */
304     public void testAddMonths() {
305       SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
307       SerialDate d2 = SerialDate.addMonths(1, d1);
308       assertEquals(30, d2.getDayOfMonth());
309       assertEquals(6, d2.getMonth());
310       assertEquals(2004, d2.getYYYY());
311
312       SerialDate d3 = SerialDate.addMonths(2, d1);
```

```
313        assertEquals(31, d3.getDayOfMonth());
314        assertEquals(7, d3.getMonth());
315        assertEquals(2004, d3.getYYYY());
316
317        SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1,
   d1));
318        assertEquals(30, d4.getDayOfMonth());
319        assertEquals(7, d4.getMonth());
320        assertEquals(2004, d4.getYYYY());
321      }
322 }
```

Listado B-3
MonthConstants.java

```
1  /* ==============================================================
2  * JCommon : biblioteca gratuita de clases de propósito general para Java(tm)
3  * ==============================================================
4  *
5  * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
6  *
7  * Información del proyecto: http://www.jfree.org/jcommon/index.html
8  *
9  * Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
10 * bajo las condiciones de la Licencia pública general GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
12 * otra versión posterior (de su elección).
13 *
14 * Esta biblioteca se distribuye con la intención de que sea útil, pero
15 * SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
16 * e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general
   GNU
17 * si necesita más información al respecto.
18 *
19 * Debería haber recibido una copia de la Licencia pública general GNU
20 * junto a esta biblioteca; en caso contrario, contacte con la Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * EE.UU.
23 *
24 * [Java es una marca comercial o marca comercial registrada de Sun
25 * Microsystems, Inc. en Estados Unidos y otros países.]
26 *
27 * ----------------------
28 * MonthConstants.java
```

```
29 * ----------------------
30 * (C) Copyright 2002, 2003, de Object Refinery Limited.
31 *
32 * Autor original: David Gilbert (para Object Refinery Limited);
33 * Colaborador(es): -;
34 *
35 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Cambios
38 * ----------
39 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
40 *
41 */
42
43 package org.jfree.date;
44
45 /**
46 * Constantes útiles para los meses. NO son equivalentes a las
47 * constantes definidas por java.util.Calendar (donde JANUARY=0 y
   DECEMBER=11).
48 * <P>
49 * Se usa en las clases SerialDate y RegularTimePeriod.
50 *
51 * @author David Gilbert
52 */
53 public interface MonthConstants {
54
55     /** Constante para Enero. */
56     public static final int JANUARY = 1;
57
58     /** Constante para Febrero. */
59     public static final int FEBRUARY = 2;
60
61     /** Constante para Marzo. */
62     public static final int MARCH = 3;
63
64     /** Constante para Abril. */
65     public static final int APRIL = 4;
66
67     /** Constante para Mayo. */
68     public static final int MAY = 5;
69
70     /** Constante para Junio. */
71     public static final int JUNE = 6;
72
```

```
73      /** Constante para Julio. */
74      public static final int JULY = 7;
75
76      /** Constante para Agosto. */
77      public static final int AUGUST = 8;
78
79      /** Constante para Septiembre. */
80      public static final int SEPTEMBER = 9;
81
82      /** Constante para Octubre. */
83      public static final int OCTOBER = 10;
84
85      /** Constante para Noviembre. */
86      public static final int NOVEMBER = 11;
87
88      /** Constante para Diciembre. */
89      public static final int DECEMBER = 12;
90
91 }
```

Listado B-4
BobsSerialDateTest.java

```
1   package org.jfree.date.junit;
2
3   import junit.framework.TestCase;
4   import org.jfree.date.*;
5   import static org.jfree.date.SerialDate.*;
6
7   import java.util.*;
8
9   public class BobsSerialDateTest extends TestCase {
10
11       public void testIsValidWeekdayCode() throws Exception {
12         for (int day = 1; day <= 7; day++)
13           assertTrue(isValidWeekdayCode(day));
14         assertFalse(isValidWeekdayCode(0));
15         assertFalse(isValidWeekdayCode(8));
16       }
17
18       public void testStringToWeekdayCode() throws Exception {
19
20         assertEquals(-1, stringToWeekdayCode("Hello"));
21         assertEquals(MONDAY, stringToWeekdayCode("Monday"));
```

```
22        assertEquals(MONDAY, stringToWeekdayCode(“Mon”));
23  //todo assertEquals(MONDAY,stringToWeekdayCode(“monday”));
24  // assertEquals(MONDAY,stringToWeekdayCode(“MONDAY”));
25  // assertEquals(MONDAY, stringToWeekdayCode(“mon”));
26
27        assertEquals(TUESDAY, stringToWeekdayCode(“Tuesday”));
28        assertEquals(TUESDAY, stringToWeekdayCode(“Tue”));
29        // assertEquals(TUESDAY,stringToWeekdayCode(“tuesday”));
30  // assertEquals(TUESDAY,StringToWeekdayCode(“TUESDAY”));
31  // assertEquals(TUESDAY, stringToWeekdayCode(“tue”));
32  // assertEquals(TUESDAY, stringToWeekdayCode(“tues”));
33
34        assertEquals(WEDNESDAY, stringToWeekdayCode (“Wednesday”));
35        assertEquals(WEDNESDAY, stringToWeekdayCode(“Wed”));
36        // assertEquals(WEDNESDAY,stringToWeekdayCode(“wednesday”));
37  // assertEquals(WEDNESDAY,stringToWeekdayCode(“WEDNESDAY”));
38  // assertEquals(WEDNESDAY, StringToWeekdayCode(“wed”));
39
40        assertEquals(THURSDAY, stringToWeekdayCode(“Thursday”));
41        assertEquals(THURSDAY, stringToWeekdayCode(“Thu”));
42  // assertEquals(THURSDAY,stringToWeekdayCode(“thursday”));
43  // assertEquals(THURSDAY,stringToWeekdayCode(“THURSDAY”));
44  // assertEquals(THURSDAY, stringToWeekdayCode(“thu”));
45  // assertEquals(THURSDAY, stringToWeekdayCode(“thurs”));
46
47        assertEquals(FRIDAY, stringToWeekdayCode(“Friday”));
48        assertEquals(FRIDAY, stringToWeekdayCode(“Fri”));
49  // assertEquals(FRIDAY,stringToWeekdayCode(“friday”));
50  // assertEquals(FRIDAY,stringToWeekdayCode(“FRIDAY”));
51  // assertEquals(FRIDAY, StringToWeekdayCode(“fri”));
52
53        assertEquals(SATURDAY, stringToWeekdayCode(“Saturday”));
54        assertEquals(SATURDAY, stringToWeekdayCode(“Sat”));
55        // assertEquals(SATURDAY,stringToWeekdayCode(“saturday”));
56  // assertEquals(SATURDAY,stringToWeekdayCode(“SATURDAY”));
57  // assertEquals(SATURDAY, stringToWeekdayCode(“sat”));
58
59        assertEquals(SUNDAY, stringToWeekdayCode(“Sunday”));
60        assertEquals(SUNDAY, stringToWeekdayCode(“Sun”));
61  // assertEquals(SUNDAY,stringToWeekdayCode(“sunday”));
62  // assertEquals(SUNDAY,stringToWeekdayCode(“SUNDAY”));
63  // assertEquals(SUNDAY, stringToWeekdayCode(“sun”));
64      }
65
66      public void testWeekdayCodeToString() throws Exception {
```

```
67        assertEquals(“Sunday”, weekdayCodeToString(SUNDAY));
68        assertEquals(“Monday”, weekdayCodeToString(MONDAY));
69        assertEquals(“Tuesday”, weekdayCodeToString(TUESDAY));
70        assertEquals(“Wednesday”, weekdayCodeToString(WEDNESDAY));
71        assertEquals(“Thursday”, weekdayCodeToString(THURSDAY));
72        assertEquals(“Friday”, weekdayCodeToString(FRIDAY));
73        assertEquals(“Saturday”, weekdayCodeToString(SATURDAY));
74      }
75
76      public void testIsValidMonthCode() throws Exception {
77        for (int i = 1; i <= 12; i++)
78          assertTrue(isValidMonthCode(i));
79        assertFalse(isValidMonthCode(0));
80        assertFalse(isValidMonthCode(13));
81      }
82
83      public void testMonthToQuarter() throws Exception {
84        assertEquals(1, monthCodeToQuarter(JANUARY));
85        assertEquals(1, monthCodeToQuarter(FEBRUARY));
86        assertEquals(1, monthCodeToQuarter(MARCH));
87        assertEquals(2, monthCodeToQuarter(APRIL));
88        assertEquals(2, monthCodeToQuarter(MAY));
89        assertEquals(2, monthCodeToQuarter(JUNE));
90        assertEquals(3, monthCodeToQuarter(JULY));
91        assertEquals(3, monthCodeToQuarter(AUGUST));
92        assertEquals(3, monthCodeToQuarter(SEPTEMBER));
93        assertEquals(4, monthCodeToQuarter(OCTOBER));
94        assertEquals(4, monthCodeToQuarter(NOVEMBER));
95        assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97        try {
98          monthCodeToQuarter(-1);
99          fail(“Invalid Month Code should throw exception”);
100       } catch (IllegalArgumentException e) {
101       }
102     }
103
104     public void testMonthCodeToString() throws Exception {
105       assertEquals(“January”, monthCodeToString(JANUARY));
106       assertEquals(“February”, monthCodeToString(FEBRUARY));
107       assertEquals(“March”, monthCodeToString(MARCH));
108       assertEquals(“April”, monthCodeToString(APRIL));
109       assertEquals(“May”, monthCodeToString(MAY));
110       assertEquals(“June”, monthCodeToString(JUNE));
111       assertEquals(“July”, monthCodeToString(JULY));
```

```
112        assertEquals(“August”, monthCodeToString(AUGUST));
113        assertEquals(“September”, monthCodeToString(SEPTEMBER));
114        assertEquals(“October”, monthCodeToString(OCTOBER));
115        assertEquals(“November”, monthCodeToString(NOVEMBER));
116        assertEquals(“December”, monthCodeToString(DECEMBER));
117
118        assertEquals(“Jan”, monthCodeToString(JANUARY, true));
119        assertEquals(“Feb”, monthCodeToString(FEBRUARY, true));
120        assertEquals(“Mar”, monthCodeToString(MARCH, true));
121        assertEquals(“Apr”, monthCodeToString(APRIL, true));
122        assertEquals(“May”, monthCodeToString(MAY, true));
123        assertEquals(“Jun”, monthCodeToString(JUNE, true));
124        assertEquals(“Jul”, monthCodeToString(JULY, true));
125        assertEquals(“Aug”, monthCodeToString(AUGUST, true));
126        assertEquals(“Sep”, monthCodeToString(SEPTEMBER, true));
127        assertEquals(“Oct”, monthCodeToString(OCTOBER, true));
128        assertEquals(“Nov”, monthCodeToString(NOVEMBER, true));
129        assertEquals(“Dec”, monthCodeToString(DECEMBER, true));
130
131        try {
132          monthCodeToString(-1);
133          fail(“Invalid month code should throw exception”);
134        } catch (IllegalArgumentException e) {
135        }
136
137      }
138
139      public void testStringToMonthCode() throws Exception {
140        assertEquals(JANUARY,stringToMonthCode(“1”));
141        assertEquals(FEBRUARY,stringToMonthCode(“2”));
142        assertEquals(MARCH,stringToMonthCode(“3”));
143        assertEquals(APRIL,stringToMonthCode(“4”));
144        assertEquals(MAY,stringToMonthCode(“5”));
145        assertEquals(JUNE,stringToMonthCode(“6”));
146        assertEquals(JULY,stringToMonthCode(“7”));
147        assertEquals(AUGUST,stringToMonthCode(“8”));
148        assertEquals(SEPTEMBER,stringToMonthCode(“9”));
149        assertEquals(OCTOBER,stringToMonthCode(“10”));
150        assertEquals(NOVEMBER, stringToMonthCode(“11”));
151        assertEquals(DECEMBER,stringToMonthCode(“12”));
152
153 //todo assertEquals(-1, stringToMonthCode(“0”));
154 // assertEquals(-1, stringToMonthCode(“13”));
155
156        assertEquals(-1,stringToMonthCode(“Hello”));
```

```

      for (int m = 1; m <= 12; m++) {
        assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
        assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
      }

// assertEquals(1,stringToMonthCode("jan"));
// assertEquals(2,stringToMonthCode("feb"));
// assertEquals(3,stringToMonthCode("mar"));
// assertEquals(4,stringToMonthCode("apr"));
// assertEquals(5,stringToMonthCode("may"));
// assertEquals(6,stringToMonthCode("jun"));
// assertEquals(7,stringToMonthCode("jul"));
// assertEquals(8,stringToMonthCode("aug"));
// assertEquals(9,stringToMonthCode("sep"));
// assertEquals(10,stringToMonthCode("oct"));
// assertEquals(11,stringToMonthCode("nov"));
// assertEquals(12,stringToMonthCode("dec"));

// assertEquals(1,stringToMonthCode("JAN"));
// assertEquals(2,stringToMonthCode("FEB"));
// assertEquals(3,stringToMonthCode("MAR"));
// assertEquals(4,stringToMonthCode("APR"));
// assertEquals(5,stringToMonthCode("MAY"));
// assertEquals(6,stringToMonthCode("JUN"));
// assertEquals(7,stringToMonthCode("JUL"));
// assertEquals(8,stringToMonthCode("AUG"));
// assertEquals(9,stringToMonthCode("SEP"));
// assertEquals(10,stringToMonthCode("OCT"));
// assertEquals(11,stringToMonthCode("NOV"));
// assertEquals(12,stringToMonthCode("DEC"));

// assertEquals(1,stringToMonthCode("january"));
// assertEquals(2,stringToMonthCode("february"));
// assertEquals(3,stringToMonthCode("march"));
// assertEquals(4,stringToMonthCode("april"));
// assertEquals(5,stringToMonthCode("may"));
// assertEquals(6,stringToMonthCode("june"));
// assertEquals(7,stringToMonthCode("july"));
// assertEquals(8,stringToMonthCode("august"));
// assertEquals(9,stringToMonthCode("september"));
// assertEquals(10,stringToMonthCode("october"));
// assertEquals(11,stringToMonthCode("november"));
// assertEquals(12,stringToMonthCode("december"));

```

```
202 // assertEquals(1,stringToMonthCode("JANUARY"));
203 // assertEquals(2,stringToMonthCode("FEBRUARY"));
204 // assertEquals(3,stringToMonthCode("MAR"));
205 // assertEquals(4,stringToMonthCode("APRIL"));
206 // assertEquals(5,stringToMonthCode("MAY"));
207 // assertEquals(6,stringToMonthCode("JUNE"));
208 // assertEquals(7,stringToMonthCode("JULY"));
209 // assertEquals(8,stringToMonthCode("AUGUST"));
210 // assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 // assertEquals(10,stringToMonthCode("OCTOBER"));
212 // assertEquals(11,stringToMonthCode("NOVEMBER"));
213 // assertEquals(12,stringToMonthCode("DECEMBER"));
214     }
215
216     public void testIsValidWeekInMonthCode() throws Exception {
217       for (int w = 0; w <= 4; w++) {
218         assertTrue(isValidWeekInMonthCode(w));
219       }
220       assertFalse(isValidWeekInMonthCode(5));
221     }
222
223     public void testIsLeapYear() throws Exception {
224       assertFalse(isLeapYear(1900));
225       assertFalse(isLeapYear(1901));
226       assertFalse(isLeapYear(1902));
227       assertFalse(isLeapYear(1903));
228       assertTrue(isLeapYear(1904));
229       assertTrue(isLeapYear(1908));
230       assertFalse(isLeapYear(1955));
231       assertTrue(isLeapYear(1964));
232       assertTrue(isLeapYear(1980));
233       assertTrue(isLeapYear(2000));
234       assertFalse(isLeapYear(2001));
235       assertFalse(isLeapYear(2100));
236     }
237
238     public void testLeapYearCount() throws Exception {
239       assertEquals(0, leapYearCount(1900));
240       assertEquals(0, leapYearCount(1901));
241       assertEquals(0, leapYearCount(1902));
242       assertEquals(0, leapYearCount(1903));
243       assertEquals(1, leapYearCount(1904));
244       assertEquals(1, leapYearCount(1905));
245       assertEquals(1, leapYearCount(1906));
246       assertEquals(1, leapYearCount(1907));
```

```
247       assertEquals(2, leapYearCount(1908));
248       assertEquals(24, leapYearCount(1999));
249       assertEquals(25, leapYearCount(2001));
250       assertEquals(49, leapYearCount(2101));
251       assertEquals(73, leapYearCount(2201));
252       assertEquals(97, leapYearCount(2301));
253       assertEquals(122, leapYearCount(2401));
254     }
255
256     public void testLastDayOfMonth() throws Exception {
257       assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258       assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259       assertEquals(31, lastDayOfMonth(MARCH, 1901));
260       assertEquals(30, lastDayOfMonth(APRIL, 1901));
261       assertEquals(31, lastDayOfMonth(MAY, 1901));
262       assertEquals(30, lastDayOfMonth(JUNE, 1901));
263       assertEquals(31, lastDayOfMonth(JULY, 1901));
264       assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265       assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266       assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267       assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268       assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269       assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270     }
271
272     public void testAddDays() throws Exception {
273       SerialDate newYears = d(1, JANUARY, 1900);
274       assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275       assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276       assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277       assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278     }
279
280     private static SpreadsheetDate d(int day, int month, int year)
            { return new SpreadsheetDate(day, month, year); }
281
282     public void testAddMonths() throws Exception {
283       assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY,
    1900)));
284       assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY,
    1900)));
285       assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY,
    1900)));
286       assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY,
    1900)));
287       assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY,
```

```
1900)));
      assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));

      assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
      assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));

      assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));

    }

    public void testAddYears() throws Exception {
      assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
      assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
      assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
      assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
    }

    public void testGetPreviousDayOfWeek() throws Exception {
      assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
      assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
      assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
      assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));

      try {
        getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
        fail("Invalid day of week code should throw exception");
      } catch (IllegalArgumentException e) {
      }
    }

    public void testGetFollowingDayOfWeek() throws Exception {
// assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
      assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
      assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));

```

```
322        try {
323          getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324          fail("Invalid day of week code should throw exception");
325        } catch (IllegalArgumentException e) {
326        }
327      }
328
329      public void testGetNearestDayOfWeek() throws Exception {
330        assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16,
    APRIL, 2006)));
331        assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17,
    APRIL, 2006)));
332        assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18,
    APRIL, 2006)));
333        assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19,
    APRIL, 2006)));
334        assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20,
    APRIL, 2006)));
335        assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21,
    APRIL, 2006)));
336        assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22,
    APRIL, 2006)));
337
338 //todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16,
    APRIL, 2006)));
339        assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17,
    APRIL, 2006)));
340        assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18,
    APRIL, 2006)));
341        assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19,
    APRIL, 2006)));
342        assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20,
    APRIL, 2006)));
343        assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21,
    APRIL, 2006)));
344        assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22,
    APRIL, 2006)));
345
346 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16,
    APRIL, 2006)));
347 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17,
    APRIL, 2006)));
348        assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18,
    APRIL, 2006)));
349        assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19,
    APRIL, 2006)));
350        assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20,
    APRIL, 2006)));
351        assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21,
```

```
APRIL, 2006)));
        assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));

// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
// assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
        assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
        assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
        assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
        assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));

// assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
// assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
        assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
        assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
        assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));

// assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
// assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
// assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
        assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));

// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16,
```

```
APRIL, 2006)));
// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
// assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
// assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
      assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));

      try {
        getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
        fail("Invalid day of week code should throw exception");
      } catch (IllegalArgumentException e) {
      }
    }

    public void testEndOfCurrentMonth() throws Exception {
      SerialDate d = SerialDate.createInstance(2);
      assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
      assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
      assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
      assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
      assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
      assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
      assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
      assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
      assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
      assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
      assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
      assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
      assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
```

```
408     }
409
410     public void testWeekInMonthToString() throws Exception {
411       assertEquals(“First”,weekInMonthToString(FIRST_WEEK_IN_MONTH));
412       assertEquals(“Second”,weekInMonthToString(SECOND_WEEK_IN_MONTH));
413       assertEquals(“Third”,weekInMonthToString(THIRD_WEEK_IN_MONTH));
414       assertEquals(“Fourth”,weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415       assertEquals(“Last”,weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo try {
418 // weekInMonthToString(-1);
419 // fail(“Invalid week code should throw exception”);
420 // } catch (IllegalArgumentException e) {
421 // }
422     }
423
424     public void testRelativeToString() throws Exception {
425       assertEquals(“Preceding”,relativeToString(PRECEDING));
426       assertEquals(“Nearest”,relativeToString(NEAREST));
427       assertEquals(“Following”,relativeToString(FOLLOWING));
428
429 //todo try {
430 // relativeToString(-1000);
431 // fail(“Invalid relative code should throw exception”);
432 // } catch (IllegalArgumentException e) {
433 // }
434     }
435
436     public void testCreateInstanceFromDDMMYYYY() throws Exception {
437       SerialDate date = createInstance(1, JANUARY, 1900);
438       assertEquals(1,date.getDayOfMonth());
439       assertEquals(JANUARY,date.getMonth());
440       assertEquals(1900,date.getYYYY());
441       assertEquals(2,date.toSerial());
442     }
443
444     public void testCreateInstanceFromSerial() throws Exception {
445       assertEquals(d(1, JANUARY, 1900),createInstance(2));
446       assertEquals(d(1, JANUARY, 1901), createInstance(367));
447     }
448
449     public void testCreateInstanceFromJavaDate() throws Exception {
450       assertEquals(d(1, JANUARY, 1900),
                 createInstance(new GregorianCalendar(1900,0,1).getTime()));
451       assertEquals(d(1, JANUARY, 2006),
```

```
                    createInstance(new GregorianCalendar(2006,0,1).getTime()));
    }

    public static void main(String[] args) {
      junit.textui.TestRunner.run(BobsSerialDateTest.class);
    }
}
```

Listado B-5
SpreadsheetDate.java.

```
1  /* ===============================================================
2  * JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
3  * ===============================================================
4  *
5  * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
6  *
7  * Información del proyecto: http://www.jfree.org/jcommon/index.html
8  *
9  * Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
10 * bajo las condiciones de la Licencia pública general GNU publicada por
11 * la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
12 * otra versión posterior (de su elección).
13 *
14 * Esta biblioteca se distribuye con la intención de que sea útil, pero
15 * SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
16 * e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general
   GNU
17 * si necesita más información al respecto.
18 *
19 * Debería haber recibido una copia de la Licencia pública general GNU
20 * junto a esta biblioteca; en caso contrario, contacte con la Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * EE.UU.
23 *
24 * [Java es una marca comercial o marca comercial registrada de Sun
25 * Microsystems, Inc. en Estados Unidos y otros países.]
26 *
27 * --------------------------
28 * SpreadsheetDate.java
29 * --------------------------
30 * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
31 *
32 * Autor original: David Gilbert (por Object Refinery Limited);
33 * Colaboradores(s): -;
34 *
35 * $Id: SpeadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36 *
37 * Cambios
38 * ----------
39 * 11-Oct-2001 : Version 1 (DG);
```

```
40  * 05-Nov-2001 : Se añaden los métodos getDescription() y setDescription()
    (DG);
41  * 12-Nov-2001 : Se cambia el nombre ExcelDate.java por SpreadsheetDate.java
    (DG);
42  * Se corrige un error a la hora de calcular el día, mes y año a
43  * partir del número de serie (DG);
44  * 24-Jan-2002 : Se corrige un error a la hora de calcular el número de
    serie a
45  * partir del día, mes y año. Gracias a Trevor Kills por el informe (DG);
46  * 29-May-2002 : Se añade el método equals(Object) (SourceForge ID 558850)
    (DG);
47  * 03-Oct-2002 : Se corrigen los errores detectados por Checkstyle (DG);
48  * 13-Mar-2003 : Implementación de Serializable (DG);
49  * 04-Sep-2003 : Métodos isInRange() completados (DG);
50  * 05-Sep-2003 : Implementación de Comparable (DG);
51  * 21-Oct-2003 : Se añade el método hashCode() (DG);
52  *
53  /*
54
55  package org.jfree.date;
56
57  import java.util.Calendar;
58  import java.util.Date;
59  /**
60
61  * Representa una con un entero, de forma similar a la
62  * implementación en Microsoft Excel. El intervalo de fechas admitido es
63  * 1-Ene-1900 a 31-Dic-9999.
64  * <P>
65  * Recuerde que Excel tiene un error que reconoce el año
66  * 1900 como bisiesto cuando en realidad no lo es. Encontrará más
67  * información en el sitio Web de Microsoft, en el artículo Q181370:
68  * <P>
69  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70  * <P>
71  * Excel usa como convención que el 1-Ene-1900 = 1. Esta clase usa la
72  * convención de que 1-Ene-1900 = 2.
73  * Como resultado, el número de día de esta clase será diferente al de
74  * Excel para enero y febrero de 1900… pero Excel añade un día
75  * más (29-Feb-1900 que en realidad no existe) y a partir de ahí
76  * los números de los días coinciden.
77  *
78  * @author David Gilbert
79  */
80  public class SpreadsheetDate extends SerialDate {
81
```

```
82      /** Para serialización. */
83      private static final long serialVersionUID = -2039586705374454461L;
84
85      /**
86      * El número de día (1-Ene-1900 = 2, 2-Ene-1900 = 3…, 31-Dic-9999 =
87      * 2958465).
88      */
89      private int serial;
90
91      /** El día del mes (de 1 a 28, 29, 30 o 31 en función del mes). */
92      private int day;
93
94      /** El mes del año (de 1 a 12). */
95      private int month;
96
97      /** El año (de 1900 a 9999). */
98      private int year;
99
100     /** Una descripción opcional para la fecha. */
101     private String description;
102
103     /**
104     * Crear una nueva instancia de la fecha.
105     *
106     * @param day el día (entre 1 y 28/29/30/31).
107     * @param month el mes (entre 1 y 12).
108     * @param year el año (entre 1900 y 9999).
109     */
110     public SpreadsheetDate(final int day, final int month, final int year)
    {
111
112       if ((year >= 1900) && (year <= 9999)) {
113         this.year = year;
114       }
115       else {
116         throw new IllegalArgumentException(
117           “The ‘year’ argument must be in range 1900 to 9999.”
118         );
119       }
120
121       if ((month >= MonthConstants.JANUARY)
122            && (month <= MonthConstants.DECEMBER)) {
123         this.month = month;
124       }
125       else {
```

```
126         throw new IllegalArgumentException(
127           “The ‘month’ argument must be in the range 1 to 12.”
128         );
129       }
130
131       if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132         this.day = day;
133       }
134       else {
135         throw new IllegalArgumentException(“Invalid ‘day’ argument.”);
136       }
137
138       // es necesario sincronizar el número de serie con el día-mes-año…
139       this.serial = calcSerial(day, month, year);
140
141       this.description = null;
142
143     }
144
145     /**
146     * Constructor estándar: crear un nuevo objeto de fecha que representa
    el
147     * número de día especificado (que debe estar comprendido entre 2 y
    2958465).
148     *
149     * @param serial número de serie para el día (entre 2 y 2958465).
150     */
151     public SpreadsheetDate(final int serial) {
152
153       if ((serial >= SERIAL_LOWER_BOUND) && (serial <=
    SERIAL_UPPER_BOUND)) {
154         this.serial = serial;
155       }
156       else {
157         throw new IllegalArgumentException(
158           “SpreadsheetDate: Serial must be in range 2 to 2958465.”);
159       }
160
161       // el día-mes-año debe estar sincronizado con el número de serie…
162       calcDayMonthYear();
163
164     }
165
166     /**
167     * Devuelve la descripción adjuntada a la fecha. No es
168     * obligatorio que la fecha tenga una descripción, pero resulta útil
```

```
169      * en algunas aplicaciones.
170      *
171      * @return La descripción adjuntada a la fecha.
172      */
173      public String getDescription() {
174        return this.description;
175      }
176
177      /**
178      * Establece la descripción de la fecha.
179      *
180      * @param description la descripción de esta fecha (<code>null</code>
181      * se permite)
182      */
183      public void setDescription(final String description) {
184        this.description = description;
185      }
186
187      /**
188      * Devuelve el número de serie de la fecha, siendo el 1 de enero 1900 =
    2
189      * (se corresponde, casi totalmente, al sistema de numeración empleado
    en
190      * Microsoft Excel para Windows y Lotus 1-2-3).
191      *
192      * @return El número de serie de la fecha.
193      */
194      public int toSerial() {
195        return this.serial;
196      }
197
198      /**
199      * Devuelve una <code>java.util.Date</code> equivalente a esta fecha.
200      *
201      * @return La fecha.
202      */
203      public Date toDate() {
204        final Calendar calendar = Calendar.getInstance();
205        calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206        return calendar.getTime();
207      }
208
209      /**
210      * Devuelve el año (con un intervalo válido de 1900 a 9999).
211      *
212      * @return El año.
```

```
213      */
214      public int getYYYY() {
215        return this.year;
216      }
217
218      /**
219      * Devuelve el mes (Enero = 1, Febrero = 2, Marzo = 3).
220      *
221      * @return El mes del año.
222      */
223      public int getMonth() {
224        return this.month;
225      }
226
227      /**
228      * Devuelve el día del mes.
229      *
230      * @return El día del mes.
231      */
232      public int getDayOfMonth() {
233        return this.day;
234      }
235
236      /**
237      * Devuelve un código que representa el día de la semana.
238      * <P>
239      * Los códigos se definen en la clase {@link SerialDate} como:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>,
   y
242      * <code>SATURDAY</code>.
243      *
244      * @return Un código que representa el día de la semana.
245      */
246      public int getDayOfWeek() {
247        return (this.serial + 6) % 7 + 1;
248      }
249
250      /**
251      * Prueba la igualdad de esta fecha con un objeto arbitrario.
252      * <P>
253      * Este método SÓLO devuelve true si el objeto es una instancia de la
254      * clase base {@link SerialDate} y representa el mismo día que
255      * {@link SpreadsheetDate}.
256      *
```

```
257     * @param object el objeto que comparar (se permite <code>null</code>).
258     *
259     * @return Un valor booleano.
260     */
261     public boolean equals(final Object object) {
262
263       if (object instanceof SerialDate) {
264         final SerialDate s = (SerialDate) object;
265         return (s.toSerial() == this.toSerial());
266       }
267       else {
268         return false;
269       }
270
271     }
272
273     /**
274     * Devuelve un código hash para la instancia de este objeto.
275     *
276     * @return Un código hash.
277     */
278     public int hashCode() {
279       return toSerial();
280     }
281
282     /**
283     * Devuelve la diferencia (en días) entre esta fecha y la
284     * ‘otra’ fecha especificada.
285     *
286     * @param other la fecha con la que se compara.
287     *
288     * @return La diferencia (en días) entre esta fecha y la
289     * otra’ fecha especificada.
290     */
291     public int compare(final SerialDate other) {
292       return this.serial – other.toSerial();
293     }
294
295     /**
296     * Implementa el método necesario para la interfaz Comparable.
297     *
298     * @param other el otro objeto (normalmente otro SerialDate).
299     *
300     * @return Un entero negativo, cero o un entero positivo si este objeto
301     * es menor que, igual o mayor que el objeto especificado.
```

```
302      */
303      public int compareTo(final Object other) {
304        return compare((SerialDate) other);
305      }
306
307      /**
308      * Devuelve true si esta SerialDate representa la misma fecha que la
309      * SerialDate especificada.
310      *
311      * @param other la fecha con la que se compara.
312      *
313      * @return <code>true</code> si esta SerialDate representa la misma
    fecha que
314      * la otra SerialDate especificada.
315      */
316      public boolean isOn(final SerialDate other) {
317        return (this.serial == other.toSerial());
318      }
319
320      /**
321      * Devuelve true si esta SerialDate representa una fecha anterior a
322      * la SerialDate especificada.
323      *
324      * @param other la fecha con la que se compara.
325      *
326      * @return <code>true</code> si esta SerialDate representa una fecha
    anterior a
327      * la SerialDate especificada.
328      */
329      public boolean isBefore(final SerialDate other) {
330        return (this.serial < other.toSerial());
331      }
332
333      /**
334      * Devuelve true si esta SerialDate representa la misma fecha que la
335      * SerialDate especificada.
336      *
337      * @param other la fecha con la que se compara.
338      *
339      * @return <code>true</code> si esta SerialDate representa la misma
    fecha
340      * que la SerialDate especificada.
341      */
342      public boolean isOnOrBefore(final SerialDate other) {
343        return (this.serial <= other.toSerial());
344      }
```

```
345
346     /**
347     * Devuelve true si esta SerialDate representa la misma fecha que la
348     * SerialDate especificada.
349     *
350     * @param other la fecha con la que se compara.
351     *
352     * @return <code>true</code> si esta SerialDate representa la misma
    fecha
353     * que la SerialDate especificada.
354     */
355     public boolean isAfter(final SerialDate other) {
356       return (this.serial > other.toSerial());
357     }
358
359     /**
360     * Devuelve true si esta SerialDate representa la misma fecha que la
361     * SerialDate especificada.
362     *
363     * @param other la fecha con la que se compara.
364     *
365     * @return <code>true</code> si esta SerialDate representa la misma
    fecha
366     * que la SerialDate especificada.
367     */
368     public boolean isOnOrAfter(final SerialDate other) {
369       return (this.serial >= other.toSerial());
370     }
371
372     /**
373     * Devuelve <code>true</code> si {@link SerialDate} se encuentra en el
374     * intervalo especificado (INCLUSIVE). El orden de fecha de d1 y d2 no
    es
375     * importante.
376     *
377     * @param d1 una fecha límite para el rango.
378     * @param d2 la otra fecha límite para el rango.
379     *
380     * @return Un valor booleano.
381     */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383       return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387     * Devuelve true si esta SerialDate se encuentra en el intervalo
```

```
especificado
     * (el invocador especifica si los puntos finales se incluyen o no). El orden
     * de d1 y d2 no es importante.
     *
     * @param d1 una fecha límite para el rango.
     * @param d2 la otra fecha límite para el rango.
     * @param include un código que controla si la fecha inicial y final
     * se incluyen en el intervalo.
     *
     * @return <code>true</code> si esta SerialDate se encuentra en el intervalo
     * especificado.
     */
     public boolean isInRange(final SerialDate d1, final SerialDate d2,
       final int include) {
       final int s1 = d1.toSerial();
       final int s2 = d2.toSerial();
       final int start = Math.min(s1, s2);
       final int end = Math.max(s1, s2);

       final int s = toSerial();
       if (include == SerialDate.INCLUDE_BOTH) {
         return (s >= start && s <= end);
       }
       else if (include == SerialDate.INCLUDE_FIRST) {
         return (s >= start && s < end);
       }
       else if (include == SerialDate.INCLUDE_SECOND) {
         return (s > start && s <= end);
       }
       else {
         return (s > start && s < end);
       }
     }

     /**
     * Calcular el número de serie a partir del día, mes y año.
     * <P>
     * 1-Ene-1900 = 2.
     *
     * @param d el día.
     * @param m el mes.
     * @param y el año.
     *
     * @return el número de serie a partir del día, mes y año.
```

```
431     */
432     private int calcSerial(final int d, final int m, final int y) {
433       final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434       int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435       if (m > MonthConstants.FEBRUARY) {
436         if (SerialDate.isLeapYear(y)) {
437           mm = mm + 1;
438         }
439       }
440       final int dd = d;
441       return yy + mm + dd + 1;
442     }
443
444     /**
445     * Calcular el día, mes y año a partir del número de serie.
446     */
447     private void calcDayMonthYear() {
448
449       // obtener el año a partir del número de serie de la fecha
450       final int days = this.serial - SERIAL_LOWER_BOUND;
451       // sobrevalorado ya que ignoramos los días bisiestos
452       final int overestimatedYYYY = 1900 + (days / 365);
453       final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454       final int nonleapdays = days - leaps;
455       // subestimado ya que sobrevaloramos los años
456       int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458       if (underestimatedYYYY == overestimatedYYYY) {
459         this.year = underestimatedYYYY;
460       }
461       else {
462         int ss1 = calcSerial(1, 1, underestimatedYYYY);
463         while (ss1 <= this.serial) {
464           underestimatedYYYY = underestimatedYYYY + 1;
465           ss1 = calcSerial(1, 1, underestimatedYYYY);
466         }
467         this.year = underestimatedYYYY - 1;
468       }
469
470       final int ss2 = calcSerial(1, 1, this.year);
471
472       int[] daysToEndOfPrecedingMonth
473         = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475       if (isLeapYear(this.year)) {
```

```
476         daysToEndOfPrecedingMonth
477           = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478       }
479
480       // get the month from the serial date
481       int mm = 1;
482       int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483       while (sss < this.serial) {
484         mm = mm + 1;
485         sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486       }
487       this.month = mm - 1;
488
489       // el resto es d(+1);
490       this.day = this.serial - ss2
491              - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

Listado B-6
RelativeDayOfWeekRule.java

```
1   /* ==============================================================
2   * JCommon : biblioteca gratuita de clases de propósito general para
    Java(tm)
3   * ==============================================================
4   *
5   * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
6   *
7   * Información del proyecto: http://www.jfree.org/jcommon/index.html
8   *
9   * Esta biblioteca es software gratuito; puede distribuirla y/o modificarla
10  * bajo las condiciones de la Licencia pública general GNU publicada por
11  * la Free Software Foundation; ya sea la versión 2.1 de la licencia, u
12  * otra versión posterior (de su elección).
13  *
14  * Esta biblioteca se distribuye con la intención de que sea útil, pero
15  * SIN GARANTÍA ALGUNA, incluida la garantía implícita de COMERCIABILIDAD
16  * e IDONEIDAD PARA UN DETERMINADO FIN. Consulte la Licencia pública general
    GNU
17  * si necesita más información al respecto.
18  *
```

```
19 * Debería haber recibido una copia de la Licencia pública general GNU
20 * junto a esta biblioteca; en caso contrario, contacte con la Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * EE.UU.
23 *
24 * [Java es una marca comercial o marca comercial registrada de Sun
25 * Microsystems, Inc. en Estados Unidos y otros países.]
26 *
27 * --------------------------
28 * RelativeDayOfWeekRule.java
29 * --------------------------
30 * (C) Copyright 2000-2003, de Object Refinery Limited y colaboradores.
31 *
32 * Autor original: David Gilbert (por Object Refinery Limited);
33 * Colaboradores(s): -;
34 *
35 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Cambios (26-Oct-2001)
38 * --------------------------
39 * 26-Oct-2001 : Se cambió el paquete por com.jrefinery.date.*;
40 * 03-Oct-2002 : Se corrigen los errores detectados por Checkstyle (DG);
41 *
42 */
43
44 package org.jfree.date;
45
46 /**
47 * Una regla de fechas anuales que devuelve una fecha por cada año en
   función de
48 * (a) una regla de referencia; (b) un día de la semana y (c) un parámetro
   de
49 * selección (SerialDate.PRECEDING, SerialDate.NEAREST,
   SerialDate.FOLLOWING).
50 * <P>
51 * Por ejemplo, el Viernes Santo se puede especificado ‘el Viernes ANTERIOR
   al
52 * domingo de Resurrección.
53 *
54 * @author David Gilbert
55 */
56 public class RelativeDayOfWeekRule extends AnnualDateRule {
57
58      /** Una referencia a la regla de fechas anuales sobre la que se basa
   esta regla. */
59      private AnnualDateRule subrule;
```

```
60
61      /**
62      * El día de la semana (SerialDate.MONDAY, SerialDate.TUESDAY, etc).
63      */
64      private int dayOfWeek;
65
66      /** Indica que día de la semana (PRECEDING, NEAREST o FOLLOWING). */
67      private int relative;
68
69      /**
70      * Constructor predeterminado: genera una regla para el lunes siguiente
    al 1 de enero.
71      */
72      public RelativeDayOfWeekRule() {
73        this(new DayAndMonthRule(), SerialDate.MONDAY,
    SerialDate.FOLLOWING);
74      }
75
76      /**
77      * Constructor estándar: genera una regla en función de la subregla
    proporcionada.
78      *
79      * @param subrule la regla que determina la fecha de referencia.
80      * @param dayOfWeek el día de la semana relativo a la fecha de
    referencia.
81      * @param relative indica “qué” día de la semana (anterior, más próximo
82      * o posterior).
83      */
84      public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85        final int dayOfWeek, final int relative) {
86        this.subrule = subrule;
87        this.dayOfWeek = dayOfWeek;
88        this.relative = relative;
89      }
90
91      /**
92      * Devuelve la subregla (también denominada regla de referencia).
93      *
94      * @return La regla de fechas anuales que determina la fecha de
    referencia para
95      * esta regla.
96      */
97      public AnnualDateRule getSubrule() {
98        return this.subrule;
99      }
100
```

```
101     /**
102     * Establece la subregla.
103     *
104     * @param subrule la regla de fechas anuales que determina la fecha de
105     * referencia para esta regla.
106     */
107     public void setSubrule(final AnnualDateRule subrule) {
108       this.subrule = subrule;
109     }
110
111     /**
112     * Devuelve el día de la semana de esta regla.
113     *
114     * @return el día de la semana de esta regla.
115     */
116     public int getDayOfWeek() {
117       return this.dayOfWeek;
118     }
119
120     /**
121     * Establece el día de la semana de esta regla.
122     *
123     * @param dayOfWeek el día de la semana de (SerialDate.MONDAY,
124     * SerialDate.TUESDAY, etc.).
125     */
126     public void setDayOfWeek(final int dayOfWeek) {
127       this.dayOfWeek = dayOfWeek;
128     }
129
130     /**
131     * Devuelve el atributo ‘relativo’ que determina “qué”
132     * día de la semana nos interesa (SerialDate.PRECEDING,
133     * SerialDate.NEAREST o SerialDate.FOLLOWING).
134     *
135     * @return El atributo ‘relativo’.
136     */
137     public int getRelative() {
138       return this.relative;
139     }
140
141     /**
142     * Establece el atributo ‘relativo’ (SerialDate.PRECEDING,
    SerialDate.NEAREST,
143     * SerialDate.FOLLOWING).
144     *
```

```
     * @param relative determina “qué” día de la semana se selecciona con esta
     * regla.
     */
     public void setRelative(final int relative) {
       this.relative = relative;
     }

     /**
     * Crea un clon de esta regla.
     *
     * @return un clon de esta regla.
     *
     * @throws CloneNotSupportedException nunca debe producirse.
     */
     public Object clone() throws CloneNotSupportedException {
       final RelativeDayOfWeekRule duplicate
         = (RelativeDayOfWeekRule) super.clone();
       duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
       return duplicate;
     }

     /**
     * Devuelve la fecha generada por esta regla, para el año especificado.
     *
     * @param year el año (1900 &lt;= year &lt;= 9999).
     *
     * @return La fecha generada por esta regla para un año concreto (posiblemente
     * <code>null</code>).
     */
     public SerialDate getDate(final int year) {

       // comprobar argumento…
       if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
         || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
         throw new IllegalArgumentException(
           “RelativeDayOfWeekRule.getDate(): year outside valid range.”);
       }

       // calcular la fecha…
       SerialDate result = null;
       final SerialDate base = this.subrule.getDate(year);

       if (base != null) {
         switch (this.relative) {
```

```
189          case(SerialDate.PRECEDING):
190            result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                base);
192            break;
193          case(SerialDate.NEAREST):
194            result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                base);
196            break;
197          case(SerialDate.FOLLOWING):
198            result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                base);
200            break;
201          default:
202            break;
203        }
204      }
205      return result;
206
207    }
208
209 }
```

Listado B-7
DayDate.java (Final)

```
1   /* ==============================================================
2   * JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
3   * ==============================================================
4   *
5   * (C) Copyright 2000-2005, de Object Refinery Limited y colaboradores.
…
36  */
37  package org.jfree.date;
38
39  import java.io.Serializable;
40  import java.util.*;
41
42  /**
43  * Una clase abstracta que representa fechas inmutables con una precisión de
44  * un día. La implementación asigna cada fecha a un entero que
45  * representa un número ordinal de días de un origen fijo.
46  *
47  * ¿Por qué no usar java.útil.Date? Lo haremos, cuando tenga sentido. En
    ocasiones,
```

```
48 * java.util.Date puede ser demasiado precisa; representa un instante en el
   tiempo,
49 * con una precisión de 1/1000 de segundo (y la fecha depende de la
50 * zona horaria). En ocasiones solo querremos representar un día concreto
   (como el 21
51 * de enero de 2015) sin preocuparnos de la hora del día, la
52 * zona horaria u otros aspectos. Para eso hemos definido SerialDate.
53 *
54 * Usar DayDateFactory.makeDate para crear una instancia.
55 *
56 * @author David Gilbert
57 * @author Robert C. Martin realizó gran parte de la refactorización.
58 */
59
60 public abstract class DayDate implements Comparable, Serializable {
61      public abstract int getOrdinalDay();
62      public abstract int getYear();
63      public abstract Month getMonth();
64      public abstract int getDayOfMonth();
65
66      protected abstract Day getDayOfWeekForOrdinalZero();
67
68      public DayDate plusDays(int days) {
69        return DayDateFactory.makeDate(getOrdinalDay() + days);
70      }
71
72      public DayDate plusMonths(int months) {
73        int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74        int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75        int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal +
   months;
76        int resultYear = resultMonthAndYearAsOrdinal / 12;
77        int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 +
   Month.JANUARY.toInt();
78        Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79        int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth,
   resultYear);
80        return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81      }
82
83      public DayDate plusYears(int years) {
84        int resultYear = getYear() + years;
85        int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(),
   resultYear);
86        return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87      }
```

```

    private int correctLastDayOfMonth(int day, Month month, int year) {
      int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
      if (day > lastDayOfMonth)
        day = lastDayOfMonth;
      return day;
    }

    public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
      int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
      if (offsetToTarget >= 0)
        offsetToTarget -= 7;
      return plusDays(offsetToTarget);
    }

    public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
      int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
      if (offsetToTarget <= 0)
        offsetToTarget += 7;
      return plusDays(offsetToTarget);
    }

    public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
      int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
      int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
      int offsetToPreviousTarget = offsetToFutureTarget - 7;

      if (offsetToFutureTarget > 3)
        return plusDays(offsetToPreviousTarget);
      else
        return plusDays(offsetToFutureTarget);
    }

    public DayDate getEndOfMonth() {
      Month month = getMonth();
      int year = getYear();
      int lastDay = DateUtil.lastDayOfMonth(month, year);
      return DayDateFactory.makeDate(lastDay, month, year);
    }

    public Date toDate() {
      final Calendar calendar = Calendar.getInstance();
      int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
```

```
      calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
      return calendar.getTime();
    }

    public String toString() {
      return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
    }

    public Day getDayOfWeek() {
      Day startingDay = getDayOfWeekForOrdinalZero();
      int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
      int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
      return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
    }

    public int daysSince(DayDate date) {
      return getOrdinalDay() - date.getOrdinalDay();
    }

    public boolean isOn(DayDate other) {
      return getOrdinalDay() == other.getOrdinalDay();
    }

    public boolean isBefore(DayDate other) {
      return getOrdinalDay() < other.getOrdinalDay();
    }

    public boolean isOnOrBefore(DayDate other) {
      return getOrdinalDay() <= other.getOrdinalDay();
    }

    public boolean isAfter(DayDate other) {
      return getOrdinalDay() > other.getOrdinalDay();
    }

    public boolean isOnOrAfter(DayDate other) {
      return getOrdinalDay() >= other.getOrdinalDay();
    }

    public boolean isInRange(DayDate d1, DayDate d2) {
      return isInRange(d1, d2, DateInterval.CLOSED);
    }

    public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
```

```
175        int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176        int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177        return interval.isIn(getOrdinalDay(), left, right);
178      }
179 }
```

Listado B-8
Month.java (Final)

```
1  package org.jfree.date;
2
3  import java.text.DateFormatSymbols;
4
5  public enum Month {
6       JANUARY(1), FEBRUARY(2), MARCH(3),
7       APRIL(4), MAY(5), JUNE(6),
8       JULY(7), AUGUST(8), SEPTEMBER(9),
9       OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10      private static DateFormatSymbols dateFormatSymbols = new
   DateFormatSymbols();
11      private static final int[] LAST_DAY_OF_MONTH =
12        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14      private int index;
15
16      Month(int index) {
17        this.index = index;
18      }
19
20      public static Month fromInt(int monthIndex) {
21        for (Month m : Month.values()) {
22          if (m.index == monthIndex)
23            return m;
24        }
25        throw new IllegalArgumentException("Invalid month index " +
   monthIndex);
26      }
27
28      public int lastDay() {
29        return LAST_DAY_OF_MONTH[index];
30      }
31
32      public int quarter() {
33        return 1 + (index - 1) / 3;
```

```
34      }
35
36      public String toString() {
37        return dateFormatSymbols.getMonths()[index - 1];
38      }
39
40      public String toShortString() {
41        return dateFormatSymbols.getShortMonths()[index - 1];
42      }
43
44      public static Month parse(String s) {
45        s = s.trim();
46        for (Month m : Month.values())
47          if (m.matches(s))
48            return m;
49
50        try {
51          return fromInt(Integer.parseInt(s));
52        }
53        catch (NumberFormatException e) {}
54        throw new IllegalArgumentException("Invalid month " + s);
55      }
56
57      private boolean matches(String s) {
58        return s.equalsIgnoreCase(toString()) ||
59        s.equalsIgnoreCase(toShortString());
60      }
61
62      public int toInt() {
63        return index;
64      }
65 }
```

Listado B-9
Day.java (Final)

```
1  package org.jfree.date;
2
3  import java.util.Calendar;
4  import java.text.DateFormatSymbols;
5
6  public enum Day {
7       MONDAY(Calendar.MONDAY),
8       TUESDAY(Calendar.TUESDAY),
```

```
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);

    private final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

    Day(int day) {
      index = day;
    }

    public static Day fromInt(int index) throws IllegalArgumentException {
      for (Day d : Day.values())
        if (d.index == index)
          return d;
      throw new IllegalArgumentException(
        String.format("Illegal day index: %d.", index));
    }

    public static Day parse(String s) throws IllegalArgumentException {
      String[] shortWeekdayNames =
        dateSymbols.getShortWeekdays();
      String[] weekDayNames =
        dateSymbols.getWeekdays();

      s = s.trim();
      for (Day day : Day.values()) {
        if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
            s.equalsIgnoreCase(weekDayNames[day.index])) {
          return day;
        }
      }
      throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
    }

    public String toString() {
      return dateSymbols.getWeekdays()[index];
    }

    public int toInt() {
      return index;
    }
```

```
}
```

Listado B-10
DateInterval.java (Final)

```
package org.jfree.date;

public enum DateInterval {
     OPEN {
       public boolean isIn(int d, int left, int right) {
         return d > left && d < right;
       }
     },
     CLOSED_LEFT {
       public boolean isIn(int d, int left, int right) {
         return d >= left && d < right;
       }
     },
     CLOSED_RIGHT {
       public boolean isIn(int d, int left, int right) {
         return d > left && d <= right;
       }
     },
     CLOSED {
       public boolean isIn(int d, int left, int right) {
         return d >= left && d <= right;
       }
     };

     public abstract boolean isIn(int d, int left, int right);
}
```

Listado B-11
WeekInMonth.java (Final)

```
package org.jfree.date;

public enum WeekInMonth {
     FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
     private final int index;

     WeekInMonth(int index) {
       this.index = index;
```

```
9        }
10
11       public int toInt() {
12         return index;
13       }
14  }
```

Listado B-12
WeekdayRange.java (Final)

```
1   package org.jfree.date;
2
3   public enum WeekdayRange {
4        LAST, NEAREST, NEXT
5   }
```

Listado B-13
DateUtil.java (Final)

```
1  package org.jfree.date;
2
3  import java.text.DateFormatSymbols;
4
5  public class DateUtil {
6       private static DateFormatSymbols dateFormatSymbols = new
   DateFormatSymbols();
7
8       public static String[] getMonthNames() {
9         return dateFormatSymbols.getMonths();
10      }
11
12      public static boolean isLeapYear(int year) {
13        boolean fourth = year % 4 == 0;
14        boolean hundredth = year % 100 == 0;
15        boolean fourHundredth = year % 400 == 0;
16        return fourth && (!hundredth || fourHundredth);
17      }
18
19      public static int lastDayOfMonth(Month month, int year) {
20        if (month == Month.FEBRUARY && isLeapYear(year))
21          return month.lastDay() + 1;
22        else
23          return month.lastDay();
```

```
24      }
25
26      public static int leapYearCount(int year) {
27        int leap4 = (year - 1896) / 4;
28        int leap100 = (year - 1800) / 100;
29        int leap400 = (year - 1600) / 400;
30        return leap4 - leap100 + leap400;
31      }
32 }
```

Listado B-14
DayDateFactory.java (Final)

```
1  package org.jfree.date;
2
3  public abstract class DayDateFactory {
4       private static DayDateFactory factory = new SpreadsheetDateFactory();
5       public static void setInstance(DayDateFactory factory) {
6         DayDateFactory.factory = factory;
7       }
8
9       protected abstract DayDate _makeDate(int ordinal);
10      protected abstract DayDate _makeDate(int day, Month month, int year);
11      protected abstract DayDate _makeDate(int day, int month, int year);
12      protected abstract DayDate _makeDate(java.util.Date date);
13      protected abstract int _getMinimumYear();
14      protected abstract int _getMaximumYear();
15
16      public static DayDate makeDate(int ordinal) {
17        return factory._makeDate(ordinal);
18      }
19
20      public static DayDate makeDate(int day, Month month, int year) {
21        return factory._makeDate(day, month, year);
22      }
23
24      public static DayDate makeDate(int day, int month, int year) {
25        return factory._makeDate(day, month, year);
26      }
27
28      public static DayDate makeDate(java.util.Date date) {
29        return factory._makeDate(date);
30      }
```

```

    public static int getMinimumYear() {
      return factory._getMinimumYear();
    }

    public static int getMaximumYear() {
      return factory._getMaximumYear();
    }
}
```

Listado B-15
SpreadsheetDateFactory.java (Final)

```
package org.jfree.date;

import java.util.*;

public class SpreadsheetDateFactory extends DayDateFactory {
     public DayDate _makeDate(int ordinal) {
       return new SpreadsheetDate(ordinal);
     }

     public DayDate _makeDate(int day, Month month, int year) {
       return new SpreadsheetDate(day, month, year);
     }

     public DayDate _makeDate(int day, int month, int year) {
       return new SpreadsheetDate(day, month, year);
     }

     public DayDate _makeDate(Date date) {
       final GregorianCalendar calendar = new GregorianCalendar();
       calendar.setTime(date);
       return new SpreadsheetDate(
         calendar.get(Calendar.DATE),
         Month.fromInt(calendar.get(Calendar.MONTH) + 1),
         calendar.get(Calendar.YEAR));
     }

     protected int _getMinimumYear() {
       return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
     }

     protected int _getMaximumYear() {
```

```
32        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33      }
34 }
```

Listado B-16
SpreadsheetDate.java (Final)

```
1   /* ===============================================================
2   * JCommon: biblioteca gratuita de clases de propósito general para Java(tm)
3   * ===============================================================
4   *
5   * (C) Copyright 2000-2005, de Object Refinery Limited y Colaboradores.
6   *
…
52  *
53  */
54
55  package org.jfree.date;
56
57  import static org.jfree.date.Month.FEBRUARY;
58
59  import java.util.*;
60
61  /**
62  * Representa una fecha con un entero, de forma similar a la
63  * implementación en Microsoft Excel. El intervalo de fechas admitido es
64  * del 1-Ene-1900 al 31-Dic-9999.
65  * <p/>
66  * Recuerde que Excel tiene un error que reconoce el año
67  * 1900 como bisiesto cuando en realidad no lo es. Encontrará más
68  * información en el sitio de Microsoft, en el artículo Q181370:
69  * <p/>
70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
71  * <p/>
72  * Excel usa como convención que el 1-Ene-1900 = 1. Esta clase usa la
73  * convención de que el 1-Ene-1900 = 2.
74  * Como resultado, el número de día de esta clase será diferente al de
75  * Excel para enero y febrero de 1900… pero Excel añade un día
76  * más (29-Feb-1900 que en realidad no existe) y a partir de ahí
77  * los números de los días coinciden.
78  *
79  * @author David Gilbert
80  */
81  public class SpreadsheetDate extends DayDate {
```

```
82      public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
83      public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
84      public static final int MINIMUM_YEAR_SUPPORTED = 1900;
85      public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
86      static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87        {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88      static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH
    =
89        {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91      private int ordinalDay;
92      private int day;
93      private Month month;
94      private int year;
95
96      public SpreadsheetDate(int day, Month month, int year) {
97        if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
98          throw new IllegalArgumentException(
99            “The ‘year’ argument must be in range ” +
100           MINIMUM_YEAR_SUPPORTED + “ to ” + MAXIMUM_YEAR_SUPPORTED + “.”);
101       if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102         throw new IllegalArgumentException(“Invalid ‘day’ argument.”);
103
104       this.year = year;
105       this.month = month;
106       this.day = day;
107       ordinalDay = calcOrdinal(day, month, year);
108     }
109
110     public SpreadsheetDate(int day, int month, int year) {
111       this(day, Month.fromInt(month), year);
112     }
113
114     public SpreadsheetDate(int serial) {
115       if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116         throw new IllegalArgumentException(
117           “SpreadsheetDate: Serial must be in range 2 to 2958465.”);
118
119       ordinalDay = serial;
120       calcDayMonthYear();
121     }
122
123     public int getOrdinalDay() {
124       return ordinalDay;
125     }
```

```

    public int getYear() {
      return year;
    }

    public Month getMonth() {
      return month;
    }

    public int getDayOfMonth() {
      return day;
    }

    protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}

    public boolean equals(Object object) {
      if (!(object instanceof DayDate))
        return false;

      DayDate date = (DayDate) object;
      return date.getOrdinalDay() == getOrdinalDay();
    }

    public int hashCode() {
      return getOrdinalDay();
    }

    public int compareTo(Object other) {
      return daysSince((DayDate) other);
    }

    private int calcOrdinal(int day, Month month, int year) {
      int leapDaysForYear = DateUtil.leapYearCount(year - 1);
      int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
      int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
      if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
        daysUpToMonth++;
      int daysInMonth = day - 1;
      return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
    }

    private void calcDayMonthYear() {
      int days = ordinalDay - EARLIEST_DATE_ORDINAL;
```

```
      int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
      int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
      int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;

      year = huntForYearContaining(ordinalDay, underestimatedYear);
      int firstOrdinalOfYear = firstOrdinalOfYear(year);
      month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
      day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
    }

    private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
      int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
      int aMonth = 1;
      while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
        aMonth++;

      return Month.fromInt(aMonth - 1);
    }

    private int daysBeforeThisMonth(int aMonth) {
      if (DateUtil.isLeapYear(year))
        return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
      else
        return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
    }

    private int huntForYearContaining(int anOrdinalDay, int startingYear) {
      int aYear = startingYear;
      while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
        aYear++;

      return aYear - 1;
    }

    private int firstOrdinalOfYear(int year) {
      return calcOrdinal(1, Month.JANUARY, year);
    }

    public static DayDate createInstance(Date date) {
      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(date);
      return new SpreadsheetDate(calendar.get(Calendar.DATE),
```

```
                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
                calendar.get(Calendar.YEAR));

      }
}
```

Epílogo

En 2005, mientras asistía a la conferencia Agile en Denver (EE.UU.), Elisabeth Hedrickson[125] me dio una pulsera verde parecida a la que Lance Armstrong popularizó hace unos años. En ésta se leía *Test Obsessed* (Obsesionado por las pruebas). Me la puse y la lucí con orgullo. Desde que aprendí el TDD de Kent Beck en 1999, sin duda el desarrollo controlado por pruebas me ha obsesionado.

Pero sucedió algo extraño. No me podía quitar la pulsera. No porque se hubiera quedado físicamente pegada, sino porque estaba moralmente pegada.

La pulsera resumía mi ética profesional.

Era un indicador visible de mi compromiso por crear el mejor código posible. Si me la hubiera quitado habría traicionado a esa ética y a ese compromiso.

Y todavía la llevo en la muñeca.

Cuando escribo código, la veo ahí. Es un recordatorio constante de la promesa que me hice de escribir código limpio.

Robert Cecil “Uncle Bob” Martin (Palo Alto California, Estados Unidos, 1952). Es un prestigioso desarrollador de *software* desde 1970 y consultor internacional desde 1990. Es fundador y presidente de Object Mentor, Inc., un equipo de experimentados consultores que ayudan a clientes de todo el mundo en diferentes campos de la programación como C++, Java, C#, Ruby, Programación orientada a objetos (POO), patrones de diseño, UML, metodologías ágiles y programación eXtreme.

Notas

[1] [Beck07]. <<

[2] Cuando Ignaz Semmelweis recomendó en 1847 que los médicos se lavaran las manos, su propuesta fue rechazada aludiendo que los doctores estaban demasiado ocupados para hacerlo entre paciente y paciente. <<

[3] http://www.pragmaticprogrammer.com/booksellers/2004-12.html. <<

[4] [Knuth92]. <<

[5] Es una adaptación del mensaje de despedida de Robert Stephenson Smyth Baden-Powell a los Scouts: «Intentad dejar este mundo un poco mejor de como os lo encontrasteis…». <<

[6] Como veremos más adelante, aunque un contenedor *sea* una `lista`, no conviene codificar el tipo de contenedor en el nombre. <<

[7] Imagine que se crea una variable con el nombre `klass` sólo porque el nombre `class` se ha usado en otro elemento. <<

[8] Uncle Bob solía hacerlo en C++ pero ha abandonado esta práctica ya que no es necesario en los IDE modernos. <<

[9] http://java.sun.com/products/javabeans/docs/spec.html. <<

[10] Una herramienta de pruebas de código abierto (www.fitnese.org). <<

[11] Una herramienta de código abierto para probar unidades para Java (www.junit.org). <<

[12] Le pregunté a Kent si todavía conservaba una copia, pero no la encontró. Busqué en mis viejos ordenadores, pero nada. Solamente se conserva el recuerdo de aquél programa. <<

[13] El lenguaje LOGO usaba la palabra clave `TO` al igual que Ruby y Python usaban `def`. Por tanto, todas las funciones comenzaban por `TO`, lo que tenía un efecto interesante en cómo se diseñaban. <<

[14] [KP78], p. 37. <<

[15] Y, por supuesto, se incluyen cadenas if/else. <<

[16] a. http://en.wikipedia.org/wiki/Single_responsibility_principle

b. http://www.objectmentor.com/resources/articles/srp.pdf <<

[17] a. http://en.wikipedia.org/wiki/Open/closed_principle

b. http://www.objectmentor.com/resources/articles/ocp.pdf <<

[18] [GOF]. <<

[19] Terminé la refactorización de un módulo que usaba la forma dinámica. Conseguí convertir el módulo `outputStream` en un campo de la clase y las invocaciones de `writeField` a formato monódico. El resultado fue mucho más limpio. <<

[20] Existen algunos que creen que pueden evitar volver a compilar e implementar, y nos hemos encargado de ellos. <<

[21] Ejemplo de principio abierto/cerrado (OCP) [PPP02]. <<

[22] El principio DRY. [PRAG]. <<

[23] [SP72]. <<

[24] [KP78], p. 144. <<

[25] La tendencia actual de los IDE de comprobar la ortografía de los comentarios será un bálsamo para los que tenemos que leer gran cantidad de código. <<

[26] El cuadro muestra sigma/2 por encima y debajo de la media. Asumo que la distribución de la longitud de archivos no es normal, por lo que la desviación estándar no es matemáticamente precisa. Pero aquí el objetivo no es la precisión, sino la sensación. <<

[27] Es lo contrario a lo que sucede en lenguajes como Pascal, C y C++ que obligan a definir, o al menos a declarar, las funciones antes de usarlas. <<

[28] ¿A quien voy a engañar? Sigo siendo programador de lenguajes de ensamblado. En este caso, el hábito sí hace al monje. <<

[29] Siempre existe una solución conocida por los diseñadores orientados a objetos con experiencia: VISITOR o entrega dual, por ejemplo. Pero son técnicas costosas y suelen devolver la estructura de un programa por procedimientos. <<

[30] http://es.wikipedia.org/wiki/Ley_de_Demeter. <<

[31] De la estructura Apache. <<

[32] En ocasiones se denomina Feature Envy (Envidia de las características), de [Refactoring]. <<

[33] [Martin]. <<

[34] [BeckTDD], pp. 136-137. <<

[35] Véase el patrón del adaptador en [GOF]. <<

[36] Más información al respecto en [WELC]. <<

[37] *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE Software, Mayo/Junio 2007 (Vol. 24, No. 3) pp. 32-36 http://doi.ieeecomputersociety.org/10.1109/MS.2007.85 <<

[38] http://fitnesse.org/FitNesse.AcceptanceTestPatterns. <<

[39] Véase el apartado sobre asignaciones mentales del capítulo 2. <<

[40] Véase la entrada de Dave Astel: http://www.artima.com/weblogs/viewpost.jsp?thread=35578 <<

[41] [RSpec]. <<

[42] [GOF]. <<

[43] ¡Cíñase al código! <<

[44] Materiales de formación de Object Mentor. <<

[45] [RDD]. <<

[46] Encontrará más Información sobre este principio en [PPP]. <<

[47] [Knuth92]. <<

[48] [PPP]. <<

[49] [PPP]. <<

[50] [Mezzaros07]. <<

[51] [GOF]. <<

[52] Véase, por ejemplo, [Fowler]. <<

[53] Véase [Spring], También existe una estructura Spring.NET. <<

[54] No olvide que la creación de instancias/evaluación tardía es sólo una optimización, puede que prematura. <<

[55] Sistema de administración de base de datos. <<

[56] Consulte [AOSD] si necesita información general sobre aspectos y [AspectJ] y [Colyer] para Información concreta de AspectJ. <<

[57] No se necesita la modificación manual del código fuente de destino. <<

[58] Véase [CGLIB], [ASM] y [Javassist]. <<

[59] Si necesita ejemplos más detallados de la API Proxy y ejemplos de uso, consulte [Goetz]. <<

[60] AOP se suele confundir con técnicas empleadas para implementarlo, como la intercepción y envoltorio de métodos a través de `proxies`. El verdadero valor de un sistema AOP es la capacidad para especificar comportamientos del sistema de forma concisa y modular. <<

[61] Véase [Spring] y [JBoss]. Java puro significa sin AspectJ. <<

[62] Adaptado de www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25. <<

[63] [GOF]. <<

[64] El ejemplo se puede simplificar mediante mecanismos que usen convenciones y anotaciones de Java 5 para reducir la cantidad necesaria de lógica de conexión explícita. <<

[65] Adaptado de http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html. <<

[66] Véase [AspectJ] y [Colyer]. <<

[67] No confundir con la práctica de diseño anticipado. BDUF es la práctica de diseñar todo por adelantado antes de implementar nada. <<

[68] Existe una cantidad significativa de exploración iterativa y detalles de análisis, incluso una vez iniciada la construcción. <<

[69] El término fue empleado por primera vez por [Kolence]. <<

[70] El trabajo de [Alexander] ha sido una gran influencia para la comunidad de *software*. <<

[71] Véase, por ejemplo, [DSL]. [JMock] es un buen ejemplo de API de Java que crea un DSL. <<

[72] [XPE]. <<

[73] [GOF]. <<

[74] Correspondencia privada. <<

[75] Rayos cósmicos, repeticiones, etc. <<

[76] Véase el apéndice A. <<

[77] Véase el apéndice A. <<

[78] [PPP]. <<

[79] Véase el apéndice A. <<

[80] [PRAG]. <<

[81] [Lea99]. <<

[82] http://en.wikipedia.org/wiki/Producer-consumer. <<

[83] http://en.wikipedia.org/wiki/Readers-writers_problem. <<

[84] http://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_filósofos. <<

[85] Véase el apéndice A. <<

[86] Una sección crítica es cualquier sección de código que debe protegerse de usos simultáneos por parte del programa para que sea correcta. <<

[87] Véase el apéndice A. <<

[88] Véase el apéndice A. <<

[89] ¿Sabía que el modelo de procesos de Java no garantiza el procesamiento preventivo? Los SO modernos sí lo hacen, de modo que lo obtiene de forma gratuita. No obstante, la MVJ no lo garantiza. <<

[90] No es estrictamente el caso. Como la MVJ no garantiza los procesos preventivos, un determinado algoritmo puede que siempre funcione en un SO que no prevea los procesos. Lo contrario también es posible, pero por distintos motivos. <<

[91]
https://www.ibm.com/developerworks/community/groups/service/html/comm
lang=es&communityUuid=18d10b14-e2c8-4780-bace-9af1fc463cc0. <<

[92] Hace poco modifiqué este módulo para Ruby. Tenía una séptima parte del tamaño original y una mejor estructura. <<

[93] Para evitar este tipo de sorpresas, añadí una nueva prueba de unidad que invocaba todas las pruebas de FitNesse. <<

[94] *JUnit Pocket Guide*, Kent Beck, O'Reilly. 2004. p. 43. <<

[95] Véase el capítulo 1. <<

[96] Una solución mejor sería que el Javadoc presentara todos los comentarios con formato previo, para que tengan el mismo aspecto en el código y en el documento. <<

[97] Algunos revisores de este texto no comparten esta decisión. Sostienen que en una estructura de código abierto es más recomendable ejercer control manual sobre el ID de serie para que los cambios mínimos del *software* no invaliden las fechas señalizadas antiguas. Me parece justo. Sin embargo, al menos el fallo, aunque sea inconveniente, tiene un motivo evidente. Por otra parte, si el autor de la clase se olvida de actualizar el ID, el modo de fallo será indefinido y puede que silencioso. Creo que la moraleja es que no debe esperar a deserializar entre versiones. <<

[98] [GOF]. <<

[99] Ibid. <<

[100] Ibid. <<

[101] [Simmons04], p. 73. <<

[102] [Refactoring]. <<

[103] [Beck97]. <<

[104] [Refactoring]. <<

[105] http://es.wikipedia.org/wiki/Principio_de_la_Mínima_Sorpresa <<

[106] [PRAG]. <<

[107] [GOF]. <<

[108] [GOF]. <<

[109] [Refactoring]. <<

[110] En concreto, el Principio de Responsabilidad Única, el Principio Abierto/Cerrado y el Principio de Cierre Común. Véase [PPP]. <<

[111] [Beck97], p. 108. <<

[112] [Beck07]. <<

[113] Es distinto saber cómo funciona el código y saber si el algoritmo se encargará de realizar la tarea para la que se necesita. Es habitual desconocer si un algoritmo es el adecuado. Desconocer lo que hace el código es indolencia. <<

[114] O mejor todavía, una clase Money que use enteros. <<

[115] [PRAG]. p. 138. <<

[116] Véase la cita de Ward Cunningham del capítulo 1. <<

[117] [DDD]. <<

[118] Puede comprobar personalmente el código antes y después, y revisar las versiones con y sin subprocesos, que veremos en un apartado posterior. <<

[119] Es una comparación simplificada, pero para los objetivos de este ejercicio es un modelo válido. <<

[120] De hecho, la interfaz `Iterator` es incompatible con subprocesos por naturaleza. No se diseñó para usar varios subprocesos, de modo que no debería sorprenderle. <<

[121] Por ejemplo, alguien añade un resultado de depuración y el problema desaparece. El código de depuración corrige el problema, pero permanece en el sistema. <<

[122] Siglas de *There ain't no such thing as a free lunch* (Todo tiene un precio). <<

[123] http://www.haifa.ibm.com/projects/verification/contest/index.html <<

[124] Véase [Lea99] p. 191. <<

[125] http://www.qualitytree.com/ <<

Eloquent JavaScript

Third Edition

A Modern Introduction to Programming

Marijn Haverbeke

no starch press

Eloquent JavaScript

3ra edición

Marijn Haverbeke

Copyright © 2018 por Marijn Haverbeke

Este libro está bajo la licencia Creative Commons Atribución-NoComercial 3.0 (http://creativecommons.org/licenses/by-nc/3.0/). El código también puede considerarse bajo una licencia MIT (http://opensource.org/licenses/MIT).

Las ilustraciones han sido contribuidas por varios artistas: Cover and chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with regexper.com by Jeff Avallone. Village photograph in Chapter 11 by Fabrice Creuzot. Game concept for Chapter 15 by Thomas Palef.

The third edition of Eloquent JavaScript was made possible by 325 financial backers.

You can buy a print version of this book, with an extra bonus chapter included, printed by No Starch Press at http://a-fwd.com/com=marijhaver-20&asin-com=1593279507.

Contents

Introducción 1
Acerca de la programación 2
Por qué el lenguaje importa 3
¿Qué es JavaScript? 6
Código, y qué hacer con él 8
Descripción general de este libro 9
Convenciones tipográficas 9

1 Valores, Tipos, y Operadores 11
Valores 11
Números 12
Strings 15
Operadores unarios 17
Valores Booleanos 17
Valores vacíos 20
Conversión de tipo automática 20
Resumen 22

2 Estructura de Programa 23
Expresiones y declaraciones 23
Vinculaciones 24
Nombres vinculantes 26
El entorno 27
Funciones 27
La función console.log 28
Valores de retorno 28
Flujo de control 29
Ejecución condicional 29
Ciclos while y do 31
Indentando Código 33
Ciclos for 34
Rompiendo un ciclo 35

Actualizando vinculaciones de manera sucinta 35
Despachar en un valor con switch 36
Capitalización 37
Comentarios 38
Resumen 38
Ejercicios 39

3 Funciones 41
Definiendo una función 41
Vinculaciones y alcances 42
Funciones como valores 45
Notación de declaración 45
Funciones de flecha 46
La pila de llamadas 47
Argumentos Opcionales 48
Cierre 50
Recursión 51
Funciones crecientes 54
Funciones y efectos secundarios 57
Resumen 58
Ejercicios 58

4 Estructuras de Datos: Objetos y Arrays 60
El Hombre Ardilla 60
Conjuntos de datos 61
Propiedades 62
Métodos 63
Objetos 64
Mutabilidad 66
El diario del licántropo 68
Calculando correlación 70
Ciclos de array 71
El análisis final 72
Arrayología avanzada 74
Strings y sus propiedades 76
Parámetros restantes 77
El objeto Math 78
Desestructurar 80
JSON 81
Resumen 82

Ejercicios 83

5 Funciones de Orden Superior 86
Abstracción 87
Abstrayendo la repetición 88
Funciones de orden superior 89
Conjunto de datos de códigos 90
Filtrando arrays 91
Transformando con map 92
Resumiendo con reduce 93
Composabilidad 94
Strings y códigos de caracteres 96
Reconociendo texto 98
Resumen 99
Ejercicios 100

6 La Vida Secreta de los Objetos 101
Encapsulación 101
Métodos 102
Prototipos 103
Clases 105
Notación de clase 106
Sobreescribiendo propiedades derivadas 107
Mapas 109
Polimorfismo 111
Símbolos 111
La interfaz de iterador 113
Getters, setters y estáticos 115
Herencia 117
El operador instanceof 118
Resumen 119
Ejercicios 120

7 Proyecto: Un Robot 122
VillaPradera 122
La tarea 124
Datos persistentes 126
Simulación 127
La ruta del camión de correos 129
Búsqueda de rutas 129

Ejercicios 132

8 Bugs y Errores 134
Lenguaje 134
Modo estricto 135
Tipos 136
Probando 137
Depuración 138
Propagación de errores 140
Excepciones 141
Limpiando después de excepciones 143
Captura selectiva 145
Afirmaciones 147
Resumen 148
Ejercicios 148

9 Expresiones Regulares 150
Creando una expresión regular 150
Probando por coincidencias 151
Conjuntos de caracteres 151
Repitiendo partes de un patrón 153
Agrupando subexpresiones 154
Coincidencias y grupos 154
La clase Date (“Fecha”) 156
Palabra y límites de string 157
Patrones de elección 158
Las mecánicas del emparejamiento 158
Retrocediendo 159
El método replace 161
Codicia 163
Creando objetos RegExp dinámicamente 164
El método search 165
La propiedad lastIndex 166
Análisis de un archivo INI 168
Caracteres internacionales 170
Resumen 171
Ejercicios 173

10 Módulos 175
Módulos 175

Paquetes 176
Módulos improvisados 177
Evaluando datos como código 178
CommonJS 179
Módulos ECMAScript 182
Construyendo y empaquetando 183
Diseño de módulos 184
Resumen 186
Ejercicios 187

11 Programación Asincrónica 189
Asincronicidad 189
Tecnología cuervo 191
Devolución de llamadas 192
Promesas 194
Fracaso 196
Las redes son difíciles 197
Colecciones de promesas 200
Inundación de red 201
Enrutamiento de mensajes 202
Funciones asíncronas 205
Generadores 207
El ciclo de evento 208
Errores asincrónicos 209
Resumen 211
Ejercicios 211

12 Proyecto: Un Lenguaje de Programación 213
Análisis 213
The evaluator 218
Special forms 219
The environment 221
Functions 222
Compilation 223
Cheating 224
Exercises 225

13 El Modelo de Objeto del Documento 227
La estructura del documento 227
Árboles 228

El estándar . . . 229
Moviéndose a través del árbol . . . 230
Buscar elementos . . . 232
Actualizar el documento . . . 233
Crear nodos . . . 233
Atributos . . . 235
Layout . . . 236
Estilización . . . 238
Estilos en Cascada . . . 240
Selectores de consulta . . . 241
Posicionamiento y animaciones . . . 242
Resumen . . . 245
Ejercicios . . . 245

14 Manejo de Eventos 248
Manejador de eventos . . . 248
Eventos y nodos DOM . . . 249
Objetos de evento . . . 250
Propagación . . . 251
Acciones por defecto . . . 252
Eventos de teclado . . . 253
Eventos de puntero . . . 255
Eventos de desplazamiento . . . 259
Eventos de foco . . . 260
Evento de carga . . . 261
Eventos y el ciclo de eventos . . . 261
Temporizadores . . . 262
Antirrebote . . . 263
Resumen . . . 265
Ejercicios . . . 265

15 Proyecto: Un Juego de Plataforma 267
El juego . . . 267
La tecnología . . . 268
Niveles . . . 268
Leyendo un nivel . . . 269
Actors . . . 271
Encapsulation as a burden . . . 275
Drawing . . . 275
Motion and collision . . . 281

Actor updates 284
Tracking keys 286
Running the game 286
Exercises 289

16 Node.js 290
Antecedentes 290
El comando Node 291
Módulos 292
Instalación con NPM 293
El módulo del sistema de archivos 296
The HTTP module 297
Streams 299
A file server 301
Summary 306
Exercises 307

17 Proyecto: Sitio web para compartir habilidades 309
Diseño 309
Long Polling 310
HTTP interface 311
The server 314
The client 321
Exercises 327

Exercise Hints 329
Estructura de Programa 329
Funciones 330
Estructuras de Datos: Objetos y Arrays 331
Funciones de Orden Superior 333
La Vida Secreta de los Objetos 334
Proyecto: Un Robot 335
Bugs y Errores 336
Expresiones Regulares 336
Módulos 337
Programación Asincrónica 339
Proyecto: Un Lenguaje de Programación 340
El Modelo de Objeto del Documento 341
Manejo de Eventos 342
Proyecto: Un Juego de Plataforma 343

Node.js 344
Proyecto: Sitio web para compartir habilidades 345

"Nosotros creemos que estamos creando el sistema para nuestros propios propósitos. Creemos que lo estamos haciendo a nuestra propia imagen... Pero la computadora no es realmente como nosotros. Es una proyección de una parte muy delgada de nosotros mismos: esa porción dedicada a la lógica, el orden, la reglas y la claridad."

—Ellen Ullman, Close to the Machine: Technophilia and its Discontents

Introducción

Este es un libro acerca de instruir computadoras. Hoy en dia las computadoras son tan comunes como los destornilladores (aunque bastante más complejas que estos), y hacer que hagan exactamente lo que quieres que hagan no siempre es fácil.

Si la tarea que tienes para tu computadora es común, y bien entendida, tal y como mostrarte tu correo electrónico o funcionar como una calculadora, puedes abrir la aplicación apropiada y ponerte a trabajar en ella. Pero para realizar tareas únicas o abiertas, es posible que no haya una aplicación disponible.

Ahí es donde la programación podría entrar en juego. La *programación* es el acto de construir un *programa*—un conjunto de instrucciones precisas que le dicen a una computadora qué hacer. Porque las computadoras son bestias tontas y pedantes, la programación es fundamentalmente tediosa y frustrante.

Afortunadamente, si puedes superar eso, y tal vez incluso disfrutar el rigor de pensar en términos que las máquinas tontas puedan manejar, la programación puede ser muy gratificante. Te permite hacer en segundos cosas que tardarían *siglos* a mano. Es una forma de hacer que tu herramienta computadora haga cosas que antes no podía. Ademas proporciona de un maravilloso ejercicio en pensamiento abstracto.

La mayoría de la programación se realiza con lenguajes de programación. Un *lenguaje de programación* es un lenguaje artificialmente construido que se utiliza para instruir ordenadores. Es interesante que la forma más efectiva que hemos encontrado para comunicarnos con una computadora es bastante parecida a la forma que usamos para comunicarnos entre nosotros. Al igual que los lenguajes humanos, los lenguajes de computación permiten que las palabras y frases sean combinadas de nuevas maneras, lo que nos permite expresar siempre nuevos conceptos.

Las interfaces basadas en lenguajes, que en un momento fueron la principal forma de interactuar con las computadoras para la mayoría de las personas, han sido en gran parte reemplazadas con interfaces más simples y limitadas. Pero todavía están allí, si sabes dónde mirar.

En un punto, las interfaces basadas en lenguajes, como las terminales BASIC

y DOS de los 80 y 90, eran la principal forma de interactuar con las computadoras. Estas han sido reemplazados en gran medida por interfaces visuales, las cuales son más fáciles de aprender pero ofrecen menos libertad. Los lenguajes de computadora todavía están allí, si sabes dónde mirar. Uno de esos lenguajes, JavaScript, está integrado en cada navegador web moderno y, por lo tanto, está disponible en casi todos los dispositivos.

Este libro intentará familiarizarte lo suficiente con este lenguaje para poder hacer cosas útiles y divertidas con él.

Acerca de la programación

Además de explicar JavaScript, también introduciré los principios básicos de la programación. La programación, en realidad, es difícil. Las reglas fundamentales son típicamente simples y claras, pero los programas construidos en base a estas reglas tienden a ser lo suficientemente complejas como para introducir sus propias reglas y complejidad. De alguna manera, estás construyendo tu propio laberinto, y es posible que te pierdas en él.

Habrá momentos en los que leer este libro se sentirá terriblemente frustrante. Si eres nuevo en la programación, habrá mucho material nuevo para digerir. Gran parte de este material sera entonces *combinado* en formas que requerirán que hagas conexiones adicionales.

Depende de ti hacer el esfuerzo necesario. Cuando estés luchando para seguir el libro, no saltes a ninguna conclusión acerca de tus propias capacidades. Estás bien, sólo tienes que seguir intentando. Tomate un descanso, vuelve a leer algún material, y asegúrate de leer y comprender los programas de ejemplo y ejercicios. Aprender es un trabajo duro, pero todo lo que aprendes se convertirá en tuyo, y hará que el aprendizaje subsiguiente sea más fácil.

> Cuando la acción deja de servirte, reúne información; cuando la información deja de servirte, duerme..
>
> —Ursula K. Le Guin, La Mano Izquierda De La Oscuridad

Un programa son muchas cosas. Es una pieza de texto escrita por un programador, es la fuerza directriz que hace que la computadora haga lo que hace, son datos en la memoria de la computadora, y sin embargo controla las acciones realizadas en esta misma memoria. Las analogías que intentan comparar programas a objetos con los que estamos familiarizados tienden a fallar. Una analogía que es superficialmente adecuada es el de una máquina —muchas partes separadas tienden a estar involucradas—, y para hacer que todo funcione, tenemos

que considerar la formas en las que estas partes se interconectan y contribuyen a la operación de un todo.

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

Un programa es un edificio de pensamiento. No cuesta nada construirlo, no pesa nada, y crece fácilmente bajo el teclear de nuestras manos.

Pero sin ningún cuidado, el tamaño de un programa y su complejidad crecerán sin control, confundiendo incluso a la persona que lo creó. Mantener programas bajo control es el problema principal de la programación. Cuando un programa funciona, es hermoso. El arte de la programación es la habilidad de controlar la complejidad. Un gran programa es moderado, hecho simple en su complejidad.

Algunos programadores creen que esta complejidad se maneja mejor mediante el uso de solo un pequeño conjunto de técnicas bien entendidas en sus programas. Ellos han compuesto reglas estrictas ("mejores prácticas") que prescriben la forma que los programas deberían tener, y se mantienen cuidadosamente dentro de su pequeña y segura zona.

Esto no solamente es aburrido, sino que también es ineficaz. Problemas nuevos a menudo requieren soluciones nuevas. El campo de la programación es joven y todavía se esta desarrollando rápidamente, y es lo suficientemente variado como para tener espacio para aproximaciones salvajemente diferentes. Hay muchos errores terribles que hacer en el diseño de programas, así que ve adelante y comételos para que los entiendas mejor. La idea de cómo se ve un buen programa se desarrolla con la práctica, no se aprende de una lista de reglas.

Por qué el lenguaje importa

Al principio, en el nacimiento de la informática, no habían lenguajes de programación. Los programas se veían mas o menos 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
```

Ese es un programa que suma los números del 1 al 10 entre ellos e imprime el resultado: `1 + 2 + ... + 10 = 55`. Podría ser ejecutado en una simple máquina hipotética. Para programar las primeras computadoras, era necesario colocar grandes arreglos de interruptores en la posición correcta o perforar agujeros en tarjetas de cartón y dárselos a la computadora. Probablemente puedas imaginarte lo tedioso y propenso a errores que era este procedimiento. Incluso escribir programas simples requería de mucha inteligencia y disciplina. Los complejos eran casi inconcebibles.

Por supuesto, ingresar manualmente estos patrones arcanos de bits (los unos y ceros) le dieron al programador un profundo sentido de ser un poderoso mago. Y eso tiene que valer algo en términos de satisfacción laboral.

{{index memoria, instrucción}}

Cada línea del programa anterior contiene una sola instrucción. Podría ser escrito en español así:

1. Almacenar el número 0 en la ubicación de memoria 0.
2. Almacenar el número 1 en la ubicación de memoria 1.
3. Almacenar el valor de la ubicación de memoria 1 en la ubicación de memoria 2.
4. Restar el número 11 del valor en la ubicación de memoria 2.
5. Si el valor en la ubicación de memoria 2 es el número 0, continuar con la instrucción 9.
6. Sumar el valor de la ubicación de memoria 1 a la ubicación de memoria 0.
7. Sumar el número 1 al valor de la ubicación de memoria 1.
8. Continuar con la instrucción 3.
9. Imprimir el valor de la ubicación de memoria 0.

Aunque eso ya es más legible que la sopa de bits, es aún difícil de entender. Usar nombres en lugar de números para las instrucciones y ubicaciones de memoria ayuda:

```
 Establecer "total" como 0.
 Establecer "cuenta" como 1.
[loop]
 Establecer "comparar" como "cuenta".
 Restar 11 de "comparar".
 Si "comparar" es cero, continuar en [fin].
 Agregar "cuenta" a "total".
 Agregar 1 a "cuenta".
 Continuar en [loop].
[fin]
 Imprimir "total".
```

¿Puedes ver cómo funciona el programa en este punto? Las primeras dos líneas le dan a dos ubicaciones de memoria sus valores iniciales: se usará `total` para construir el resultado de la computación, y `cuenta` hará un seguimiento del número que estamos mirando actualmente. Las líneas usando `comparar` son probablemente las más extrañas. El programa quiere ver si `cuenta` es igual a 11 para decidir si puede detener su ejecución. Debido a que nuestra máquina hipotética es bastante primitiva, esta solo puede probar si un número es cero y hace una decisión (o salta) basándose en eso. Por lo tanto, usa la ubicación de memoria etiquetada como `comparar` para calcular el valor de `cuenta - 11` y toma una decisión basada en ese valor. Las siguientes dos líneas agregan el valor de `cuenta` al resultado e incrementan `cuenta` en 1 cada vez que el programa haya decidido que `cuenta` todavía no es 11.

Aquí está el mismo programa en JavaScript:

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

Esta versión nos da algunas mejoras más. La más importante, ya no hay necesidad de especificar la forma en que queremos que el programa salte hacia adelante y hacia atrás. El constructo del lenguaje `while` se ocupa de eso. Este continúa ejecutando el bloque de código (envuelto en llaves) debajo de el, siempre y cuando la condición que se le dio se mantenga. Esa condición es `cuenta <= 10`, lo que significa "*cuenta* es menor o igual a 10". Ya no tenemos que crear un valor temporal y compararlo con cero, lo cual era un detalle

poco interesante. Parte del poder de los lenguajes de programación es que se encargan por nosotros de los detalles sin interés.

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

{{index "sum function", "range function", abstracción, function}}

Finalmente, aquí está cómo se vería el programa si tuviéramos acceso a las las convenientes operaciones `rango` y `suma` disponibles, que respectivamente crean una colección de números dentro de un rango y calculan la suma de una colección de números:

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

La moraleja de esta historia es que el mismo programa se puede expresar en formas largas y cortas, ilegibles y legibles. La primera versión del programa era extremadamente oscura, mientras que esta última es casi Español: muestra en el `log` de la consola la `suma` del `rango` de los números 1 al 10. (En capítulos posteriores veremos cómo definir operaciones como `suma` y `rango`.)

Un buen lenguaje de programación ayuda al programador permitiéndole hablar sobre las acciones que la computadora tiene que realizar en un nivel superior. Ayuda a omitir detalles poco interesantes, proporciona bloques de construcción convenientes (como `while` y `console.log`), te permite que definas tus propios bloques de construcción (como `suma` y `rango`), y hace que esos bloques sean fáciles de componer.

¿Qué es JavaScript?

JavaScript se introdujo en 1995 como una forma de agregar programas a páginas web en el navegador Netscape Navigator. El lenguaje ha sido desde entonces adoptado por todos los otros navegadores web principales. Ha hecho que las aplicaciones web modernas sean posibles: aplicaciones con las que puedes interactuar directamente, sin hacer una recarga de página para cada acción. JavaScript también es utilizado en sitios web más tradicionales para proporcionar diversas formas de interactividad e ingenio.

Es importante tener en cuenta que JavaScript casi no tiene nada que ver con el lenguaje de programación llamado Java. El nombre similar fue inspirado por consideraciones de marketing, en lugar de buen juicio. Cuando JavaScript estaba siendo introducido, el lenguaje Java estaba siendo fuertemente comercializado y estaba ganando popularidad. Alguien pensó que era una buena idea

intentar cabalgar sobre este éxito. Ahora estamos atrapados con el nombre.

Después de su adopción fuera de Netscape, un documento estándar fue escrito para describir la forma en que debería funcionar el lenguaje JavaScript, para que las diversas piezas de software que decían ser compatibles con JavaScript en realidad estuvieran hablando del mismo lenguaje. Este se llamo el Estándar ECMAScript, por Ecma International que hizo la estandarización. En la práctica, los términos ECMAScript y JavaScript se puede usar indistintamente, son dos nombres para el mismo lenguaje.

Hay quienes dirán cosas *terribles* sobre JavaScript. Muchas de estas cosas son verdaderas. Cuando estaba comenzando a escribir algo en JavaScript por primera vez, rápidamente comencé a despreciarlo. El lenguaje aceptaba casi cualquier cosa que escribiera, pero la interpretaba de una manera que era completamente diferente de lo que quería decir. Por supuesto, esto tenía mucho que ver con el hecho de que no tenía idea de lo que estaba haciendo, pero hay un problema real aquí: JavaScript es ridículamente liberal en lo que permite. La idea detrás de este diseño era que haría a la programación en JavaScript más fácil para los principiantes. En realidad, lo que mas hace es que encontrar problemas en tus programas sea más difícil porque el sistema no los señalará por ti.

Sin embargo, esta flexibilidad también tiene sus ventajas. Deja espacio para muchas técnicas que son imposibles en idiomas más rígidos, y como verás (por ejemplo en el Capítulo 10) se pueden usar para superar algunas de las deficiencias de JavaScript. Después de aprender el idioma correctamente y luego de trabajar con él por un tiempo, he aprendido a *querer* a JavaScript.

Ha habido varias versiones de JavaScript. ECMAScript versión 3 fue la versión mas ampliamente compatible en el momento del ascenso de JavaScript a su dominio, aproximadamente entre 2000 y 2010. Durante este tiempo, se trabajó en marcha hacia una ambiciosa versión 4, que planeaba una serie de radicales mejoras y extensiones al lenguaje. Cambiar un lenguaje vivo y ampliamente utilizado de una manera tan radical resultó ser políticamente difícil, y el trabajo en la versión 4 fue abandonado en 2008, lo que llevó a la versión 5, mucho menos ambiciosa, que se publicaría en el 2009. Luego, en 2015, una actualización importante, incluyendo algunas de las ideas planificadas para la versión 4, fue realizada. Desde entonces hemos tenido actualizaciones nuevas y pequeñas cada año.

El hecho de que el lenguaje esté evolucionando significa que los navegadores deben mantenerse constantemente al día, y si estás usando uno más antiguo, puede que este no soporte todas las mejoras. Los diseñadores de lenguajes tienen cuidado de no realizar cualquier cambio que pueda romper los programas ya existentes, de manera que los nuevos navegadores puedan todavía ejecutar

programas viejos. En este libro, usaré la versión 2017 de JavaScript.

Los navegadores web no son las únicas plataformas en las que se usa JavaScript. Algunas bases de datos, como MongoDB y CouchDB, usan JavaScript como su lenguaje de scripting y consultas. Varias plataformas para programación de escritorio y servidores, más notablemente el proyecto Node.js (el tema del Capítulo 20) proporcionan un entorno para programar en JavaScript fuera del navegador.

Código, y qué hacer con él

Código es el texto que compone los programas. La mayoría de los capítulos en este libro contienen bastante. Creo que leer código y escribir código son partes indispensables del aprendizaje para programar. Trata de no solo echar un vistazo a los ejemplos, léelos atentamente y entiéndelos. Esto puede ser algo lento y confuso al principio, pero te prometo que rápidamente vas agarrar el truco. Lo mismo ocurre con los ejercicios. No supongas que los entiendes hasta que hayas escrito una solución funcional para resolverlos.

Te recomiendo que pruebes tus soluciones a los ejercicios en un intérprete real de JavaScript. De esta forma, obtendrás retroalimentación inmediata acerca de que si esta funcionando lo que estás haciendo, y, espero, serás tentado a experimentar e ir más allá de los ejercicios.

La forma más fácil de ejecutar el código de ejemplo en el libro y experimentar con él, es buscarlo en la versión en línea del libro en *eloquentjavascript.net*. Alli puedes hacer clic en cualquier ejemplo de código para editar y ejecutarlo y ver el resultado que produce. Para trabajar en los ejercicios, ve a *eloquentjavascript.net/code*, que proporciona el código de inicio para cada ejercicio de programación y te permite ver las soluciones.

Si deseas ejecutar los programas definidos en este libro fuera de la caja de arena del libro, se requiere cierto cuidado. Muchos ejemplos se mantienen por si mismos y deberían de funcionar en cualquier entorno de JavaScript. Pero código en capítulos más avanzados a menudo se escribe para un entorno específico (el navegador o Node.js) y solo puede ser ejecutado allí. Además, muchos capítulos definen programas más grandes, y las piezas de código que aparecen en ellos dependen de otras piezas o de archivos externos. La caja de arena en el sitio web proporciona enlaces a archivos Zip que contienen todos los scripts y archivos de datos necesarios para ejecutar el código de un capítulo determinado.

Descripción general de este libro

Este libro contiene aproximadamente tres partes. Los primeros 12 capítulos discuten el lenguaje JavaScript en sí. Los siguientes siete capítulos son acerca de los navegadores web y la forma en la que JavaScript es usado para programarlos. Finalmente, dos capítulos están dedicados a Node.js, otro entorno en donde programar JavaScript.

A lo largo del libro, hay cinco *capítulos de proyectos*, que describen programas de ejemplo más grandes para darte una idea de la programación real. En orden de aparición, trabajaremos en la construcción de un robot de delivery, un lenguaje de programación, un juego de plataforma, un programa de paint y un sitio web dinámico.

La parte del lenguaje del libro comienza con cuatro capítulos para presentar la estructura básica del lenguaje de JavaScript. Estos introducen estructuras de control (como la palabra `while` que ya viste en esta introducción), funciones (escribir tus propios bloques de construcción), y estructuras de datos. Después de estos, seras capaz de escribir programas simples. Luego, los Capítulos 5 y 6 introducen técnicas para usar funciones y objetos y asi escribir código más *abstracto* y de manera que puedas mantener la complejidad bajo control.

Después de un primer capítulo de proyecto, la primera parte del libro continúa con los capítulos sobre manejo y solución de errores, en expresiones regulares (una herramienta importante para trabajar con texto), en modularidad (otra defensa contra la complejidad), y en programación asincrónica (que se encarga de eventos que toman tiempo). El segundo capítulo de proyecto concluye la primera parte del libro.

La segunda parte, Capítulos 13 a 19, describe las herramientas a las que el JavaScript en un navegador tiene acceso. Aprenderás a mostrar cosas en la pantalla (Capítulos 14 y 17), responder a entradas de usuario (Capitulo 15), y a comunicarte a través de la red (Capitulo 18). Hay dos capítulos de proyectos en este parte.

Después de eso, el Capítulo 20 describe Node.js, y el Capitulo 21 construye un pequeño sistema web usando esta herramienta.

Convenciones tipográficas

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

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

Algunas veces, para mostrar el resultado que produce un programa, la salida esperada se escribe después de el, con dos diagonales y una flecha en frente.

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

¡Buena suerte!

"Debajo de la superficie de la máquina, el programa se mueve. Sin esfuerzo, se expande y se contrae. En gran armonía, los electrones se dispersan y se reagrupan. Las figuras en el monitor son tan solo ondas sobre el agua. La esencia se mantiene invisible debajo de la superficie."

—Master Yuan-Ma, The Book of Programming

Chapter 1

Valores, Tipos, y Operadores

Dentro del mundo de la computadora, solo existen datos. Puedes leer datos, modificar datos, crear nuevos datos—pero todo lo que no sean datos, no puede ser mencionado. Toda estos datos están almacenados como largas secuencias de bits, y por lo tanto, todos los datos son fundamentalmente parecidos.

Los *bits* son cualquier tipo de cosa que pueda tener dos valores, usualmente descritos como ceros y unos. Dentro de la computadora, estos toman formas tales como cargas eléctricas altas o bajas, una señal fuerte o débil, o un punto brillante u opaco en la superficie de un CD. Cualquier pedazo de información discreta puede ser reducida a una secuencia de ceros y unos y, de esa manera ser representada en bits.

Por ejemplo, podemos expresar el numero 13 en bits. Funciona de la misma manera que un número decimal, pero en vez de 10 diferentes dígitos, solo tienes 2, y el peso de cada uno aumenta por un factor de 2 de derecha a izquierda. Aquí tenemos los bits que conforman el número 13, con el peso de cada dígito mostrado debajo de el:

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

Entonces ese es el número binario 00001101, o 8 + 4 + 1, o 13.

Valores

Imagina un mar de bits—un océano de ellos. Una computadora moderna promedio tiene mas de 30 billones de bits en su almacenamiento de datos volátiles (memoria funcional). El almacenamiento no volátil (disco duro o equivalente) tiende a tener unas cuantas mas ordenes de magnitud.

Para poder trabajar con tales cantidades de bits sin perdernos, debemos separarlos en porciones que representen pedazos de información. En un entorno de JavaScript, esas porciones son llamadas *valores*. Aunque todos los valores

están hechos de bits, estos juegan papeles diferentes. Cada valor tiene un tipo que determina su rol. Algunos valores son números, otros son pedazos de texto, otros son funciones, y asi sucesivamente.

Para crear un valor, solo debemos de invocar su nombre. Esto es conveniente. No tenemos que recopilar materiales de construcción para nuestros valores, o pagar por ellos. Solo llamamos su nombre, y *woosh*, ahi lo tienes. Estos no son realmente creados de la nada, por supuesto. Cada valor tiene que ser almacenado en algún sitio, y si quieres usar una cantidad gigante de valores al mismo tiempo, puede que te quedes sin memoria. Afortunadamente, esto solo es un problema si los necesitas todos al mismo tiempo. Tan pronto como dejes de utilizar un valor, este se disipará, dejando atrás sus bits para que estos sean reciclados como material de construcción para la próxima generación de valores.

Este capitulo introduce los elementos atómicos de los programas en JavaScript, estos son, los tipos de valores simples y los operadores que actúan en tales valores.

Números

Valores del tipo *number* (número) son, como es de esperar, valores numéricos. En un programa hecho en JavaScript, se escriben de la siguiente manera:

```
13
```

Utiliza eso en un programa, y ocasionara que el patron de bits que representa el número 13 sea creado dentro de la memoria de la computadora.

JavaScript utiliza un número fijo de bits, específicamente 64 de ellos, para almacenar un solo valor numérico. Solo existen una cantidad finita de patrones que podemos crear con 64 bits, lo que significa que la cantidad de números diferentes que pueden ser representados es limitada. Para una cantidad de N dígitos decimales, la cantidad de números que pueden ser representados es 10^{N}. Del mismo modo, dados 64 dígitos binarios, podemos representar 2^{64} números diferentes, lo que es alrededor de 18 mil trillones (un 18 con 18 ceros más). Eso es muchísimo.

La memoria de un computador solía ser mucho mas pequeña que en la actualidad, y las personas tendían a utilizar grupos de 8 o 16 bits para representar sus números. Era común accidentalmente *desbordar* esta limitación— terminando con un número que no cupiera dentro de la cantidad dada de bits. Hoy en día, incluso computadoras que caben dentro de tu bolsillo poseen de bastante

memoria, por lo que somos libres de usar pedazos de memoria de 64 bits, y solamente nos tenemos que preocupar por desbordamientos de memoria cuando lidiamos con números verdaderamente astronómicos.

A pesar de esto, no todos los números enteros por debajo de 18 mil trillones caben en un número de JavaScript. Esos bits también almacenan números negativos, por lo que un bit indica el signo de un número. Un problema mayor es que los números no enteros tienen que ser representados también. Para hacer esto, algunos de los bits son usados para almacenar la posición del punto decimal. El número entero mas grande que puede ser almacenado está en el rango de los 9 trillones (15 ceros)—lo cual es todavía placenteramente inmenso.

Los números fraccionarios se escriben usando un punto:

```
9.81
```

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

```
2.998e8
```

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

Los cálculos con números enteros (también llamados *integers*) mas pequeños a los 9 trillones anteriormente mencionados están garantizados a ser siempre precisos. Desafortunadamente, los calculos con números fraccionarios, generalmente no lo son. Así como π (pi) no puede ser precisamente expresado por un número finito de números decimales, muchos números pierden algo de precisión cuando solo hay 64 bits disponibles para almacenarlos. Esto es una pena, pero solo causa problemas prácticos en situaciones especificas. Lo importante es que debemos ser consciente de estas limitaciones y tratar a los números fraccionarios como aproximaciones, no como valores precisos.

Aritmética

Lo que mayormente se hace con los números es aritmética. Operaciones aritméticas tales como la adición y la multiplicación, toman dos valores numéricos y producen un nuevo valor a raíz de ellos. Asi es como lucen en JavaScript:

```
100 + 4 * 11
```

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

¿Pero el ejemplo significa "agrega 4 y 100, y multiplica el resultado por 11", o es la multiplicación aplicada antes de la adición? Como quizás hayas podido adivinar, la multiplicación sucede primero. Pero asi como en las matemáticas, puedes cambiar este orden envolviendo la adición en paréntesis:

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

Para sustraer, existe el operador `-`, y la división puede ser realizada con el operador `/`.

Cuando operadores aparecen juntos sin paréntesis, el orden en el cual son aplicados es determinado por la *precedencia* de los operadores. El ejemplo muestra que la multiplicación es aplicada antes que la adición. El operador `/` tiene la misma precedencia que `*`. Lo mismo aplica para `+` y `-`. Cuando operadores con la misma precedencia aparecen uno al lado del otro, como en `1 - 2 + 1`, estos se aplican de izquierda a derecha: `(1 - 2)+ 1`.

Estas reglas de precedencia no son algo de lo que deberias preocuparte. Cuando tengas dudas, solo agrega un paréntesis.

Existe otro operador aritmético que quizás no reconozcas inmediatamente. El símbolo `%` es utilizado para representar la operación de *residuo*. `X % Y` es el residuo de dividir `X` entre `Y`. Por ejemplo, `314 % 100` produce `14`, y `144 % 12` produce `0`. La precedencia del residuo es la la misma que la de la multiplicación y la división. Frecuentemente veras que este operador es tambien conocido como *modulo*.

Números especiales

Existen 3 valores especiales en JavaScript que son considerados números pero que no se comportan como números normales.

Los primeros dos son `Infinity` y `-Infinity`, los cuales representan las infinidades positivas y negativas. `Infinity - 1` aun es `Infinity`, y asi sucesivamente. A pesar de esto, no confíes mucho en computaciones que dependan de infinidades. Estas no son matemáticamente confiables, y puede que muy rápidamente nos resulten en el próximo número especial: `NaN`.

`NaN` significa "no es un número" ("Not A Number"), aunque *sea* un valor del tipo numérico. Obtendras este resultado cuando, por ejemplo, trates de

calcular `0 / 0` (cero dividido entre cero), `Infinity - Infinity`, o cualquier otra cantidad de operaciones numéricas que no produzcan un resultado significante.

Strings

El próximo tipo de dato básico es el *string*. Los Strings son usados para representar texto. Son escritos encerrando su contenido en comillas:

```
`Debajo en el mar`
"Descansa en el océano"
'Flota en el océano'
```

Puedes usar comillas simples, comillas dobles, o comillas invertidas para representar strings, siempre y cuando las comillas al principio y al final coincidan.

Casi todo puede ser colocado entre comillas, y JavaScript construirá un valor string a partir de ello. Pero algunos caracteres son mas difíciles. Te puedes imaginar que colocar comillas entre comillas podría ser difícil. Los *Newlines* (los caracteres que obtienes cuando presionas la tecla de Enter) solo pueden ser incluidos cuando el string está encapsulado con comillas invertidas (`\`).

Para hacer posible incluir tales caracteres en un string, la siguiente notación es utilizada: cuando una barra invertida (`\`) es encontrada dentro de un texto entre comillas, indica que el carácter que le sigue tiene un significado especial. Esto se conoce como *escapar* el carácter. Una comilla que es precedida por una barra invertida no representará el final del string sino que formara parte del mismo. Cuando el carácter `n` es precedido por una barra invertida, este se interpreta como un Newline (salto de linea). De la mima forma, `t` después de una barra invertida, se interpreta como un character de tabulación. Toma como referencia el siguiente string:

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

El texto actual es este:

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

Se encuentran, por supuesto, situaciones donde queremos que una barra invertida en un string solo sea una barra invertida, y no un carácter especial. Si dos barras invertidas prosiguen una a la otra, serán colapsadas y sólo una per-

manecerá en el valor resultante del string. Asi es como el string "*Un carácter de salto de linea es escrito así: "\n".*" puede ser expresado:

```
Un carácter de salto de linea es escrito así: \"\\n\"."
```

También los strings deben de ser modelados como una serie de bits para poder existir dentro del computador. La forma en la que JavaScript hace esto es basada en el estándar *Unicode*. Este estándar asigna un número a todo carácter que alguna vez pudieras necesitar, incluyendo caracteres en Griego, Árabe, Japones, Armenio, y asi sucesivamente. Si tenemos un número para representar cada carácter, un string puede ser descrito como una secuencia de números.

Y eso es lo que hace JavaScript. Pero hay una complicación: La representación de JavaScript usa 16 bits por cada elemento string, en el cual caben 2^{16} números diferentes. Pero Unicode define mas caracteres que aquellos—aproximadamente el doble, en este momento. Entonces algunos caracteres, como muchos emojis, necesitan ocupar dos "posiciones de caracteres" en los strings de JavaScript. Volveremos a este tema en el Capitulo 5.

Los strings no pueden ser divididos, multiplicados, o substraidos, pero el operador `+` *puede* ser utilizado en ellos. No los agrega, sino que los *concatena*—pega dos strings juntos. La siguiente línea producirá el string `"concatenar"`:

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

Los valores string tienen un conjunto de funciones (*métodos*) asociadas, que pueden ser usadas para realizar operaciones en ellos. Regresaremos a estas en el Capítulo 4.

Los strings escritos con comillas simples o dobles se comportan casi de la misma manera—La unica diferencia es el tipo de comilla que necesitamos para escapar dentro de ellos. Los strings de comillas inversas, usualmente llamados *plantillas literales*, pueden realizar algunos trucos más. Mas alla de permitir saltos de lineas, pueden también incrustar otros valores.

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

Cuando escribes algo dentro de `${}` en una plantilla literal, el resultado será computado, convertido a string, e incluido en esa posición. El ejemplo anterior produce "*la mitad de 100 es 50*".

Operadores unarios

No todo los operadores son simbolos. Algunos se escriben como palabras. Un ejemplo es el operador `typeof`, que produce un string con el nombre del tipo de valor que le demos.

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

Usaremos `console.log` en los ejemplos de código para indicar que que queremos ver el resultado de alguna evaluación. Mas acerca de esto esto en el proximo capitulo.

En los otros operadores que hemos visto hasta ahora, todos operaban en dos valores, pero `typeof` sola opera con un valor. Los operadores que usan dos valores son llamados operadores *binarios*, mientras que aquellos operadores que usan uno son llamados operadores *unarios*. El operador menos puede ser usado tanto como un operador binario o como un operador unario.

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

Valores Booleanos

Es frecuentemente util tener un valor que distingue entre solo dos posibilidades, como “si”, y “no”, o “encendido” y “apagado”. Para este propósito, JavaScript tiene el tipo *Boolean*, que tiene solo dos valores: true (verdadero) y false (falso) que se escriben de la misma forma.

Comparación

Aquí se muestra una forma de producir valores Booleanos:

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

Los signos > y < son tradicionalmente símbolos para “mayor que” y “menor que”, respectivamente. Ambos son operadores binarios. Aplicarlos resulta en un valor Boolean que indica si la condición que indican se cumple.

Los Strings pueden ser comparados de la misma forma.

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

La forma en la que los strings son ordenados, es aproximadamente alfabético, aunque no realmente de la misma forma que esperaríamos ver en un diccionario: las letras mayúsculas son siempre “menores que” las letras minúsculas, así que `"Z" < "a"`, y caracteres no alfabéticos (como `!`, `-` y demás) son también incluidos en el ordenamiento. Cuando comparamos strings, JavaScript evalúa los caracteres de izquierda a derecha, comparando los códigos Unicode uno por uno.

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

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

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

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

Se supone que `NaN` denota el resultado de una computación sin sentido, y como tal, no es igual al resultado de ninguna *otra* computación sin sentido.

Operadores lógicos

También existen algunas operaciones que pueden ser aplicadas a valores Booleanos. JavaScript soporta tres operadores lógicos: *and*, *or*, y *not*. Estos pueden ser usados para “razonar” acerca de valores Booleanos.

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

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

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

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

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

Not se escribe como un signo de exclamación (`!`). Es un operador unario que voltea el valor dado—`!true` produce `false` y `!false` produce `true`.

Cuando estos operadores Booleanos son mezclados con aritmética y con otros operadores, no siempre es obvio cuando son necesarios los paréntesis. En la práctica, usualmente puedes manejarte bien sabiendo que de los operadores que hemos visto hasta ahora, `||` tiene la menor precedencia, luego le sigue `&&`, luego le siguen los operadores de comparación (`>`, `==`, y demás), y luego el resto. Este orden ha sido determinado para que en expresiones como la siguiente, la menor cantidad de paréntesis posible sea necesaria:

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

El ultimo operador lógico que discutiremos no es unario, tampoco binario, sino *ternario*, esto es, que opera en tres valores. Es escrito con un signo de interrogación y dos puntos, de esta forma:

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

Este es llamado el operador *condicional* (o algunas veces simplemente operador *ternario* ya que solo existe uno de este tipo). El valor a la izquierda del signo de interrogación "decide" cual de los otros dos valores sera retornado. Cuando es verdadero, elige el valor de en medio, y cuando es falso, el valor de la derecha.

Valores vacíos

Existen dos valores especiales, escritos como `null` y `undefined`, que son usados para denotar la ausencia de un valor *significativo*. Son en si mismos valores, pero no traen consigo información.

Muchas operaciones en el lenguaje que no producen un valor significativo (veremos algunas mas adelante), producen `undefined` simplemente porque tienen que producir *algún* valor.

La diferencia en significado entre `undefined` y `null` es un accidente del diseño de JavaScript, y realmente no importa la mayor parte del tiempo. En los casos donde realmente tendríamos que preocuparnos por estos valores, mayormente recomiendo que los trates como intercambiables.

Conversión de tipo automática

En la Introducción, mencione que JavaScript tiende a salirse de su camino para aceptar casi cualquier programa que le demos, incluso programas que hacen cosas extrañas. Esto es 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 de valor "incorrecto", JavaScript silenciosamente convertirá ese valor al tipo que necesita, utilizando una serie de reglas que frecuentemente no dan el resultado que quisieras o esperarías. Esto es llamado *coercion de tipo*. El `null` en la primera expresión se torna `0`, y el `"5"`en la segunda expresión se torna `5` (de string a número). Sin embargo, en la tercera expresión, `+` intenta realizar una concatenación de string antes que una adición numérica, entonces el `1` es convertido a `"1"` (de número a string)

Cuando algo que no se traduce a un número en una manera obvia (tal como `"cinco"` o `undefined`) es convertido a un número, obtenemos el valor `NaN`. Operaciones aritméticas subsecuentes con `NaN`, continúan produciendo `NaN`, asi

que si te encuentras obteniendo uno de estos valores en algun lugar inesperado, busca por coerciones de tipo accidentales.

Cuando se utiliza `==` para comparar valores del mismo tipo, el desenlace es fácil de predecir: debemos de obtener verdadero cuando ambos valores son lo mismo, excepto en el caso de `NaN`. Pero cuando los tipos difieren, JavaScript utiliza una serie de reglas complicadas y confusas para determinar que hacer. En la mayoria de los casos, solo tratara de convertir uno de estos valores al tipo del otro valor. Sin embargo, cuando `null` o `undefined` ocurren en cualquiera de los lados del operador, este produce verdadero solo si ambos lados son valores o `null` o `undefined`.

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

Este comportamiento es frecuentemente util. Cuando queremos probar si un valor tiene un valor real en vez de `null` o `undefined`, puedes compararlo con `null` usando el operador `==` (o `!=`).

Pero que pasa si queremos probar que algo se refiere precisamente al valor `false`? Las reglas para convertir strings y números a valores Booleanos, dice que `0`, `NaN`, y el string vació (`""`) cuentan como `false`, mientras que todos los otros valores cuentan como `true`. Debido a esto, expresiones como `0 == false`, y `"" == false` son también verdaderas. Cuando no queremos ninguna conversion de tipo automática, existen otros dos operadores adicionales: `===` y `!==`. El primero prueba si un valor es *precisamente* igual al otro, y el segundo prueba si un valor no es precisamente igual. Entonces `"" === false` es falso, como es de esperarse.

Recomiendo usar el operador de comparación de tres caracteres de una manera defensiva para prevenir que conversiones de tipo inesperadas te estorben. Pero cuando estés seguro de que el tipo va a ser el mismo en ambos lados, no es problemático utilizar los operadores mas cortos.

Corto circuito de operadores lógicos

Los operadores lógicos `&&` y `||`, manejan valores de diferentes tipos de una forma peculiar. Ellos convertirán el valor en su lado izquierdo a un tipo Booleano para decidir que hacer, pero dependiendo del operador y el resultado de la conversión, devolverán o el valor *original* de la izquierda o el valor de la derecha.

El operador `||`, por ejemplo, devolverá el valor de su izquierda cuando este

puede ser convertido a verdadero y de ser lo contrario devolverá el valor de la derecha. Esto tiene el efecto esperado cuando los valores son Booleanos, pero se comporta de una forma algo análoga con valores de otros tipos.

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

Podemos utilizar esta funcionalidad como una forma de recurrir a un valor por defecto. Si tenemos un valor que puede estar vacío, podemos usar `||` después de este para remplazarlo con otro valor. Si el valor inicial puede ser convertido a falso, obtendra el reemplazo en su lugar.

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

Otra propiedad importante de estos dos operadores es que la parte de su derecha solo es evaluada si es necesario. En el caso de de `true || X`, no importa que sea `X`—aun si es una pieza del programa que hace algo *terrible*—el resultado será verdadero, y `X` nunca sera evaluado. Lo mismo sucede con `false && X`, que es falso e ignorará `X`. Esto es llamado *evaluación de corto circuito*.

El operador condicional funciona de manera similar. Del segundo y tercer valor, solo el que es seleccionado es evaluado.

Resumen

Observamos cuatro tipos de valores de JavaScript en este capítulo: números, textos (`strings`), Booleanos, y valores indefinidos.

Tales valores son creados escribiendo su nombre (`true`, `null`) o valor (`13`, `"abc"`). Puedes combinar y transformar valores con operadores. Vimos operadores binarios para aritmética (`+`, `-`, `*`, `/`, y `%`), concatenación de strings (`+`), comparaciones (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`), y lógica (`&&`, `||`), así también como varios otros operadores unarios (`-` para negar un número, `!` para negar lógicamente, y `typeof` para saber el valor de un tipo) y un operador ternario (`?:`) para elegir uno de dos valores basándose en un tercer valor.

Esto te dá la información suficiente para usar JavaScript como una calculadora de bolsillo, pero no para mucho más. El próximo capitulo empezará a juntar estas expresiones para formar programas básicos.

"Y mi corazón brilla de un color rojo brillante bajo mi piel transparente y translúcida, y tienen que administrarme 10cc de JavaScript para conseguir que regrese. (respondo bien a las toxinas en la sangre.) Hombre, esa cosa es increible!"

—_why, Why's (Poignant) Guide to Ruby

Chapter 2

Estructura de Programa

En este capítulo, comenzaremos a hacer cosas que realmente se pueden llamar *programación*. Expandiremos nuestro dominio del lenguaje JavaScript más allá de los sustantivos y fragmentos de oraciones que hemos visto hasta ahora, al punto donde podemos expresar prosa significativa.

Expresiones y declaraciones

En el Capítulo 1, creamos valores y les aplicamos operadores a ellos para obtener nuevos valores. Crear valores de esta manera es la sustancia principal de cualquier programa en JavaScript. Pero esa sustancia tiene que enmarcarse en una estructura más grande para poder ser útil. Así que eso es lo que veremos a continuación.

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

Esto demuestra parte de la belleza de una interfaz basada en un lenguaje. Las expresiones pueden contener otras expresiones de una manera muy similar a como las sub-oraciones en los lenguajes humanos están anidadas, una sub-oración puede contener sus propias sub-oraciones, y así sucesivamente. Esto nos permite construir expresiones que describen cálculos arbitrariamente complejos.

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

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

```
1;
!false;
```

Sin embargo, es un programa inútil. Una expresión puede estar feliz solo con producir un valor, que luego pueda ser utilizado por el código circundante. Una declaración es independiente por si misma, por lo que equivale a algo solo si afecta al mundo. Podría mostrar algo en la pantalla—eso cuenta como cambiar el mundo—o podría cambiar el estado interno de la máquina en una manera que afectará a las declaraciones que vengan después de ella. Estos cambios se llaman *efecto secundarios*. Las declaraciones en el ejemplo anterior solo producen los valores `1` y `true` y luego inmediatamente los tira a la basura. Esto no deja ninguna huella en el mundo. Cuando ejecutes este programa, nada observable ocurre.

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 próxima línea serán tratada como parte de la misma declaración. Las reglas para saber cuando se puede omitir con seguridad son algo complejas y propensas a errores. Asi que en este libro, cada declaración que necesite un punto y coma siempre tendra uno. Te recomiendo que hagas lo mismo, al menos hasta que hayas aprendido más sobre las sutilezas de los puntos y comas que puedan ser omitidos.

Vinculaciones

Cómo mantiene un programa un estado interno? Cómo recuerda cosas? Hasta ahora hemos visto cómo producir nuevos valores a partir de valores anteriores, pero esto no cambia los valores anteriores, y el nuevo valor tiene que ser usado inmediatamente o se disipará nuevamente. Para atrapar y mantener valores, JavaScript proporciona una cosa llamada *vinculación*, o *variable*:

```
let atrapado = 5 * 5;
```

Ese es un segundo tipo de declaración. La palabra especial (*palabra clave*) `let` indica que esta oración va a definir una vinculación. Le sigue el nombre de la vinculación y, si queremos darle un valor inmediatamente, un operador `=` y una expresión.

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

Después de que una vinculación haya sido definida, su nombre puede usarse como una expresión. El valor de tal expresión es el valor que la vinculación mantiene actualmente. Aquí hay un ejemplo:

```
let diez = 10;
```

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

Cuando una vinculación señala a un valor, eso no significa que esté atada a ese valor para siempre. El operador `=` puede usarse en cualquier momento en vinculaciones existentes para desconectarlas de su valor actual y hacer que ellas apuntan a uno nuevo:

```
let humor = "ligero";
console.log(humor);
// → ligero
humor = "oscuro";
console.log(humor);
// → oscuro
```

Deberías imaginar a las vinculaciones como tentáculos, en lugar de cajas. Ellas no *contienen* valores; ellas los *agarran*—dos vinculaciones pueden referirse al mismo valor. Un programa solo puede acceder a los valores que todavía pueda referenciar. Cuando necesitas recordar algo, creces un tentáculo para aferrarte a él o vuelves a conectar uno de tus tentáculos existentes a ese algo.

Veamos otro ejemplo. Para recordar la cantidad de dólares que Luigi aún te debe, creas una vinculación. Y luego, cuando él te pague de vuelta $35, le das a esta vinculación un nuevo valor:

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

Cuando defines una vinculación sin darle un valor, el tentáculo no tiene nada que agarrar, por lo que termina en solo aire. Si pides el valor de una vinculación vacía, obtendrás el valor `undefined`.

Una sola declaración `let` puede definir múltiples vinculaciones. Las definiciones deben estar separadas por comas.

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

Las palabras `var` y `const` también pueden ser usadas para crear vinculaciones,

en una manera similar a `let`.

```
var nombre = "Ayda";
const saludo = "Hola ";
console.log(saludo + nombre);
// → Hola Ayda
```

La primera, `var` (abreviatura de "variable"), es la forma en la que se declaraban las vinculaciones en JavaScript previo al 2015. Volveremos a la forma precisa en que difiere de `let` en el próximo capítulo. Por ahora, recuerda que generalmente hace lo mismo, pero raramente la usaremos en este libro porque tiene algunas propiedades confusas.

La palabra `const` representa una *constante*. Define una vinculación constante, que apunta al mismo valor por el tiempo que viva. Esto es útil para vinculaciones que le dan un nombre a un valor para que fácilmente puedas consultarlo más adelante.

Nombres vinculantes

Los nombres de las vinculaciones pueden ser cualquier palabra. Los dígitos pueden ser parte de los nombres de las vinculaciones pueden—`catch22` es un nombre válido, por ejemplo—pero el nombre no debe comenzar con un dígito. El nombre de una vinculación puede incluir signos de dólar (`$`) o caracteres de subrayado (`_`), pero no otros signos de puntuación o caracteres especiales.

Las palabras con un significado especial, como `let`, son *palabras claves*, y no pueden usarse como nombres vinculantes. También hay una cantidad de palabras que están "reservadas para su uso" en futuras versiones de JavaScript, que tampoco pueden ser usadas como nombres vinculantes. La lista completa de palabras clave y palabras reservadas es bastante larga:

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

No te preocupes por memorizarlas. Cuando crear una vinculación produzca un error de sintaxis inesperado, observa si estas tratando de definir una palabra reservada.

El entorno

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

Funciones

Muchos de los valores proporcionados por el entorno predeterminado tienen el tipo *función*. Una función es una pieza de programa envuelta en un valor. Dichos valores pueden ser *aplicados* para ejecutar el programa envuelto. Por ejemplo, en un entorno navegador, la vinculación `prompt` sostiene una función que muestra un pequeño cuadro de diálogo preguntando por entrada del usuario. Esta se usa así:

```
prompt("Introducir contraseña");
```

Ejecutar una función tambien se conoce como *invocarla*, *llamarla*, o *aplicarla*. Puedes llamar a una función poniendo paréntesis después de una expresión que produzca un valor de función. Usualmente usarás directamente el nombre de la vinculación que contenga la función. Los valores entre los paréntesis se dan al programa dentro de la función. En el ejemplo, la función `prompt` usa el string que le damos como el texto a mostrar en el cuadro de diálogo. Los valores dados a las funciones se llaman *argumentos*. Diferentes funciones pueden necesitar un número diferente o diferentes tipos de argumentos

La función `prompt` no se usa mucho en la programación web moderna, sobre todo porque no tienes control sobre la forma en como se ve la caja de diálogo resultante, pero puede ser útil en programas de juguete y experimentos.

La función console.log

En los ejemplos, utilicé `console.log` para dar salida a los valores. La mayoría de los sistemas de JavaScript (incluidos todos los navegadores web modernos y Node.js) proporcionan una función `console.log` que escribe sus argumentos en *algun* dispositivo de salida de texto. En los navegadores, esta salida aterriza en la consola de JavaScript. Esta parte de la interfaz del navegador está oculta por defecto, pero la mayoría de los navegadores la abren cuando presionas F12 o, en Mac, Command-Option-I. Si eso no funciona, busca en los menús un elemento llamado "herramientas de desarrollador" o algo similar.

Aunque los nombres de las vinculaciones no puedan contener carácteres de puntos, `console.log` tiene uno. Esto es porque `console.log` no es un vinculación simple. En realidad, es una expresión que obtiene la propiedad `log` del valor mantenido por la vinculación `console`. Averiguaremos qué significa esto 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 debido a los efectos secundarios que ellas producen. Las funciones también pueden producir valores, en cuyo caso no necesitan tener un efecto secundario para ser útil. Por ejemplo, la función `Math.max` toma cualquier cantidad de argumentos numéricos y devuelve el mayor de ellos.

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

Cuando una función produce un valor, se dice que *retorna* ese valor. Todo lo que produce un valor es una expresión en JavaScript, lo que significa que las llamadas a funciones se pueden usar dentro de expresiones más grandes. aquí una llamada a `Math.min`, que es lo opuesto a `Math.max`, se usa como parte de una expresión de adición:

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

El próximo capítulo explica cómo escribir tus propias funciones.

Flujo de control

Cuando tu programa contiene más de una declaración, las declaraciones se ejecutan como si fueran una historia, de arriba a abajo. Este programa de ejemplo tiene dos declaraciones. La primera le pide al usuario un número, y la segunda, que se ejecuta después de la primera, muestra el cuadrado de ese número.

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

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

Aquí está la representación esquemática (bastante trivial) de un flujo de control en línea recta:

Ejecución condicional

No todos los programas son caminos rectos. Podemos, por ejemplo, querer crear un camino de ramificación, donde el programa toma la rama adecuada basadandose en la situación en cuestión. Esto se llama *ejecución condicional*.

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

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

Con esta modificación, si ingresas la palabra “loro”, no se mostrara ninguna salida.

La palabra clave `if` ejecuta u omite una declaración dependiendo del valor de una expresión booleana. La expresión decisiva se escribe después de la palabra clave, entre paréntesis, seguida de la declaración a ejecutar.

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

La declaración debajo del `if` está envuelta en llaves (`{`y `}`) en este ejemplo. Estos pueden usarse para agrupar cualquier cantidad de declaraciones en una sola declaración, llamada un *bloque*. Podrías también haberlas omitido en este caso, ya que solo tienes una sola declaración, pero para evitar tener que pensar si se necesitan o no, la mayoría de los programadores en JavaScript las usan en cada una de sus declaraciones envueltas como esta. Seguiremos esta convención en la mayoria de este libro, a excepción de la ocasional declaración de una sola linea.

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

A menudo no solo tendrás código que se ejecuta cuando una condición es verdadera, pero también código que maneja el otro caso. Esta ruta alternativa está representado por la segunda flecha en el diagrama. La palabra clave `else` se puede usar, junto con `if`, para crear dos caminos de ejecución alternativos, de una manera separada.

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

Si tenemos más de dos rutas a elegir, múltiples pares de `if`/`else` se pueden “encadenar”. Aquí hay un ejemplo:

```
let numero = Number(prompt("Elige un numero"));
```

```
if (numero < 10) {
  console.log("Pequeño");
} else if (numero < 100) {
  console.log("Mediano");
} else {
  console.log("Grande");
}
```

El programa primero comprobará si `numero` es menor que 10. Si lo es, eligira esa rama, mostrara `"Pequeño"`, y está listo. Si no es así, toma la rama `else`, que a su vez contiene un segundo `if`. Si la segunda condición (`< 100`) es verdadera, eso significa que el número está entre 10 y 100, y `"Mediano"` se muestra. Si no es así, la segunda y última la rama `else` es elegida.

El esquema de este programa se ve así:

Ciclos while y do

Considera un programa que muestra todos los números pares de 0 a 12. Una forma de escribir esto es la siguiente:

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

Eso funciona, pero la idea de escribir un programa es hacer de algo *menos* trabajo, no más. Si necesitáramos todos los números pares menores a 1.000, este enfoque sería poco práctico. Lo que necesitamos es una forma de ejecutar una pieza de código multiples veces. Esta forma de flujo de control es llamada un *ciclo* (o “loop”):

El flujo de control de ciclos nos permite regresar a algún punto del programa en donde estábamos antes y repetirlo con nuestro estado del programa actual. Si combinamos esto con una vinculación que cuenta, podemos hacer algo como esta:

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

Una declaración que comienza con la palabra clave `while` crea un ciclo. La palabra `while` es seguida por una expresión en paréntesis y luego por una declaración, muy similar a `if`. El bucle sigue ingresando a esta declaración siempre que la expresión produzca un valor que dé `true` cuando sea convertida a Boolean.

La vinculación `numero` demuestra la forma en que una vinculaciónpuede seguir el progreso de un programa. Cada vez que el ciclo se repite, `numero` obtiene un valor que es 2 más que su valor anterior. Al comienzo de cada repetición, se compara con el número 12 para decidir si el trabajo del programa está terminado.

Como un ejemplo que realmente hace algo útil, ahora podemos escribir un programa que calcula y muestra el valor de 2^{10} (2 a la 10). Usamos dos vinculaciones: una para realizar un seguimiento de nuestro resultado y una para contar cuántas veces hemos multiplicado este resultado por 2. El ciclo prueba si la segunda vinculación ha llegado a 10 todavía y, si no, actualiza ambas vinculaciones.

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

El contador también podría haber comenzado en `1` y chequear para `<= 10`, pero, por razones que serán evidentes en el Capítulo 4, es una buena idea ir acostumbrandose a contar desde 0.

Un ciclo `do` es una estructura de control similar a un ciclo `while`. Difiere solo en un punto: un ciclo `do` siempre ejecuta su cuerpo al menos una vez, y comienza a chequear si debe detenerse solo después de esa primera ejecución. Para reflejar esto, la prueba aparece después del cuerpo del ciclo:

```
let tuNombre;
do {
  tuNombre = prompt("Quien eres?");
} while (!tuNombre);
console.log(tuNombre);
```

Este programa te obligará a ingresar un nombre. Preguntará de nuevo y de nuevo hasta que obtenga algo que no sea un string vacío. Aplicar el operador `!` convertirá un valor a tipo Booleano antes de negarlo y todos los strings, excepto `""` seran convertidas a `true`. Esto significa que el ciclo continúa dando vueltas hasta que proporciones un nombre no-vacío.

Indentando Código

En los ejemplos, he estado agregando espacios adelante de declaraciones que son parte de una declaración más grande. Estos no son necesarios—la computadora aceptará el programa normalmente sin ellos. De hecho, incluso las nuevas líneas en los programas son opcionales. Podrías escribir un programa en una sola línea inmensa si asi quisieras.

El rol de esta indentación dentro de los bloques es hacer que la estructura del código se destaque. En código donde se abren nuevos bloques dentro de otros bloques, puede ser difícil ver dónde termina un bloque y donde comienza el otro. Con la indentación apropiada, la forma visual de un programa corresponde a la forma de los bloques dentro de él. Me gusta usar dos espacios para cada bloque abierto, pero los gustos varían—algunas personas usan cuatro espacios, y algunas personas usan carácteres de tabulación. Lo cosa importante es que cada bloque nuevo agregue la misma cantidad de espacio.

```
if (false != true) {
  console.log("Esto tiene sentido.");
  if (1 < 2) {
    console.log("Ninguna sorpresa alli.");
```

```
  }
}
```

La mayoría de los editores de código ayudaran indentar automáticamente las nuevas líneas con la cantidad adecuada.

Ciclos for

Muchos ciclos siguen el patrón visto en los ejemplos de `while`. Primero una vinculación "contador" se crea para seguir el progreso del ciclo. Entonces viene un ciclo `while`, generalmente con una expresión de prueba que verifica si el contador ha alcanzado su valor final. Al final del cuerpo del ciclo, el el contador se actualiza para mantener un seguimiento del progreso.

Debido a que este patrón es muy común, JavaScript y otros lenguajes similares proporcionan una forma un poco más corta y más completa, el ciclo `for`:

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

Este programa es exactamente equivalente al ejemplo anterior de impresión de números pares. El único cambio es que todos las declaraciónes que están relacionadas con el "estado" del ciclo estan agrupadas después del `for`.

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

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

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

Rompiendo un ciclo

Hacer que la condición del ciclo produzca `false` no es la única forma en que el ciclo puede terminar. Hay una declaración especial llamada `break` ("romper") que tiene el efecto de inmediatamente saltar afuera del ciclo circundante.

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

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

Usar el operador restante (`%`) es una manera fácil de probar si un número es divisible por otro número. Si lo es, el residuo de su división es cero.

El constructo `for` en el ejemplo no tiene una parte que verifique cuando finalizar el ciclo. Esto significa que el ciclo nunca se detendrá a menos que se ejecute la declaración `break` dentro de el.

Si eliminases esa declaración `break` o escribieras accidentalmente una condición final que siempre produciera `true`, tu programa estaria atrapado en un *ciclo infinito*. Un programa atrapado en un ciclo infinito nunca terminará de ejecutarse, lo que generalmente es algo malo.

La palabra clave `continue` ("continuar") es similar a `break`, en que influye el progreso de un ciclo. Cuando `continue` se encuentre en el cuerpo de un ciclo, el control salta afuera del cuerpo y continúa con la siguiente iteración del ciclo.

Actualizando vinculaciones de manera sucinta

Especialmente cuando realices un ciclo, un programa a menudo necesita "actualizar" una vinculación para mantener un valor basadandose en el valor anterior de esa vinculación.

```
contador = contador + 1;
```

JavaScript provee de un atajo para esto:

```
contador += 1;
```

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

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

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

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

Despachar en un valor con switch

No es poco común que el código se vea así:

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

Existe un constructo llamado `switch` que está destinada a expresar tales "despachos" de una manera más directa. Desafortunadamente, la sintaxis que JavaScript usa para esto (que heredó de la línea lenguajes de programación C/Java) es algo incómoda—una cadena de declaraciones `if` podria llegar a verse mejor. Aquí hay un ejemplo:

```
switch (prompt("Como esta el clima?")) {
  case "lluvioso":
    console.log("Recuerda salir con un paraguas.");
    break;
  case "soleado":
    console.log("Vistete con poca ropa.");
  case "nublado":
    console.log("Ve afuera.");
    break;
  default:
    console.log("Tipo de clima desconocido!");
    break;
}
```

Puedes poner cualquier número de etiquetas de `case` dentro del bloque abierto por `switch`. El programa comenzará a ejecutarse en la etiqueta que corresponde al valor que se le dio a `switch`, o en `default` si no se encuentra ningún valor que coincida. Continuará ejecutándose, incluso a través de otras etiquetas, hasta que llegue a una declaración `break`. En algunos casos, como en el caso `"soleado"` del ejemplo, esto se puede usar para compartir algo de código entre casos (recomienda salir para ambos climas soleado y nublado). Pero ten cuidado—es fácil olvidarse de `break`, lo que hará que el programa ejecute código que no quieres que sea ejecutado.

Capitalización

Los nombres de vinculaciones no pueden contener espacios, sin embargo, a menudo es útil usar múltiples palabras para describir claramente lo que representa la vinculación. Estas son más o menos tus opciones para escribir el nombre de una vinculación con varias palabras en ella:

```
pequeñatortugaverde
pequeña_tortuga_verde
PequeñaTortugaVerde
pequeñaTortugaVerde
```

El primer estilo puede ser difícil de leer. Me gusta mucho el aspecto del estilo con los guiones bajos, aunque ese estilo es algo fastidioso de escribir. Las funciones estándar de JavaScript, y la mayoría de los programadores de JavaScript, siguen el estilo de abajo: capitalizan cada palabra excepto la primera. No es difícil acostumbrarse a pequeñas cosas así, y programar con estilos de nombres mixtos pueden ser algo discordante para leer, así que seguiremos esta convención.

En algunos casos, como en la función `Number`, la primera letra de la vinculación también está en mayúscula. Esto se hizo para marcar esta función como un constructor. Lo que es un constructor quedará claro en el Capítulo 6. Por ahora, lo importante es no ser molestado por esta aparente falta de consistencia.

Comentarios

A menudo, el código en si mismo no transmite toda la información que deseas que un programa transmita a los lectores humanos, o lo transmite de una manera tan críptica que la gente quizás no lo entienda. En otras ocasiones, podrías simplemente querer incluir algunos pensamientos relacionados como parte de tu programa. Esto es para lo qué son los *comentarios*.

Un comentario es una pieza de texto que es parte de un programa pero que es completamente ignorado por la computadora. JavaScript tiene dos formas de escribir comentarios. Para escribir un comentario de una sola línea, puede usar dos caracteres de barras inclinadas (`//`) y luego el texto del comentario después.

```
let balanceDeCuenta = calcularBalance(cuenta);
// Es un claro del bosque donde canta un río
balanceDeCuenta.ajustar();
// Cuelgan enloquecidamente de las hierbas harapos de plata
let reporte = new Reporte();
// Donde el sol de la orgullosa montaña luce:
añadirAReporte(balanceDeCuenta, reporte);
// Un pequeño valle espumoso de luz.
```

Un comentario `//` va solo haste el final de la línea. Una sección de texto entre `/*` y `*/` se ignorará en su totalidad, independientemente de si contiene saltos de línea. Esto es útil para agregar bloques de información sobre un archivo o un pedazo de programa.

```
/*
  Primero encontré este número garabateado en la parte posterior de
  un viejo cuaderno. Desde entonces, a menudo lo he visto,
  apareciendo en números de teléfono y en los números de serie de
  productos que he comprado. Obviamente me gusta, así que
  decidí quedármelo
*/
const miNumero = 11213;
```

Resumen

Ahora sabes que un programa está construido a partir de declaraciones, las cuales a veces pueden contener más declaraciones. Las declaraciones tienden a

contener expresiones, que a su vez se pueden construir a partir de expresiones mas pequeñas.

Poner declaraciones una despues de otras te da un programa que es ejecutado de arriba hacia abajo. Puedes introducir alteraciones en el flujo de control usando declaraciones condicionales (`if`, `else`, y `switch`) y ciclos (`while`, `do`, y `for`).

Las vinculaciones se pueden usar para archivar datos bajo un nombre, y son utiles para el seguimiento de estado en tu programa. El entorno es el conjunto de vinculaciones que se definen. Los sistemas de JavaScript siempre incluyen por defecto un número de vinculaciones estándar útiles en tu entorno.

Las funciones son valores especiales que encapsulan una parte del programa. Puedes invocarlas escribiendo `nombreDeLaFuncion(argumento1, argumento2)`. Tal llamada a función es una expresión, y puede producir un valor.

Ejercicios

Si no estas seguro de cómo probar tus soluciones para los ejercicios, consulta la introducción.

Cada ejercicio comienza con una descripción del problema. Lee eso y trata de resolver el ejercicio. Si tienes problemas, considera leer las pistas en el final del libro. Las soluciones completas para los ejercicios no estan incluidas en este libro, pero puedes encontrarlas en línea en *eloquentjavascript.net/code*. Si quieres aprender algo de los ejercicios, te recomiendo mirar a las soluciones solo despues de que hayas resuelto el ejercicio, o al menos despues de que lo hayas intentando resolver por un largo tiempo y tengas un ligero dolor de cabeza.

Ciclo de un triángulo

Escriba un ciclo que haga siete llamadas a `console.log` para generar el siguiente triángulo:

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

Puede ser útil saber que puedes encontrar la longitud de un string escribiendo `.length` después de él:

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

FizzBuzz

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

Cuando tengas eso funcionando, modifica tu programa para imprimir `"FizzBuzz"`, para números que sean divisibles entre 3 y 5 (y aún imprimir `"Fizz"` o `"Buzz"` para números divisibles por solo uno de ellos).

(Esta es en realidad una pregunta de entrevista que se ha dicho elimina un porcentaje significativo de candidatos a programadores. Así que si la puedes resolver, tu valor en el mercado laboral acaba de subir).

Tablero de ajedrez

Escribe un programa que cree un string que represente una cuadrícula de 8 × 8, usando caracteres de nueva línea para separar las líneas. En cada posición de la cuadrícula hay un espacio o un carácter "#". Los caracteres deberían de formar un tablero de ajedrez.

Pasar este string a `console.log` debería mostrar algo como esto:

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

Cuando tengas un programa que genere este patrón, define una vinculación `tamaño = 8` y cambia el programa para que funcione con cualquier `tamaño`, dando como salida una cuadrícula con el alto y ancho dados.

"La gente piensa que las ciencias de la computación son el arte de los genios, pero la verdadera realidad es lo opuesto, estas solo consisten en mucha gente haciendo cosas que se construyen una sobre la otra, al igual que un muro hecho de piedras pequeñas."

—Donald Knuth

Chapter 3

Funciones

Las funciones son el pan y la mantequilla de la programación en JavaScript. El concepto de envolver una pieza de programa en un valor tiene muchos usos. Esto nos da una forma de estructurar programas más grandes, de reducir la repetición, de asociar nombres con subprogramas y de aislar estos subprogramas unos con otros.

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

En promedio, un tipico adulto que hable español tiene unas 20,000 palabras en su vocabulario. Pocos lenguajes de programación vienen con 20,000 comandos ya incorporados en el. Y el vocabulario que *está* disponible tiende a ser más precisamente definido, y por lo tanto menos flexible, que en el lenguaje humano. Por lo tanto, nosotros por lo general *tenemos* que introducir nuevos conceptos para evitar repetirnos demasiado.

Definiendo una función

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

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

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

Una función es creada con una expresión que comienza con la palabra clave `function` ("función"). Las funciones tienen un conjunto de *parámetros* (en este caso, solo `x`) y un *cuerpo*, que contiene las declaraciones que deben ser

ejecutadas cuando se llame a la función. El cuerpo de la función de una función creada de esta manera siempre debe estar envuelto en llaves, incluso cuando consista en una sola declaración.

Una función puede tener múltiples parámetros o ningún parámetro en absoluto. En el siguiente ejemplo, `hacerSonido` no lista ningún nombre de parámetro, mientras que `potencia` enumera dos:

```
const hacerSonido = function() {
  console.log("Pling!");
};

hacerSonido();
// → Pling!

const potencia = function(base, exponente) {
  let resultado = 1;
  for (let 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 `hacerSonido`, cuyo único resultado es un efecto secundario. Una declaración de `return` determina el valor que es retornado por la función. Cuando el control se encuentre con tal declaración, inmediatamente salta de la función actual y devuelve el valor retornado al código que llamó la función. Una declaración `return` sin una expresión después de ella hace que la función retorne `undefined`. Funciones que no tienen una declaración `return` en absoluto, como `hacerSonido`, similarmente retornan `undefined`.

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

Vinculaciones y alcances

Cada vinculación tiene un *alcace*, que correspone a la parte del programa en donde la vinculación es visible. Para vinculaciones definidas fuera de cualquier

función o bloque, el alcance es todo el programa—puedes referir a estas vinculaciones en donde sea que quieras. Estas son llamadas *globales*.

Pero las vinculaciones creadas como parámetros de función o declaradas dentro de una función solo puede ser referenciadas en esa función. Estas se llaman *locales*. Cada vez que se llame a la función, se crean nuevas instancias de estas vinculaciones. Esto proporciona cierto aislamiento entre funciones—cada llamada de función actúa sobre su pequeño propio mundo (su entorno local), y a menudo puede ser entendida sin saber mucho acerca de lo qué está pasando en el entorno global.

Vinculaciones declaradas con `let` y `const` son, de hecho, locales al *bloque* donde esten declarados, así que si creas uno de esas dentro de un ciclo, el código antes y después del ciclo no puede "verlas". En JavaScript anterior a 2015, solo las funciones creaban nuevos alcances, por lo que las vinculaciones de estilo-antiguo, creadas con la palabra clave `var`, son visibles a lo largo de toda la función en la que aparecen—o en todo el alcance global, si no están dentro de una función.

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y no es visible desde aqui
console.log(x + z);
// → 40
```

Cada alcance puede "mirar afuera" hacia al alcance que lo rodee, por lo que `x` es visible dentro del bloque en el ejemplo. La excepción es cuando vinculaciones múltiples tienen el mismo nombre—en ese caso, el código solo puede ver a la vinculación más interna. Por ejemplo, cuando el código dentro de la función `dividirEnDos` se refiera a `numero`, estara viendo su *propio* `numero`, no el `numero` en el alcance global.

```
const dividirEnDos = function(numero) {
  return numero / 2;
};

let numero = 10;
console.log(dividirEnDos(100));
// → 50
```

```
console.log(numero);
// → 10
```

Alcance anidado

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

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

```
const humus = function(factor) {
  const ingrediente = function(cantidad, unidad, nombre) {
    let cantidadIngrediente = cantidad * factor;
    if (cantidadIngrediente > 1) {
      unidad += "s";
    }
    console.log(`${cantidadIngrediente} ${unidad} ${nombre}`);
  };
  ingrediente(1, "lata", "garbanzos");
  ingrediente(0.25, "taza", "tahini");
  ingrediente(0.25, "taza", "jugo de limón");
  ingrediente(1, "clavo", "ajo");
  ingrediente(2, "cucharada", "aceite de oliva");
  ingrediente(0.5, "cucharadita", "comino");
};
```

El código dentro de la función `ingrediente` puede ver la vinculación `factor` de la función externa. Pero sus vinculaciones locales, como `unidad` o `cantidadIngrediente` , no son visibles para la función externa.

En resumen, cada alcance local puede ver también todos los alcances locales que lo contengan. El conjunto de vinculaciones visibles dentro de un bloque está determinado por el lugar de ese bloque en el texto del programa. Cada alcance local puede tambien ver todos los alcances locales que lo contengan, y todos los alcances pueden ver el alcance global. Este enfoque para la visibilidad de vinculaciones es llamado *alcance léxico*.

Funciones como valores

Las vinculaciones de función simplemente actúan como nombres para una pieza específica del programa. Tal vinculación se define una vez y nunca cambia. Esto hace que sea fácil confundir la función con su nombre.

Pero los dos son diferentes. Un valor de función puede hacer todas las cosas que otros valores pueden hacer—puedes usarlo en expresiones arbitrarias, no solo llamarlo. Es posible almacenar un valor de función en una nueva vinculación, pasarla como argumento a una función, y así sucesivamente. Del mismo modo, una vinculación que contenga una función sigue siendo solo una vinculación regular y se le puede asignar un nuevo valor, asi:

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

En el Capitulo 5, discutiremos las cosas interesantes que se pueden hacer al pasar valores de función a otras funciones.

Notación de declaración

Hay una forma ligeramente más corta de crear una vinculación de función. Cuando la palabra clave `function` es usada al comienzo de una declaración, funciona de una manera diferente.

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

Esta es una *declaración* de función. La declaración define la vinculación `cuadrado` y la apunta a la función dada. Esto es un poco mas facil de escribir, y no requiere un punto y coma después de la función.

Hay una sutileza con esta forma de definir una función.

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

function futuro() {
  return "Nunca tendran autos voladores";
```

```
}
```

Este código funciona, aunque la función esté definida *debajo* del código que lo usa. Las declaraciones de funciones no son parte del flujo de control regular de arriba hacia abajo. Estas son conceptualmente trasladadas a la cima de su alcance y pueden ser utilizadas por todo el código en ese alcance. Esto es a veces útil porque nos da la libertad de ordenar el código en una forma que nos parezca significativa, sin preocuparnos por tener que definir todas las funciones antes de que sean utilizadas.

Funciones de flecha

Existe una tercera notación para funciones, que se ve muy diferente de las otras. En lugar de la palabra clave `function`, usa una flecha (`=>`) compuesta de los caracteres igual y mayor que (no debe ser confundida con el operador igual o mayor que, que se escribe `>=`).

```
const potencia = (base, exponente) => {
  let resultado = 1;
  for (let cuenta = 0; cuenta < exponente; cuenta++) {
    resultado *= base;
  }
  return resultado;
};
```

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

Cuando solo haya un solo nombre de parámetro, los paréntesis alrededor de la lista de parámetros pueden ser omitidos. Si el cuerpo es una sola expresión, en lugar de un bloque en llaves, esa expresión será retornada por parte de la función. Asi que estas dos definiciones de `cuadrado` hacen la misma cosa:

```
const cuadrado1 = (x) => { return x * x; };
const cuadrado2 = x => x * x;
```

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

```
const bocina = () => {
```

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

No hay una buena razón para tener ambas funciones de flecha y expresiones `function` en el lenguaje. Aparte de un detalle menor, que discutiremos en Capítulo 6, estas hacen lo mismo. Las funciones de flecha se agregaron en 2015, principalmente para que fuera posible escribir pequeñas expresiones de funciones de una manera menos verbosa. Las usaremos mucho en el Capitulo 5.

La pila de llamadas

La forma en que el control fluye a través de las funciones es algo complicado. Vamos a écharle un vistazo más de cerca. Aquí hay un simple programa que hace unas cuantas llamadas de función:

```
function saludar(quien) {
  console.log("Hola " + quien);
}
saludar("Harry");
console.log("Adios");
```

Un recorrido por este programa es más o menos así: la llamada a `saludar` hace que el control salte al inicio de esa función (línea 2). La función llama a `console.log`, la cual toma el control, hace su trabajo, y entonces retorna el control a la línea 2. Allí llega al final de la función `saludar`, por lo que vuelve al lugar que la llamó, que es la línea 4. La línea que sigue llama a `console.log` nuevamente. Después que esta función retorna, el programa llega a su fin.

Podríamos mostrar el flujo de control esquemáticamente de esta manera:

We could show the flow of control schematically like this:

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

Ya que una función tiene que regresar al lugar donde fue llamada cuando esta

retorna, la computadora debe recordar el contexto de donde sucedió la llamada. En un caso, `console.log` tiene que volver a la función `saludar` cuando está lista. En el otro caso, vuelve al final del programa.

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

Almacenar esta pila requiere espacio en la memoria de la computadora. Cuando la pila crece demasiado grande, la computadora fallará con un mensaje como "fuera de espacio de pila" o "demasiada recursividad". El siguiente código ilustra esto haciendo una pregunta realmente difícil a la computadora, que causara un ir y venir infinito entre las dos funciones. Mejor dicho, *sería* infinito, si la computadora tuviera una pila infinita. Como son las cosas, nos quedaremos sin espacio, o "explotaremos la pila".

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

Argumentos Opcionales

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

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

Definimos `cuadrado` con solo un parámetro. Sin embargo, cuando lo llamamos con tres, el lenguaje no se queja. Este ignora los argumentos extra y calcula el cuadrado del primero.

JavaScript es de extremadamente mente-abierta sobre la cantidad de argumentos que puedes pasar a una función. Si pasa demasiados, los adicionales son ignorados. Si pasas muy pocos, a los parámetros faltantes se les asigna el valor `undefined`.

La desventaja de esto es que es posible—incluso probable—que accidentalmente pases la cantidad incorrecta de argumentos a las funciones. Y nadie te dira nada acerca de eso.

La ventaja es que este comportamiento se puede usar para permitir que una función sea llamada con diferentes cantidades de argumentos. Por ejemplo, esta función `menos` intenta imitar al operador `-` actuando ya sea en uno o dos argumentos

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

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

Si escribes un operador `=` después un parámetro, seguido de una expresión, el valor de esa expresión reemplazará al argumento cuando este no sea dado.

Por ejemplo, esta versión de `potencia` hace que su segundo argumento sea opcional. Si este no es proporcionado o si pasas el valor `undefined`, este se establecerá en dos y la función se comportará como `cuadrado`.

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

console.log(potencia(4));
// → 16
console.log(potencia(2, 6));
// → 64
```

En el próximo capítulo, veremos una forma en el que el cuerpo de una función puede obtener una lista de todos los argumentos que son pasados. Esto es útil porque hace posible que una función acepte cualquier cantidad de argumentos. Por ejemplo, `console.log` hace esto—muetra en la consola todos los valores que se le den.

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

Cierre

La capacidad de tratar a las funciones como valores, combinado con el hecho de que las vinculaciones locales se vuelven a crear cada vez que una sea función es llamada, trae a la luz una pregunta interesante. Qué sucede con las vinculaciones locales cuando la llamada de función que los creó ya no está activa?

El siguiente código muestra un ejemplo de esto. Define una función, `envolverValor` , que crea una vinculación local. Luego retorna una función que accede y devuelve esta vinculación local.

```
function envolverValor(n) {
  let local = n;
  return () => local;
}

let envolver1 = envolverValor(1);
let envolver2 = envolverValor(2);
console.log(envolver1());
// → 1
console.log(envolver2());
// → 2
```

Esto está permitido y funciona como es de esperar—ambas instancias de las vinculaciones todavía pueden ser accedidas. Esta situación es una buena demostración del hecho de que las vinculaciones locales se crean de nuevo para cada llamada, y que las diferentes llamadas no pueden pisotear las distintas vinculaciones locales entre sí.

Esta característica—poder hacer referencia a una instancia específica de una vinculación local en un alcance encerrado—se llama *cierre*. Una función que que hace referencia a vinculaciones de alcances locales alrededor de ella es llamada *un* cierre. Este comportamiento no solo te libera de tener que preocuparte por la duración de las vinculaciones pero también hace posible usar valores de funciones en algunas formas bastante creativas.

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

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

let duplicar = multiplicador(2);
console.log(duplicar(5));
// → 10
```

La vinculación explícita `local` del ejemplo `envolverValor` no es realmente necesaria ya que un parámetro es en sí misma una vinculación local.

Pensar en programas de esta manera requiere algo de práctica. Un buen modelo mental es pensar en los valores de función como que contienen tanto el código en su cuerpo tanto como el entorno en el que se crean. Cuando son llamadas, el cuerpo de la función ve su entorno original, no el entorno en el que se realiza la llamada.

En el ejemplo, se llama a `multiplicador` y esta crea un entorno en el que su parámetro `factor` está ligado a 2. El valor de función que retorna, el cual se almacena en `duplicar`, recuerda este entorno. Asi que cuando es es llamada, multiplica su argumento por 2.

Recursión

Está perfectamente bien que una función se llame a sí misma, siempre que no lo haga tanto que desborde la pila. Una función que se llama a si misma es llamada *recursiva*. La recursión permite que algunas funciones sean escritas en un estilo diferente. Mira, 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 bastante parecido a la forma en la que los matemáticos definen la exponenciación y posiblemente describa el concepto más claramente que la

variante con el ciclo. La función se llama a si misma muchas veces con cada vez exponentes más pequeños para lograr la multiplicación repetida.

Pero esta implementación tiene un problema: en las implementaciones típicas de JavaScript, es aproximadamente 3 veces más lenta que la versión que usa un ciclo. Correr a través de un ciclo simple es generalmente más barato en terminos de memoria que llamar a una función multiples veces.

El dilema de velocidad versus elegancia es interesante. Puedes verlo como una especie de compromiso entre accesibilidad-humana y accesibilidad-maquina. Casi cualquier programa se puede hacer más rápido haciendolo más grande y complicado. El programador tiene que decidir acerca de cual es un equilibrio apropiado.

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

Preocuparse por la eficiencia puede ser una distracción. Es otro factor más que complica el diseño del programa, y cuando estás haciendo algo que ya es difícil, añadir algo más de lo que preocuparse puede ser paralizante.

Por lo tanto, siempre comienza escribiendo algo que sea correcto y fácil de comprender. Si te preocupa que sea demasiado lento—lo que generalmente no sucede, ya que la mayoría del código simplemente no se ejecuta con la suficiente frecuencia como para tomar cantidades significativas de tiempo—puedes medir luego y mejorar si es necesario.

La recursión no siempre es solo una alternativa ineficiente a los ciclos. Algunos problemas son realmente más fáciles de resolver con recursión que con ciclos. En la mayoría de los casos, estos son problemas que requieren explorar o procesar varias "ramas", cada una de las cuales podría ramificarse de nuevo en aún más ramas.

Considera este acertijo: comenzando desde el número 1 y repetidamente agregando 5 o multiplicando por 3, una cantidad infinita de números nuevos pueden ser producidos. ¿Cómo escribirías una función que, dado un número, intente encontrar una secuencia de tales adiciones y multiplicaciones que produzca ese número?

Por ejemplo, se puede llegar al número 13 multiplicando primero por 3 y luego agregando 5 dos veces, mientras que el número 15 no puede ser alcanzado de ninguna manera.

Aquí hay una solución recursiva:

```
function encontrarSolucion(objetivo) {
```

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

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

Ten en cuenta que este programa no necesariamente encuentra la secuencia de operaciones *mas corta*. Este está satisfecho cuando encuentra cualquier secuencia que funcione.

Está bien si no ves cómo funciona el programa de inmediato. Vamos a trabajar a través de él, ya que es un gran ejercicio de pensamiento recursivo.

La función interna `encontrar` es la que hace uso de la recursión real. Esta toma dos argumentos, el número actual y un string que registra cómo se ha alcanzado este número. Si encuentra una solución, devuelve un string que muestra cómo llegar al objetivo. Si no puede encontrar una solución a partir de este número, retorna `null`.

Para hacer esto, la función realiza una de tres acciones. Si el número actual es el número objetivo, la historia actual es una forma de llegar a ese objetivo, por lo que es retornada. Si el número actual es mayor que el objetivo, no tiene sentido seguir explorando esta rama ya que tanto agregar como multiplicar solo hara que el número sea mas grande, por lo que retorna `null`. Y finalmente, si aún estamos por debajo del número objetivo, la función intenta ambos caminos posibles que comienzan desde el número actual llamandose a sí misma dos veces, una para agregar y otra para multiplicar. Si la primera llamada devuelve algo que no es `null`, esta es retornada. De lo contrario, se retorna la segunda llamada, independientemente de si produce un string o el valor `null`.

Para comprender mejor cómo esta función produce el efecto que estamos buscando, veamos todas las llamadas a `encontrar` que se hacen cuando buscamos una solución para el número 13.

```
encontrar(1, "1")
```

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

La indentación indica la profundidad de la pila de llamadas. La primera vez que `encontrar` es llamada, comienza llamandose a sí misma para explorar la solución que comienza con `(1 + 5)`. Esa llamada hara uso de la recursión aún más para explorar *cada* solución continuada que produzca un número menor o igual a el número objetivo. Como no encuentra uno que llegue al objetivo, retorna `null` a la primera llamada. Ahí el operador `||` genera la llamada que explora `(1 * 3)` para que esta suceda. Esta búsqueda tiene más suerte—su primera llamada recursiva, a través de *otra* llamada recursiva, encuentra al número objetivo. Esa llamada más interna retorna un string, y cada uno de los operadores `||` en las llamadas intermedias pasa ese string a lo largo, en última instancia retornando la solución.

Funciones crecientes

Hay dos formas más o menos naturales para que las funciones sean introducidas en los programas.

La primera es que te encuentras escribiendo código muy similar múltiples veces. Preferiríamos no hacer eso. Tener más código significa más espacio para que los errores se oculten y más material que leer para las personas que intenten entender el programa. Entonces tomamos la funcionalidad repetida, buscamos un buen nombre para ella, y la ponemos en una función.

La segunda forma es que encuentres que necesitas alguna funcionalidad que aún no has escrito y parece que merece su propia función. Comenzarás por nombrar a la función y luego escribirás su cuerpo. Incluso podrías comenzar a escribir código que use la función antes de que definas a la función en sí misma.

Que tan difícil te sea encontrar un buen nombre para una función es una buena indicación de cuán claro es el concepto que está tratando de envolver.

Veamos un ejemplo.

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

```
007 Vacas
011 Pollos
```

Esto pide una función de dos argumentos—el numero de vacas y el numero de pollos. Vamos a programar.

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

Escribir `.length` después de una expresión de string nos dará la longitud de dicho string. Por lo tanto, los ciclos `while` seguiran sumando ceros delante del string de numeros hasta que este tenga al menos tres caracteres de longitud.

Misión cumplida! Pero justo cuando estamos por enviar el código a la agricultora (junto con una considerable factura), ella nos llama y nos dice que ella también comenzó a criar cerdos, y que si no podríamos extender el software para imprimir cerdos también?

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

```
function imprimirEtiquetaAlcochadaConCeros(numero, etiqueta) {
  let stringNumero = String(numero);
  while (stringNumero.length < 3) {
    stringNumero = "0" + stringNumero;
  }
```

```
  console.log(`${stringNumero} ${etiqueta}`);
}

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

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

Funciona! Pero ese nombre, `imprimirEtiquetaAlcochadaConCeros`, es un poco incómodo. Combina tres cosas—impresión, alcochar con ceros y añadir una etiqueta—en una sola función.

En lugar de sacar la parte repetida de nuestro programa al por mayor, intentemos elegir un solo *concepto*.

```
function alcocharConCeros(numero, amplitud) {
  let string = String(numero);
  while (string.length < amplitud) {
    string = "0" + string;
  }
  return string;
}

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

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

Una función con un nombre agradable y obvio como `alcocharConCeros` hace que sea más fácil de entender lo que hace para alguien que lee el código. Y tal función es útil en situaciones más alla de este programa en específico. Por ejemplo, podrías usarla para ayudar a imprimir tablas de números en una manera alineada.

Que tan inteligente y versátil *deberia* de ser nuestra función? Podríamos escribir cualquier cosa, desde una función terriblemente simple que solo pueda alcochar un número para que tenga tres caracteres de ancho, a un complicado sistema generalizado de formateo de números que maneje números fraccionar-

ios, números negativos, alineación de puntos decimales, relleno con diferentes caracteres, y así sucesivamente.

Un principio útil es no agregar mucho ingenio a menos que estes absolutamente seguro de que lo vas a necesitar. Puede ser tentador escribir "frameworks" generalizados para cada funcionalidad que encuentres. Resiste ese impulso. No realizarás ningún trabajo real de esta manera—solo estarás escribiendo código que nunca usarás.

Funciones y efectos secundarios

Las funciones se pueden dividir aproximadamente en aquellas que se llaman por su efectos secundarios y aquellas que son llamadas por su valor de retorno. (Aunque definitivamente también es posible tener tanto efectos secundarios como devolver un valor en una misma función.)

La primera función auxiliar en el ejemplo de la granja, `imprimirEtiquetaAlcochadaConCeros`, se llama por su efecto secundario: imprime una línea. La segunda versión, `alcocharConCeros`, se llama por su valor de retorno. 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 las funciones que directamente realizan efectos secundarios.

Una función *pura* es un tipo específico de función de producción-de-valores que no solo no tiene efectos secundarios pero que tampoco depende de los efectos secundarios de otro código—por ejemplo, no lee vinculaciones globales cuyos valores puedan cambiar. Una función pura tiene la propiedad agradable de que cuando se le llama con los mismos argumentos, siempre produce el mismo valor (y no hace nada más). Una llamada a tal función puede ser sustituida por su valor de retorno sin cambiar el significado del código. Cuando no estás seguro de que una función pura esté funcionando correctamente, puedes probarla simplemente llamándola, y saber que si funciona en ese contexto, funcionará en cualquier contexto. Las funciones no puras tienden a requerir más configuración para poder ser probadas.

Aún así, no hay necesidad de sentirse mal cuando escribas funciones que no son puras o de hacer una guerra santa para purgarlas de tu código. Los efectos secundarios a menudo son útiles. No habría forma de escribir una versión pura de `console.log`, por ejemplo, y `console.log` es bueno de tener. Algunas operaciones también son más fáciles de expresar de una manera eficiente cuando usamos efectos secundarios, por lo que la velocidad de computación puede ser una razón para evitar la pureza.

Resumen

Este capítulo te enseñó a escribir tus propias funciones. La palabra clave `function`, cuando se usa como una expresión, puede crear un valor de función. Cuando se usa como una declaración, se puede usar para declarar una vinculación y darle una función como su valor. Las funciones de flecha son otra forma más de crear funciones.

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

// Declara g para ser una función
function g(a, b) {
  return a * b * 3.5;
}

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

Un aspecto clave en para comprender a las funciones es comprender los alcances. Cada bloque crea un nuevo alcance. Los parámetros y vinculaciones declaradas en un determinado alcance son locales y no son visibles desde el exterior. Vinculaciones declaradas con `var` se comportan de manera diferente—terminan en el alcance de la función más cercana o en el alcance global.

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

Ejercicios

Mínimo

El capítulo anterior introdujo la función estándar `Math.min` que devuelve su argumento más pequeño. Nosotros podemos construir algo como eso ahora. Escribe una función `min` que tome dos argumentos y retorne su mínimo.

Recursión

Hemos visto que `%` (el operador de residuo) se puede usar para probar si un número es par o impar usando `% 2` para ver si es divisible entre dos. Aquí hay

otra manera de definir si un número entero positivo es par o impar:

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

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

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

Conteo de frijoles

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

Escribe una función `contarFs` que tome un string como su único argumento y devuelva un número que indica cuántos caracteres "F" en mayúsculas haya en el string.

Despues, escribe una función llamada `contarCaracteres` que se comporte como `contarFs`, excepto que toma un segundo argumento que indica el carácter que debe ser contado (en lugar de contar solo caracteres "F" en mayúscula). Reescribe `contarFs` para que haga uso de esta nueva función.

"En dos ocasiones me han preguntado, 'Dinos, Sr. Babbage, si pones montos equivocadas en la máquina, saldrán las respuestas correctas? [...] No soy capaz de comprender correctamente el tipo de confusión de ideas que podrían provocar tal pregunta."

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

Chapter 4

Estructuras de Datos: Objetos y Arrays

Los números, los booleanos y los strings son los átomos que constituyen las estructuras de datos. Sin embargo, muchos tipos de información requieren más de un átomo. Los *objetos* nos permiten agrupar valores—incluidos otros objetos— para construir estructuras más complejas.

Los programas que hemos construido hasta ahora han estado limitados por el hecho de que estaban operando solo en tipos de datos simples. Este capítulo introducira estructuras de datos básicas. Al final de el, sabrás lo suficiente como para comenzar a escribir programas útiles.

El capítulo trabajara a través de un ejemplo de programación más o menos realista, presentando nuevos conceptos según se apliquen al problema en cuestión. El código de ejemplo a menudo se basara en funciones y vinculaciones que fueron introducidas anteriormente en el texto.

La caja de arena en línea para el libro (*eloquentjavascript.net/code*] proporciona una forma de ejecutar código en el contexto de un capítulo en específico. Si decides trabajar con los ejemplos en otro entorno, asegúrate de primero descargar el código completo de este capítulo desde la página de la caja de arena.

El Hombre Ardilla

De vez en cuando, generalmente entre las ocho y las diez de la noche, Jacques se encuentra a si mismo transformándose en un pequeño roedor peludo con una cola espesa.

Por un lado, Jacques está muy contento de no tener la licantropía clásica. Convertirse en una ardilla causa menos problemas que convertirse en un lobo. En lugar de tener que preocuparse por accidentalmente comerse al vecino (*eso* sería incómodo), le preocupa ser comido por el gato del vecino. Después de dos ocasiones en las que se despertó en una rama precariamente delgada de la copa de un roble, desnudo y desorientado, Jacques se ha dedicado a bloquear las puertas y ventanas de su habitación por la noche y pone algunas nueces en

el piso para mantenerse ocupado.

Eso se ocupa de los problemas del gato y el árbol. Pero Jacques preferiría deshacerse de su condición por completo. Las ocurrencias irregulares de la transformación lo hacen sospechar que estas podrían ser provocadas por algo en especifico. Por un tiempo, creyó que solo sucedia en los días en los que el había estado cerca de árboles de roble. Pero evitar los robles no detuvo el problema.

Cambiando a un enfoque más científico, Jacques ha comenzado a mantener un registro diario de todo lo que hace en un día determinado y si su forma cambio. Con esta información el espera reducir las condiciones que desencadenan las transformaciones.

Lo primero que el necesita es una estructura de datos para almacenar esta información.

Conjuntos de datos

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

Podríamos ponernos creativos con los strings—después de todo, los strings pueden tener cualquier longitud, por lo que podemos poner una gran cantidad de datos en ellos—y usar `"2 3 5 7 11"` como nuestra representación. Pero esto es incómodo. Tendrías que extraer los dígitos de alguna manera y convertirlos a números para acceder a ellos.

Afortunadamente, JavaScript proporciona un tipo de datos específicamente para almacenar secuencias de valores. Es llamado *array* y está escrito como una lista de valores entre corchetes, separados por comas.

```
let listaDeNumeros = [2, 3, 5, 7, 11];
console.log(listaDeNumeros[2]);
// → 5
console.log(listaDeNumeros[0]);
// → 2
console.log(listaDeNumeros[2 - 1]);
// → 3
```

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

El primer índice de un array es cero, no uno. Entonces el primer elemento es alcanzado con `listaDeNumeros[0]`. El conteo basado en cero tiene una larga tradición en el mundo de la tecnología, y en ciertas maneras tiene mucho sentido, pero toma algo de tiempo acostumbrarse. Piensa en el índice como la cantidad de elementos a saltar, contando desde el comienzo del array.

Propiedades

Hasta ahora hemos visto algunas expresiones sospechosas como `miString.length` (para obtener la longitud de un string) y `Math.max` (la función máxima) en capítulos anteriores. Estas son expresiones que acceden a la *propiedad* de algún valor. En el primer caso, accedemos a la propiedad `length` de el valor en `miString`. En el segundo, accedemos a la propiedad llamada `max` en el objeto `Math` (que es una colección de constantes y funciones relacionadas con las matemáticas).

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

```
null.length;
// → TypeError: null has no properties
```

Las dos formas principales de acceder a las propiedades en JavaScript son con un punto y con corchetes. Tanto `valor.x` como `valor[x]` acceden una propiedad en `valor`—pero no necesariamente la misma propiedad. La diferencia está en cómo se interpreta `x`. Cuando se usa un punto, la palabra después del punto es el nombre literal de la propiedad. Cuando usas corchetes, la expresión entre corchetes es *evaluada* para obtener el nombre de la propiedad. Mientras `valor.x` obtiene la propiedad de `valor` llamada "x", `valor[x]` intenta evaluar la expresión `x` y usa el resultado, convertido en un string, como el nombre de la propiedad.

Entonces, si sabes que la propiedad que te interesa se llama *color*, dices `valor.color`. Si quieres extraer la propiedad nombrado por el valor mantenido en la vinculación `i`, dices `valor[i]`. Los nombres de las propiedades son strings. Pueden ser cualquier string, pero la notación de puntos solo funciona con nombres que se vean como nombres de vinculaciones válidos. Entonces, si quieres acceder a una propiedad llamada *2* o *Juan Perez*, debes usar corchetes: `valor[2]` o `valor["Juan Perez"]`.

Los elementos en un array son almacenados como propiedades del array,

usando números como nombres de propiedad. Ya que no puedes usar la notación de puntos con números, y que generalmente quieres utilizar una vinculación que contenga el índice de cualquier manera, debes de usar la notación de corchetes para llegar a ellos.

La propiedad `length` de un array nos dice cuántos elementos este tiene. Este nombre de propiedad es un nombre de vinculación válido, y sabemos su nombre en avance, así que para encontrar la longitud de un array, normalmente escribes `array.length` ya que es más fácil de escribir que `array["length"]`.

Métodos

Ambos objetos de string y array contienen, además de la propiedad `length`, una serie de propiedades que tienen valores de función.

```
let ouch = "Ouch";
console.log(typeof ouch.toUpperCase);
// → function
console.log(ouch.toUpperCase());
// → OUCH
```

Cada string tiene una propiedad `toUpperCase` ("a mayúsculas"). Cuando se llame, regresará una copia del string en la que todas las letras han sido convertido a mayúsculas. También hay `toLowerCase` ("a minúsculas"), que hace lo contrario.

Curiosamente, a pesar de que la llamada a `toUpperCase` no pasa ningún argumento, la función de alguna manera tiene acceso al string `"Ouch"`, el valor de cuya propiedad llamamos. Cómo funciona esto se describe en el Capítulo 6.

Las propiedades que contienen funciones generalmente son llamadas *metodos* del valor al que pertenecen. Como en, "`toUpperCase` es un método de string".

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

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

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

Estos nombres algo tontos son los términos tradicionales para las operaciones en una *pila*. Una pila, en programación, es una estructura de datos que te permite agregar valores a ella y volverlos a sacar en el orden opuesto, de modo que lo que se agregó de último se elimine primero. Estas son comunes en la programación—es posible que recuerdes la pila de llamadas en el capítulo anterior, que es una instancia de la misma idea.

Objetos

De vuelta al Hombre-Ardilla. Un conjunto de entradas diarias puede ser representado como un array. Pero estas entradas no consisten en solo un número o un string—cada entrada necesita almacenar una lista de actividades y un valor booleano que indica si Jacques se convirtió en una ardilla o no. Idealmente, nos gustaría agrupar estos en un solo valor y luego agrupar estos valores en un array de registro de entradas.

Los valores del tipo *objeto* son colecciones arbitrarias de propiedades. Una forma de crear un objeto es mediante el uso de llaves como una expresión.

```
let dia1 = {
  ardilla: false,
  eventos: ["trabajo", "toque un arbol", "pizza", "salir a correr"]
};
console.log(dia1.ardilla);
// → false
console.log(dia1.lobo);
// → undefined
dia1.lobo = false;
console.log(dia1.lobo);
// → false
```

Dentro de las llaves, hay una lista de propiedades separadas por comas. Cada propiedad tiene un nombre seguido de dos puntos y un valor. Cuando un objeto está escrito en varias líneas, indentar como en el ejemplo ayuda con la legibilidad. Las propiedades cuyos nombres no sean nombres válidos de vinculaciones o números válidos deben estar entre comillas.

```
let descripciones = {
  trabajo: "Fui a trabajar",
  "toque un arbol": "Toque un arbol"
```

```
};
```

Esto significa que las llaves tienen *dos* significados en JavaScript. Al comienzo de una declaración, comienzan un bloque de declaraciones. En cualquier otra posición, describen un objeto. Afortunadamente, es raramente útil comenzar una declaración con un objeto en llaves, por lo que la ambigüedad entre estas dos acciones no es un gran problema.

Leer una propiedad que no existe te dará el valor `undefined`.

Es posible asignarle un valor a una expresión de propiedad con un operador `=`. Esto reemplazará el valor de la propiedad si ya tenia uno o crea una nueva propiedad en el objeto si no fuera así.

Para volver brevemente a nuestro modelo de vinculaciones como tentáculos—Las vinculaciones de propiedad son similares. Ellas *agarran* valores, pero otras vinculaciones y propiedades pueden estar agarrando esos mismos valores. Puedes pensar en los objetos como pulpos con cualquier cantidad de tentáculos, cada uno de los cuales tiene un nombre tatuado en él.

El operador `delete` ("eliminar") corta un tentáculo de dicho pulpo. Es un operador unario que, cuando se aplica a la propiedad de un objeto, eliminará la propiedad nombrada de dicho objeto. Esto no es algo que hagas todo el tiempo, pero es posible.

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

El operador binario `in` ("en"), cuando se aplica a un string y un objeto, te dice si ese objeto tiene una propiedad con ese nombre. La diferencia entre darle un valor de `undefined` a una propiedad y eliminarla realmente es que, en el primer caso, el objeto todavía *tiene* la propiedad (solo que no tiene un valor muy interesante), mientras que en el segundo caso la propiedad ya no está presente e `in` retornara `false`.

Para saber qué propiedades tiene un objeto, puedes usar la función `Object.keys`. Le das un objeto y devuelve un array de strings—los nombres de las

propiedades del objeto.

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

Hay una función `Object.assign` que copia todas las propiedades de un objeto a otro.

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

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

Representaremos el diario de Jacques como un array de objetos.

```
let diario = [
  {eventos: ["trabajo", "toque un arbol", "pizza",
             "sali a correr", "television"],
   ardilla: false},
  {eventos: ["trabajo", "helado", "coliflor",
             "lasaña", "toque un arbol", "me cepille los dientes"],
   ardilla: false},
  {eventos: ["fin de semana", "monte la bicicleta", "descanso", "
     nueces",
             "cerveza"],
   ardilla: true},
  /* y asi sucesivamente... */
];
```

Mutabilidad

Llegaremos a la programación real *pronto*. Pero primero, hay una pieza más de teoría por entender.

Vimos que los valores de objeto pueden ser modificados. Los tipos de valores discutidos en capítulos anteriores, como números, strings y booleanos, son todos *inmutables*—es imposible cambiar los valores de aquellos tipos. Puedes

combinarlos y obtener nuevos valores a partir de ellos, pero cuando tomas un valor de string específico, ese valor siempre será el mismo. El texto dentro de él no puede ser cambiado. Si tienes un string que contiene `"gato"`, no es posible que otro código cambie un carácter en tu string para que deletree `"rato"`.

Los objetos funcionan de una manera diferente. Tu *puedes* cambiar sus propiedades, haciendo que un único valor de objeto tenga contenido diferente en diferentes momentos.

Cuando tenemos dos números, 120 y 120, podemos considerarlos el mismo número precisamente, ya sea que hagan referencia o no a los mismos bits físicos. Con los objetos, hay una diferencia entre tener dos referencias a el mismo objeto y tener dos objetos diferentes que contengan las mismas propiedades. Considera el siguiente código:

```
let objeto1 = {valor: 10};
let objeto2 = objeto1;
let objeto3 = {valor: 10};

console.log(objeto1 == objeto2);
// → true
console.log(objeto1 == objeto3);
// → false

objeto1.valor = 15;
console.log(objeto2.valor);
// → 15
console.log(objeto3.valor);
// → 10
```

Las vinculaciones `objeto1` y `objeto2` agarran el *mismo* objeto, que es la razon por la cual cambiar `objeto1` también cambia el valor de `objeto2`. Se dice que tienen la misma *identidad*. La vinculación `objeto3` apunta a un objeto diferente, que inicialmente contiene las mismas propiedades que `objeto1` pero vive una vida separada.

Las vinculaciones también pueden ser cambiables o constantes, pero esto es independiente de la forma en la que se comportan sus valores. Aunque los valores numéricos no cambian, puedes usar una vinculación `let` para hacer un seguimiento de un número que cambia al cambiar el valor al que apunta la vinculación. Del mismo modo, aunque una vinculación `const` a un objeto no pueda ser cambiada en si misma y continuará apuntando al mismo objeto, los *contenidos* de ese objeto pueden cambiar.

```
const puntuacion = {visitantes: 0, locales: 0};
```

```
// Esto esta bien
puntuacion.visitantes = 1;
// Esto no esta permitido
puntuacion = {visitantes: 1, locales: 1};
```

Cuando comparas objetos con el operador `==` en JavaScript, este los compara por identidad: producirá `true` solo si ambos objetos son precisamente el mismo valor. Comparar diferentes objetos retornara `false`, incluso si tienen propiedades idénticas. No hay una operación de comparación "profunda" incorporada en JavaScript, que compare objetos por contenidos, pero es posible que la escribas tu mismo (que es uno de los ejercicios al final de este capítulo).

El diario del licántropo

Asi que Jacques inicia su intérprete de JavaScript y establece el entorno que necesita para mantener su diario.

```
let diario = [];

function añadirEntrada(eventos, ardilla) {
  diario.push({eventos, ardilla});
}
```

Ten en cuenta que el objeto agregado al diario se ve un poco extraño. En lugar de declarar propiedades como `eventos: eventos`, simplemente da un nombre de propiedad. Este es un atajo que representa lo mismo—si el nombre de propiedad en la notación de llaves no es seguido por un valor, su el valor se toma de la vinculación con el mismo nombre.

Entonces, todas las noches a las diez—o algunas veces a la mañana siguiente, después de bajar del estante superior de su biblioteca—Jacques registra el día.

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

```
añadirEntrada(["trabajo", "toque un arbol", "pizza", "sali a correr
    ",
          "television"], false);
añadirEntrada(["trabajo", "helado", "coliflor", "lasaña",
          "toque un arbol", "me cepille los dientes"], false);
añadirEntrada(["fin de semana", "monte la bicicleta", "descanso", "
    nueces",
          "cerveza"], true);
```

Una vez que tiene suficientes puntos de datos, tiene la intención de utilizar estadísticas para encontrar cuál de estos eventos puede estar relacionado con la transformación a ardilla.

La *correlación* es una medida de dependencia entre variables estadísticas. Una variable estadística no es lo mismo que una variable de programación. En las estadísticas, normalmente tienes un conjunto de *medidas*, y cada variable se mide para cada medida. La correlación entre variables generalmente se expresa como un valor que varia de -1 a 1. Una correlación de cero significa que las variables no estan relacionadas. Una correlación de uno indica que las dos están perfectamente relacionadas—si conoces una, también conoces la otra. Uno negativo también significa que las variables están perfectamente relacionadas pero que son opuestas—cuando una es verdadera, la otra es falsa.

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

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

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

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

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}} \tag{4.1}$$

(Si en este momento estas bajando el libro para enfocarte en un terrible flashback a la clase de matemática de 10° grado—espera! No tengo la intención de torturarte con infinitas páginas de notación críptica—solo esta fórmula para ahora. E incluso con esta, todo lo que haremos es convertirla en JavaScript.)

La notación n_{01} indica el número de mediciones donde la primera variable

(ardilla) es falso (0) y la segunda variable (pizza) es verdadera (1). En la tabla de pizza, n_{01} es 9.

El valor $n_{1\bullet}$ se refiere a la suma de todas las medidas donde la primera variable es verdadera, que es 5 en la tabla de ejemplo. Del mismo modo, $n_{\bullet 0}$ se refiere a la suma de las mediciones donde la segunda variable es falsa.

Entonces para la tabla de pizza, la parte arriba de la línea de división (el dividendo) sería $1{\times}76{-}4{\times}9 = 40$, y la parte inferior (el divisor) sería la raíz cuadrada de $5{\times}85{\times}10{\times}80$, o $\sqrt{340000}$. Esto da $\varphi \approx 0.069$, que es muy pequeño. Comer pizza no parece tener influencia en las transformaciones.

Calculando correlación

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

Esta es la función que calcula el coeficiente φ de tal array:

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

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

Esta es una traducción directa de la fórmula φ a JavaScript. `Math.sqrt` es la función de raíz cuadrada, proporcionada por el objeto `Math` en un entorno de JavaScript estándar. Tenemos que sumar dos campos de la tabla para

obtener campos como $n_{1\bullet}$ porque las sumas de filas o columnas no se almacenan directamente en nuestra estructura de datos.

Jacques mantuvo su diario por tres meses. El conjunto de datos resultante está disponible en la caja de arena para este capítulo(*eloquentjavascript.net/code#4*), donde se almacena en la vinculación `JOURNAL`, y en un archivo descargable.

Para extraer una tabla de dos por dos para un evento en específico del diario, debemos hacer un ciclo a traves de todas las entradas y contar cuántas veces ocurre el evento en relación a las transformaciones de ardilla.

```
function tablaPara(evento, diario) {
  let tabla = [0, 0, 0, 0];
  for (let i = 0; i < diario.length; i++) {
    let entrada = diario[i], index = 0;
    if (entrada.eventos.includes(evento)) index += 1;
    if (entrada.ardilla) index += 2;
    tabla[index] += 1;
  }
  return tabla;
}

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

Los array tienen un método `includes` ("incluye") que verifica si un valor dado existe en el array. La función usa eso para determinar si el nombre del evento en el que estamos interesados forma parte de la lista de eventos para un determinado día.

El cuerpo del ciclo en `tablaPara` determina en cual caja de la tabla cae cada entrada del diario al verificar si la entrada contiene el evento específico que nos interesa y si el evento ocurre junto con un incidente de ardilla. El ciclo luego agrega uno a la caja correcta en la tabla.

Ahora tenemos las herramientas que necesitamos para calcular las correlaciónes individuales. El único paso que queda es encontrar una correlación para cada tipo de evento que se escribio en el diario y ver si algo se destaca.

Ciclos de array

En la función `tablaPara`, hay un ciclo como este:

```
for (let i = 0; i < DIARIO.length; i++) {
```

```
  let entrada = DIARIO[i];
  // Hacer con algo con la entrada
}
```

Este tipo de ciclo es común en JavaScript clasico—ir a traves de los arrays un elemento a la vez es algo que surge mucho, y para hacer eso correrias un contador sobre la longitud del array y elegirías cada elemento en turnos.

Hay una forma más simple de escribir tales ciclos en JavaScript moderno.

```
for (let entrada of DIARIO) {
  console.log(`${entrada.eventos.length} eventos.`);
}
```

Cuando un ciclo `for` se vea de esta manera, con la palabra `of` ("de") después de una definición de variable, recorrerá los elementos del valor dado después `of`. Esto funciona no solo para arrays, sino también para strings y algunas otras estructuras de datos. Vamos a discutir *como* funciona en el Capítulo 6.

El análisis final

Necesitamos calcular una correlación para cada tipo de evento que ocurra en el conjunto de datos. Para hacer eso, primero tenemos que *encontrar* cada tipo de evento.

```
function eventosDiario(diario) {
  let eventos = [];
  for (let entrada of diario) {
    for (let evento of entrada.eventos) {
      if (!eventos.includes(evento)) {
        eventos.push(evento);
      }
    }
  }
  return eventos;
}

console.log(eventosDiario(DIARIO));
// → ["zanahoria", "ejercicio", "fin de semana", "pan", …]
```

Yendo a traves de todos los eventos, y agregando aquellos que aún no están en allí en el array `eventos`, la función recolecta cada tipo de evento.

Usando eso, podemos ver todos las correlaciones.

```
for (let evento of eventosDiario(DIARIO)) {
  console.log(evento + ":", phi(tablaPara(evento, DIARIO)));
}
// → zanahoria:      0.0140970969
// → ejercicio:      0.0685994341
// → fin de semana:  0.1371988681
// → pan:           -0.0757554019
// → pudin:         -0.0648203724
// and so on...
```

La mayoría de las correlaciones parecen estar cercanas a cero. Come zanahorias, pan o pudín aparentemente no desencadena la licantropía de ardilla. *Parece* ocurrir un poco más a menudo los fines de semana. Filtremos los resultados para solo mostrar correlaciones mayores que 0.1 o menores que -0.1.

```
for (let evento of eventosDiario(DIARIO)) {
  let correlacion = phi(tablaPara(evento, DIARIO));
  if (correlacion > 0.1 || correlacion < -0.1) {
    console.log(evento + ":", correlacion);
  }
}
// → fin de semana:           0.1371988681
// → me cepille los dientes: -0.3805211953
// → dulces:                  0.1296407447
// → trabajo:                -0.1371988681
// → spaghetti:               0.2425356250
// → leer:                    0.1106828054
// → nueces:                  0.5902679812
```

A-ha! Hay dos factores con una correlación que es claramente más fuerte que las otras. Comer nueces tiene un fuerte efecto positivo en la posibilidad de convertirse en una ardilla, mientras que cepillarse los dientes tiene un significativo efecto negativo.

Interesante. Intentemos algo.

```
for (let entrada of DIARIO) {
  if (entrada.eventos.includes("nueces") &&
     !entrada.eventos.includes("me cepille los dientes")) {
    entrada.eventos.push("dientes con nueces");
  }
}
```

```
console.log(phi(tablaPara("dientes con nueces", DIARIO)));
// → 1
```

Ese es un resultado fuerte. El fenómeno ocurre precisamente cuando Jacques come nueces y no se cepilla los dientes. Si tan solo él no hubiese sido tan flojo con su higiene dental, él nunca habría notado su aflicción.

Sabiendo esto, Jacques deja de comer nueces y descubre que sus transformaciones no vuelven.

Durante algunos años, las cosas van bien para Jacques. Pero en algún momento él pierde su trabajo. Porque vive en un país desagradable donde no tener trabajo significa que no tiene servicios médicos, se ve obligado a trabajar con a circo donde actua como *El Increible Hombre-Ardilla*, llenando su boca con mantequilla de maní antes de cada presentación.

Un día, harto de esta existencia lamentable, Jacques no puede cambiar de vuelta a su forma humana, salta a través de una grieta en la carpa del circo, y se desvanece en el bosque. Nunca se le ve de nuevo.

Arrayología avanzada

Antes de terminar el capítulo, quiero presentarte algunos conceptos extras relacionados a los objetos. Comenzaré introduciendo algunos en métodos de arrays útiles generalmente.

Vimos `push` y `pop`, que agregan y removen elementos en el final de un array, anteriormente en este capítulo. Los métodos correspondientes para agregar y remover cosas en el comienzo de un array se llaman `unshift` y `shift`.

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

Ese programa administra una cola de tareas. Agregas tareas al final de la cola al llamar `recordar("verduras")`, y cuando estés listo para hacer algo, llamas a `obtenerTarea()` para obtener (y eliminar) el elemento frontal de la cola.

La función `recordarUrgentemente` también agrega una tarea pero la agrega al frente en lugar de a la parte posterior de la cola.

Para buscar un valor específico, los arrays proporcionan un método `indexOf` ("indice de"). Este busca a través del array desde el principio hasta el final y retorna el índice en el que se encontró el valor solicitado—o -1 si este no fue encontrado. Para buscar desde el final en lugar del inicio, hay un método similar llamado `lastIndexOf` ("ultimo indice de").

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

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

Otro método fundamental de array es `slice` ("rebanar"), que toma índices de inicio y fin y retorna un array que solo tiene los elementos entre ellos. El índice de inicio es inclusivo, el índice final es exclusivo.

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

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

El método `concat` ("concatenar") se puede usar para unir arrays y asi crear un nuevo array, similar a lo que hace el operador `+` para los strings.

El siguiente ejemplo muestra tanto `concat` como `slice` en acción. Toma un array y un índice, y retorna un nuevo array que es una copia del array original pero eliminando al elemento en el índice dado:

```
function remover(array, indice) {
  return array.slice(0, indice)
    .concat(array.slice(indice + 1));
}
console.log(remover(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

Si a `concat` le pasas un argumento que no es un array, ese valor sera agregado

al nuevo array como si este fuera un array de un solo elemento.

Strings y sus propiedades

Podemos leer propiedades como `length` y `toUpperCase` de valores string. Pero si intentas agregar una nueva propiedad, esta no se mantiene.

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

Los valores de tipo string, número, y Booleano no son objetos, y aunque el lenguaje no se queja si intentas establecer nuevas propiedades en ellos, en realidad no almacena esas propiedades. Como se mencionó antes, tales valores son inmutables y no pueden ser cambiados.

Pero estos tipos tienen propiedades integradas. Cada valor de string tiene un numero de metodos. Algunos muy útiles son `slice` e `indexOf`, que se parecen a los métodos de array de los mismos nombres.

```
console.log("panaderia".slice(0, 3));
// → pan
console.log("panaderia".indexOf("a"));
// → 1
```

Una diferencia es que el `indexOf` de un string puede buscar por un string que contenga más de un carácter, mientras que el método correspondiente al array solo busca por un elemento único.

```
console.log("uno dos tres".indexOf("tres"));
// → 8
```

El método `trim` ("recortar") elimina los espacios en blanco (espacios, saltos de linea, tabulaciones y caracteres similares) del inicio y final de un string.

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

La función `alcocharConCeros` del capítulo anterior también existe como un método. Se llama `padStart` ("alcohar inicio") y toma la longitud deseada y el

carácter de relleno como argumentos.

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

Puedes dividir un string en cada ocurrencia de otro string con el metodo `split` (“dividir”), y unirlo nuevamente con `join` (“unir”).

```
let oracion = "Los pajaros secretarios se especializan en pisotear";
let palabras = oracion.split(" ");
console.log(palabras);
// → ["Los", "pajaros", "secretarios", "se", "especializan", "en", "
   pisotear"]
console.log(palabras.join(". "));
// → Los. pajaros. secretarios. se. especializan. en. pisotear
```

Se puede repetir un string con el método `repeat` (“repetir”), el cual crea un nuevo string que contiene múltiples copias concatenadas del string original.

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

Ya hemos visto la propiedad `length` en los valores de tipo string. Acceder a los caracteres individuales en un string es similar a acceder a los elementos de un array (con una diferencia que discutiremos en el Capítulo 6).

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

Parámetros restantes

Puede ser útil para una función aceptar cualquier cantidad de argumentos. Por ejemplo, `Math.max` calcula el máximo de *todos* los argumentos que le son dados.

Para escribir tal función, pones tres puntos antes del ultimo parámetro de la función, asi:

```
function maximo(...numeros) {
```

```
  let resultado = -Infinity;
  for (let numero of numeros) {
    if (numero > resultado) resultado = numero;
  }
  return resultado;
}
console.log(maximo(4, 1, 9, -2));
// → 9
```

Cuando se llame a una función como esa, el *parámetro restante* está vinculado a un array que contiene todos los argumentos adicionales. Si hay otros parámetros antes que él, sus valores no seran parte de ese array. Cuando, tal como en `maximo`, sea el único parámetro, contendrá todos los argumentos.

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

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

Esto "extiende" al array en la llamada de la función, pasando sus elementos como argumentos separados. Es posible incluir un array de esa manera, junto con otros argumentos, como en `max(9, ...numeros, 2)`.

La notación de corchetes para crear arrays permite al operador de tres-puntos extender otro array en el nuevo array:

Square bracket array notation similarly allows the triple-dot operator to spread another array into the new array:

```
let palabras = ["nunca", "entenderas"];
console.log(["tu", ...palabras, "completamente"]);
// → ["tu", "nunca", "entenderas", "completamente"]
```

El objeto Math

Como hemos visto, `Math` es una bolsa de sorpresas de utilidades relacionadas a los numeros, como `Math.max` (máximo), `Math.min` (mínimo) y `Math.sqrt` (raíz cuadrada).

El objeto `Math` es usado como un contenedor que agrupa un grupo de funcionalidades relacionadas. Solo hay un objeto `Math`, y casi nunca es útil como

un valor. Más bien, proporciona un *espacio de nombre* para que todos estas funciones y valores no tengan que ser vinculaciones globales.

Tener demasiadas vinculaciones globales "contamina" el espacio de nombres. Cuanto más nombres hayan sido tomados, es más probable que accidentalmente sobrescribas el valor de algunas vinculaciones existentes. Por ejemplo, no es es poco probable que quieras nombrar algo `max` en alguno de tus programas. Dado que la función `max` ya incorporada en JavaScript está escondida dentro del Objeto `Math`, no tenemos que preocuparnos por sobrescribirla.

Muchos lenguajes te detendrán, o al menos te advertirán, cuando estes por definir una vinculación con un nombre que ya este tomado. JavaScript hace esto para vinculaciones que hayas declarado con `let` o`const` pero-perversamente-no para vinculaciones estándar, ni para vinculaciones declaradas con `var` o `function`.

De vuelta al objeto `Math`. Si necesitas hacer trigonometría, `Math` te puede ayudar. Contiene `cos` (coseno), `sin` (seno) y `tan` (tangente), así como sus funciones inversas, `acos`, `asin`, y `atan`, respectivamente. El número π (pi)—o al menos la aproximación más cercano que cabe en un número de JavaScript—está disponible como `Math.PI`. Hay una vieja tradición en la programación de escribir los nombres de los valores constantes en mayúsculas.

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

Si los senos y los cosenos son algo con lo que no estas familiarizado, no te preocupes. Cuando se usen en este libro, en el Capítulo 14, te los explicaré.

El ejemplo anterior usó `Math.random`. Esta es una función que retorna un nuevo número pseudoaleatorio entre cero (inclusivo) y uno (exclusivo) cada vez que la llamas.

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

Aunque las computadoras son máquinas deterministas—siempre reaccionan de la misma manera manera dada la misma entrada—es posible hacer que produzcan números que parecen aleatorios. Para hacer eso, la máquina mantiene algun valor escondido, y cada vez que le pidas un nuevo número aleatorio, realiza calculos complicados en este valor oculto para crear un nuevo valor. Esta almacena un nuevo valor y retorna un número derivado de él. De esta manera, puede producir números nuevos y difíciles de predecir de una manera que *parece* aleatoria.

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

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

Multiplicar el número aleatorio por 10 nos da un número mayor que o igual a cero e inferior a 10. Como `Math.floor` redondea hacia abajo, esta expresión producirá, con la misma probabilidad, cualquier número desde 0 hasta 9.

También están las funciones `Math.ceil` (que redondea hacia arriba hasta llegar al número entero mas cercano), `Math.round` (al número entero más cercano), y `Math.abs`, que toma el valor absoluto de un número, lo que significa que niega los valores negativos pero deja los positivos tal y como estan.

Desestructurar

Volvamos a la función `phi` por un momento:

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

Una de las razones por las que esta función es incómoda de leer es que tenemos una vinculación apuntando a nuestro array, pero preferiríamos tener vinculaciones para los *elementos* del array, es decir, `let n00 = tabla[0]` y así sucesivamente. Afortunadamente, hay una forma concisa de hacer esto en JavaScript.

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

Esto también funciona para vinculaciones creadas con `let`, `var`, o `const`. Si sabes que el valor que estas vinculando es un array, puedes usar corchetes para "mirar dentro" del valor, y asi vincular sus contenidos.

Un truco similar funciona para objetos, utilizando llaves en lugar de corchetes.

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

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

JSON

Ya que las propiedades solo agarran su valor, en lugar de contenerlo, los objetos y arrays se almacenan en la memoria de la computadora como secuencias de bits que contienen las *direcciónes*—el lugar en la memoria—de sus contenidos. Asi que un array con otro array dentro de el consiste en (al menos) una región de memoria para el array interno, y otra para el array externo, que contiene (entre otras cosas) un número binario que representa la posición del array interno.

Si deseas guardar datos en un archivo para más tarde, o para enviarlo a otra computadora a través de la red, tienes que convertir de alguna manera estos enredos de direcciones de memoria a una descripción que se puede almacenar o enviar. Supongo, que *podrías* enviar toda la memoria de tu computadora junto con la dirección del valor que te interesa, pero ese no parece el mejor enfoque.

Lo que podemos hacer es *serializar* los datos. Eso significa que son convertidos a una descripción plana. Un formato de serialización popular llamado *JSON* (pronunciado "Jason"), que significa JavaScript Object Notation ("Notación de Objetos JavaScript"). Es ampliamente utilizado como un formato de almacenamiento y comunicación de datos en la Web, incluso en otros lenguajes diferentes a JavaScript.

JSON es similar a la forma en que JavaScript escribe arrays y objetos, con algunas restricciones. Todos los nombres de propiedad deben estar rodeados por

comillas dobles, y solo se permiten expresiones de datos simples—sin llamadas a función, vinculaciones o cualquier otra cosa que involucre computaciones reales. Los comentarios no están permitidos en JSON.

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

```
{
  "ardilla": false,
  "eventos": ["trabajo", "toque un arbol", "pizza", "sali a correr"]
}
```

JavaScript nos da las funciones `JSON.stringify` y `JSON.parse` para convertir datos hacia y desde este formato. El primero toma un valor en JavaScript y retorna un string codificado en JSON. La segunda toma un string como ese y lo convierte al valor que este codifica.

```
let string = JSON.stringify({ardilla: false,
                             eventos: ["fin de semana"]});
console.log(string);
// → {"ardilla":false,"eventos":["fin de semana"]}
console.log(JSON.parse(string).eventos);
// → ["fin de semana"]
```

Resumen

Los objetos y arrays (que son un tipo específico de objeto) proporcionan formas de agrupar varios valores en un solo valor. Conceptualmente, esto nos permite poner un montón de cosas relacionadas en un bolso y correr alredor con el bolso, en lugar de envolver nuestros brazos alrededor de todas las cosas individuales, tratando de aferrarnos a ellas por separado.

La mayoría de los valores en JavaScript tienen propiedades, las excepciones son `null` y `undefined`. Se accede a las propiedades usando `valor.propiedad` o `valor["propiedad"]`. Los objetos tienden a usar nombres para sus propiedades y almacenar más o menos un conjunto fijo de ellos. Los arrays, por el otro lado, generalmente contienen cantidades variables de valores conceptualmente idénticos y usa números (comenzando desde 0) como los nombres de sus propiedades.

Hay *algunas* propiedades con nombre en los arrays, como `length` y un numero de metodos. Los métodos son funciones que viven en propiedades y (por lo general) actuan sobre el valor del que son una propiedad.

Puedes iterar sobre los arrays utilizando un tipo especial de ciclo `for`—`for (let elemento of array)`.

Ejercicios

La suma de un rango

La introducción de este libro aludía a lo siguiente como una buena forma de calcular la suma de un rango de números:

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

Escribe una función `rango` que tome dos argumentos, `inicio` y `final`, y retorne un array que contenga todos los números desde `inicio` hasta (e incluyendo) `final`.

Luego, escribe una función `suma` que tome un array de números y retorne la suma de estos números. Ejecuta el programa de ejemplo y ve si realmente retorna 55.

Como una misión extra, modifica tu función `rango` para tomar un tercer argumento opcional que indique el valor de "paso" utilizado para cuando construyas el array. Si no se da ningún paso, los elementos suben en incrementos de uno, correspondiedo al comportamiento anterior. La llamada a la función `rango(1, 10, 2)` deberia retornar `[1, 3, 5, 7, 9]`. Asegúrate de que también funcione con valores de pasos negativos para que `rango(5, 2, -1)` produzca `[5, 4, 3, 2]`.

Revirtiendo un array

Los arrays tienen un método `reverse` que cambia al array invirtiendo el orden en que aparecen sus elementos. Para este ejercicio, escribe dos funciones, `revertirArray` y `revertirArrayEnSuLugar`. El primero, `revertirArray`, toma un array como argumento y produce un *nuevo* array que tiene los mismos elementos pero en el orden inverso. El segundo, `revertirArrayEnSuLugar`, hace lo que hace el método`reverse`: *modifica* el array dado como argumento invirtiendo sus elementos. Ninguno de los dos puede usar el método `reverse` estándar.

Pensando en las notas acerca de los efectos secundarios y las funciones puras en el capítulo anterior, qué variante esperas que sea útil en más situaciones? Cuál corre más rápido?

Una lista

Los objetos, como conjuntos genéricos de valores, se pueden usar para construir todo tipo de estructuras de datos. Una estructura de datos común es la *lista* (no confundir con un array). Una lista es un conjunto anidado de objetos, con el primer objeto conteniendo una referencia al segundo, el segundo al tercero, y así sucesivamente.

```
let lista = {
  valor: 1,
  resto: {
    valor: 2,
    resto: {
      valor: 3,
      resto: null
    }
  }
};
```

Los objetos resultantes forman una cadena, como esta:

Algo bueno de las listas es que pueden compartir partes de su estructura. Por ejemplo, si creo dos nuevos valores `{valor: 0, resto: lista}` y `{valor: -1, resto: lista}` (con `lista` refiriéndose a la vinculación definida anteriormente), ambos son listas independientes, pero comparten la estructura que conforma sus últimos tres elementos. La lista original también sigue siendo una lista válida de tres elementos.

Escribe una función `arrayAListа` que construya una estructura de lista como el que se muestra arriba cuando se le da `[1, 2, 3]` como argumento. También escribe una función `listaAArray` que produzca un array de una lista. Luego agrega una función de utilidad `preceder`, que tome un elemento y una lista y creé una nueva lista que agrega el elemento al frente de la lista de entrada, y `posicion`, que toma una lista y un número y retorne el elemento en la posición dada en la lista (con cero refiriéndose al primer elemento) o `undefined` cuando no exista tal elemento.

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

Comparación profunda

El operador `==` compara objetos por identidad. Pero a veces preferirias comparar los valores de sus propiedades reales.

Escribe una función `igualdadProfunda` que toma dos valores y retorne `true` solo si tienen el mismo valor o son objetos con las mismas propiedades, donde los valores de las propiedades sean iguales cuando comparadas con una llamada recursiva a `igualdadProfunda`.

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

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

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

—C.A.R. Hoare, 1980 ACM Turing Award Lecture

Chapter 5

Funciones de Orden Superior

Un programa grande es un programa costoso, y no solo por el tiempo que se necesita para construirlo. El tamaño casi siempre involucra complejidad, y la complejidad confunde a los programadores. A su vez, los programadores confundidos, introducen errores en los programas. Un programa grande entonces proporciona de mucho espacio para que estos bugs se oculten, haciéndolos difíciles de encontrar.

Volvamos rapidamente a los dos últimos programas de ejemplo en la introducción. El primero es auto-contenido y solo tiene seis líneas de largo:

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

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

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

Cuál es más probable que contenga un bug?

Si contamos el tamaño de las definiciones de `suma` y `rango`, el segundo programa también es grande—incluso puede que sea más grande que el primero. Pero aún así, argumentaria que es más probable que sea correcto.

Es más probable que sea correcto porque la solución se expresa en un vocabulario que corresponde al problema que se está resolviendo. Sumar un rango de números no se trata acerca de ciclos y contadores. Se trata acerca de rangos y sumas.

Las definiciones de este vocabulario (las funciones `suma` y `rango`) seguirán involucrando ciclos, contadores y otros detalles incidentales. Pero ya que expresan conceptos más simples que el programa como un conjunto, son más

fáciles de realizar correctamente.

Abstracción

En el contexto de la programación, estos tipos de vocabularios suelen ser llamados *abstracciones*. Las abstracciones esconden detalles y nos dan la capacidad de hablar acerca de los problemas a un nivel superior (o más abstracto).

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

> Coloque 1 taza de guisantes secos por persona en un recipiente. Agregue agua hasta que los guisantes esten bien cubiertos. Deje los guisantes en agua durante al menos 12 horas. Saque los guisantes del agua y pongalos en una cacerola para cocinar. Agregue 4 tazas de agua por persona. Cubra la sartén y mantenga los guisantes hirviendo a fuego lento durante dos horas. Tome media cebolla por persona. Cortela en piezas con un cuchillo. Agréguela a los guisantes. Tome un tallo de apio por persona. Cortelo en pedazos con un cuchillo. Agréguelo a los guisantes. Tome una zanahoria por persona. Cortela en pedazos. Con un cuchillo! Agregarla a los guisantes. Cocine por 10 minutos más.

Y la segunda receta:

> Por persona: 1 taza de guisantes secos, media cebolla picada, un tallo de apio y una zanahoria.
>
> Remoje los guisantes durante 12 horas. Cocine a fuego lento durante 2 horas en 4 tazas de agua (por persona). Picar y agregar verduras. Cocine por 10 minutos más.

La segunda es más corta y fácil de interpretar. Pero necesitas entender algunas palabras más relacionadas a la cocina—*remojar*, *cocinar a fuego lento*, *picar*, y, supongo, *verduras*.

Cuando programamos, no podemos confiar en que todas las palabras que necesitaremos estaran esperando por nosotros en el diccionario. Por lo tanto, puedes caer en el patrón de la primera receta—resolviendo los pasos precisos que debe realizar la computadora, uno por uno, ciego a los conceptos de orden superior que estos expresan.

En la programación, es una habilidad útil, darse cuenta cuando estás trabajando en un nivel de abstracción demasiado bajo.

Abstrayendo la repetición

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

Es común que un programa haga algo una determinada cantidad de veces. Puedes escribir un ciclo `for` para eso, de esta manera:

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

Podemos abstraer "hacer algo N veces" como una función? Bueno, es fácil escribir una función que llame a `console.log` N cantidad de veces.

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

Pero, y si queremos hacer algo más que loggear los números? Ya que "hacer algo" se puede representar como una función y que las funciones solo son valores, podemos pasar nuestra acción como un valor de función.

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

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

No es necesario que le pases una función predefinida a `repetir`. A menudo, desearas crear un valor de función al momento en su lugar.

```
let etiquetas = [];
repetir(5, i => {
  etiquetas.push(`Unidad ${i + 1}`);
});
console.log(etiquetas);
```

```
// → ["Unidad 1", "Unidad 2", "Unidad 3", "Unidad 4", "Unidad 5"]
```

Esto está estructurado un poco como un ciclo `for`—primero describe el tipo de ciclo, y luego provee un cuerpo. Sin embargo, el cuerpo ahora está escrito como un valor de función, que está envuelto en el paréntesis de la llamada a `repetir`. Por eso es que tiene que cerrarse con el corchete de cierre *y* paréntesis de cierre. En casos como este ejemplo, donde el cuerpo es una expresión pequeña y única, podrias tambien omitir las llaves y escribir el ciclo en una sola línea.

Funciones de orden superior

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

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

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

Y puedes tener funciones que cambien otras funciones.

```
function ruidosa(funcion) {
  return (...argumentos) => {
    console.log("llamando con", argumentos);
    let resultado = funcion(...argumentos);
    console.log("llamada con", argumentos, ", retorno", resultado);
    return resultado;
  };
}
ruidosa(Math.min)(3, 2, 1);
// → llamando con [3, 2, 1]
```

```
// → llamada con [3, 2, 1] , retorno 1
```

Incluso puedes escribir funciones que proporcionen nuevos tipos de flujo de control.

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

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

Hay un método de array incorporado, `forEach` que proporciona algo como un ciclo `for/of` como una función de orden superior.

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

Conjunto de datos de códigos

Un área donde brillan las funciones de orden superior es en el procesamiento de datos. Para procesar datos, necesitaremos algunos datos reales. Este capítulo usara un conjunto de datos acerca de códigos—sistema de escrituras como Latin, Cirílico, o Arábico.

Recuerdas Unicode del Capítulo 1, el sistema que asigna un número a cada carácter en el lenguaje escrito. La mayoría de estos carácteres están asociados a un código específico. El estandar contiene 140 codigos diferentes—81 de los cuales todavía están en uso hoy, y 59 que son históricos.

Aunque solo puedo leer con fluidez los caracteres en Latin, aprecio el hecho de que las personas estan escribiendo textos en al menos 80 diferentes sistemas de escritura, muchos de los cuales ni siquiera reconocería. Por ejemplo, aquí está una muestra de escritura a mano en Tamil.

இன்னா செய்தாரை ஒறுத்தல் அவர்நாண
நன்னயம் செய்து விடல்.

El conjunto de datos de ejemplo contiene algunos piezas de información acerca de los 140 codigos definidos en Unicode. Este esta disponible en la caja de arena para este capítulo (*eloquentjavascript.net/code#5*) como la vinculación `SCRIPTS`. La vinculación contiene un array de objetos, cada uno de los cuales describe un codigo.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Tal objeto te dice el nombre del codigo, los rangos de Unicode asignados a él, la dirección en la que está escrito, la tiempo de origen (aproximado), si todavía está en uso, y un enlace a más información. La dirección en la que esta escrito puede ser `"ltr"` (left-to-right) para izquierda a derecha, `"rtl"` (right-to-left) para derecha a izquierda (la forma en que se escriben los textos en árabe y en hebreo), o `"ttb"` (top-to-bottom) para de arriba a abajo (como con la escritura de Mongolia).

La propiedad `ranges` contiene un array de rangos de caracteres Unicode, cada uno de los cuales es un array de dos elementos que contiene límites inferior y superior. Se asignan los códigos de caracteres dentro de estos rangos al codigo. El limite más bajo es inclusivo (el código 994 es un carácter Copto) y el límite superior es no-inclusivo (el código 1008 no lo es).

Filtrando arrays

Para encontrar los codigos en el conjunto de datos que todavía están en uso, la siguiente función podría ser útil. Filtra hacia afuera los elementos en un array que no pasen una prueba:

```
function filtrar(array, prueba) {
  let pasaron = [];
  for (let elemento of array) {
```

```
    if (prueba(elemento)) {
      pasaron.push(elemento);
    }
  }
  return pasaron;
}

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

La función usa el argumento llamado `prueba`, un valor de función, para llenar una "brecha" en el cálculo—el proceso de decidir qué elementos recolectar.

Observa cómo la función `filtrar`, en lugar de eliminar elementos del array existente, crea un nuevo array solo con los elementos que pasan la prueba. Esta función es *pura*. No modifica el array que se le es dado.

Al igual que `forEach`, `filtrar` es un método de array estándar, este esta incorporado como `filter`. El ejemplo definió la función solo para mostrar lo que hace internamente. A partir de ahora, la usaremos así en su lugar:

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

Transformando con map

Digamos que tenemos un array de objetos que representan codigos, producidos al filtrar el array `SCRIPTS` de alguna manera. Pero queremos un array de nombres, que es más fácil de inspeccionar

El método `map` ("mapear") transforma un array al aplicar una función a todos sus elementos y construir un nuevo array a partir de los valores retornados. El nuevo array tendrá la misma longitud que el array de entrada, pero su contenido ha sido *mapeado* a una nueva forma en base a la función.

```
function map(array, transformar) {
  let mapeados = [];
  for (let elemento of array) {
    mapeados.push(transformar(elemento));
  }
  return mapeados;
}
```

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

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

Resumiendo con reduce

Otra cosa común que hacer con arrays es calcular un valor único a partir de ellos. Nuestro ejemplo recurrente, sumar una colección de números, es una instancia de esto. Otro ejemplo sería encontrar el codigo con la mayor cantidad de caracteres.

La operación de orden superior que representa este patrón se llama *reduce* ("reducir")—a veces también llamada *fold* ("doblar"). Esta construye un valor al repetidamente tomar un solo elemento del array y combinándolo con el valor actual. Al sumar números, comenzarías con el número cero y, para cada elemento, agregas eso a la suma.

Los parámetros para `reduce` son, además del array, una función de combinación y un valor de inicio. Esta función es un poco menos sencilla que `filter` y `map`, así que mira atentamente:

```
function reduce(array, combinar, inicio) {
  let actual = inicio;
  for (let elemento of array) {
    actual = combinar(actual, elemento);
  }
  return actual;
}

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

El método de array estándar `reduce`, que por supuesto corresponde a esta función tiene una mayor comodidad. Si tu array contiene al menos un elemento, tienes permitido omitir el argumento `inicio`. El método tomará el primer elemento del array como su valor de inicio y comienza a reducir a partir del segundo elemento.

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

Para usar `reduce` (dos veces) para encontrar el codigo con la mayor cantidad de caracteres, podemos escribir algo como esto:

```
function cuentaDeCaracteres(codigo) {
  return codigo.ranges.reduce((cuenta, [desde, hasta]) => {
    return cuenta + (hasta - desde);
  }, 0);
}

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

La función `cuentaDeCaracteres` reduce los rangos asignados a un codigo sumando sus tamaños. Ten en cuenta el uso de la desestructuración en el parámetro lista de la función reductora. La segunda llamada a `reduce` luego usa esto para encontrar el codigo más grande al comparar repetidamente dos scripts y retornando el más grande.

El codigo Han tiene más de 89,000 caracteres asignados en el Estándar Unicode, por lo que es, por mucho, el mayor sistema de escritura en el conjunto de datos. Han es un codigo (a veces) usado para texto chino, japonés y coreano. Esos idiomas comparten muchos caracteres, aunque tienden a escribirlos de manera diferente. El consorcio Unicode (con sede en EE.UU.) decidió tratarlos como un único sistema de escritura para ahorrar códigos de caracteres. Esto se llama *unificación Han* y aún enoja bastante a algunas personas.

Composabilidad

Considera cómo habríamos escrito el ejemplo anterior (encontrar el código más grande) sin funciones de orden superior. El código no es mucho peor.

```
let mayor = null;
for (let codigo of SCRIPTS) {
  if (mayor == null ||
      cuentaDeCaracteres(mayor) < cuentaDeCaracteres(codigo)) {
    mayor = codigo;
  }
}
console.log(mayor);
// → {name: "Han", …}
```

Hay algunos vinculaciones más, y el programa tiene cuatro líneas más. Pero todavía es bastante legible.

Las funciones de orden superior comienzan a brillar cuando necesitas *componer* operaciones. Como ejemplo, vamos a escribir código que encuentre el año de origen promedio para los codigos vivos y muertos en el conjunto de datos.

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

console.log(Math.round(promedio(
  SCRIPTS.filter(codigo => codigo.living).map(codigo => codigo.year)
      )));
// → 1185
console.log(Math.round(promedio(
  SCRIPTS.filter(codigo => !codigo.living).map(codigo => codigo.year
      ))));
// → 209
```

Entonces, los codigos muertos en Unicode son, en promedio, más antiguos que los vivos. Esta no es una estadística terriblemente significativa o sorprendente. Pero espero que aceptes que el código utilizado para calcularlo no es difícil de leer. Puedes verlo como una tubería: comenzamos con todos los codigos, filtramos los vivos (o muertos), tomamos los años de aquellos, los promediamos, y redondeamos el resultado.

Definitivamente también podrías haber escribir este codigo como un gran ciclo.

```
let total = 0, cuenta = 0;
for (let codigo of SCRIPTS) {
  if (codigo.living) {
    total += codigo.year;
    cuenta += 1;
  }
}
console.log(Math.round(total / cuenta));
// → 1185
```

Pero es más difícil de ver qué se está calculando y cómo. Y ya que los resultados intermedios no se representan como valores coherentes, sería mucho más trabajo extraer algo así como `promedio` en una función aparte.

En términos de lo que la computadora realmente está haciendo, estos dos enfoques también son bastante diferentes. El primero creará nuevos arrays al ejecutar `filter` y `map`, mientras que el segundo solo computa algunos números, haciendo menos trabajo. Por lo general, puedes permitirte el enfoque legible, pero si estás procesando arrays enormes, y haciendolo muchas veces, el estilo menos abstracto podría ser mejor debido a la velocidad extra.

Strings y códigos de caracteres

Un uso del conjunto de datos sería averiguar qué código esta usando una pieza de texto. Veamos un programa que hace esto.

Recuerda que cada codigo tiene un array de rangos para los códigos de caracteres asociados a el. Entonces, dado un código de carácter, podríamos usar una función como esta para encontrar el codigo correspondiente (si lo hay):

```
function codigoCaracter(codigo_caracter) {
  for (let codigo of SCRIPTS) {
    if (codigo.ranges.some(([desde, hasta]) => {
      return codigo_caracter >= desde && codigo_caracter < hasta;
    })) {
      return codigo;
    }
  }
  return null;
}

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

El método `some` (“alguno”) es otra función de orden superior. Toma una función de prueba y te dice si esa función retorna verdadero para cualquiera de los elementos en el array.

Pero cómo obtenemos los códigos de los caracteres en un string?

En el Capítulo 1 mencioné que los strings de JavaScript estan codificados como una secuencia de números de 16 bits. Estos se llaman *unidades de código*. Inicialmente se suponía que un código de carácter Unicode encajara dentro de esa unidad (lo que da un poco más de 65,000 caracteres). Cuando quedó claro que esto no seria suficiente, muchas personas se resistieron a la necesidad de usar más memoria por carácter. Para apaciguar estas preocupaciones, UTF-16, el formato utilizado por los strings de JavaScript, fue inventado. Este describe la mayoría de los caracteres mas comunes usando una sola unidad de código de

16 bits, pero usa un par de dos de esas unidades para otros caracteres.

Al dia de hoy UTF-16 generalmente se considera como una mala idea. Parece casi intencionalmente diseñado para invitar a errores. Es fácil escribir programas que pretenden que las unidades de código y caracteres son la misma cosa. Y si tu lenguaje no usa caracteres de dos unidades, esto parecerá funcionar simplemente bien. Pero tan pronto como alguien intente usar dicho programa con algunos menos comunes caracteres chinos, este se rompe. Afortunadamente, con la llegada del emoji, todo el mundo ha empezado a usar caracteres de dos unidades, y la carga de lidiar con tales problemas esta bastante mejor distribuida.

Desafortunadamente, las operaciones obvias con strings de JavaScript, como obtener su longitud a través de la propiedad `length` y acceder a su contenido usando corchetes, trata solo con unidades de código.

```
// Dos caracteres emoji, caballo y zapato
let caballoZapato = "🐎👟";
console.log(caballoZapato.length);
// → 4
console.log(caballoZapato[0]);
// → ((Medio-carácter inválido))
console.log(caballoZapato.charCodeAt(0));
// → 55357 (Código del medio-carácter)
console.log(caballoZapato.codePointAt(0));
// → 128052 (Código real para emoji de caballo)
```

El método `charCodeAt` de JavaScript te da una unidad de código, no un código de carácter completo. El método `codePointAt`, añadido despues, si da un carácter completo de Unicode. Entonces podríamos usarlo para obtener caracteres de un string. Pero el argumento pasado a `codePointAt` sigue siendo un índice en la secuencia de unidades de código. Entonces, para hacer un ciclo a traves de todos los caracteres en un string, todavía tendríamos que lidiar con la cuestión de si un carácter ocupa una o dos unidades de código.

En el capítulo anterior, mencioné que el ciclo `for/of` también se puede usar en strings. Como `codePointAt`, este tipo de ciclo se introdujo en un momento en que las personas eran muy conscientes de los problemas con UTF-16. Cuando lo usas para hacer un ciclo a traves de un string, te da caracteres reales, no unidades de código.

```
let dragonRosa = "🐉🌹";
for (let caracter of dragonRosa) {
  console.log(caracter);
```

```
}
// → 🐉
// → 🌹
```

Si tienes un caracter (que será un string de unidades de uno o dos códigos), puedes usar `codePointAt(0)` para obtener su código.

Reconociendo texto

Tenemos una función `codigoCaracter` y una forma de correctamente hacer un ciclo a traves de caracteres. El siguiente paso sería contar los caracteres que pertenecen a cada codigo. La siguiente abstracción de conteo será útil para eso:

```
function contarPor(elementos, nombreGrupo) {
  let cuentas = [];
  for (let elemento of elementos) {
    let nombre = nombreGrupo(elemento);
    let conocido = cuentas.findIndex(c => c.nombre == nombre);
    if (conocido == -1) {
      cuentas.push({nombre, cuenta: 1});
    } else {
      cuentas[conocido].cuenta++;
    }
  }
  return cuentas;
}

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

La función `contarPor` espera una colección (cualquier cosa con la que podamos hacer un ciclo `for/of`) y una función que calcula un nombre de grupo para un elemento dado. Retorna un array de objetos, cada uno de los cuales nombra un grupo y te dice la cantidad de elementos que se encontraron en ese grupo.

Utiliza otro método de array—`findIndex` ("encontrar index"). Este método es algo así como `indexOf`, pero en lugar de buscar un valor específico, este encuentra el primer valor para el cual la función dada retorna verdadero. Como `indexOf`, retorna -1 cuando no se encuentra dicho elemento.

Usando `contarPor`, podemos escribir la función que nos dice qué codigos se usan en una pieza de texto.

```
function codigosTexto(texto) {
  let codigos = contarPor(texto, caracter => {
    let codigo = codigoCaracter(caracter.codePointAt(0));
    return codigo ? codigo.name : "ninguno";
  }).filter(({name}) => name != "ninguno");

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

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

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

La función primero cuenta los caracteres por nombre, usando `codigoCaracter` para asignarles un nombre, y recurre al string `"ninguno"` para caracteres que no son parte de ningún codigo. La llamada `filter` deja afuera las entrada para `"ninguno"` del array resultante, ya que no estamos interesados en esos caracteres.

Para poder calcular porcentajes, primero necesitamos la cantidad total de caracteres que pertenecen a un codigo, lo que podemos calcular con `reduce`. Si no se encuentran tales caracteres, la función retorna un string específico. De lo contrario, transforma las entradas de conteo en strings legibles con `map` y luego las combina con `join`.

Resumen

Ser capaz de pasar valores de función a otras funciones es un aspecto profundamente útil de JavaScript. Nos permite escribir funciones que modelen calculos con "brechas" en ellas. El código que llama a estas funciones pueden llenar estas brechas al proporcionar valores de función.

Los arrays proporcionan varios métodos útiles de orden superior. Puedes usar `forEach` para recorrer los elementos en un array. El método `filter` retorna un nuevo array que contiene solo los elementos que pasan una función de predicado. Transformar un array al poner cada elemento a través de una función se hace con `map`. Puedes usar `reduce` para combinar todos los elementos en una array a un solo valor. El método `some` prueba si algun elemento coincide con una función de predicado determinada. Y `findIndex` encuentra la posición

del primer elemento que coincide con un predicado.

Ejercicios

Aplanamiento

Use el método `reduce` en combinación con el método `concat` para "aplanar" un array de arrays en un único array que tenga todos los elementos de los arrays originales.

Tu propio ciclo

Escriba una función de orden superior llamada `ciclo` que proporcione algo así como una declaración de ciclo `for`. Esta toma un valor, una función de prueba, una función de actualización y un cuerpo de función. En cada iteración, primero ejecuta la función de prueba en el valor actual del ciclo y se detiene si esta retorna falso. Luego llama al cuerpo de función, dándole el valor actual. Y finalmente, llama a la función de actualización para crear un nuevo valor y comienza desde el principio.

Cuando definas la función, puedes usar un ciclo regular para hacer los ciclos reales.

Cada

De forma análoga al método `some`, los arrays también tienen un método `every` ("cada"). Este retorna true cuando la función dada devuelve verdadero para *cada* elemento en el array. En cierto modo, `some` es una versión del operador `||` que actúa en arrays, y `every` es como el operador `&&`.

Implementa `every` como una función que tome un array y una función predicado como parámetros. Escribe dos versiones, una usando un ciclo y una usando el método `some`.

Dirección de Escritura Dominante

Escriba una función que calcule la dirección de escritura dominante en un string de texto. Recuerde que cada objeto de codigo tiene una propiedad `direction` que puede ser `"ltr"` (de izquierda a derecha), `"rtl"` (de derecha a izquierda), o `"ttb"` (arriba a abajo).

La dirección dominante es la dirección de la mayoría de los caracteres que tienen un código asociado a ellos. Las funciones `codigoCaracter` y `contarPor` definidas anteriormente en el capítulo probablemente seran útiles aquí.

"Un tipo de datos abstracto se realiza al escribir un tipo especial de programa [...] que define el tipo en base a las operaciones que puedan ser realizadas en él."

—Barbara Liskov, Programming with Abstract Data Types

Chapter 6

La Vida Secreta de los Objetos

El Capítulo 4 introdujo los objetos en JavaScript. En la cultura de la programación, tenemos una cosa llamada *programación orientada a objetos*, la cual es un conjunto de técnicas que usan objetos (y conceptos relacionados) como el principio central de la organización del programa.

Aunque nadie realmente está de acuerdo con su definición exacta, la programación orientada a objetos ha contribuido al diseño de muchos lenguajes de programación, incluyendo JavaScript. Este capítulo describirá la forma en la que estas ideas pueden ser aplicadas en JavaScript.

Encapsulación

La idea central en la programación orientada a objetos es dividir a los programas en piezas más pequeñas y hacer que cada pieza sea responsable de gestionar su propio estado.

De esta forma, los conocimientos acerca de como funciona una parte del programa pueden mantenerse *locales* a esa pieza. Alguien trabajando en otra parte del programa no tiene que recordar o ni siquiera tener una idea de ese conocimiento. Cada vez que los detalles locales cambien, solo el código directamente a su alrededor debe ser actualizado.

Las diferentes piezas de un programa como tal, interactúan entre sí a través de *interfaces*, las cuales son conjuntos limitados de funciones y vinculaciones que proporcionan funcionalidades útiles en un nivel más abstracto, ocultando asi su implementación interna.

Tales piezas del programa se modelan usando objetos. Sus interfaces consisten en un conjunto específico de métodos y propiedades. Las propiedades que son parte de la interfaz se llaman *publicas*. Las otras, las cuales no deberian ser tocadas por el código externo , se les llama *privadas*.

Muchos lenguajes proporcionan una forma de distinguir entre propiedades publicas y privadas, y ademas evitarán que el código externo pueda acceder a las privadas por completo. JavaScript, una vez más tomando el enfoque

minimalista, no hace esto. Todavía no, al menos—hay trabajo en camino para agregar esto al lenguaje.

Aunque el lenguaje no tenga esta distinción incorporada, los programadores de JavaScript *estan* usando esta idea con éxito .Típicamente, la interfaz disponible se describe en la documentación o en los comentarios. También es común poner un carácter de guión bajo (`_`) al comienzo de los nombres de las propiedades para indicar que estas propiedades son privadas.

Separar la interfaz de la implementación es una gran idea. Esto usualmente es llamado *encapsulación*.

Métodos

Los métodos no son más que propiedades que tienen valores de función. Este es un método simple:

```
let conejo = {};
conejo.hablar = function(linea) {
  console.log(`El conejo dice '${linea}'`);
};

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

Por lo general, un método debe hacer algo en el objeto con que se llamó. Cuando una función es llamada como un método—buscada como una propiedad y llamada inmediatamente, como en `objeto.metodo()`—la vinculación llamada `this` (“este”) en su cuerpo apunta automáticamente al objeto en la cual fue llamada.

```
function hablar(linea) {
  console.log(`El conejo ${this.tipo} dice '${linea}'`);
}
let conejoBlanco = {tipo: "blanco", hablar};
let conejoHambriento = {tipo: "hambriento", hablar};

conejoBlanco.hablar("Oh mis orejas y bigotes, " +
                    "que tarde se esta haciendo!");
// → El conejo blanco dice 'Oh mis orejas y bigotes, que
//   tarde se esta haciendo!'
conejoHambriento.hablar("Podria comerme una zanahoria ahora mismo.")
    ;
// → El conejo hambriento dice 'Podria comerme una zanahoria ahora
```

```
mismo.'
```

Puedes pensar en `this` como un parámetro extra que es pasado en una manera diferente. Si quieres pasarlo explícitamente, puedes usar el método `call` ("llamar") de una función, que toma el valor de `this` como primer argumento y trata a los argumentos adicionales como parámetros normales.

```
hablar.call(conejoHambriento, "Burp!");
// → El conejo hambriento dice 'Burp!'
```

Como cada función tiene su propia vinculación `this`, cuyo valor depende de la forma en como esta se llama, no puedes hacer referencia al `this` del alcance envolvente en una función regular definida con la palabra clave `function`.

Las funciones de flecha son diferentes—no crean su propia vinculación `this`, pero pueden ver la vinculación`this` del alcance a su alrededor. Por lo tanto, puedes hacer algo como el siguiente código, que hace referencia a `this` desde adentro de una función local:

```
function normalizar() {
  console.log(this.coordinadas.map(n => n / this.length));
}
normalizar.call({coordinadas: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

Si hubieras escrito el argumento para `map` usando la palabra clave `function`, el código no funcionaría.

Prototipos

Observa atentamente.

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

Saqué una propiedad de un objeto vacío. Magia!

Bueno, en realidad no. Simplemente he estado ocultando información acerca de como funcionan los objetos en JavaScript. En adición a su conjunto de

propiedades, la mayoría de los objetos también tienen un *prototipo*. Un prototipo es otro objeto que se utiliza como una reserva de propiedades alternativa. Cuando un objeto recibe una solicitud por una propiedad que este no tiene, se buscará en su prototipo la propiedad, luego en el prototipo del prototipo y asi sucesivamente.

Asi que, quién es el prototipo de ese objeto vacío? Es el gran prototipo ancestral, la entidad detrás de casi todos los objetos, `Object.prototype` ("Objeto.prototipo").

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

Como puedes adivinar, `Object.getPrototypeOf` ("Objeto.obtenerPrototipoDe") retorna el prototipo de un objeto.

Las relaciones prototipo de los objetos en JavaScript forman una estructura en forma de árbol, y en la raíz de esta estructura se encuentra `Object.prototype` . Este proporciona algunos métodos que pueden ser accedidos por todos los objetos, como `toString`, que convierte un objeto en una representación de tipo string.

Muchos objetos no tienen `Object.prototype` directamente como su prototipo, pero en su lugar tienen otro objeto que proporciona un conjunto diferente de propiedades predeterminadas. Las funciones derivan de `Function.prototype`, y los arrays derivan de `Array.prototype`.

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

Tal prototipo de objeto tendrá en si mismo un prototipo, a menudo `Object.prototype`, por lo que aún proporciona indirectamente métodos como `toString`.

Puede usar `Object.create` para crear un objeto con un prototipo especifico.

```
let conejoPrototipo = {
  hablar(linea) {
    console.log(`El conejo ${this.tipo} dice '${linea}'`);
```

```
  }
};
let conejoAsesino = Object.create(conejoPrototipo);
conejoAsesino.tipo = "asesino";
conejoAsesino.hablar("SKREEEE!");
// → El conejo asesino dice 'SKREEEE!'
```

Una propiedad como `hablar(linea)` en una expresión de objeto es un atajo para definir un método. Esta crea una propiedad llamada `hablar` y le da una función como su valor.

El conejo "prototipo" actúa como un contenedor para las propiedades que son compartidas por todos los conejos. Un objeto de conejo individual, como el conejo asesino, contiene propiedades que aplican solo a sí mismo—en este caso su tipo—y deriva propiedades compartidas desde su prototipo.

Clases

El sistema de prototipos en JavaScript se puede interpretar como un enfoque informal de un concepto orientado a objetos llamado *clasees*. Una clase define la forma de un tipo de objeto—qué métodos y propiedades tiene este. Tal objeto es llamado una *instancia* de la clase.

Los prototipos son útiles para definir propiedades en las cuales todas las instancias de una clase compartan el mismo valor, como métodos. Las propiedades que difieren por instancia, como la propiedad `tipo` en nuestros conejos, necesitan almacenarse directamente en los objetos mismos.

Entonces, para crear una instancia de una clase dada, debes crear un objeto que derive del prototipo adecuado, pero *también* debes asegurarte de que, en sí mismo, este objeto tenga las propiedades que las instancias de esta clase se supone que tengan. Esto es lo que una función *constructora* hace.

```
function crearConejo(tipo) {
  let conejo = Object.create(conejoPrototipo);
  conejo.tipo = tipo;
  return conejo;
}
```

JavaScript proporciona una manera de hacer que la definición de este tipo de funciones sea más fácil. Si colocas la palabra clave `new` ("new") delante de una llamada de función, la función sera tratada como un constructor. Esto significa que un objeto con el prototipo adecuado es creado automáticamente,

vinculado a `this` en la función, y retornado al final de la función.

El objeto prototipo utilizado al construir objetos se encuentra al tomar la propiedad `prototype` de la función constructora.

```
function Conejo(tipo) {
  this.tipo = tipo;
}
Conejo.prototype.hablar = function(linea) {
  console.log(`El conejo ${this.tipo} dice '${linea}'`);
};

let conejoRaro = new Conejo("raro");
```

Los constructores (todas las funciones, de hecho) automáticamente obtienen una propiedad llamada `prototype`, que por defecto contiene un objeto simple y vacío, que deriva de `Object.prototype`. Puedes sobrescribirlo con un nuevo objeto si asi quieres. O puedes agregar propiedades al objeto ya existente, como lo hace el ejemplo.

Por convención, los nombres de los constructores tienen la primera letra en mayúscula para que se puedan distinguir fácilmente de otras funciones.

Es importante entender la distinción entre la forma en que un prototipo está asociado con un constructor (a través de su propiedad `prototype`) y la forma en que los objetos *tienen* un prototipo (que se puede encontrar con `Object.getPrototypeOf`). El prototipo real de un constructor es `Function.prototype`, ya que los constructores son funciones. Su *propiedad* `prototype` contiene el prototipo utilizado para las instancias creadas a traves de el.

```
console.log(Object.getPrototypeOf(Conejo) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(conejoRaro) ==
            Conejo.prototype);
// → true
```

Notación de clase

Entonces, las clasees en JavaScript son funciones constructoras con una propiedad prototipo. Así es como funcionan, y hasta 2015, esa era la manera en como tenías que escribirlas. Estos días, tenemos una notación menos incómoda.

```
class Conejo {
  constructor(tipo) {
    this.tipo = tipo;
  }
  hablar(linea) {
    console.log(`El conejo ${this.tipo} dice '${linea}'`);
  }
}

let conejoAsesino = new Conejo("asesino");
let conejoNegro = new Conejo("negro");
```

La palabra clave `class` ("clase") comienza una declaración de clase, que nos permite definir un constructor y un conjunto de métodos, todo en un solo lugar. Cualquier número de métodos se pueden escribir dentro de las llaves de la declaración. El metodo llamado `constructor` es tratado de una manera especial. Este proporciona la función constructora real, que estará vinculada al nombre `Conejo`. Los otros metodos estaran empacados en el prototipo de ese constructor. Por lo tanto, la declaración de clase anterior es equivalente a la definición de constructor en la sección anterior. Solo que se ve mejor.

Actualmente las declaraciones de clase solo permiten que los *metodos*—propiedades que contengan funciones—puedan ser agregados al prototipo. Esto puede ser algo inconveniente para cuando quieras guardar un valor no-funcional allí. La próxima versión del lenguaje probablemente mejore esto. Por ahora, tú puedes crear tales propiedades al manipular directamente el prototipo después de haber definido la clase.

Al igual que `function`, `class` se puede usar tanto en posiciones de declaración como de expresión. Cuando se usa como una expresión, no define una vinculación, pero solo produce el constructor como un valor. Tienes permitido omitir el nombre de clase en una expresión de clase.

```
let objeto = new class { obtenerPalabra() { return "hola"; } };
console.log(objeto.obtenerPalabra());
// → hola
```

Sobreescribiendo propiedades derivadas

Cuando le agregas una propiedad a un objeto, ya sea que esté presente en el prototipo o no, la propiedad es agregada al objeto *en si mismo*. Si ya había una

propiedad con el mismo nombre en el prototipo, esta propiedad ya no afectará al objeto, ya que ahora está oculta detrás de la propiedad del propio objeto.

```
Rabbit.prototype.dientes = "pequeños";
console.log(conejoAsesino.dientes);
// → pequeños
conejoAsesino.dientes = "largos, filosos, y sangrientos";
console.log(conejoAsesino.dientes);
// → largos, filosos, y sangrientos
console.log(conejoNegro.dientes);
// → pequeños
console.log(Rabbit.prototype.dientes);
// → pequeños
```

El siguiente diagrama esboza la situación después de que este código ha sido ejecutado. Los prototipos de `Conejo` y `Object` se encuentran detrás de `conejoAsesino` como una especie de telón de fondo, donde las propiedades que no se encuentren en el objeto en sí mismo puedan ser buscadas.

Sobreescribir propiedades que existen en un prototipo puede ser algo útil que hacer. Como muestra el ejemplo de los dientes de conejo, esto se puede usar para expresar propiedades excepcionales en instancias de una clase más genérica de objetos, dejando que los objetos no-excepcionales tomen un valor estándar desde su prototipo.

También puedes sobreescribir para darle a los prototipos estándar de función y array un método diferente `toString` al del objeto prototipo básico.

```
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 al de una llamada

`.join(",")` en él—pone comas entre los valores del array. Llamar directamente a `Object.prototype.toString` con un array produce un string diferente. Esa función no sabe acerca de los arrays, por lo que simplemente pone la palabra *object* y el nombre del tipo entre corchetes.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Mapas

Vimos a la palabra *map* usada en el capítulo anterior para una operación que transforma una estructura de datos al aplicar una función en sus elementos.

Un *mapa* (sustantivo) es una estructura de datos que asocia valores (las llaves) con otros valores. Por ejemplo, es posible que desees mapear nombres a edades. Es posible usar objetos para esto.

```
let edades = {
  Boris: 39,
  Liang: 22,
  Júlia: 62
};

console.log(`Júlia tiene ${edades["Júlia"]}`);
// → Júlia tiene 62
console.log("Se conoce la edad de Jack?", "Jack" in edades);
// → Se conoce la edad de Jack? false
console.log("Se conoce la edad de toString?", "toString" in edades);
// → Se conoce la edad de toString? true
```

Aquí, los nombres de las propiedades del objeto son los nombres de las personas, y los valores de las propiedades sus edades. Pero ciertamente no incluimos a nadie llamado toString en nuestro mapa. Sin embargo, debido a que los objetos simples se derivan de `Object.prototype`, parece que la propiedad está ahí.

Como tal, usar objetos simples como mapas es peligroso. Hay varias formas posibles de evitar este problema. Primero, es posible crear objetos sin *ningun* prototipo. Si pasas `null` a `Object.create`, el objeto resultante no se derivará de `Object.prototype` y podra ser usado de forma segura como un mapa.

```
console.log("toString" in Object.create(null));
```

```
// → false
```

Los nombres de las propiedades de los objetos deben ser strings. Si necesitas un mapa cuyas claves no puedan ser convertidas fácilmente a strings—como objetos—no puedes usar un objeto como tu mapa.

Afortunadamente, JavaScript viene con una clase llamada `Map` que esta escrita para este propósito exacto. Esta almacena un mapeo y permite cualquier tipo de llaves.

```
let edades = new Map();
edades.set("Boris", 39);
edades.set("Liang", 22);
edades.set("Júlia", 62);

console.log(`Júlia tiene ${edades.get("Júlia")}`);
// → Júlia tiene 62
console.log("Se conoce la edad de Jack?", edades.has("Jack"));
// → Se conoce la edad de Jack? false
console.log(edades.has("toString"));
// → false
```

Los métodos `set` (“establecer”),`get` (“obtener”), y `has` (“tiene”) son parte de la interfaz del objeto `Map`. Escribir una estructura de datos que pueda actualizarse rápidamente y buscar en un gran conjunto de valores no es fácil, pero no tenemos que preocuparnos acerca de eso. Alguien más lo hizo por nosotros, y podemos utilizar esta simple interfaz para usar su trabajo.

Si tienes un objeto simple que necesitas tratar como un mapa por alguna razón, es útil saber que `Object.keys` solo retorna las llaves propias del objeto, no las que estan en el prototipo. Como alternativa al operador `in`, puedes usar el método`hasOwnProperty` (“tienePropiaPropiedad”), el cual ignora el prototipo del objeto.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Polimorfismo

Cuando llamas a la función `String` (que convierte un valor a un string) en un objeto, llamará al método `toString` en ese objeto para tratar de crear un string significativo a partir de el. Mencioné que algunos de los prototipos estándar definen su propia versión de `toString` para que puedan crear un string que contenga información más útil que `"[object Object]"`. También puedes hacer eso tú mismo.

```
Conejo.prototype.toString = function() {
  return `un conejo ${this.tipo}`;
};

console.log(String(conejoNegro));
// → un conejo negro
```

Esta es una instancia simple de una idea poderosa. Cuando un pedazo de código es escrito para funcionar con objetos que tienen una cierta interfaz—en este caso, un método `toString`—cualquier tipo de objeto que soporte esta interfaz se puede conectar al código, y simplemente funcionará.

Esta técnica se llama *polimorfismo*. El código polimórfico puede funcionar con valores de diferentes formas, siempre y cuando soporten la interfaz que este espera.

Mencioné en el Capítulo 4 que un ciclo `for/of` puede recorrer varios tipos de estructuras de datos. Este es otro caso de polimorfismo—tales ciclos esperan que la estructura de datos exponga una interfaz específica, lo que hacen los arrays y strings. Y también puedes agregar esta interfaz a tus propios objetos! Pero antes de que podamos hacer eso, necesitamos saber qué son los símbolos.

Símbolos

Es posible que múltiples interfaces usen el mismo nombre de propiedad para diferentes cosas. Por ejemplo, podría definir una interfaz en la que se suponga que el método `toString` convierte el objeto a una pieza de hilo. No sería posible para un objeto ajustarse a esa interfaz y al uso estándar de `toString`.

Esa sería una mala idea, y este problema no es muy común. La mayoria de los programadores de JavaScript simplemente no piensan en eso. Pero los diseñadores del lenguaje, cuyo *trabajo* es pensar acerca de estas cosas, nos han proporcionado una solución de todos modos.

Cuando afirmé que los nombres de propiedad son strings, eso no fue del todo

preciso. Usualmente lo son, pero también pueden ser *símbolos*. Los símbolos son valores creados con la función `Symbol`. A diferencia de los strings, los símbolos recién creados son únicos—no puedes crear el mismo símbolo dos veces.

```
let simbolo = Symbol("nombre");
console.log(simbolo == Symbol("nombre"));
// → false
Conejo.prototype[simbolo] = 55;
console.log(conejoNegro[simbolo]);
// → 55
```

El string que pases a `Symbol` es incluido cuando lo conviertas a string, y puede hacer que sea más fácil reconocer un símbolo cuando, por ejemplo, lo muestres en la consola. Pero no tiene sentido más allá de eso—múltiples símbolos pueden tener el mismo nombre.

Al ser únicos y utilizables como nombres de propiedad, los símbolos son adecuados para definir interfaces que pueden vivir pacíficamente junto a otras propiedades, sin importar cuáles sean sus nombres.

```
const simboloToString = Symbol("toString");
Array.prototype[simboloToString] = function() {
  return `${this.length} cm de hilo azul`;
};

console.log([1, 2].toString());
// → 1,2
console.log([1, 2][simboloToString]());
// → 2 cm de hilo azul
```

Es posible incluir propiedades de símbolos en expresiones de objetos y clases usando corchetes alrededor del nombre de la propiedad. Eso hace que se evalúe el nombre de la propiedad, al igual que la notación de corchetes para acceder propiedades, lo cual nos permite hacer referencia a una vinculación que contiene el símbolo.

```
let objetoString = {
  [simboloToString]() { return "una cuerda de cañamo"; }
};
console.log(objetoString[simboloToString]());
// → una cuerda de cañamo
```

La interfaz de iterador

Se espera que el objeto dado a un ciclo `for/of` sea *iterable*. Esto significa que tenga un método llamado con el símbolo `Symbol.iterator` (un valor de símbolo definido por el idioma, almacenado como una propiedad de la función `Symbol`).

Cuando sea llamado, ese método debe retornar un objeto que proporcione una segunda interfaz, *iteradora*. Esta es la cosa real que realiza la iteración. Tiene un método `next` ("siguiente") que retorna el siguiente resultado. Ese resultado debería ser un objeto con una propiedad `value` ("valor"), que proporciona el siguiente valor, si hay uno, y una propiedad `done` ("listo") que debería ser cierta cuando no haya más resultados y falso de lo contrario.

Ten en cuenta que los nombres de las propiedades `next`, `value` y `done` son simples strings, no símbolos. Solo `Symbol.iterator`, que probablemente sea agregado a un *monton* de objetos diferentes, es un símbolo real.

Podemos usar directamente esta interfaz nosotros mismos.

```
let iteradorOK = "OK"[Symbol.iterator]();
console.log(iteradorOK.next());
// → {value: "O", done: false}
console.log(iteradorOK.next());
// → {value: "K", done: false}
console.log(iteradorOK.next());
// → {value: undefined, done: true}
```

Implementemos una estructura de datos iterable. Construiremos una clase *matriz*, que actuara como un array bidimensional.

```
class Matriz {
  constructor(ancho, altura, elemento = (x, y) => undefined) {
    this.ancho = ancho;
    this.altura = altura;
    this.contenido = [];

    for (let y = 0; y < altura; y++) {
      for (let x = 0; x < ancho; x++) {
        this.contenido[y * ancho + x] = elemento(x, y);
      }
    }
  }

  obtener(x, y) {
    return this.contenido[y * this.ancho + x];
  }
```

```
  establecer(x, y, valor) {
    this.contenido[y * this.ancho + x] = valor;
  }
}
```

La clase almacena su contenido en un único array de elementos *altura* × *ancho*. Los elementos se almacenan fila por fila, por lo que, por ejemplo, el tercer elemento en la quinta fila es (utilizando indexación basada en cero) almacenado en la posición $4 \times ancho + 2$.

La función constructora toma un ancho, una altura y una función opcional de contenido que se usará para llenar los valores iniciales. Hay métodos `obtener` y `establecer` para recuperar y actualizar elementos en la matriz.

Al hacer un ciclo sobre una matriz, generalmente estás interesado en la posición tanto de los elementos como de los elementos en sí mismos, así que haremos que nuestro iterador produzca objetos con propiedades `x`, `y`, y `value` ("valor").

```
class IteradorMatriz {
  constructor(matriz) {
    this.x = 0;
    this.y = 0;
    this.matriz = matriz;
  }

  next() {
    if (this.y == this.matriz.altura) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matriz.obtener(this.x, this.y)};
    this.x++;
    if (this.x == this.matriz.ancho) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

La clase hace un seguimiento del progreso de iterar sobre una matriz en sus propiedades `x` y `y`. El método `next` ("siguiente") comienza comprobando si la parte inferior de la matriz ha sido alcanzada. Si no es así, *primero* crea el objeto que contiene el valor actual y *luego* actualiza su posición, moviéndose a

la siguiente fila si es necesario.

Configuremos la clase `Matriz` para que sea iterable. A lo largo de este libro, Ocasionalmente usaré la manipulación del prototipo después de los hechos para agregar métodos a clases, para que las piezas individuales de código permanezcan pequeñas y autónomas. En un programa regular, donde no hay necesidad de dividir el código en pedazos pequeños, declararias estos métodos directamente en la clase.

```
Matriz.prototype[Symbol.iterator] = function() {
  return new IteradorMatriz(this);
};
```

Ahora podemos recorrer una matriz con `for/of`.

```
let matriz = new Matriz(2, 2, (x, y) => `valor ${x},${y}`);
for (let {x, y, value} of matriz) {
  console.log(x, y, value);
}
// → 0 0 valor 0,0
// → 1 0 valor 1,0
// → 0 1 valor 0,1
// → 1 1 valor 1,1
```

Getters, setters y estáticos

A menudo, las interfaces consisten principalmente de métodos, pero también está bien incluir propiedades que contengan valores que no sean de función. Por ejemplo, los objetos `Map` tienen una propiedad `size` ("tamaño") que te dice cuántas claves hay almacenanadas en ellos.

Ni siquiera es necesario que dicho objeto calcule y almacene tales propiedades directamente en la instancia. Incluso las propiedades que pueden ser accedidas directamente pueden ocultar una llamada a un método. Tales métodos se llaman *getters*, y se definen escribiendo `get` ("obtener") delante del nombre del método en una expresión de objeto o declaración de clase.

```
let tamañoCambiante = {
  get tamaño() {
    return Math.floor(Math.random() * 100);
  }
};
```

```
console.log(tamañoCambiante.tamaño);
// → 73
console.log(tamañoCambiante.tamaño);
// → 49
```

Cuando alguien lee desde la propiedad `tamaño` de este objeto, el método asociado es llamado. Puedes hacer algo similar cuando se escribe en una propiedad, usando un *setter*.

```
class Temperatura {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(valor) {
    this.celsius = (valor - 32) / 1.8;
  }

  static desdeFahrenheit(valor) {
    return new Temperatura((valor - 32) / 1.8);
  }
}

let temp = new Temperatura(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

La clase `Temperatura` te permite leer y escribir la temperatura ya sea en grados Celsius o grados Fahrenheit, pero internamente solo almacena Celsius y convierte automáticamente a Celsius en el getter y setter `fahrenheit`.

Algunas veces quieres adjuntar algunas propiedades directamente a tu función constructora, en lugar de al prototipo. Tales métodos no tienen acceso a una instancia de clase, pero pueden, por ejemplo, ser utilizados para proporcionar formas adicionales de crear instancias.

Dentro de una declaración de clase, métodos que tienen `static` ("estatico") escrito antes su nombre son almacenados en el constructor. Entonces, la clase `Temperatura` te permite escribir `Temperature.desdeFahrenheit(100)` para crear

una temperatura usando grados Fahrenheit.

Herencia

Algunas matrices son conocidas por ser *simétricas*. Si duplicas una matriz simétrico alrededor de su diagonal de arriba-izquierda a derecha-abajo, esta se mantiene igual. En otras palabras, el valor almacenado en x,y es siempre el mismo al de y,x.

Imagina que necesitamos una estructura de datos como `Matriz` pero que haga cumplir el hecho de que la matriz es y siga siendo simétrica. Podríamos escribirla desde cero, pero eso implicaría repetir algo de código muy similar al que ya hemos escrito.

El sistema de prototipos en JavaScript hace posible crear una *nueva* clase, parecida a la clase anterior, pero con nuevas definiciones para algunas de sus propiedades. El prototipo de la nueva clase deriva del antiguo prototipo, pero agrega una nueva definición para, por ejemplo, el método `set`.

En términos de programación orientada a objetos, esto se llama *herencia*. La nueva clase hereda propiedades y comportamientos de la vieja clase.

```
class MatrizSimetrica extends Matriz {
  constructor(tamaño, elemento = (x, y) => undefined) {
    super(tamaño, tamaño, (x, y) => {
      if (x < y) return elemento(y, x);
      else return elemento(x, y);
    });
  }

  set(x, y, valor) {
    super.set(x, y, valor);
    if (x != y) {
      super.set(y, x, valor);
    }
  }
}

let matriz = new MatrizSimetrica(5, (x, y) => `${x},${y}`);
console.log(matriz.get(2, 3));
// → 3,2
```

El uso de la palabra `extends` indica que esta clase no debe estar basada directamente en el prototipo de `Objeto` predeterminado, pero de alguna otra clase. Esta se llama la *superclase*. La clase derivada es la *subclase*.

Para inicializar una instancia de `MatrizSimetrica`, el constructor llama a su constructor de superclase a través de la palabra clave `super`. Esto es necesario porque si este nuevo objeto se comporta (más o menos) como una `Matriz`, va a necesitar las propiedades de instancia que tienen las matrices. En orden para asegurar que la matriz sea simétrica, el constructor ajusta el método `contenido` para intercambiar las coordenadas de los valores por debajo del diagonal.

El método `set` nuevamente usa `super`, pero esta vez no para llamar al constructor, pero para llamar a un método específico del conjunto de metodos de la superclase. Estamos redefiniendo `set` pero queremos usar el comportamiento original. Ya que `this.set` se refiere al *nuevo* método`set`, llamarlo no funcionaria. Dentro de los métodos de clase, `super` proporciona una forma de llamar a los métodos tal y como se definieron en la superclase.

La herencia nos permite construir tipos de datos ligeramente diferentes a partir de tipos de datos existentes con relativamente poco trabajo. Es una parte fundamental de la tradición orientada a objetos, junto con la encapsulación y el polimorfismo. Pero mientras que los últimos dos son considerados como ideas maravillosas en la actualidad, la herencia es más controversial.

Mientras que la encapsulación y el polimorfismo se pueden usar para *separar* piezas de código entre sí, reduciendo el enredo del programa en general, la herencia fundamentalmente vincula las clases, creando *mas* enredo. Al heredar de una clase, generalmente tienes que saber más sobre cómo funciona que cuando simplemente la usas. La herencia puede ser una herramienta útil, y la uso de vez en cuando en mis propios programas, pero no debería ser la primera herramienta que busques, y probablemente no deberías estar buscando oportunidades para construir jerarquías (árboles genealógicos de clases) de clases en una manera activa.

El operador instanceof

Ocasionalmente es útil saber si un objeto fue derivado de una clase específica. Para esto, JavaScript proporciona un operador binario llamado `instanceof` ("instancia de").

```
console.log(
  new MatrizSimetrica(2) instanceof MatrizSimetrica);
// → true
console.log(new MatrizSimetrica(2) instanceof Matriz);
// → true
console.log(new Matriz(2, 2) instanceof MatrizSimetrica);
// → false
console.log([1] instanceof Array);
```

```
// → true
```

El operador verá a través de los tipos heredados, por lo que una `MatrizSimetrica` es una instancia de `Matriz`. El operador también se puede aplicar a constructores estándar como `Array`. Casi todos los objetos son una instancia de `Object`.

Resumen

Entonces los objetos hacen más que solo tener sus propias propiedades. Ellos tienen prototipos, que son otros objetos. Estos actuarán como si tuvieran propiedades que no tienen mientras su prototipo tenga esa propiedad. Los objetos simples tienen `Object.prototype` como su prototipo.

Los constructores, que son funciones cuyos nombres generalmente comienzan con una mayúscula, se pueden usar con el operador `new` para crear nuevos objetos. El prototipo del nuevo objeto será el objeto encontrado en la propiedad `prototype` del constructor. Puedes hacer un buen uso de esto al poner las propiedades que todos los valores de un tipo dado comparten en su prototipo. Hay una notación de `class` que proporciona una manera clara de definir un constructor y su prototipo.

Puedes definir getters y setters para secretamente llamar a métodos cada vez que se acceda a la propiedad de un objeto. Los métodos estáticos son métodos almacenados en el constructor de clase, en lugar de su prototipo.

El operador `instanceof` puede, dado un objeto y un constructor, decir si ese objeto es una instancia de ese constructor.

Una cosa útil que hacer con los objetos es especificar una interfaz para ellos y decirle a todos que se supone que deben hablar con ese objeto solo a través de esa interfaz. El resto de los detalles que componen tu objeto ahora estan *encapsulados*, escondidos detrás de la interfaz.

Más de un tipo puede implementar la misma interfaz. El código escrito para utilizar una interfaz automáticamente sabe cómo trabajar con cualquier cantidad de objetos diferentes que proporcionen la interfaz. Esto se llama *polimorfismo*.

Al implementar múltiples clases que difieran solo en algunos detalles, puede ser útil escribir las nuevas clases como *subclases* de una clase existente, *heredando* parte de su comportamiento.

Ejercicios

Un tipo vector

Escribe una clase `Vec` que represente un vector en un espacio de dos dimensiones. Toma los parámetros (numericos) `x` y `y`, que debería guardar como propiedades del mismo nombre.

Dale al prototipo de `Vector` dos métodos, `mas` y `menos`, los cuales toman otro vector como parámetro y retornan un nuevo vector que tiene la suma o diferencia de los valores x y y de los dos vectores (`this` y el parámetro).

Agrega una propiedad getter llamada `longitud` al prototipo que calcule la longitud del vector—es decir, la distancia del punto (x, y) desde el origen (0, 0).

Conjuntos

El entorno de JavaScript estándar proporciona otra estructura de datos llamada `Set` ("Conjunto"). Al igual que una instancia de `Map`, un conjunto contiene una colección de valores. Pero a diferencia de `Map`, este no asocia valores con otros—este solo rastrea qué valores son parte del conjunto. Un valor solo puede ser parte de un conjunto una vez—agregarlo de nuevo no tiene ningún efecto.

Escribe una clase llamada `Conjunto`. Como `Set`, debe tener los métodos `add` ("añadir"), `delete` ("eliminar"), y `has` ("tiene"). Su constructor crea un conjunto vacío, `añadir` agrega un valor al conjunto (pero solo si no es ya un miembro), `eliminar` elimina su argumento del conjunto (si era un miembro) y `tiene` retorna un valor booleano que indica si su argumento es un miembro del conjunto.

Usa el operador `===`, o algo equivalente como `indexOf`, para determinar si dos valores son iguales.

Proporcionale a la clase un método estático `desde` que tome un objeto iterable como argumento y cree un grupo que contenga todos los valores producidos al iterar sobre el.

Conjuntos Iterables

Haz iterable la clase `Conjunto` del ejercicio anterior. Puedes remitirte a la sección acerca de la interfaz del iterador anteriormente en el capítulo si ya no recuerdas muy bien la forma exacta de la interfaz.

Si usaste un array para representar a los miembros del conjunto, no solo retornes el iterador creado llamando al método `Symbol.iterator` en el array. Eso funcionaría, pero frustra el propósito de este ejercicio.

Está bien si tu iterador se comporta de manera extraña cuando el conjunto es modificado durante la iteración.

Tomando un método prestado

Anteriormente en el capítulo mencioné que el metodo `hasOwnProperty` de un objeto puede usarse como una alternativa más robusta al operador `in` cuando quieras ignorar las propiedades del prototipo. Pero, ¿y si tu mapa necesita incluir la palabra `"hasOwnProperty"`? Ya no podrás llamar a ese método ya que la propiedad del objeto oculta el valor del método.

¿Puedes pensar en una forma de llamar `hasOwnProperty` en un objeto que tiene una propia propiedad con ese nombre?

"[...] la pregunta de si las Maquinas Pueden Pensar [...] es tan relevante como la pregunta de si los Submarinos Pueden Nadar."

—Edsger Dijkstra, The Threats to Computing Science

Chapter 7

Proyecto: Un Robot

En los capítulos de "proyectos", dejaré de golpearte con teoría nueva por un breve momento y en su lugar vamos a trabajar juntos en un programa. La teoría es necesaria para aprender a programar, pero leer y entender programas reales es igual de importante.

Nuestro proyecto en este capítulo es construir un autómata, un pequeño programa que realiza una tarea en un mundo virtual. Nuestro autómata será un robot de entregas por correo que recoge y deja paquetes.

VillaPradera

El pueblo de VillaPradera no es muy grande. Este consiste de 11 lugares con 14 caminos entre ellos. Puede ser describido con este array de caminos:

```
const caminos = [
  "Casa de Alicia-Casa de Bob",        "Casa de Alicia-Cabaña",
  "Casa de Alicia-Oficina de Correos", "Casa de Bob-Ayuntamiento",
  "Casa de Daria-Casa de Ernie",       "Casa de Daria-Ayuntamiento",
  "Casa de Ernie-Casa de Grete",       "Casa de Grete-Granja",
  "Casa de Grete-Tienda",              "Mercado-Granja",
  "Mercado-Oficina de Correos",        "Mercado-Tienda",
  "Mercado-Ayuntamiento",              "Tienda-Ayuntamiento"
];
```

La red de caminos en el pueblo forma un *grafo*. Un grafo es una colección de puntos (lugares en el pueblo) con líneas entre ellos (caminos). Este grafo será el mundo por el que nuestro robot se movera.

El array de strings no es muy fácil de trabajar. En lo que estamos interesados es en los destinos a los que podemos llegar desde un lugar determinado. Vamos a convertir la lista de caminos en una estructura de datos que, para cada lugar, nos diga a donde se pueda llegar desde allí.

```
function construirGrafo(bordes) {
  let grafo = Object.create(null);
  function añadirBorde(desde, hasta) {
    if (grafo[desde] == null) {
      grafo[desde] = [hasta];
    } else {
      grafo[desde].push(hasta);
    }
  }
  for (let [desde, hasta] of bordes.map(c => c.split("-"))) {
    añadirBorde(desde, hasta);
    añadirBorde(hasta, desde);
  }
  return grafo;
}

const grafoCamino = construirGrafo(roads);
```

Dado un conjunto de bordes, `construirGrafo` crea un objeto de mapa que,

para cada nodo, almacena un array de nodos conectados.

Utiliza el método `split` para ir de los strings de caminos, que tienen la forma `"Comienzo-Final"`, a arrays de dos elementos que contienen el inicio y el final como strings separados.

La tarea

Nuestro robot se moverá por el pueblo. Hay paquetes en varios lugares, cada uno dirigido a otro lugar. El robot tomara paquetes cuando los encuentre y los entregara cuando llegue a sus destinos.

El autómata debe decidir, en cada punto, a dónde ir después. Ha finalizado su tarea cuando se han entregado todos los paquetes.

Para poder simular este proceso, debemos definir un mundo virtual que pueda describirlo. Este modelo nos dice dónde está el robot y dónde estan los paquetes. Cuando el robot ha decidido moverse a alguna parte, necesitamos actualizar el modelo para reflejar la nueva situación.

Si estás pensando en términos de programación orientada a objetos, tu primer impulso podría ser comenzar a definir objetos para los diversos elementos en el mundo. Una clase para el robot, una para un paquete, tal vez una para los lugares. Estas podrían tener propiedades que describen su estado actual, como la pila de paquetes en un lugar, que podríamos cambiar al actualizar el mundo.

Esto está mal.

Al menos, usualmente lo esta. El hecho de que algo suena como un objeto no significa automáticamente que debe ser un objeto en tu programa. Escribir por reflejo las clases para cada concepto en tu aplicación tiende a dejarte con una colección de objetos interconectados donde cada uno tiene su propio estado interno y cambiante. Tales programas a menudo son difíciles de entender y, por lo tanto, fáciles de romper.

En lugar de eso, condensemos el estado del pueblo hasta el mínimo conjunto de valores que lo definan. Está la ubicación actual del robot y la colección de paquetes no entregados, cada uno de los cuales tiene una ubicación actual y una dirección de destino. Eso es todo.

Y mientras estamos en ello, hagámoslo de manera que no *cambiemos* este estado cuándo se mueva el robot, sino calcular un *nuevo* estado para la situación después del movimiento.

```
class EstadoPueblo {
  constructor(lugar, paquetes) {
    this.lugar = lugar;
    this.paquetes = paquetes;
```

```
  }

  mover(destino) {
    if (!grafoCamino[this.lugar].includes(destino)) {
      return this;
    } else {
      let paquetes = this.paquetes.map(p => {
        if (p.lugar != this.lugar) return p;
        return {lugar: destino, direccion: p.direccion};
      }).filter(p => p.lugar != p.direccion);
      return new EstadoPueblo(destino, paquetes);
    }
  }
}
```

En el método `mover` es donde ocurre la acción. Este primero verifica si hay un camino que va del lugar actual al destino, y si no, retorna el estado anterior, ya que este no es un movimiento válido.

Luego crea un nuevo estado con el destino como el nuevo lugar del robot. Pero también necesita crear un nuevo conjunto de paquetes—los paquetes que el robot esta llevando (que están en el lugar actual del robot) necesitan de moverse tambien al nuevo lugar. Y paquetes que están dirigidos al nuevo lugar donde deben de ser entregados—es decir, deben de eliminarse del conjunto de paquetes no entregados. La llamada a `map` se encarga de mover los paquetes, y la llamada a `filter` hace la entrega.

Los objetos de paquete no se modifican cuando se mueven, sino que se vuelven a crear. El método `movee` nos da un nuevo estado de aldea, pero deja el viejo completamente intacto

```
let primero = new EstadoPueblo(
  "Oficina de Correos",
  [{lugar: "Oficina de Correos", direccion: "Casa de Alicia"}]
);
let siguiente = primero.mover("Casa de Alicia");

console.log(siguiente.lugar);
// → Casa de Alicia
console.log(siguiente.parcels);
// → []
console.log(primero.lugar);
// → Oficina de Correos
```

Mover hace que se entregue el paquete, y esto se refleja en el próximo estado. Pero el estado inicial todavía describe la situación donde el robot está en la oficina de correos y el paquete aun no ha sido entregado.

Datos persistentes

Las estructuras de datos que no cambian se llaman *inmutables* o *persistentes*. Se comportan de manera muy similar a los strings y números en que son quienes son, y se quedan así, en lugar de contener diferentes cosas en diferentes momentos.

En JavaScript, casi todo *puede* ser cambiado, así que trabajar con valores que se supone que sean persistentes requieren cierta restricción. Hay una función llamada `Object.freeze` ("Objeto.congelar") que cambia un objeto de manera que escribir en sus propiedades sea ignorado. Podrías usar eso para asegurarte de que tus objetos no cambien, si quieres ser cuidadoso. La congelación requiere que la computadora haga un trabajo extra e ignorar actualizaciones es probable que confunda a alguien tanto como para que hagan lo incorrecto. Por lo general, prefiero simplemente decirle a la gente que un determinado objeto no debe ser molestado, y espero que lo recuerden.

```
let objeto = Object.freeze({valor: 5});
objeto.valor = 10;
console.log(objeto.valor);
// → 5
```

Por qué me salgo de mi camino para no cambiar objetos cuando el lenguaje obviamente está esperando que lo haga?

Porque me ayuda a entender mis programas. Esto es acerca de manejar la complejidad nuevamente. Cuando los objetos en mi sistema son cosas fijas y estables, puedo considerar las operaciones en ellos de forma aislada—moverse a la casa de Alicia desde un estado de inicio siempre produce el mismo nuevo estado. Cuando los objetos cambian con el tiempo, eso agrega una dimensión completamente nueva de complejidad a este tipo de razonamiento.

Para un sistema pequeño como el que estamos construyendo en este capítulo, podriamos manejar ese poco de complejidad adicional. Pero el límite más importante sobre qué tipo de sistemas podemos construir es cuánto podemos entender. Cualquier cosa que haga que tu código sea más fácil de entender hace que sea posible construir un sistema más ambicioso.

Lamentablemente, aunque entender un sistema basado en estructuras de datos persistentes es más fácil, *diseñar* uno, especialmente cuando tu lenguaje

de programación no ayuda, puede ser un poco más difícil. Buscaremos oportunidades para usar estructuras de datos persistentes en este libro, pero también utilizaremos las modificables.

Simulación

Un robot de entregas mira al mundo y decide en qué dirección que quiere moverse. Como tal, podríamos decir que un robot es una función que toma un objeto `EstadoPueblo` y retorna el nombre de un lugar cercano.

Ya que queremos que los robots sean capaces de recordar cosas, para que puedan hacer y ejecutar planes, también les pasamos su memoria y les permitimos retornar una nueva memoria. Por lo tanto, lo que retorna un robot es un objeto que contiene tanto la dirección en la que quiere moverse como un valor de memoria que se le sera regresado la próxima vez que se llame.

```
function correrRobot(estado, robot, memoria) {
  for (let turno = 0;; turno++) {
    if (estado.paquetes.length == 0) {
      console.log(`Listo en ${turno} turnos`);
      break;
    }
    let accion = robot(estado, memoria);
    estado = estado.mover(accion.direccion);
    memoria = accion.memoria;
    console.log(`Moverse a ${accion.direccion}`);
  }
}
```

Considera lo que un robot tiene que hacer para “resolver” un estado dado. Debe recoger todos los paquetes visitando cada ubicación que tenga un paquete, y entregarlos visitando cada lugar al que se dirige un paquete, pero solo después de recoger el paquete.

Cuál es la estrategia más estúpida que podría funcionar? El robot podría simplemente caminar hacia una dirección aleatoria en cada vuelta. Eso significa, con gran probabilidad, que eventualmente se encontrara con todos los paquetes, y luego también en algún momento llegara a todos los lugares donde estos deben ser entregados.

Aqui esta como se podria ver eso:

```
function eleccionAleatoria(array) {
  let eleccion = Math.floor(Math.random() * array.length);
```

```
  return array[eleccion];
}

function robotAleatorio(estado) {
  return {direccion: eleccionAleatoria(grafoCamino[estado.lugar])};
}
```

Recuerda que `Math.random ()` retorna un número entre cero y uno, pero siempre debajo de uno. Multiplicar dicho número por la longitud de un array y luego aplicarle `Math.floor` nos da un índice aleatorio para el array.

Como este robot no necesita recordar nada, ignora su segundo argumento (recuerda que puedes llamar a las funciones en JavaScript con argumentos adicionales sin efectos negativos) y omite la propiedad `memoria` en su objeto retornado.

Para poner en funcionamiento este sofisticado robot, primero necesitaremos una forma de crear un nuevo estado con algunos paquetes. Un método estático (escrito aquí al agregar directamente una propiedad al constructor) es un buen lugar para poner esa funcionalidad.

```
EstadoPueblo.aleatorio = function(numeroDePaquetes = 5) {
  let paquetes = [];
  for (let i = 0; i < numeroDePaquetes; i++) {
    let direccion = eleccionAleatoria(Object.keys(grafoCamino));
    let lugar;
    do {
      lugar = eleccionAleatoria(Object.keys(grafoCamino));
    } while (lugar == direccion);
    paquetes.push({lugar, direccion});
  }
  return new EstadoPueblo("Oficina de Correos", paquetes);
};
```

No queremos paquetes que sean enviados desde el mismo lugar al que están dirigidos. Por esta razón, el bucle `do` sigue seleccionando nuevos lugares cuando obtenga uno que sea igual a la dirección.

Comencemos un mundo virtual.

```
correrRobot(EstadoPueblo.aleatorio(), robotAleatorio);
// → Moverse a Mercado
// → Moverse a Ayuntamiento
// →…
// → Listo en 63 turnos
```

Le toma muchas vueltas al robot para entregar los paquetes, porque este no está planeando muy bien. Nos ocuparemos de eso pronto.

La ruta del camión de correos

Deberíamos poder hacer algo mucho mejor que el robot aleatorio. Una mejora fácil sería tomar una pista de la forma en que como funciona la entrega de correos en el mundo real. Si encontramos una ruta que pasa por todos los lugares en el pueblo, el robot podría ejecutar esa ruta dos veces, y en ese punto esta garantizado que ha entregado todos los paquetes. Aquí hay una de esas rutas (comenzando desde la Oficina de Correos).

```
const rutaCorreo = [
  "Casa de Alicia", "Cabaña", "Casa de Alicia", "Casa de Bob",
  "Ayuntamiento", "Casa de Daria", "Casa de Ernie",
  "GCasa de Grete", "Tienda", "Casa de Grete", "Granja",
  "Mercado", "Oficina de Correos"
];
```

Para implementar el robot que siga la ruta, necesitaremos hacer uso de la memoria del robot. El robot mantiene el resto de su ruta en su memoria y deja caer el primer elemento en cada vuelta.

```
function robotRuta(estado, memoria) {
  if (memoria.length == 0) {
    memoria = rutaCorreo;
  }
  return {direction: memoria[0], memoria: memoria.slice(1)};
}
```

Este robot ya es mucho más rápido. Tomará un máximo de 26 turnos (dos veces la ruta de 13 pasos), pero generalmente seran menos.

Búsqueda de rutas

Aún así, realmente no llamaría seguir ciegamente una ruta fija comportamiento inteligente. El robot podría funcionar más eficientemente si ajustara su comportamiento al trabajo real que necesita hacerse.

Para hacer eso, tiene que ser capaz de avanzar deliberadamente hacia un determinado paquete, o hacia la ubicación donde se debe entregar un paquete. Al hacer eso, incluso cuando el objetivo esté a más de un movimiento de distancia, requiere algún tipo de función de búsqueda de ruta.

El problema de encontrar una ruta a través de un grafo es un típico *problema de búsqueda*. Podemos decir si una solución dada (una ruta) es una solución válida, pero no podemos calcular directamente la solución de la misma manera que podríamos para 2 + 2. En cambio, tenemos que seguir creando soluciones potenciales hasta que encontremos una que funcione.

El número de rutas posibles a través de un grafo es infinito. Pero cuando buscamos una ruta de A a B, solo estamos interesados en aquellas que comienzan en A. Tampoco nos importan las rutas que visitan el mismo lugar dos veces, definitivamente esa no es la ruta más eficiente en cualquier sitio. Entonces eso reduce la cantidad de rutas que el buscador de rutas tiene que considerar.

De hecho, estamos más interesados en la ruta *mas corta*. Entonces queremos asegurarnos de mirar las rutas cortas antes de mirar las más largas. Un buen enfoque sería "crecer" las rutas desde el punto de partida, explorando cada lugar accesible que aún no ha sido visitado, hasta que una ruta alcanze la meta. De esa forma, solo exploraremos las rutas que son potencialmente interesantes, y encontremos la ruta más corta (o una de las rutas más cortas, si hay más de una) a la meta.

Aquí hay una función que hace esto:

```
function encontrarRuta(grafo, desde, hasta) {
  let trabajo = [{donde: desde, ruta: []}];
  for (let i = 0; i < trabajo.length; i++) {
    let {donde, ruta} = trabajo[i];
    for (let lugar of grafo[donde]) {
      if (lugar == hasta) return ruta.concat(lugar);
      if (!trabajo.some(w => w.donde == lugar)) {
        trabajo.push({donde: lugar, ruta: ruta.concat(lugar)});
      }
    }
  }
}
```

La exploración tiene que hacerse en el orden correcto—los lugares que fueron alcanzados primero deben ser explorados primero. No podemos explorar de inmediato un lugar apenas lo alcanzamos, porque eso significaría que los lugares alcanzados *desde allí* también se explorarían de inmediato, y así sucesivamente, incluso aunque puedan haber otros caminos más cortos que aún no han sido

explorados.

Por lo tanto, la función mantiene una *lista de trabajo*. Esta es un array de lugares que deberían explorarse a continuación, junto con la ruta que nos llevó ahí. Esta comienza solo con la posición de inicio y una ruta vacía.

La búsqueda luego opera tomando el siguiente elemento en la lista y explorando eso, lo que significa que todos los caminos que van desde ese lugar son mirados. Si uno de ellos es el objetivo, una ruta final puede ser retornada. De lo contrario, si no hemos visto este lugar antes, un nuevo elemento se agrega a la lista. Si lo hemos visto antes, ya que estamos buscando primero rutas cortas, hemos encontrado una ruta más larga a ese lugar o una precisamente tan larga como la existente, y no necesitamos explorarla.

Puedes imaginar esto visualmente como una red de rutas conocidas que se arrastran desde el lugar de inicio, creciendo uniformemente hacia todos los lados (pero nunca enredándose de vuelta a si misma). Tan pronto como el primer hilo llegue a la ubicación objetivo, ese hilo se remonta al comienzo, dándonos asi nuestra ruta.

Nuestro código no maneja la situación donde no hay más elementos de trabajo en la lista de trabajo, porque sabemos que nuestro gráfico está *conectado*, lo que significa que se puede llegar a todos los lugares desde todos los otros lugares. Siempre podremos encontrar una ruta entre dos puntos, y la búsqueda no puede fallar.

```
function robotOrientadoAMetas({lugar, paquetes}, ruta) {
  if (ruta.length == 0) {
    let paquete = paquetes[0];
    if (paquete.lugar != lugar) {
      ruta = encontrarRuta(grafoCamino, lugar, paquete.lugar);
    } else {
      ruta = encontrarRuta(grafoCamino, lugar, paquete.direccion);
    }
  }
  return {direccion: ruta[0], memoria: ruta.slice(1)};
}
```

Este robot usa su valor de memoria como una lista de instrucciones para moverse, como el robot que sigue la ruta. Siempre que esa lista esté vacía, este tiene que descubrir qué hacer a continuación. Toma el primer paquete no entregado en el conjunto y, si ese paquete no se ha recogido aún, traza una ruta hacia el. Si el paquete *ha* sido recogido, todavía debe ser entregado, por lo que el robot crea una ruta hacia la dirección de entrega en su lugar.

Este robot generalmente termina la tarea de entregar 5 paquetes en 16 turnos

aproximadamente. Un poco mejor que `robotRuta`, pero definitivamente no es óptimo.

Ejercicios

Midiendo un robot

Es difícil comparar objetivamente robots simplemente dejándolos resolver algunos escenarios. Tal vez un robot acaba de conseguir tareas más fáciles, o el tipo de tareas en las que es bueno, mientras que el otro no.

Escribe una función `compararRobots` que toma dos robots (y su memoria de inicio). Debe generar 100 tareas y dejar que cada uno de los robots resuelvan cada una de estas tareas. Cuando terminen, debería generar el promedio de pasos que cada robot tomó por tarea.

En favor de lo que es justo, asegúrate de la misma tarea a ambos robots, en lugar de generar diferentes tareas por robot.

Eficiencia del robot

Puedes escribir un robot que termine la tarea de entrega más rápido que `robotOrientadoAMetas`? Si observas el comportamiento de ese robot, qué obviamente cosas estúpidas este hace? Cómo podrían mejorarse?

Si resolviste el ejercicio anterior, es posible que desees utilizar tu función `compararRobots` para verificar si has mejorado al robot.

Conjunto persistente

La mayoría de las estructuras de datos proporcionadas en un entorno de JavaScript estándar no son muy adecuadas para usos persistentes. Los arrays tienen los métodos `slice` y `concat`, que nos permiten fácilmente crear nuevos arrays sin dañar al anterior. Pero `Set`, por ejemplo, no tiene métodos para crear un nuevo conjunto con un elemento agregado o eliminado.

Escribe una nueva clase `ConjuntoP`, similar a la clase `Conjunto` del Capitulo 6, que almacena un conjunto de valores. Como `Grupo`, tiene métodos `añadir`, `eliminar`, y `tiene`.

Su método `añadir`, sin embargo, debería retornar una *nueva* instancia de `ConjuntoP` con el miembro dado agregado, y dejar la instancia anterior sin cambios. Del mismo modo, `eliminar` crea una nueva instancia sin un miembro dado.

La clase debería funcionar para valores de cualquier tipo, no solo strings. Esta *no* tiene que ser eficiente cuando se usa con grandes cantidades de valores.

El constructor no deberia ser parte de la interfaz de la clase (aunque definitivamente querrás usarlo internamente). En cambio, allí hay una instancia vacía, `ConjuntoP.vacio`, que se puede usar como un valor de inicio.

Por qué solo necesitas un valor `ConjuntoP.vacio`, en lugar de tener una función que crea un nuevo mapa vacío cada vez?

"Arreglar errores es dos veces mas difícil que escribir el código en primer lugar. Por lo tanto, si escribes código de la manera más inteligente posible, eres, por definición, no lo suficientemente inteligente como para depurarlo."

—Brian Kernighan and P.J. Plauger, The Elements of Programming Style

Chapter 8

Bugs y Errores

Los defectos en los programas de computadora usualmente se llaman *bugs* (o "insectos"). Este nombre hace que los programadores se sientan bien al imaginarlos como pequeñas cosas que solo sucede se arrastran hacia nuestro trabajo. En la realidad, por supuesto, nosotros mismos los ponemos allí.

Si un programa es un pensamiento cristalizado, puedes categorizar en grandes rasgos a los bugs en aquellos causados al confundir los pensamientos, y los causados por cometer errores al convertir un pensamiento en código. El primer tipo es generalmente más difícil de diagnosticar y corregir que el último.

Lenguaje

Muchos errores podrían ser señalados automáticamente por la computadora, si esta supiera lo suficiente sobre lo que estamos tratando de hacer. Pero aquí la soltura de JavaScript es un obstáculo. Su concepto de vinculaciones y propiedades es lo suficientemente vago que rara vez atrapará errores ortograficos antes de ejecutar el programa. E incluso entonces, te permite hacer algunas cosas claramente sin sentido, como calcular `true * "mono"`.

Hay algunas cosas de las que JavaScript se queja. Escribir un programa que no siga la gramática del lenguaje inmediatamente hara que la computadora se queje. Otras cosas, como llamar a algo que no sea una función o buscar una propiedad en un valor indefinido, causará un error que sera reportado cuando el programa intente realizar la acción.

Pero a menudo, tu cálculo sin sentido simplemente producirá `NaN` (no es un número) o un valor indefinido. Y el programa continuara felizmente, convencido de que está haciendo algo significativo. El error solo se manifestara más tarde, después de que el valor falso haya viajado a traves de varias funciones. Puede no desencadenar un error en absoluto, pero en silencio causara que la salida del programa sea incorrecta. Encontrar la fuente de tales problemas puede ser algo difícil.

El proceso de encontrar errores—bugs—en los programas se llama *depuración*.

Modo estricto

JavaScript se puede hacer un *poco* más estricto al habilitar el *modo estricto*. Esto se hace al poner el string `"use strict"` ("usar estricto") en la parte superior de un archivo o cuerpo de función. Aquí hay un ejemplo:

```
function puedesDetectarElProblema() {
  "use strict";
  for (contador = 0; contador < 10; contador++) {
    console.log("Feliz feliz");
  }
}

puedesDetectarElProblema();
// → ReferenceError: contador is not defined
```

Normalmente, cuando te olvidas de poner `let` delante de tu vinculación, como con `contador` en el ejemplo, JavaScript silenciosamente crea una vinculación global y utiliza eso. En el modo estricto, se reportara un error en su lugar. Esto es muy útil. Sin embargo, debe tenerse en cuenta que esto no funciona cuando la vinculación en cuestión ya existe como una vinculación global. En ese caso, el ciclo aún sobrescribirá silenciosamente el valor de la vinculación.

Otro cambio en el modo estricto es que la vinculación `this` contiene el valor `undefined` en funciones que no se llamen como métodos. Cuando se hace una llamada fuera del modo estricto, `this` se refiere al objeto del alcance global, que es un objeto cuyas propiedades son vinculaciones globales. Entonces, si llamas accidentalmente a un método o constructor incorrectamente en el modo estricto, JavaScript producirá un error tan pronto trate de leer algo de `this`, en lugar de escribirlo felizmente al alcance global.

Por ejemplo, considera el siguiente código, que llama una función constructora sin la palabra clave `new` de modo que su `this` *no* hara referencia a un objeto recién construido:

```
function Persona(nombre) { this.nombre = nombre; }
let ferdinand = Persona("Ferdinand"); // oops
console.log(nombre);
// → Ferdinand
```

Así que la llamada fraudulenta a `Persona` tuvo éxito pero retorno un valor indefinido y creó la vinculación `nombre` global. En el modo estricto, el resultado es diferente.

```
"use strict";
function Persona(nombre) { this.nombre = nombre; }
let ferdinand = Persona("Ferdinand"); // olvide new
// → TypeError: Cannot set property 'nombre' of undefined
```

Se nos dice inmediatamente que algo está mal. Esto es útil.

Afortunadamente, los constructores creados con la notación `class` siempre se quejan si se llaman sin `new`, lo que hace que esto sea menos un problema incluso en el modo no-estricto.

El modo estricto hace algunas cosas más. No permite darle a una función múltiples parámetros con el mismo nombre y elimina ciertas características problemáticas del lenguaje por completo (como la declaración `with` ("con"), la cual esta tan mal, que no se discute en este libro).

En resumen, poner `"use strict"` en la parte superior de tu programa rara vez duele y puede ayudarte a detectar un problema.

Tipos

Algunos lenguajes quieren saber los tipos de todas tus vinculaciones y expresiones incluso antes de ejecutar un programa. Estos te dirán de una vez cuando uses un tipo de una manera inconsistente. JavaScript solo considera a los tipos cuando ejecuta el programa, e incluso a menudo intentara convertir implícitamente los valores al tipo que espera, por lo que no es de mucha ayuda

Aún así, los tipos proporcionan un marco útil para hablar acerca de los programas. Muchos errores provienen de estar confundido acerca del tipo de valor que entra o sale de una función. Si tienes esa información anotada, es menos probable que te confundas.

Podrías agregar un comentario como arriba de la función `robotOrientadoAMetas` del último capítulo, para describir su tipo.

```
// (EstadoMundo, Array) → {direccion: string, memoria: Array}
function robotOrientadoAMetas(estado, memoria) {
  // ...
}
```

Hay varias convenciones diferentes para anotar programas de JavaScript con tipos.

Una cosa acerca de los tipos es que necesitan introducir su propia complejidad para poder describir suficiente código como para poder ser útil. Cual crees que sería el tipo de la función `eleccionAleatoria` que retorna un elemento aleatorio de un array? Deberías introducir un *tipo variable*, T, que puede representar cualquier tipo, para que puedas darle a `eleccionAleatoria` un tipo como `([T])→T` (función de un array de Ts a a T).

Cuando se conocen los tipos de un programa, es posible que la computadora haga un *chequeo* por ti, señalando los errores antes de que el programa sea ejecutado. Hay varios dialectos de JavaScript que agregan tipos al lenguaje y y los verifica. El más popular se llama TypeScript. Si estás interesado en agregarle más rigor a tus programas, te recomiendo que lo pruebes.

En este libro, continuaremos usando código en JavaScript crudo, peligroso y sin tipos.

Probando

Si el lenguaje no va a hacer mucho para ayudarnos a encontrar errores, tendremos que encontrarlos de la manera difícil: ejecutando el programa y viendo si hace lo correcto.

Hacer esto a mano, una y otra vez, es una muy mala idea. No solo es es molesto, también tiende a ser ineficaz, ya que lleva demasiado tiempo probar exhaustivamente todo cada vez que haces un cambio en tu programa.

Las computadoras son buenas para las tareas repetitivas, y las pruebas son las tareas repetitivas ideales. Las pruebas automatizadas es el proceso de escribir un programa que prueba otro programa. Escribir pruebas consiste en algo más de trabajo que probar manualmente, pero una vez que lo haz hecho, ganas un tipo de superpoder: solo te tomara unos segundos verificar que tu programa todavía se comporta correctamente en todas las situaciones para las que escribiste tu prueba. Cuando rompas algo, lo notarás inmediatamente, en lugar aleatoriomente encontrarte con el problema en algún momento posterior.

Las pruebas usualmente toman la forma de pequeños programas etiquetados que verifican algún aspecto de tu código. Por ejemplo, un conjunto de pruebas para el método (estándar, probablemente ya probado por otra persona) `toUpperCase` podría verse así:

```
function probar(etiqueta, cuerpo) {
  if (!cuerpo()) console.log(`Fallo: ${etiqueta}`);
}
```

```
probar("convertir texto Latino a mayúscula", () => {
  return "hola".toUpperCase() == "HOLA";
});
probar("convertir texto Griego a mayúsculas", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΪΡΕΤΕ";
});
probar("no convierte caracteres sin mayúsculas", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
});
```

Escribir pruebas de esta manera tiende a producir código bastante repetitivo e incómodo. Afortunadamente, existen piezas de software que te ayudan a construir y ejecutar colecciones de pruebas (*suites de prueba*) al proporcionar un lenguaje (en forma de funciones y métodos) adecuado para expresar pruebas y obtener información informativa cuando una prueba falla. Estos generalmente se llaman *corredores de pruebas*.

Algunos programas son más fáciles de probar que otros programas. Por lo general, con cuantos más objetos externos interactúe el código, más difícil es establecer el contexto en el cual probarlo. El estilo de programación mostrado en el capítulo anterior, que usa valores persistentes auto-contenidos en lugar de cambiar objetos, tiende a ser fácil de probar.

Depuración

Una vez que notes que hay algo mal con tu programa porque se comporta mal o produce errores, el siguiente paso es descubir *cual* es el problema.

A veces es obvio. El mensaje de error apuntará a una línea específica de tu programa, y si miras la descripción del error y esa línea de código, a menudo puedes ver el problema.

Pero no siempre. A veces, la línea que provocó el problema es simplemente el primer lugar en donde un valor extraño producido en otro lugar es usado de una manera inválida. Si has estado resolviendo los ejercicios en capítulos anteriores, probablemente ya habrás experimentado tales situaciones.

El siguiente programa de ejemplo intenta convertir un número entero a un string en una base dada (decimal, binario, etc.) al repetidamente seleccionar el último dígito y luego dividiendo el número para deshacerse de este dígito. Pero la extraña salida que produce sugiere que tiene un error.

```
function numeroAString(n, base = 10) {
```

```
  let resultado = "", signo = "";
  if (n < 0) {
    signo = "-";
    n = -n;
  }
  do {
    resultado = String(n % base) + resultado;
    n /= base;
  } while (n > 0);
  return signo + resultado;
}
console.log(numeroAString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e…-3181.3
```

Incluso si ya ves el problema, finge por un momento que no lo has hecho. Sabemos que nuestro programa no funciona bien, y queremos encontrar por qué.

Aquí es donde debes resistir el impulso de comenzar a hacer cambios aleatorios en el código para ver si eso lo mejora. En cambio, *piensa*. Analiza lo que está sucediendo y piensa en una teoría de por qué podría ser sucediendo. Luego, haz observaciones adicionales para probar esta teoría—o si aún no tienes una teoría, haz observaciones adicionales para ayudarte a que se te ocurra una.

Poner algunas llamadas estratégicas a `console.log` en el programa es una buena forma de obtener información adicional sobre lo que está haciendo el programa. En en este caso, queremos que `n` tome los valores `13`, `1` y luego `0`. Vamos a escribir su valor al comienzo del ciclo.

```
13
1.3
0.13
0.013…

1.5e-323
```

Exacto. Dividir 13 entre 10 no produce un número entero. En lugar de `n /= base`, lo que realmente queremos es `n = Math.floor(n / base)` para que el número sea correctamente "desplazado" hacia la derecha.

Una alternativa al uso de `console.log` para echarle un vistazo al comportamiento del programa es usar las capacidades del *depurador* de tu navegador. Los navegadores vienen con la capacidad de establecer un *punto de interrupción* en una línea específico de tu código. Cuando la ejecución del programa

alcanza una línea con un punto de interrupción, este entra en pausa, y puedes inspeccionar los valores de las vinculaciones en ese punto. No entraré en detalles, ya que los depuradores difieren de navegador en navegador, pero mira las herramientas de desarrollador en tu navegador o busca en la Web para obtener más información.

Otra forma de establecer un punto de interrupción es incluir una declaración `debugger` (que consiste simplemente de esa palabra clave) en tu programa. Si las herramientas de desarrollador en tu navegador están activas, el programa pausará cada vez que llegue a tal declaración.

Propagación de errores

Desafortunadamente, no todos los problemas pueden ser prevenidos por el programador. Si tu programa se comunica con el mundo exterior de alguna manera, es posible obtener una entrada malformada, sobrecargarse con el trabajo, o la red falle en la ejecución.

Si solo estás programando para ti mismo, puedes permitirte ignorar tales problemas hasta que estos ocurran. Pero si construyes algo que va a ser utilizado por cualquier otra persona, generalmente quieres que el programa haga algo mejor que solo estrellarse. A veces lo correcto es tomar la mala entrada en zancada y continuar corriendo. En otros casos, es mejor informar al usuario lo que salió mal y luego darse por vencido. Pero en cualquier situación, el programa tiene que hacer algo activamente en respuesta al problema.

Supongamos que tienes una función `pedirEntero` que le pide al usuario un número entero y lo retorna. Qué deberías retornar si la entrada por parte del usuario es "naranja"?

Una opción es hacer que retorne un valor especial. Opciones comunes para tales valores son `null`, `undefined`, o -1.

```
function pedirEntero(pregunta) {
  let resultado = Number(prompt(pregunta));
  if (Number.isNaN(resultado)) return null;
  else return resultado;
}

console.log(pedirEntero("Cuantos arboles ves?"));
```

Ahora cualquier código que llame a `pedirEntero` debe verificar si un número real fue leído y, si eso falla, de alguna manera debe recuperarse—tal vez preguntando nuevamente o usando un valor predeterminado. O podría de nuevo

retornar un valor especial a *su* llamada para indicar que no pudo hacer lo que se pidió.

En muchas situaciones, principalmente cuando los errores son comunes y la persona que llama debe tenerlos explícitamente en cuenta, retornar un valor especial es una buena forma de indicar un error. Sin embargo, esto tiene sus desventajas. Primero, qué pasa si la función puede retornar cada tipo de valor posible? En tal función, tendrás que hacer algo como envolver el resultado en un objeto para poder distinguir el éxito del fracaso.

```
function ultimoElemento(array) {
  if (array.length == 0) {
    return {fallo: true};
  } else {
    return {elemento: array[array.length - 1]};
  }
}
```

El segundo problema con retornar valores especiales es que puede conducir a código muy incómodo. Si un fragmento de código llama a `pedirEntero` 10 veces, tiene que comprobar 10 veces si `null` fue retornado. Y si su respuesta a encontrar `null` es simplemente retornar `null` en sí mismo, los llamadores de esa función a su vez tendrán que verificarlo, y así sucesivamente.

Excepciones

Cuando una función no puede continuar normalmente, lo que nos *gustaría* hacer es simplemente detener lo que estamos haciendo e inmediatamente saltar a un lugar que sepa cómo manejar el problema. Esto es lo que el *manejo de excepciones* hace.

Las excepciones son un mecanismo que hace posible que el código que se encuentre con un problema *produzca* (o *lance*) una excepción. Una excepción puede ser cualquier valor. Producir una se asemeja a un retorno súper-cargado de una función: salta no solo de la función actual sino también fuera de sus llamadores, todo el camino hasta la primera llamada que comenzó la ejecución actual. Esto se llama *desenrollando la pila*. Puede que recuerdes que la pila de llamadas de función fue mencionada en el Capítulo 3. Una excepción se aleja de esta pila, descartando todos los contextos de llamadas que encuentra.

Si las excepciones siempre se acercaran al final de la pila, estas no serían de mucha utilidad. Simplemente proporcionarían una nueva forma de explotar tu programa. Su poder reside en el hecho de que puedes establecer

"obstáculos" a lo largo de la pila para *capturar* la excepción, cuando esta esta se dirige hacia abajo. Una vez que hayas capturado una excepción, puedes hacer algo con ella para abordar el problema y luego continuar ejecutando el programa.

Aquí hay un ejemplo:

```
function pedirDireccion(pregunta) {
  let resultado = prompt(pregunta);
  if (resultado.toLowerCase() == "izquierda") return "I";
  if (resultado.toLowerCase() == "derecha") return "D";
  throw new Error("Dirección invalida: " + resultado);
}

function mirar() {
  if (pedirDireccion("Hacia que dirección quieres ir?") == "I") {
    return "una casa";
  } else {
    return "dos osos furiosos";
  }
}

try {
  console.log("Tu ves", mirar());
} catch (error) {
  console.log("Algo incorrecto sucedio: " + error);
}
```

La palabra clave `throw` ("producir") se usa para generar una excepción. La captura de una se hace al envolver un fragmento de código en un bloque `try` ("intentar"), seguido de la palabra clave `catch` ("atrapar"). Cuando el código en el bloque `try` cause una excepción para ser producida, se evalúa el bloque `catch`, con el nombre en paréntesis vinculado al valor de la excepción. Después de que el bloque `catch` finaliza, o si el bloque `try` finaliza sin problemas, el programa procede debajo de toda la declaración `try/catch`.

En este caso, usamos el constructor `Error` para crear nuestro valor de excepción. Este es un constructor (estándar) de JavaScript que crea un objeto con una propiedad `message` ("mensaje"). En la mayoría de los entornos de JavaScript, las instancias de este constructor también recopilan información sobre la pila de llamadas que existía cuando se creó la excepción, algo llamado *seguimiento de la pila*. Esta información se almacena en la propiedad `stack` ("pila") y puede ser útil al intentar depurar un problema: esta nos dice la función donde ocurrió el problema y qué funciones realizaron la llamada fallida.

Ten en cuenta que la función `mirar` ignora por completo la posibilidad de que `pedirDireccion` podría salir mal. Esta es la gran ventaja de las excepciones: el código de manejo de errores es necesario solamente en el punto donde el error ocurre y en el punto donde se maneja. Las funciones en el medio puede olvidarse de todo.

Bueno, casi...

Limpiando después de excepciones

El efecto de una excepción es otro tipo de flujo de control. Cada acción que podría causar una excepción, que es prácticamente cualquier llamada de función y acceso a propiedades, puede causar al control dejar tu codigo repentinamente.

Eso significa que cuando el código tiene varios efectos secundarios, incluso si parece que el flujo de control "regular" siempre sucederá, una excepción puede evitar que algunos de ellos sucedan.

Aquí hay un código bancario realmente malo.

```
const cuentas = {
  a: 100,
  b: 0,
  c: 20
};

function obtenerCuenta() {
  let nombreCuenta = prompt("Ingrese el nombre de la cuenta");
  if (!cuentas.hasOwnProperty(nombreCuenta)) {
    throw new Error(`La cuenta "${nombreCuenta}" no existe`);
  }
  return nombreCuenta;
}

function transferir(desde, cantidad) {
  if (cuentas[desde] < cantidad) return;
  cuentas[desde] -= cantidad;
  cuentas[obtenerCuenta()] += cantidad;
}
```

La función `transferir` transfiere una suma de dinero desde una determinada cuenta a otra, pidiendo el nombre de la otra cuenta en el proceso. Si se le da un nombre de cuenta no válido, `obtenerCuenta` arroja una excepción.

Pero `transferir` *primero* remueve el dinero de la cuenta, y *luego* llama a `obtenerCuenta` antes de añadirlo a la otra cuenta. Si esto es interrumpido por

una excepción en ese momento, solo hará que el dinero desaparezca.

Ese código podría haber sido escrito de una manera un poco más inteligente, por ejemplo al llamar `obtenerCuenta` antes de que se comience a mover el dinero. Pero a menudo problemas como este ocurren de maneras más sutiles. Incluso funciones que no parece que lanzarán una excepción podría hacerlo en circunstancias excepcionales o cuando contienen un error de programador.

Una forma de abordar esto es usar menos efectos secundarios. De nuevo, un estilo de programación que calcula nuevos valores en lugar de cambiar los datos existentes ayuda. Si un fragmento de código deja de ejecutarse en el medio de crear un nuevo valor, nadie ve el valor a medio terminar, y no hay ningún problema.

Pero eso no siempre es práctico. Entonces, hay otra característica que las declaraciones `try` tienen. Estas pueden ser seguidas por un bloque `finally` ("finalmente") en lugar de o además de un bloque `catch`. Un bloque `finally` dice "no importa lo que pase, ejecuta este código después de intentar ejecutar el código en el bloque `try`."

```
function transferir(desde, cantidad) {
  if (cuentas[desde] < cantidad) return;
  let progreso = 0;
  try {
    cuentas[desde] -= cantidad;
    progreso = 1;
    cuentas[obtenerCuenta()] += cantidad;
    progreso = 2;
  } finally {
    if (progreso == 1) {
      cuentas[desde] += cantidad;
    }
  }
}
```

Esta versión de la función rastrea su progreso, y si, cuando este terminando, se da cuenta de que fue abortada en un punto donde habia creado un estado de programa inconsistente, repara el daño que hizo.

Ten en cuenta que, aunque el código `finally` se ejecuta cuando una excepción deja el bloque `try`, no interfiere con la excepción. Después de que se ejecuta el bloque `finally`, la pila continúa desenrollandose.

Escribir programas que funcionan de manera confiable incluso cuando aparecen excepciones en lugares inesperados es muy difícil. Muchas personas simplemente no se molestan, y porque las excepciones suelen reservarse para cir-

cunstancias excepcionales, el problema puede ocurrir tan raramente que nunca siquiera es notado. Si eso es algo bueno o algo realmente malo depende de cuánto daño hará el software cuando falle.

Captura selectiva

Cuando una excepción llega hasta el final de la pila sin ser capturada, esta es manejada por el entorno. Lo que esto significa difiere entre los entornos. En los navegadores, una descripción del error generalmente sera escrita en la consola de JavaScript (accesible a través de las herramientas de desarrollador del navegador). Node.js, el entorno de JavaScript sin navegador que discutiremos en el Capítulo 20, es más cuidadoso con la corrupción de datos. Aborta todo el proceso cuando ocurre una excepción no manejada.

Para los errores de programador, solo dejar pasar al error es a menudo lo mejor que puedes hacer. Una excepción no manejada es una forma razonable de señalizar un programa roto, y la consola de JavaScript, en los navegadores moderno, te proporcionan cierta información acerca de qué llamdas de función estaban en la pila cuando ocurrió el problema.

Para problemas que se *espera* que sucedan durante el uso rutinario, estrellarse con una excepción no manejada es una estrategia terrible.

Usos inválidos del lenguaje, como hacer referencia a vinculaciones inexistentes, buscar una propiedad en `null`, o llamar a algo que no sea una función, también dará como resultado que se levanten excepciones. Tales excepciones también pueden ser atrapadas.

Cuando se ingresa en un cuerpo `catch`, todo lo que sabemos es que *algo* en nuestro cuerpo `try` provocó una excepción. Pero no sabemos *que*, o *cual* excepción este causó.

JavaScript (en una omisión bastante evidente) no proporciona soporte directo para la captura selectiva de excepciones: o las atrapas todas o no atrapas nada. Esto hace que sea tentador *asumir* que la excepción que obtienes es en la que estabas pensando cuando escribiste el bloque `catch`.

Pero puede que no. Alguna otra suposición podría ser violada, o es posible que hayas introducido un error que está causando una excepción. Aquí está un ejemplo que *intenta* seguir llamando `pedirDireccion` hasta que obtenga una respuesta válida:

```
for (;;) {
  try {
    let direccion = peirDirrecion("Donde?"); // ← error tipografico!
    console.log("Tu elegiste ", direccion);
```

```
    break;
  } catch (e) {
    console.log ("No es una dirección válida. Inténtalo de nuevo");
  }
}
```

El constructo `for (;;)` es una forma de crear intencionalmente un ciclo que no termine por si mismo. Salimos del ciclo solamente una cuando dirección válida sea dada. *Pero* escribimos mal `pedirDireccion`, lo que dará como resultado un error de "variable indefinida". Ya que el bloque `catch` ignora por completo su valor de excepción (`e`), suponiendo que sabe cuál es el problema, trata erróneamente al error de vinculación como indicador de una mala entrada. Esto no solo causa un ciclo infinito, también "entierra" el útil mensaje de error acerca de la vinculación mal escrita.

Como regla general, no incluyas excepciones a menos de que sean con el propósito de "enrutarlas" hacia alguna parte—por ejemplo, a través de la red para decirle a otro sistema que nuestro programa se bloqueó. E incluso entonces, piensa cuidadosamente sobre cómo podrias estar ocultando información.

Por lo tanto, queremos detectar un tipo de excepción *específico*. Podemos hacer esto al revisar el bloque `catch` si la excepción que tenemos es en la que estamos interesados y relanzar de otra manera. Pero como hacemos para reconocer una excepción?

Podríamos comparar su propiedad `message` con el mensaje de error que sucede estamos esperando. Pero esa es una forma inestable de escribir código—estariamos utilizando información destinada al consumo humano (el mensaje) para tomar una decisión programática. Tan pronto como alguien cambie (o traduzca) el mensaje, el código dejaria de funcionar.

En vez de esto, definamos un nuevo tipo de error y usemos `instanceof` para identificarlo.

```
class ErrorDeEntrada extends Error {}

function pedirDireccion(pregunta) {
  let resultado = prompt(pregunta);
  if (resultado.toLowerCase() == "izquierda") return "I";
  if (resultado.toLowerCase() == "derecha") return "D";
  throw new ErrorDeEntrada("Direccion invalida: " + resultado);
}
```

La nueva clase de error extiende `Error`. No define su propio constructor, lo que significa que hereda el constructor `Error`, que espera un mensaje de string como argumento. De hecho, no define nada—la clase está vacía. Los objetos `ErrorDeEntrada` se comportan como objetos `Error`, excepto que tienen una clase diferente por la cual podemos reconocerlos.

Ahora el ciclo puede atraparlos con mas cuidado.

```
for (;;) {
  try {
    let direccion = pedirDireccion("Donde?");
    console.log("Tu eliges ", direccion);
    break;
  } catch (e) {
    if (e instanceof ErrorDeEntrada) {
      console.log ("No es una dirección válida. Inténtalo de nuevo")
          ;
    } else {
      throw e;
    }
  }
}
```

Esto capturará solo las instancias de `error` y dejará que las excepciones no relacionadas pasen a través. Si reintroduce el error tipográfico, el error de la vinculación indefinida será reportado correctamente.

Afirmaciones

Las *afirmaciones* son comprobaciones dentro de un programa que verifican que algo este en la forma en la que se supone que debe estar. Se usan no para manejar situaciones que puedan aparecer en el funcionamiento normal, pero para encontrar errores hechos por el programador.

Si, por ejemplo, `primerElemento` se describe como una función que nunca se debería invocar en arrays vacíos, podríamos escribirla así:

```
function primerElemento(array) {
  if (array.length == 0) {
    throw new Error("primerElemento llamado con []");
  }
  return array[0];
}
```

Ahora, en lugar de silenciosamente retornar `undefined` (que es lo que obtienes cuando lees una propiedad de array que no existe), esto explotará fuertemente tu programa tan pronto como lo uses mal. Esto hace que sea menos probable que tales errores pasen desapercibidos, y sea más fácil encontrar su causa cuando estos ocurran.

No recomiendo tratar de escribir afirmaciones para todos los tipos posibles de entradas erroneas. Eso sería mucho trabajo y llevaría a código muy ruidoso. Querrás reservarlas para errores que son fáciles de hacer (o que te encuentras haciendo constantemente).

Resumen

Los errores y las malas entradas son hechos de la vida. Una parte importante de la programación es encontrar, diagnosticar y corregir errores. Los problemas pueden será más fáciles de notar si tienes un conjunto de pruebas automatizadas o si agregas afirmaciones a tus programas.

Por lo general, los problemas causados por factores fuera del control del programa deberían ser manejados con gracia. A veces, cuando el problema pueda ser manejado localmente, los valores de devolución especiales son una buena forma de rastrearlos. De lo contrario, las excepciones pueden ser preferibles.

Al lanzar una excepción, se desenrolla la pila de llamadas hasta el próximo bloque `try/catch` o hasta el final de la pila. Se le dará el valor de excepción al bloque `catch` que lo atrape, que debería verificar que en realidad es el tipo esperado de excepción y luego hacer algo con eso. Para ayudar a controlar el impredecible flujo de control causado por las excepciones, los bloques `finally` se pueden usar para asegurarte de que una parte del código *siempre* se ejecute cuando un bloque termina.

Ejercicios

Reintentar

Digamos que tienes una función `multiplicacionPrimitiva` que, en el 20 por ciento de los casos, multiplica dos números, y en el otro 80 por ciento, genera una excepción del tipo `FalloUnidadMultiplicadora`. Escribe una función que envuelva esta torpe función y solo siga intentando hasta que una llamada tenga éxito, después de lo cual retorna el resultado.

Asegúrete de solo manejar las excepciones que estás tratando de manejar.

La caja bloqueada

Considera el siguiente objeto (bastante artificial):

```
const caja = {
  bloqueada: true,
  desbloquear() { this.bloqueada = false; },
  bloquear() { this.bloqueada = true;  },
  _contenido: [],
  get contenido() {
    if (this.bloqueada) throw new Error("Bloqueada!");
    return this._contenido;
  }
};
```

Es solo una caja con una cerradura. Hay un array en la caja, pero solo puedes accederlo cuando la caja esté desbloqueada. Acceder directamente a la propiedad privada `_contenido` está prohibido.

Escribe una función llamada `conCajaDesbloqueada` que toma un valor de función como su argumento, desbloquea la caja, ejecuta la función y luego se asegura de que la caja se bloquee nuevamente antes de retornar, independientemente de si la función argumento retorno normalmente o lanzo una excepción.

"Algunas personas, cuando confrontadas con un problema, piensan 'Ya sé, usaré expresiones regulares.' Ahora tienen dos problemas."

—Jamie Zawinski

Chapter 9

Expresiones Regulares

Las herramientas y técnicas de la programación sobreviven y se propagan de una forma caótica y evolutiva. No siempre son los bonitas o las brillantes las que ganan, sino más bien las que funcionan lo suficientemente bien dentro del nicho correcto o que sucede se integran con otra pieza exitosa de tecnología.

En este capítulo, discutiré una de esas herramientas, *expresiones regulares*. Las expresiones regulares son una forma de describir patrones en datos de tipo string. Estas forman un lenguaje pequeño e independiente que es parte de JavaScript y de muchos otros lenguajes y sistemas.

Las expresiones regulares son terriblemente incómodas y extremadamente útiles. Su sintaxis es críptica, y la interfaz de programación que JavaScript proporciona para ellas es torpe. Pero son una poderosa herramienta para inspeccionar y procesar cadenas. Entender apropiadamente a las expresiones regulares te hará un programador más efectivo.

Creando una expresión regular

Una expresión regular es un tipo de objeto. Puede ser construido con el constructor `RegExp` o escrito como un valor literal al envolver un patrón en caracteres de barras diagonales (`/`).

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Ambos objetos de expresión regular representan el mismo patrón: un carácter *a* seguido por una *b* seguida de una *c*.

Cuando se usa el constructor `RegExp`, el patrón se escribe como un string normal, por lo que las reglas habituales se aplican a las barras invertidas.

La segunda notación, donde el patrón aparece entre caracteres de barras diagonales, trata a las barras invertidas de una forma diferente. Primero, dado que una barra diagonal termina el patrón, tenemos que poner una barra in-

vertida antes de cualquier barra diagonal que queremos sea *parte* del patrón. En adición, las barras invertidas que no sean parte de códigos especiales de caracteres (como `\n`) seran *preservadas*, en lugar de ignoradas, ya que están en strings, y cambian el significado del patrón. Algunos caracteres, como los signos de interrogación pregunta y los signos de adición, tienen significados especiales en las expresiones regulares y deben ir precedidos por una barra inversa si se pretende que representen al caracter en sí mismo.

```
let dieciochoMas = /dieciocho\+/;
```

Probando por coincidencias

Los objetos de expresión regular tienen varios métodos. El más simple es `test` ("probar"). Si le pasas un string, retornar un Booleano diciéndote si el string contiene una coincidencia del patrón en la expresión.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

Una expresión regular que consista solamente de caracteres no especiales simplemente representara esa secuencia de caracteres. Si *abc* ocurre en cualquier parte del string con la que estamos probando (no solo al comienzo), `test` retornara `true`.

Conjuntos de caracteres

Averiguar si un string contiene *abc* bien podría hacerse con una llamada a `indexOf`. Las expresiones regulares nos permiten expresar patrones más complicados.

Digamos que queremos encontrar cualquier número. En una expresión regular, poner un conjunto de caracteres entre corchetes hace que esa parte de la expresión coincida con cualquiera de los caracteres entre los corchetes.

Ambas expresiones coincidiran con todas los strings que contengan un dígito:

```
console.log(/[0123456789]/.test("en 1992"));
// → true
console.log(/[0-9]/.test("en 1992"));
```

```
// → true
```

Dentro de los corchetes, un guion (`-`) entre dos caracteres puede ser utilizado para indicar un rango de caracteres, donde el orden es determinado por el número Unicode del carácter. Los caracteres 0 a 9 estan uno al lado del otro en este orden (códigos 48 a 57), por lo que `[0-9]` los cubre a todos y coincide con cualquier dígito.

Un numero de caracteres comunes tienen sus propios atajos incorporados. Los dígitos son uno de ellos: `\d` significa lo mismo que `[0-9]`.

- `\d` Cualquier caracter dígito
- `\w` Un caracter alfanumérico
- `\s` Cualquier carácter de espacio en blanco (espacio, tabulación, nueva línea y similar)
- `\D` Un caracter que *no* es un dígito
- `\W` Un caracter no alfanumérico
- `\S` Un caracter que no es un espacio en blanco
- `.` Cualquier caracter a excepción de una nueva línea

Por lo que podrías coincidir con un formato de fecha y hora como 30-01-2003 15:20 con la siguiente expresión:

```
let fechaHora = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(fechaHora.test("30-01-2003 15:20"));
// → true
console.log(fechaHora.test("30-jan-2003 15:20"));
// → false
```

Eso se ve completamente horrible, no? La mitad de la expresión son barras invertidas, produciendo un ruido de fondo que hace que sea difícil detectar el patrón real que queremos expresar. Veremos una versión ligeramente mejorada de esta expresión más tarde.

Estos códigos de barra invertida también pueden usarse dentro de corchetes. Por ejemplo, `[\d.]` representa cualquier dígito o un carácter de punto. Pero el punto en sí mismo, entre corchetes, pierde su significado especial. Lo mismo va para otros caracteres especiales, como `+`.

Para *invertir* un conjunto de caracteres, es decir, para expresar que deseas coincidir con cualquier carácter *excepto* con los que están en el conjunto—puedes escribir un carácter de intercalación (`^`) después del corchete de apertura.

```
let noBinario = /[^01]/;
console.log(noBinario.test("1100100010100110"));
// → false
```

```
console.log(noBinario.test("1100100010200110"));
// → true
```

Repitiendo partes de un patrón

Ya sabemos cómo hacer coincidir un solo dígito. Qué pasa si queremos hacer coincidir un número completo—una secuencia de uno o más dígitos?

Cuando pones un signo más (`+`) después de algo en una expresión regular, este indica que el elemento puede repetirse más de una vez. Por lo tanto, `/\d+/` coincide con uno o más caracteres de dígitos.

```
console.log(/'\d+'/.test("'123'"));
// → true
console.log(/'\d+'/.test("''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("''"));
// → true
```

La estrella (`*`) tiene un significado similar pero también permite que el patrón coincida cero veces. Algo con una estrella después de el nunca evitara un patrón de coincidirlo—este solo coincidirá con cero instancias si no puede encontrar ningun texto adecuado para coincidir.

Un signo de interrogación hace que alguna parte de un patrón sea *opcional*, lo que significa que puede ocurrir cero o mas veces. En el siguiente ejemplo, el carácter *h* está permitido, pero el patrón también retorna verdadero cuando esta letra no esta.

```
let reusar = /reh?usar/;
console.log(reusar.test("rehusar"));
// → true
console.log(reusar.test("reusar"));
// → true
```

Para indicar que un patrón deberia ocurrir un número preciso de veces, usa llaves. Por ejemplo, al poner `{4}` después de un elemento, hace que requiera que este ocurra exactamente cuatro veces. También es posible especificar un rango de esta manera: `{2,4}` significa que el elemento debe ocurrir al menos

dos veces y como máximo cuatro veces.

Aquí hay otra versión del patrón fecha y hora que permite días tanto en dígitos individuales como dobles, meses y horas. Es también un poco más fácil de descifrar.

```
let fechaHora = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(fechaHora.test("30-1-2003 8:45"));
// → true
```

También puedes especificar rangos de final abierto al usar llaves omitiendo el número después de la coma. Entonces, `{5,}` significa cinco o más veces.

Agrupando subexpresiones

Para usar un operador como `*` o `+` en más de un elemento a la vez, tienes que usar paréntesis. Una parte de una expresión regular que se encierre entre paréntesis cuenta como un elemento único en cuanto a los operadores que la siguen están preocupados.

```
let caricaturaLlorando = /boo+(hoo+)+/i;
console.log(caricaturaLlorando.test("Boohoooohoohooo"));
// → true
```

El primer y segundo caracter `+` aplican solo a la segunda *o* en *boo* y *hoo*, respectivamente. El tercer `+` se aplica a la totalidad del grupo `(hoo+)`, haciendo coincidir una o más secuencias como esa.

La `i` al final de la expresión en el ejemplo hace que esta expresión regular sea insensible a mayúsculas y minúsculas, lo que permite que coincida con la letra mayúscula *B* en el string que se le da de entrada, asi el patrón en sí mismo este en minúsculas.

Coincidencias y grupos

El método `test` es la forma más simple de hacer coincidir una expresión. Solo te dice si coincide y nada más. Las expresiones regulares también tienen un método `exec` ("ejecutar") que retorna `null` si no se encontró una coincidencia y retorna un objeto con información sobre la coincidencia de lo contrario.

```
let coincidencia = /\d+/.exec("uno dos 100");
console.log(coincidencia);
```

```
// → ["100"]
console.log(coincidencia.index);
// → 8
```

Un objeto retornado por `exec` tiene una propiedad `index` ("indice") que nos dice *donde* en el string comienza la coincidencia exitosa. Aparte de eso, el objeto parece (y de hecho es) un array de strings, cuyo primer elemento es el string que coincidio—en el ejemplo anterior, esta es la secuencia de dígitos que estábamos buscando.

Los valores de tipo string tienen un método `match` que se comporta de manera similar.

```
console.log("uno dos 100".match(/\d+/));
// → ["100"]
```

Cuando la expresión regular contenga subexpresiones agrupadas con paréntesis, el texto que coincida con esos grupos también aparecerá en el array. La coincidencia completa es siempre el primer elemento. El siguiente elemento es la parte que coincidio con el primer grupo (el que abre paréntesis primero en la expresión), luego el segundo grupo, y asi sucesivamente.

```
let textoCitado = /'([^']*)'/;
console.log(textoCitado.exec("ella dijo 'hola'"));
// → ["'hola'", "hola"]
```

Cuando un grupo no termina siendo emparejado en absoluto (por ejemplo, cuando es seguido de un signo de interrogación), su posición en el array de salida sera `undefined`. Del mismo modo, cuando un grupo coincida multiples veces, solo la ultima coincidencia termina en el array.

```
console.log(/mal(isimo)?/.exec("mal"));
// → ["mal", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

Los grupos pueden ser útiles para extraer partes de un string. Si no solo queremos verificar si un string contiene una fecha pero también extraerla y construir un objeto que la represente, podemos envolver paréntesis alrededor de los patrones de dígitos y tomar directamente la fecha del resultado de `exec`.

Pero primero, un breve desvío, en el que discutiremos la forma incorporada

de representar valores de fecha y hora en JavaScript.

La clase Date (``Fecha'')

JavaScript tiene una clase estándar para representar fechas—o mejor dicho, puntos en el tiempo. Se llama `Date`. Si simplemente creas un objeto fecha usando `new`, obtienes la fecha y hora actual.

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 GMT+0100 (CET)
```

También puedes crear un objeto para un tiempo específico.

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript usa una convención en donde los números de los meses comienzan en cero (por lo que Diciembre es 11), sin embargo, los números de los días comienzan en uno. Esto es confuso y tonto. Ten cuidado.

Los últimos cuatro argumentos (horas, minutos, segundos y milisegundos) son opcionales y se toman como cero cuando no se dan.

Las marcas de tiempo se almacenan como la cantidad de milisegundos desde el inicio de 1970, en la zona horaria UTC. Esto sigue una convención establecida por el "Tiempo Unix", el cual se inventó en ese momento. Puedes usar números negativos para los tiempos anteriores a 1970. Usar el método `getTime` ("obtenerTiempo") en un objeto fecha retorna este número. Es bastante grande, como te puedes imaginar.

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

Si le das al constructor `Date` un único argumento, ese argumento sera tratado como un conteo de milisegundos. Puedes obtener el recuento de milisegundos actual creando un nuevo objeto `Date` y llamando `getTime` en él o llamando a la función `Date.now`.

Los objetos de fecha proporcionan métodos como `getFullYear` ("obtenerAño-

Completo”), `getMonth` (“obtenerMes”), `getDate` (“obtenerFecha”), `getHours` (“obtenerHoras”), `getMinutes` (“obtenerMinutos”), y `getSeconds` (“obtenerSegundos”) para extraer sus componentes. Además de `getFullYear`, también existe `getYear` (“obtenerAño”), que te da como resultado un valor de año de dos dígitos bastante inútil (como `93` o `14`).

Al poner paréntesis alrededor de las partes de la expresión en las que estamos interesados, ahora podemos crear un objeto de fecha a partir de un string.

```
function obtenerFecha(string) {
  let [_, dia, mes, año] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(año, mes - 1, dia);
}
console.log(obtenerFecha("30-1-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

La vinculación `_` (guion bajo) es ignorada, y solo se usa para omitir el elemento de coincidencia completa en el array retornado por `exec`.

Palabra y límites de string

Desafortunadamente, `obtenerFecha` felizmente también extraerá la absurda fecha 00-1-3000 del string `"100-1-30000"`. Una coincidencia puede suceder en cualquier lugar del string, por lo que en este caso, esta simplemente comenzará en el segundo carácter y terminara en el penúltimo carácter.

Si queremos hacer cumplir que la coincidencia deba abarcar el string completamente, puedes agregar los marcadores `^` y `$`. El signo de intercalación (`"^"`) coincide con el inicio del string de entrada, mientras que el signo de dólar coincide con el final. Entonces, `/^\d+$/` coincide con un string compuesto por uno o más dígitos, `/^!/` coincide con cualquier string que comience con un signo de exclamación, y `/x^/` no coincide con ningun string (no puede haber una x antes del inicio del string).

Si, por el otro lado, solo queremos asegurarnos de que la fecha comience y termina en un límite de palabras, podemos usar el marcador `\b`. Un límite de palabra puede ser el inicio o el final del string o cualquier punto en el string que tenga un carácter de palabra (como en `\w`) en un lado y un carácter de no-palabra en el otro.

```
console.log(/cat/.test("concatenar"));
// → true
console.log(/\bcat\b/.test("concatenar"));
```

```
// → false
```

Ten en cuenta que un marcador de límite no coincide con un carácter real. Solo hace cumplir que la expresión regular coincida solo cuando una cierta condición se mantenga en el lugar donde aparece en el patrón.

Patrones de elección

Digamos que queremos saber si una parte del texto contiene no solo un número pero un número seguido de una de las palabras *cerdo*, *vaca*, o *pollo*, o cualquiera de sus formas plurales.

Podríamos escribir tres expresiones regulares y probarlas a su vez, pero hay una manera más agradable. El carácter de tubería (`|`) denota una elección entre el patrón a su izquierda y el patrón a su derecha. Entonces puedo decir esto:

```
let conteoAnimales = /\b\d+ (cerdo|vaca|pollo)s?\b/;
console.log(conteoAnimales.test("15 cerdo"));
// → true
console.log(conteoAnimales.test("15 cerdopollos"));
// → false
```

Los paréntesis se pueden usar para limitar la parte del patrón a la que aplica el operador de tuberia, y puedes poner varios de estos operadores unos a los lados de los otros para expresar una elección entre más de dos alternativas.

Las mecánicas del emparejamiento

Conceptualmente, cuando usas `exec` o `test` el motor de la expresión regular busca una coincidencia en tu string al tratar de hacer coincidir la expresión primero desde el comienzo del string, luego desde el segundo caracter, y así sucesivamente hasta que encuentra una coincidencia o llega al final del string. Retornara la primera coincidencia que se puede encontrar o fallara en encontrar cualquier coincidencia.

Para realmente hacer la coincidencia, el motor trata una expresión regular algo así como un diagrama de flujo. Este es el diagrama para la expresión de ganado en el ejemplo anterior:

Nuestra expresión coincide si podemos encontrar un camino desde el lado izquierdo del diagrama al lado derecho. Mantenemos una posición actual en el string, y cada vez que nos movemos a través de una caja, verificaremos que la parte del string después de nuestra posición actual coincida con esa caja.

Entonces, si tratamos de coincidir `"los 3 cerdos"` desde la posición 4, nuestro progreso a través del diagrama de flujo se vería así:

- En la posición 4, hay un límite de palabra, por lo que podemos pasar la primera caja.
- Aún en la posición 4, encontramos un dígito, por lo que también podemos pasar la segunda caja.
- En la posición 5, una ruta regresa a antes de la segunda caja (dígito), mientras que la otro se mueve hacia adelante a través de la caja que contiene un caracter de espacio simple. Hay un espacio aquí, no un dígito, asi que debemos tomar el segundo camino.
- Ahora estamos en la posición 6 (el comienzo de "cerdos") y en el camino de tres vías en el diagrama. No vemos "vaca" o "pollo" aquí, pero vemos "cerdo", entonces tomamos esa rama.
- En la posición 9, después de la rama de tres vías, un camino se salta la caja *s* y va directamente al límite de la palabra final, mientras que la otra ruta coincide con una *s*. Aquí hay un carácter *s*, no una palabra límite, por lo que pasamos por la caja *s*.
- Estamos en la posición 10 (al final del string) y solo podemos hacer coincidir una palabra límite. El final de un string cuenta como un límite de palabra, así que pasamos por la última caja y hemos emparejado con éxito este string.

Retrocediendo

La expresión regular `/\b([01]+b|[\da-f]+h|\d+)\b/` coincide con un número binario seguido de una *b*, un número hexadecimal (es decir, en base 16, con las

letras a a f representando los dígitos 10 a 15) seguido de una h, o un número decimal regular sin caracter de sufijo. Este es el diagrama correspondiente:

Al hacer coincidir esta expresión, a menudo sucederá que la rama superior (binaria) sea ingresada aunque la entrada en realidad no contenga un número binario. Al hacer coincidir el string `"103"`, por ejemplo, queda claro solo en el 3 que estamos en la rama equivocada. El string *si* coincide con la expresión, pero no con la rama en la que nos encontramos actualmente.

Entonces el "emparejador" *retrocede*. Al ingresar a una rama, este recuerda su posición actual (en este caso, al comienzo del string, justo después del primer cuadro de límite en el diagrama) para que pueda retroceder e intentar otra rama si la actual no funciona. Para el string `"103"`, después de encontrar los 3 caracteres, comenzará a probar la rama para números hexadecimales, que falla nuevamente porque no hay h después del número. Por lo tanto, intenta con la rama de número decimal. Esta encaja, y se informa de una coincidencia después de todo.

El emparejador se detiene tan pronto como encuentra una coincidencia completa. Esto significa que si múltiples ramas podrían coincidir con un string, solo la primera (ordenado por donde las ramas aparecen en la expresión regular) es usada.

El retroceso también ocurre para repetición de operadores como + y `*`. Si hace coincidir `/^.*x/` contra `"abcxe"`, la parte `.*` intentará primero consumir todo el string. El motor entonces se dará cuenta de que necesita una x para que coincida con el patrón. Como no hay x al pasar el final del string, el operador de estrella intenta hacer coincidir un caracter menos. Pero el emparejador tampoco encuentra una x después de `abcx`, por lo que retrocede nuevamente, haciendo coincidir el operador de estrella con `abc`. *Ahora* encuentra una x donde lo necesita e informa de una coincidencia exitosa de las posiciones 0 a 4.

Es posible escribir expresiones regulares que harán un *monton* de retrocesos. Este problema ocurre cuando un patrón puede coincidir con una pieza de entrada en muchas maneras diferentes. Por ejemplo, si nos confundimos mientras escribimos una expresión regular de números binarios, podríamos accidentalmente escribir algo como `/([01]+)+b/`.

Si intentas hacer coincidir eso con algunas largas series de ceros y unos sin un caracter *b* al final, el emparejador primero pasara por el ciclo interior hasta que se quede sin dígitos. Entonces nota que no hay *b*, asi que retrocede una posición, atraviesa el ciclo externo una vez, y se da por vencido otra vez, tratando de retroceder fuera del ciclo interno una vez más. Continuará probando todas las rutas posibles a través de estos dos bucles. Esto significa que la cantidad de trabajo se *duplica* con cada caracter. Incluso para unas pocas docenas de caracters, la coincidencia resultante tomará prácticamente para siempre.

El método replace

Los valores de string tienen un método `replace` ("reemplazar") que se puede usar para reemplazar parte del string con otro string.

```
console.log("papa".replace("p", "m"));
// → mapa
```

El primer argumento también puede ser una expresión regular, en cuyo caso ña primera coincidencia de la expresión regular es reemplazada. Cuando una opción `g` (para *global*) se agrega a la expresión regular, *todas* las coincidencias en el string será reemplazadas, no solo la primera.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
```

```
// → Barabadar
```

Hubiera sido sensato si la elección entre reemplazar una coincidencia o todas las coincidencias se hiciera a través de un argumento adicional en `replace` o proporcionando un método diferente, `replaceAll` ("reemplazarTodas"). Pero por alguna desafortunada razón, la elección se basa en una propiedad de los expresiones regulares en su lugar.

El verdadero poder de usar expresiones regulares con `replace` viene del hecho de que podemos referirnos a grupos coincidentes en la string de reemplazo. Por ejemplo, supongamos que tenemos una gran string que contenga los nombres de personas, un nombre por línea, en el formato `Apellido, Nombre`. Si deseamos intercambiar estos nombres y eliminar la coma para obtener un formato `Nombre Apellido`, podemos usar el siguiente código:

```
console.log(
  "Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Philip Wadler
```

Los `$1` y `$2` en el string de reemplazo se refieren a los grupos entre paréntesis del patrón. `$1` se reemplaza por el texto que coincide con el primer grupo, `$2` por el segundo, y así sucesivamente, hasta `$9`. Puedes hacer referencia a la coincidencia completa con `$&`.

Es posible pasar una función, en lugar de un string, como segundo argumento para `replace`. Para cada reemplazo, la función será llamada con los grupos coincidentes (así como con la coincidencia completa) como argumentos, y su valor de retorno se insertará en el nuevo string.

Aquí hay un pequeño ejemplo:

```
let s = "la cia y el fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → la CIA y el FBI
```

Y aquí hay uno más interesante:

```
let almacen = "1 limon, 2 lechugas, y 101 huevos";
function menosUno(coincidencia, cantidad, unidad) {
  cantidad = Number(cantidad) - 1;
```

```
  if (cantidad == 1) { // solo queda uno, remover la 's'
    unidad = unidad.slice(0, unidad.length - 1);
  } else if (cantidad == 0) {
    cantidad = "sin";
  }
  return cantidad + " " + unidad;
}
console.log(almacen.replace(/(\d+) (\w+)/g, menosUno));
// → sin limon, 1 lechuga, y 100 huevos
```

Esta función toma un string, encuentra todas las ocurrencias de un número seguido de una palabra alfanumérica, y retorna un string en la que cada ocurrencia es decrementada por uno.

El grupo `(\d+)` termina como el argumento `cantidad` para la función, y el grupo `(\w+)` se vincula a `unidad`. La función convierte `cantidad` a un número—lo que siempre funciona, ya que coincidio con `\d+`—y realiza algunos ajustes en caso de que solo quede uno o cero.

Codicia

Es posible usar `replace` para escribir una función que elimine todo los comentarios de un fragmento de código JavaScript. Aquí hay un primer intento:

```
function removerComentarios(codigo) {
  return codigo.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(removerComentarios("1 + /* 2 */3"));
// → 1 + 3
console.log(removerComentarios("x = 10;// ten!"));
// → x = 10;
console.log(removerComentarios("1 /* a */+/* b */ 1"));
// → 1  1
```

La parte anterior al operador *o* coincide con dos caracteres de barra inclinada seguido de cualquier número de caracteres que no sean nuevas lineas. La parte para los comentarios de líneas múltiples es más complicado. Usamos `[^]` (cualquier caracter que no está en el conjunto de caracteres vacíos) como una forma de unir cualquier caracter. No podemos simplemente usar un punto aquí porque los comentarios de bloque pueden continuar en una nueva línea, y el carácter del período no coincide con caracteres de nuevas lineas.

Pero la salida de la última línea parece haber salido mal. Por qué?

La parte `[^]*` de la expresión, como describí en la sección retroceder, primero coincidirá tanto como sea posible. Si eso causa un falo en la siguiente parte del patrón, el emparejador retrocede un caracter e intenta nuevamente desde allí. En el ejemplo, el emparejador primero intenta emparejar el resto del string y luego se mueve hacia atrás desde allí. Este encontrará una ocurrencia de `*/` después de retroceder cuatro caracteres y emparejar eso. Esto no es lo que queríamos, la intención era hacer coincidir un solo comentario, no ir hasta el final del código y encontrar el final del último comentario de bloque.

Debido a este comportamiento, decimos que los operadores de repetición (`+`, `*`, `?` y `{}`) son _ codiciosos, lo que significa que coinciden con tanto como pueden y retroceden desde allí. Si colocas un signo de interrogación después de ellos (`+?`, `*?`, `??`, `{}?`), se vuelven no-codiciosos y comienzan a hacer coincidir lo menos posible, haciendo coincidir más solo cuando el patrón restante no se ajuste a la coincidencia más pequeña.

Y eso es exactamente lo que queremos en este caso. Al hacer que la estrella coincida con el tramo más pequeño de caracteres que nos lleve a un `*/`, consumimos un comentario de bloque y nada más.

```
function removerComentarios(codigo) {
  return codigo.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(removerComentarios("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Una gran cantidad de errores en los programas de expresiones regulares se pueden rastrear a intencionalmente usar un operador codicioso, donde uno que no sea codicioso trabajaria mejor. Al usar un operador de repetición, considera la variante no-codiciosa primero.

Creando objetos RegExp dinámicamente

Hay casos en los que quizás no sepas el patrón exacto necesario para coincidir cuando estes escribiendo tu código. Imagina que quieres buscar el nombre del usuario en un texto y encerrarlo en caracteres de subrayado para que se destaque. Como solo sabrás el nombrar una vez que el programa se está ejecutando realmente, no puedes usar la notación basada en barras.

Pero puedes construir un string y usar el constructor `RegExp` en el. Aquí hay un ejemplo:

```
let nombre = "harry";
```

```
let texto = "Harry es un personaje sospechoso.";
let regexp = new RegExp("\\b(" + nombre + ")\\b", "gi");
console.log(texto.replace(regexp, "_$1_"));
// → _Harry_ es un personaje sospechoso.
```

Al crear los marcadores de límite `\b`, tenemos que usar dos barras invertidas porque las estamos escribiendo en un string normal, no en una expresión regular contenida en barras. El segundo argumento para el constructor `RegExp` contiene las opciones para la expresión regular—en este caso, `"gi"` para global e insensible a mayúsculas y minúsculas.

Pero, y si el nombre es `"dea+hl[]rd"` porque nuestro usuario es un nerd adolescente? Eso daría como resultado una expresión regular sin sentido que en realidad no coincidirá con el nombre del usuario.

Para solucionar esto, podemos agregar barras diagonales inversas antes de cualquier caracter que tenga un significado especial.

```
let nombre = "dea+hl[]rd";
let texto = "Este sujeto dea+hl[]rd es super fastidioso.";
let escapados = nombre.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escapados + "\\b", "gi");
console.log(texto.replace(regexp, "_$&_"));
// → Este sujeto _dea+hl[]rd_ es super fastidioso.
```

El método search

El método `indexOf` en strings no puede invocarse con una expresión regular. Pero hay otro método, `search` ("buscar"), que espera una expresión regular. Al igual que `indexOf`, retorna el primer índice en que se encontró la expresión, o -1 cuando no se encontró.

```
console.log("  palabra".search(/\S/));
// → 2
console.log("    ".search(/\S/));
// → -1
```

Desafortunadamente, no hay forma de indicar que la coincidencia debería comenzar a partir de un desplazamiento dado (como podemos con el segundo argumento para `indexOf`), que a menudo sería útil.

La propiedad lastIndex

De manera similar el método `exec` no proporciona una manera conveniente de comenzar buscando desde una posición dada en el string. Pero proporciona una manera *in*conveniente.

Los objetos de expresión regular tienen propiedades. Una de esas propiedades es `source` ("fuente"), que contiene el string de donde se creó la expresión. Otra propiedad es `lastIndex` ("ultimoIndice"), que controla, en algunas circunstancias limitadas, donde comenzará la siguiente coincidencia.

Esas circunstancias son que la expresión regular debe tener la opción global (`g`) o adhesiva (`y`) habilitada, y la coincidencia debe suceder a través del método `exec`. De nuevo, una solución menos confusa hubiese sido permitir que un argumento adicional fuera pasado a `exec`, pero la confusión es una característica esencial de la interfaz de las expresiones regulares de JavaScript.

```
let patron = /y/g;
patron.lastIndex = 3;
let coincidencia = patron.exec("xyzzy");
console.log(coincidencia.index);
// → 4
console.log(patron.lastIndex);
// → 5
```

Si la coincidencia fue exitosa, la llamada a `exec` actualiza automáticamente a la propiedad `lastIndex` para que apunte después de la coincidencia. Si no se encontraron coincidencias, `lastIndex` vuelve a cero, que es también el valor que tiene un objeto de expresión regular recién construido.

La diferencia entre las opciones globales y las adhesivas es que, cuandoa adhesivo está habilitado, la coincidencia solo tendrá éxito si comienza directamente en `lastIndex`, mientras que con global, buscará una posición donde pueda comenzar una coincidencia.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let adhesivo = /abc/y;
console.log(adhesivo.exec("xyz abc"));
// → null
```

Cuando se usa un valor de expresión regular compartido para múltiples llamadas a `exec`, estas actualizaciones automáticas a la propiedad `lastIndex`

pueden causar problemas. Tu expresión regular podría estar accidentalmente comenzando en un índice que quedó de una llamada anterior.

```
let digito = /\d/g;
console.log(digito.exec("aqui esta: 1"));
// → ["1"]
console.log(digito.exec("y ahora: 1"));
// → null
```

Otro efecto interesante de la opción global es que cambia la forma en que funciona el método `match` en strings. Cuando se llama con una expresión global, en lugar de retornar un matriz similar al retornado por `exec`,`match` encontrará *todas* las coincidencias del patrón en el string y retornar un array que contiene los strings coincidentes.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

Por lo tanto, ten cuidado con las expresiones regulares globales. Los casos donde son necesarias—llamadas a `replace` y lugares donde deseas explícitamente usar `lastIndex`—son generalmente los únicos lugares donde querras usarlas.

Ciclos sobre coincidencias

Una cosa común que hacer es escanear todas las ocurrencias de un patrón en un string, de una manera que nos de acceso al objeto de coincidencia en el cuerpo del ciclo. Podemos hacer esto usando `lastIndex` y `exec`.

```
let entrada = "Un string con 3 numeros en el... 42 y 88.";
let numero = /\b\d+\b/g;
let coincidencia;
while (coincidencia = numero.exec(entrada)) {
  console.log("Se encontro", coincidencia[0], "en", coincidencia.
      index);
}
// → Se encontro 3 en 14
//   Se encontro 42 en 33
//   Se encontro 88 en 38
```

Esto hace uso del hecho de que el valor de una expresión de asignación (`=`) es el valor asignado. Entonces al usar `coincidencia = numero.exec(entrada)`

como la condición en la declaración `while`, realizamos la coincidencia al inicio de cada iteración, guardamos su resultado en una vinculación, y terminamos de repetir cuando no se encuentran más coincidencias.

Análisis de un archivo INI

Para concluir el capítulo, veremos un problema que requiere de expresiones regulares. Imagina que estamos escribiendo un programa para recolectar automáticamente información sobre nuestros enemigos de el Internet. (No escribiremos ese programa aquí, solo la parte que lee el archivo de configuración. Lo siento.) El archivo de configuración se ve así:

```
motordebusqueda=https://duckduckgo.com/?q=$1
malevolencia=9.7

; los comentarios estan precedidos por un punto y coma...
; cada seccion contiene un enemigo individual

[larry]
nombrecompleto=Larry Doe
tipo=bravucon del preescolar
sitioweb=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
nombrecompleto=Davaeorn
tipo=hechizero malvado
directoriosalida=/home/marijn/enemies/davaeorn
```

Las reglas exactas para este formato (que es un formato ampliamente utilizado, usualmente llamado un archivo *INI*) son las siguientes:

- Las líneas en blanco y líneas que comienzan con punto y coma se ignoran.
- Las líneas envueltas en `[` y `]` comienzan una nueva sección.
- Líneas que contienen un identificador alfanumérico seguido de un carácter `=` agregan una configuración a la sección actual.
- Cualquier otra cosa no es válida.

Nuestra tarea es convertir un string como este en un objeto cuyas propiedades contengas strings para configuraciones sin sección y sub-objetos para secciones, con esos subobjetos conteniendo la configuración de la sección.

Dado que el formato debe procesarse línea por línea, dividir el archivo en líneas separadas es un buen comienzo. Usamos `string.split("\n")` para hacer esto en el Capítulo 4. Algunos sistemas operativos, sin embargo, usan no solo un carácter de nueva línea para separar lineas sino un carácter de retorno de carro seguido de una nueva línea (`"\r\n"`). Dado que el método `split` también permite una expresión regular como su argumento, podemos usar una expresión regular como `/\r?\n/` para dividir el string de una manera que permita tanto `"\n"` como `"\r\n"` entre líneas.

```
function analizarINI(string) {
  // Comenzar con un objeto para mantener los campos de nivel
     superior
  let resultado = {};
  let seccion = resultado;
  string.split(/\r?\n/).forEach(linea => {
    let coincidencia;
    if (coincidencia = linea.match(/^(\w+)=(.*)$/)) {
      seccion[coincidencia[1]] = coincidencia[2];
    } else if (coincidencia = linea.match(/^\[(.*)\]$/)) {
      seccion = resultado[coincidencia[1]] = {};
    } else if (!/^\s*(;.*)?$/.test(linea)) {
      throw new Error("Linea '" + linea + "' no es valida.");
    }
  });
  return resultado;
}

console.log(analizarINI(`
nombre=Vasilis
[direccion]
ciudad=Tessaloniki`));
// → {nombre: "Vasilis", direccion: {ciudad: "Tessaloniki"}}
```

El código pasa por las líneas del archivo y crea un objeto. Las propiedades en la parte superior se almacenan directamente en ese objeto, mientras que las propiedades que se encuentran en las secciones se almacenan en un objeto de sección separado. La vinculación `sección` apunta al objeto para la sección actual.

Hay dos tipos de de líneas significativas—encabezados de seccion o lineas de propiedades. Cuando una línea es una propiedad regular, esta se almacena en la sección actual. Cuando se trata de un encabezado de sección, se crea un nuevo objeto de sección, y `seccion` se configura para apuntar a él.

Nota el uso recurrente de `^` y `$` para asegurarse de que la expresión coincida con toda la línea, no solo con parte de ella. Dejando afuera estos resultados en código que funciona principalmente, pero que se comporta de forma extraña para algunas entradas, lo que puede ser un error difícil de rastrear.

El patrón `if (coincidencia = string.match (...))` es similar al truco de usar una asignación como condición para `while`. A menudo no estas seguro de que tu llamada a `match` tendrá éxito, para que puedas acceder al objeto resultante solo dentro de una declaración `if` que pruebe esto. Para no romper la agradable cadena de las formas `else if`, asignamos el resultado de la coincidencia a una vinculación e inmediatamente usamos esa asignación como la prueba para la declaración `if`.

Si una línea no es un encabezado de sección o una propiedad, la función verifica si es un comentario o una línea vacía usando la expresión `/^\s*(;.*)?$/`. Ves cómo funciona? La parte entre los paréntesis coincidirá con los comentarios, y el `?` asegura que también coincida con líneas que contengan solo espacios en blanco. Cuando una línea no coincida con cualquiera de las formas esperadas, la función arroja una excepción.

Caracteres internacionales

Debido a la simplista implementación inicial de JavaScript y al hecho de que este enfoque simplista fue luego establecido en piedra como comportamiento estándar, las expresiones regulares de JavaScript son bastante tontas acerca de los caracteres que no aparecen en el idioma inglés. Por ejemplo, en cuanto a las expresiones regulares de JavaScript, una "palabra caracter" es solo uno de los 26 caracteres en el alfabeto latino (mayúsculas o minúsculas), dígitos decimales, y, por alguna razón, el carácter de guion bajo. Cosas como *é* o *ß*, que definitivamente son caracteres de palabras, no coincidirán con `\w` (y *si* coincidiran con `\W` mayúscula, la categoría no-palabra).

Por un extraño accidente histórico, `\s` (espacio en blanco) no tiene este problema y coincide con todos los caracteres que el estándar Unicode considera espacios en blanco, incluyendo cosas como el (espacio de no separación) y el Separador de vocales Mongol.

Otro problema es que, de forma predeterminada, las expresiones regulares funcionan en unidades del código, como se discute en el Capítulo 5, no en caracteres reales. Esto significa que los caracteres que estan compustos de dos unidades de código se comportan de manera extraña.

```
console.log(/🍎{3}/.test("🍎🍎🍎"));
// → false
```

```
console.log(/<.>/.test("<🌹>"));
// → false
console.log(/<.>/u.test("<🌹>"));
// → true
```

El problema es que la 🍎 en la primera línea se trata como dos unidades de código, y la parte `{3}` se aplica solo a la segunda. Del mismo modo, el punto coincide con una sola unidad de código, no con las dos que componen al emoji de rosa.

Debe agregar una opción `u` (para Unicode) a tu expresión regular para hacerla tratar a tales caracteres apropiadamente. El comportamiento incorrecto sigue siendo el predeterminado, desafortunadamente, porque cambiarlo podría causar problemas en código ya existente que depende de él.

Aunque esto solo se acaba de estandarizar y aun no es, al momento de escribir este libro, ampliamente compatible con muchs nabegadores, es posible usar `\p` en una expresión regular (que debe tener habilitada la opción Unicode) para que coincida con todos los caracteres a los que el estándar Unicode lis asigna una propiedad determinada.

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode define una cantidad de propiedades útiles, aunque encontrar la que necesitas puede no ser siempre trivial. Puedes usar la notación `\p{Property=Value}` para hacer coincidir con cualquier carácter que tenga el valor dado para esa propiedad. Si el nombre de la propiedad se deja afuera, como en `\p{Name}`, se asume que el nombre es una propiedad binaria como `Alfabético` o una categoría como `Número`.

Resumen

Las expresiones regulares son objetos que representan patrones en strings. Ellas usan su propio lenguaje para expresar estos patrones.

`/abc/`	Una secuencia de caracteres
`/[abc]/`	Cualquier caracter de un conjunto de caracteres
`/[^abc]/`	Cualquier carácter que *no* este en un conjunto de caracteres
`/[0-9]/`	Cualquier caracter en un rango de caracteres
`/x+/`	Una o más ocurrencias del patrón `x`
`/x+?/`	Una o más ocurrencias, no codiciosas
`/x*/`	Cero o más ocurrencias
`/x?/`	Cero o una ocurrencia
`/x{2,4}/`	De dos a cuatro ocurrencias
`/(abc)/`	Un grupo
`/a|b|c/`	Cualquiera de varios patrones
`/\d/`	Cualquier caracter de digito
`/\w/`	Un caracter alfanumérico ("carácter de palabra")
`/\s/`	Cualquier caracter de espacio en blanco
`/./`	Cualquier caracter excepto líneas nuevas
`/\b/`	Un límite de palabra
`/^/`	Inicio de entrada
`/$/`	Fin de la entrada

Una expresión regular tiene un método `test` para probar si una determinada string coincide cn ella. También tiene un método `exec` que, cuando una coincidencia es encontrada, retorna un array que contiene todos los grupos que coincidieron. Tal array tiene una propiedad `index` que indica en dónde comenzó la coincidencia.

Los strings tienen un método `match` para coincidirlas con una expresión regular y un método `search` para buscar por una, retornando solo la posición inicial de la coincidencia. Su método `replace` puede reemplazar coincidencias de un patrón con un string o función de reemplazo.

Las expresiones regulares pueden tener opciones, que se escriben después de la barra que cierra la expresión. La opción `i` hace que la coincidencia no distinga entre mayúsculas y minúsculas. La opción `g` hace que la expresión sea *global*, que, entre otras cosas, hace que el método `replace` reemplace todas las instancias en lugar de solo la primera. La opción `y` la hace adhesivo, lo que significa que hará que no busque con anticipación y omita la parte del string cuando busque una coincidencia. La opción `u` activa el modo Unicode, lo que soluciona varios problemas alrededor del manejo de caracteres que toman dos unidades de código.

Las expresiones regulares son herramientas afiladas con un manejo incómodo. Ellas simplifican algunas tareas enormemente, pero pueden volverse inmanejables rápidamente cuando se aplican a problemas complejos. Parte de saber cómo usarlas es resistiendo el impulso de tratar de calzar cosas que no pueden

ser expresadas limpiamente en ellas.

Ejercicios

Es casi inevitable que, durante el curso de trabajar en estos ejercicios, te sentiras confundido y frustrado por el comportamiento inexplicable de alguna regular expresión. A veces ayuda ingresar tu expresión en una herramienta en línea como *debuggex.com* para ver si su visualización corresponde a lo que pretendías y a experimentar con la forma en que responde a varios strings de entrada.

Golf Regexp

Golf de Codigo es un término usado para el juego de intentar expresar un programa particular con el menor número de caracteres posible. Similarmente, *Golf de Regexp* es la práctica de escribir una expresión regular tan pequeña como sea posible para que coincida con un patrón dado, y *sólo* con ese patrón.

Para cada uno de los siguientes elementos, escribe una expresión regular para probar si alguna de las substrings dadas ocurre en un string. La expresión regular debe coincidir solo con strings que contengan una de las substrings descritas. No te preocupes por los límites de palabras a menos que sean explícitamente mencionados. Cuando tu expresión funcione, ve si puedes hacerla más pequeña.

1. *car* y *cat*
2. *pop* y *prop*
3. *ferret*, *ferry*, y *ferrari*
4. Cualquier palabra que termine *ious*
5. Un carácter de espacio en blanco seguido de un punto, coma, dos puntos o punto y coma
6. Una palabra con mas de seis letras
7. Una palabra sin la letra *e* (o *E*)

Consulta la tabla en el resumen del capítulo para ayudarte. Pruebe cada solución con algunos strings de prueba.

Estilo entre comillas

Imagina que has escrito una historia y has utilizado comillass simples en todas partes para marcar piezas de diálogo. Ahora quieres reemplazar todas las comillas de diálogo con comillas dobles, pero manteniendo las comillas simples usadas en contracciones como *aren't*.

Piensa en un patrón que distinga de estos dos tipos de uso de citas y crea una llamada al método `replace` que haga el reemplazo apropiado.

Números otra vez

Escribe una expresión que coincida solo con el estilo de números en JavaScript. Esta debe admitir un signo opcional menos *o* más delante del número, el punto decimal, y la notación de exponente—`5e-3` o `1E10`— nuevamente con un signo opcional en frente del exponente. También ten en cuenta que no es necesario que hayan dígitos delante o detrás del punto, pero el el número no puede ser solo un punto. Es decir, `.5` y `5.` son numeros válidos de JavaScript, pero solo un punto *no* lo es.

"Escriba código que sea fácil de borrar, no fácil de extender."
—Tef, Programming is Terrible

Chapter 10

Módulos

El programa ideal tiene una estructura cristalina. La forma en que funciona es fácil de explicar, y cada parte juega un papel bien definido.

Un típico programa real crece orgánicamente. Nuevas piezas de funcionalidad se agregan a medida que surgen nuevas necesidades. Estructurar—y preservar la estructura—es trabajo adicional, trabajo que solo valdra la pena en el futuro, la *siguiente* vez que alguien trabaje en el programa. Así que es tentador descuidarlo, y permitir que las partes del programa se vuelvan profundamente enredadas.

Esto causa dos problemas prácticos. En primer lugar, entender tal sistema es difícil. Si todo puede tocar todo lo demás, es difícil ver a cualquier pieza dada de forma aislada. Estas obligado a construir un entendimiento holístico de todo el asunto. En segundo lugar, si quieres usar cualquiera de las funcionalidades de dicho programa en otra situación, reescribirla podria resultar más fácil que tratar de desenredarla de su contexto.

El término "gran bola de barro" se usa a menudo para tales programas grandes, sin estructura. Todo se mantiene pegado, y cuando intentas sacar una pieza, todo se desarma y tus manos se ensucian.

Módulos

Los *módulos* son un intento de evitar estos problemas. Un módulo es una pieza del programa que especifica en qué otras piezas este depende (sus *dependencias*) y qué funcionalidad proporciona para que otros módulos usen (su *interfaz*).

Las interfaces de los módulos tienen mucho en común con las interfaces de objetos, como las vimos en el Capítulo 6. Estas hacen parte del módulo disponible para el mundo exterior y mantienen el resto privado. Al restringir las formas en que los módulos interactúan entre sí, el el sistema se parece más a un juego de LEGOS, donde las piezas interactúan a través de conectores bien definidos, y menos como barro, donde todo se mezcla con todo.

Las relaciones entre los módulos se llaman *dependencias*. Cuando un módulo

necesita una pieza de otro módulo, se dice que depende de ese módulo. Cuando este hecho está claramente especificado en el módulo en sí, puede usarse para descubrir qué otros módulos deben estar presentes para poder ser capaces de usar un módulo dado y cargar dependencias automáticamente.

Para separar módulos de esa manera, cada uno necesita su propio alcance privado.

Simplemente poner todo tu código JavaScript en diferentes archivos no satisface estos requisitos. Los archivos aún comparten el mismo espacio de nombres global. Pueden, intencionalmente o accidentalmente, interferir con las vinculaciones de cada uno. Y la estructura de dependencia sigue sin estar clara. Podemos hacerlo mejor, como veremos más adelante en el capítulo.

Diseñar una estructura de módulo ajustada para un programa puede ser difícil. En la fase en la que todavía estás explorando el problema, intentando cosas diferentes para ver que funciona, es posible que desees no preocuparte demasiado por eso, ya que puede ser una gran distracción. Una vez que tengas algo que se sienta sólido, es un buen momento para dar un paso atrás y organizarlo.

Paquetes

Una de las ventajas de construir un programa a partir de piezas separadas, y ser capaces de ejecutar esas piezas por si mismas, es que tú podrías ser capaz de aplicar la misma pieza en diferentes programas.

Pero cómo se configura esto? Digamos que quiero usar la función `analizarINI` del Capítulo 9 en otro programa. Si está claro de qué depende la función (en este caso, nada), puedo copiar todo el código necesario en mi nuevo proyecto y usarlo. Pero luego, si encuentro un error en ese código, probablemente lo solucione en el programa en el que estoy trabajando en ese momento y me olvido de arreglarlo en el otro programa.

Una vez que comience a duplicar código, rápidamente te encontraras perdiendo tiempo y energía moviendo las copias alrededor y manteniéndolas actualizadas.

Ahí es donde los *paquetes* entran. Un paquete es un pedazo de código que puede ser distribuido (copiado e instalado). Puede contener uno o más módulos, y tiene información acerca de qué otros paquetes depende. Un paquete también suele venir con documentación que explica qué es lo que hace, para que las personas que no lo escribieron todavía puedan hacer uso de el.

Cuando se encuentra un problema en un paquete, o se agrega una nueva característica, el el paquete es actualizado. Ahora los programas que dependen de él (que también pueden ser otros paquetes) pueden actualizar a la nueva

versión.

Trabajar de esta manera requiere infraestructura. Necesitamos un lugar para almacenar y encontrar paquetes, y una forma conveniente de instalar y actualizarlos. En el mundo de JavaScript, esta infraestructura es provista por NPM (*npmjs.org*).

NPM es dos cosas: un servicio en línea donde uno puede descargar (y subir) paquetes, y un programa (incluido con Node.js) que te ayuda a instalar y administrarlos.

Al momento de escribir esto, hay más de medio millón de paquetes diferentes disponibles en NPM. Una gran parte de ellos son basura, debería mencionar, pero casi todos los paquetes útiles, disponibles públicamente, se puede encontrar allí. Por ejemplo, un analizador de archivos INI, similar al uno que construimos en el Capítulo 9, está disponible bajo el nombre de paquete `ini`.

En el Capítulo 20 veremos cómo instalar dichos paquetes de forma local utilizando el programa de línea de comandos `npm`.

Tener paquetes de calidad disponibles para descargar es extremadamente valioso. Significa que a menudo podemos evitar tener que reinventar un programa que cien personas han escrito antes, y obtener una implementación sólida y bien probado con solo presionar algunas teclas.

El software es barato de copiar, por lo que una vez lo haya escrito alguien, distribuirlo a otras personas es un proceso eficiente. Pero escribirlo en el primer lugar, *es* trabajo y responder a las personas que han encontrado problemas en el código, o que quieren proponer nuevas características, es aún más trabajo.

Por defecto, tu posees el copyright del código que escribes, y otras personas solo pueden usarlo con tu permiso. Pero ya que algunas personas son simplemente agradables, y porque la publicación de un buen software puede ayudarte a hacerte un poco famoso entre los programadores, se publican muchos paquetes bajo una licencia que explícitamente permite a otras personas usarlos.

La mayoría del código en NPM esta licenciado de esta manera. Algunas licencias requieren que tu publiques también el código bajo la misma licencia del paquete que estas usando. Otras son menos exigentes, solo requieren que guardes la licencia con el código cuando lo distribuyas. La comunidad de JavaScript principalmente usa ese último tipo de licencia. Al usar paquetes de otras personas, asegúrete de conocer su licencia.

Módulos improvisados

Hasta 2015, el lenguaje JavaScript no tenía un sistema de módulos incorporado. Sin embargo, la gente había estado construyendo sistemas grandes en

JavaScript durante más de una década y ellos *necesitaban* módulos.

Así que diseñaron sus propios sistema de módulos arriba del lenguaje. Puedes usar funciones de JavaScript para crear alcances locales, y objetos para representar las interfaces de los módulos.

Este es un módulo para ir entre los nombres de los días y números (como son retornados por el método `getDay` de `Date`). Su interfaz consiste en `diaDeLaSemana.nombre` y `diaDeLaSemana.numero`, y oculta su vinculación local `nombres` dentro del alcance de una expresión de función que se invoca inmediatamente.

```
const diaDeLaSemana = function() {
  const nombres = ["Domingo", "Lunes", "Martes", "Miercoles",
                   "Jueves", "Viernes", "Sabado"];
  return {
    nombre(numero) { return nombres[numero]; },
    numero(nombre) { return nombres.indexOf(nombre); }
  };
}();

console.log(diaDeLaSemana.nombre(diaDeLaSemana.numero("Domingo")));
// → Domingo
```

Este estilo de módulos proporciona aislamiento, hasta cierto punto, pero no declara dependencias. En cambio, simplemente pone su interfaz en el alcance global y espera que sus dependencias, si hay alguna, hagan lo mismo. Durante mucho tiempo, este fue el enfoque principal utilizado en la programación web, pero ahora está mayormente obsoleto.

Si queremos hacer que las relaciones de dependencia sean parte del código, tendremos que tomar el control de las dependencias que deben ser cargadas. Hacer eso requiere que seamos capaces de ejecutar strings como código. JavaScript puede hacer esto.

Evaluando datos como código

Hay varias maneras de tomar datos (un string de código) y ejecutarlos como una parte del programa actual.

La forma más obvia es usar el operador especial `eval`, que ejecuta un string en el alcance *actual*. Esto usualmente es una mala idea porque rompe algunas de las propiedades que normalmente tienen los alcances, tal como fácilmente predecir a qué vinculación se refiere un nombre dado.

```
const x = 1;
```

```
function evaluarYRetornarX(codigo) {
  eval(codigo);
  return x;
}

console.log(evaluarYRetornarX("var x = 2"));
// → 2
```

Una forma menos aterradora de interpretar datos como código es usar el constructor `Function`. Este toma dos argumentos: un string que contiene una lista de nombres de argumentos separados por comas y un string que contiene el cuerpo de la función.

```
let masUno = Function("n", "return n + 1;");
console.log(masUno(4));
// → 5
```

Esto es precisamente lo que necesitamos para un sistema de módulos. Podemos envolver el código del módulo en una función y usar el alcance de esa función como el alcance del módulo.

CommonJS

El enfoque más utilizado para incluir módulos en JavaScript es llamado *módulos CommonJS*. Node.js lo usa, y es el sistema utilizado por la mayoría de los paquetes en NPM.

El concepto principal en los módulos CommonJS es una función llamada `require` ("requerir"). Cuando la llamas con el nombre del módulo de una dependencia, esta se asegura de que el módulo sea cargado y retorna su interfaz.

Debido a que el cargador envuelve el código del módulo en una función, los módulos obtienen automáticamente su propio alcance local. Todo lo que tienen que hacer es llamar a `require` para acceder a sus dependencias, y poner su interfaz en el objeto vinculado a `exports` ("exportaciones").

Este módulo de ejemplo proporciona una función de formateo de fecha. Utiliza dos paquetes de NPM—`ordinal` para convertir números a strings como `"1st"` y `"2nd"`, y `date-names` para obtener los nombres en inglés de los días de la semana y meses. Este exporta una sola función, `formatDate`, que toma un objeto `Date` y un string plantilla.

El string de plantilla puede contener códigos que dirigen el formato, como `YYYY` para todo el año y `Do` para el día ordinal del mes. Podrías darle un string

como "MMMM Do YYYY" para obtener resultados como “November 22nd 2017”.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

La interfaz de `ordinal` es una función única, mientras que `date-names` exporta un objeto que contiene varias cosas—los dos valores que usamos son arrays de nombres. La desestructuración es muy conveniente cuando creamos vinculaciones para interfaces importadas.

El módulo agrega su función de interfaz a `exports`, de modo que los módulos que dependen de el tengan acceso a el. Podríamos usar el módulo de esta manera:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

Podemos definir `require`, en su forma más mínima, así:

```
require.cache = Object.create(null);

function require(nombre) {
  if (!(nombre in require.cache)) {
    let codigo = leerArchivo(nombre);
    let modulo = {exportaciones: {}};
    require.cache[nombre] = modulo;
    let envolvedor = Function("require, exportaciones, modulo",
       codigo);
    envolvedor(require, modulo.exportaciones, modulo);
  }
  return require.cache[nombre].exportaciones;
```

```
}
```

En este código, `leerArchivo` es una función inventada que lee un archivo y retorna su contenido como un string. El estándar de JavaScript no ofrece tal funcionalidad—pero diferentes entornos de JavaScript, como el navegador y Node.js, proporcionan sus propias formas de acceder a archivos. El ejemplo solo pretende que `leerArchivo` existe.

Para evitar cargar el mismo módulo varias veces, `require` mantiene un (caché) almacenado de módulos que ya han sido cargados. Cuando se llama, primero verifica si el módulo solicitado ya ha sido cargado y, si no, lo carga. Esto implica leer el código del módulo, envolverlo en una función y llamárla.

La interfaz del paquete `ordinal` que vimos antes no es un objeto, sino una función. Una peculiaridad de los módulos CommonJS es que, aunque el sistema de módulos creará un objeto de interfaz vacío para ti (vinculado a `exports`), puedes reemplazarlo con cualquier valor al sobrescribir `module.exports`. Esto lo hacen muchos módulos para exportar un valor único en lugar de un objeto de interfaz.

Al definir `require`, `exportaciones` y `modulo` como parametros para la función de envoltura generada (y pasando los valores apropiados al llamarla), el cargador se asegura de que estas vinculaciones esten disponibles en el alcance del módulo.

La forma en que el string dado a `require` se traduce a un nombre de archivo real o dirección web difiere en diferentes sistemas. Cuando comienza con `"./"` o `"../"`, generalmente se interpreta como relativo al nombre del archivo actual. Entonces `"./format-date"` sería el archivo llamado `format-date.js` en el mismo directorio.

Cuando el nombre no es relativo, Node.js buscará por un paquete instalado con ese nombre. En el código de ejemplo de este capítulo, interpretaremos esos nombres como referencias a paquetes de NPM. Entraremos en más detalles sobre cómo instalar y usar los módulos de NPM en el Capítulo 20.

Ahora, en lugar de escribir nuestro propio analizador de archivos INI, podemos usar uno de NPM:

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

Módulos ECMAScript

Los módulos CommonJS funcionan bastante bien y, en combinación con NPM, han permitido que la comunidad de JavaScript comience a compartir código en una gran escala.

Pero siguen siendo un poco de un truco con cinta adhesiva. La notación es ligeramente incomoda—las cosas que agregas a `exports` no están disponibles en el alcance local, por ejemplo. Y ya que `require` es una llamada de función normal tomando cualquier tipo de argumento, no solo un string literal, puede ser difícil de determinar las dependencias de un módulo sin correr su código primero.

Esta es la razón por la cual el estándar de JavaScript introdujo su propio, sistema de módulos diferente a partir de 2015. Por lo general es llamado *módulos ES*, donde *ES* significa ECMAScript. Los principales conceptos de dependencias e interfaces siguen siendo los mismos, pero los detalles difieren. Por un lado, la notación está ahora integrada en el lenguaje. En lugar de llamar a una función para acceder a una dependencia, utilizas una palabra clave `import` ("importar") especial.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

Similarmente, la palabra clave `export` se usa para exportar cosas. Puede aparecer delante de una función, clase o definición de vinculación (`let`, `const`, o `var`).

La interfaz de un módulo ES no es un valor único, sino un conjunto de vinculaciones con nombres. El módulo anterior vincula `formatDate` a una función. Cuando importas desde otro módulo, importas la *vinculación*, no el valor, lo que significa que un módulo exportado puede cambiar el valor de la vinculación en cualquier momento, y que los módulos que la importen verán su nuevo valor.

Cuando hay una vinculación llamada `default`, esta se trata como el principal valor del módulo exportado. Si importas un módulo como `ordinal` en el ejemplo, sin llaves alrededor del nombre de la vinculación, obtienes su vinculación `default`. Dichos módulos aún pueden exportar otras vinculaciones bajo diferentes nombres ademas de su exportación por `default`.

Para crear una exportación por default, escribe `export default` antes de una expresión, una declaración de función o una declaración de clase.

```
export default ["Invierno", "Primavera", "Verano", "Otoño"];
```

Es posible renombrar la vinculación importada usando la palabra `as` ("como").

```
import {days as nombresDias} from "date-names";

console.log(nombresDias.length);
// → 7
```

Al momento de escribir esto, la comunidad de JavaScript está en proceso de adoptar este estilo de módulos. Pero ha sido un proceso lento. Tomó algunos años, después de que se haya especificado el formato paraq que los navegadores y Node.js comenzaran a soportarlo. Y a pesar de que lo soportan mayormente ahora, este soporte todavía tiene problemas, y la discusión sobre cómo dichos módulos deberían distribuirse a través de NPM todavía está en curso.

Muchos proyectos se escriben usando módulos ES y luego se convierten automáticamente a algún otro formato cuando son publicados. Estamos en período de transición en el que se utilizan dos sistemas de módulos diferentes uno al lado del otro, y es útil poder leer y escribir código en cualquiera de ellos.

Construyendo y empaquetando

De hecho, muchos proyectos de JavaScript ni siquiera están, técnicamente, escritos en JavaScript. Hay extensiones, como el dialecto de comprobación de tipos mencionado en el Capítulo 7, que son ampliamente usados. Las personas también suelen comenzar a usar extensiones planificadas para el lenguaje mucho antes de que estas hayan sido agregadas a las plataformas que realmente corren JavaScript.

Para que esto sea posible, ellos *compilan* su código, traduciéndolo del dialecto de JavaScript que eligieron a JavaScript simple y antiguo—o incluso a una versión anterior de JavaScript, para que navegadores antiguos puedan ejecutarlo.

Incluir un programa modular que consiste de 200 archivos diferentes en una página web produce sus propios problemas. Si buscar un solo archivo sobre la red tarda 50 milisegundos, cargar todo el programa tardaria 10 segundos, o tal vez la mitad si puedes cargar varios archivos simultáneamente. Eso es mucho tiempo perdido. Ya que buscar un solo archivo grande tiende a ser más rápido que buscar muchos archivos pequeños, los programadores web han comenzado a usar herramientas que convierten sus programas (los cuales cuidadosamente

estan dividos en módulos) de nuevo en un único archivo grande antes de publicarlo en la Web. Tales herramientas son llamado *empaquetadores*.

Y podemos ir más allá. Además de la cantidad de archivos, el *tamaño* de los archivos también determina qué tan rápido se pueden transferir a través de la red. Por lo tanto, la comunidad de JavaScript ha inventado *minificadores*. Estas son herramientas que toman un programa de JavaScript y lo hacen más pequeño al eliminar automáticamente los comentarios y espacios en blanco, cambia el nombre de las vinculaciones, y reemplaza piezas de código con código equivalente que ocupa menos espacio.

Por lo tanto, no es raro que el código que encuentres en un paquete de NPM o que se ejecute en una página web haya pasado por *multiples* etapas de transformación: conversión de JavaScript moderno a JavaScript histórico, del formato de módulos ES a CommonJS, empaquetado y minificado. No vamos a entrar en los detalles de estas herramientas en este libro, ya que tienden a ser aburridos y cambian rápidamente. Solo ten en cuenta que el código JavaScript que ejecutas a menudo no es el código tal y como fue escrito.

Diseño de módulos

La estructuración de programas es uno de los aspectos más sutiles de la programación. Cualquier pieza de funcionalidad no trivial se puede modelar de varias maneras.

Un buen diseño de programa es subjetivo—hay ventajas/desventajas involucradas, y cuestiones de gusto. La mejor manera de aprender el valor de una buena estructura de diseño es leer o trabajar en muchos programas y notar lo que funciona y lo qué no. No asumas que un desastroso doloroso es “solo la forma en que las cosas son ". Puedes mejorar la estructura de casi todo al ponerle mas pensamiento.

Un aspecto del diseño de módulos es la facilidad de uso. Si estás diseñando algo que está destinado a ser utilizado por varias personas—o incluso por ti mismo, en tres meses cuando ya no recuerdes los detalles de lo que hiciste—es útil si tu interfaz es simple y predicible.

Eso puede significar seguir convenciones existentes. Un buen ejemplo es el paquete `ini`. Este módulo imita el objeto estándar `JSON` al proporcionar las funciones `parse` y `stringify` (para escribir un archivo INI), y, como `JSON` , convierte entre strings y objetos simples. Entonces la interfaz es pequeña y familiar, y después de haber trabajado con ella una vez, es probable que recuerdes cómo usarla.

Incluso si no hay una función estándar o un paquete ampliamente utilizado

para imitar, puedes mantener tus módulos predecibles mediante el uso de estructuras de datos simples y haciendo una cosa única y enfocada. Muchos de los módulos de análisis de archivos INI en NPM proporcionan una función que lee directamente tal archivo del disco duro y lo analiza, por ejemplo. Esto hace que sea imposible de usar tales módulos en el navegador, donde no tenemos acceso directo al sistema de archivos, y agrega una complejidad que habría sido mejor abordada al *componer* el módulo con alguna función de lectura de archivos.

Lo que apunta a otro aspecto útil del diseño de módulos—la facilidad con la qué algo se puede componer con otro código. Módulos enfocados que que computan valores son aplicables en una gama más amplia de programas que módulos mas grandes que realizan acciones complicadas con efectos secundarios. Un lector de archivos INI que insista en leer el archivo desde el disco es inútil en un escenario donde el contenido del archivo provenga de alguna otra fuente.

Relacionadamente, los objetos con estado son a veces útiles e incluso necesarios, pero si se puede hacer algo con una función, usa una función. Varios de los lectores de archivos INI en NPM proporcionan un estilo de interfaz que requiere que primero debes crear un objeto, luego cargar el archivo en tu objeto, y finalmente usar métodos especializados para obtener los resultados. Este tipo de cosas es común en la tradición orientada a objetos, y es terrible. En lugar de hacer una sola llamada de función y seguir adelante, tienes que realizar el ritual de mover tu objeto a través de diversos estados. Y ya que los datos ahora están envueltos en un objeto de tipo especializado, todo el código que interactúa con él tiene que saber sobre ese tipo, creando interdependencias innecesarias.

A menudo no se puede evitar la definición de nuevas estructuras de datos—solo unas pocas básicas son provistos por el estándar de lenguaje, y muchos tipos de datos tienen que ser más complejos que un array o un mapa. Pero cuando el array es suficiente, usa un array.

Un ejemplo de una estructura de datos un poco más compleja es el grafo de el Capítulo 7. No hay una sola manera obvia de representar un grafo en JavaScript. En ese capítulo, usamos un objeto cuya propiedades contenian arrays de strings—los otros nodos accesibles desde ese nodo.

Hay varios paquetes de busqueda de rutas diferentes en NPM, pero ninguno de ellos usa este formato de grafo. Por lo general, estos permiten que los bordes del grafo tengan un peso, el costo o la distancia asociada a ellos, lo que no es posible en nuestra representación.

Por ejemplo, está el paquete `dijkstrajs`. Un enfoque bien conocido par la busqueda de rutas, bastante similar a nuestra función `encontrarRuta`, se llama el *algoritmo de Dijkstra*, después de Edsger Dijkstra, quien fue el primero que

lo escribió. El sufijo `js` a menudo se agrega a los nombres de los paquetes para indicar el hecho de que están escritos en JavaScript. Este paquete `dijkstrajs` utiliza un formato de grafo similar al nuestro, pero en lugar de arrays, utiliza objetos cuyos valores de propiedad son números—los pesos de los bordes.

Si quisiéramos usar ese paquete, tendríamos que asegurarnos de que nuestro grafo fue almacenado en el formato que este espera.

```
const {find_path} = require("dijkstrajs");

let grafo = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(grafo, "Oficina de Correos", "Cabaña"));
// → ["Oficina de Correos", "Casa de Alice", "Cabaña"]
```

Esto puede ser una barrera para la composición—cuando varios paquetes están usando diferentes estructuras de datos para describir cosas similares, combinarlos es difícil. Por lo tanto, si deseas diseñar para la compibilidad, averigua qué estructura de datos están usando otras personas y, cuando sea posible, sigue su ejemplo.

Resumen

Los módulos proporcionan de estructura a programas más grandes al separar el código en piezas con interfaces y dependencias claras. La interfaz es la parte del módulo que es visible desde otros módulos, y las dependencias son los otros módulos este que utiliza.

Debido a que históricamente JavaScript no proporcionó un sistema de módulos, el sistema CommonJS fue construido encima. Entonces, en algún momento, *consiguio* un sistema incorporado, que ahora coexiste incomodamente con el sistema CommonJS.

Un paquete es una porción de código que se puede distribuir por sí misma. NPM es un repositorio de paquetes de JavaScript. Puedes descargar todo tipo de paquetes útiles (e inútiles) de él.

Ejercicios

Un robot modular

Estas son las vinculaciones que el proyecto del Capítulo 7 crea:

```
caminos
construirGrafo
grafoCamino
EstadoPueblo
correrRobot
eleccionAleatoria
robotAleatorio
rutaCorreo
robotRuta
encontrarRuta
robotOrientadoAMetas
```

Si tuvieras que escribir ese proyecto como un programa modular, qué módulos crearías? Qué módulo dependería de qué otro módulo, y cómo se verían sus interfaces?

Qué piezas es probable que estén disponibles pre-escritas en NPM? Preferirias usar un paquete de NPM o escribirlas tu mismo?

Módulo de Caminos

Escribe un módulo CommonJS, basado en el ejemplo del Capítulo 7, que contenga el array de caminos y exporte la estructura de datos grafo que los representa como `grafoCamino`. Debería depender de un modulo `./grafo`, que exporta una función `construirGrafo` que se usa para construir el grafo. Esta función espera un array de arrays de dos elementos (los puntos de inicio y final de los caminos).

Dependencias circulares

Una dependencia circular es una situación en donde el módulo A depende de B, y B también, directa o indirectamente, depende de A. Muchos sistemas de módulos simplemente prohíbne esto porque cualquiera que sea el orden que elijas para cargar tales módulos, no puedes asegurarse de que las dependencias de cada módulo han sido cargadas antes de que se ejecuten.

Los modulos CommonJS permiten una forma limitada de dependencias cíclicas. Siempre que los módulos no reemplacen a su objeto `exports` predetermi-

nado, y no accedan a la interfaz de las demás hasta que terminen de cargar, las dependencias cíclicas están bien.

La función `require` dada anteriormente en este capítulo es compatible con este tipo de ciclo de dependencias. Puedes ver cómo maneja los ciclos? Qué iría mal cuando un módulo en un ciclo *reemplace* su objeto `exports` por defecto?

"Quién puede esperar tranquilamente mientras el barro se asienta?
Quién puede permanecer en calma hasta el momento de actuar?"
—Laozi, Tao Te Ching

Chapter 11

Programación Asincrónica

La parte central de una computadora, la parte que lleva a cabo los pasos individuales que componen nuestros programas, es llamada *procesador*. Los programas que hemos visto hasta ahora son cosas que mantienen al procesador ocupado hasta que hayan terminado su trabajo. La velocidad a la que algo como un ciclo que manipule números pueda ser ejecutado, depende casi completamente de la velocidad del procesador.

Pero muchos programas interactúan con cosas fuera del procesador. por ejemplo, podrian comunicarse a través de una red de computadoras o solicitar datos del disco duro—lo que es mucho más lento que obtenerlos desde la memoria.

Cuando una cosa como tal este sucediendo, sería una pena dejar que el procesador se mantenga inactivo—podría haber algún otro trabajo que este pueda hacer en el mientras tanto. En parte, esto es manejado por tu sistema operativo, que cambiará el procesador entre múltiples programas en ejecución. Pero eso no ayuda cuando queremos que un *unico* programa pueda hacer progreso mientras este espera una solicitud de red.

Asincronicidad

En un modelo de programación *sincrónico*, las cosas suceden una a la vez. Cuando llamas a una función que realiza una acción de larga duración, solo retorna cuando la acción ha terminado y puede retornar el resultado. Esto detiene tu programa durante el tiempo que tome la acción.

Un modelo *asincrónico* permite que ocurran varias cosas al mismo tiempo. Cuando comienzas una acción, tu programa continúa ejecutándose. Cuando la acción termina, el programa es informado y tiene acceso al resultado (por ejemplo, los datos leídos del disco).

Podemos comparar a la programación síncrona y asincrónica usando un pequeño ejemplo: un programa que obtiene dos recursos de la red y luego combina resultados.

En un entorno síncrono, donde la función de solicitud solo retorna una vez que ha hecho su trabajo, la forma más fácil de realizar esta tarea es realizar las solicitudes una después de la otra. Esto tiene el inconveniente de que la segunda solicitud se iniciará solo cuando la primera haya finalizado. El tiempo total de ejecución será como minimo la suma de los dos tiempos de respuesta.

La solución a este problema, en un sistema síncrono, es comenzar hilos adicionales de control. Un *hilo* es otro programa activo cuya ejecución puede ser intercalada con otros programas por el sistema operativo—ya que la mayoría de las computadoras modernas contienen múltiples procesadores, múltiples hilos pueden incluso ejecutarse al mismo tiempo, en diferentes procesadores. Un segundo hilo podría iniciar la segunda solicitud, y luego ambos subprocesos esperan a que los resultados vuelvan, después de lo cual se vuelven a resincronizar para combinar sus resultados.

En el siguiente diagrama, las líneas gruesas representan el tiempo que el programa pasa corriendo normalmente, y las líneas finas representan el tiempo pasado esperando la red. En el modelo síncrono, el tiempo empleado por la red es *parte* de la línea de tiempo para un hilo de control dado. En el modelo asincrónico, comenzar una acción de red conceptualmente causa una *división* en la línea del tiempo. El programa que inició la acción continúa ejecutándose, y la acción ocurre junto a el, notificando al programa cuando está termina.

Otra forma de describir la diferencia es que esperar que las acciones terminen es *implicito* en el modelo síncrono, mientras que es *explicito*, bajo nuestro control, en el asincrónico.

La asincronicidad corta en ambos sentidos. Hace que expresar programas que hagan algo no se ajuste al modelo de control lineal más fácil, pero también puede hacer que expresar programas que siguen una línea recta sea más incómodo. Veremos algunas formas de abordar esta incomodidad más adelante en el capítulo.

Ambas de las plataformas de programación JavaScript importantes—navegadores y Node.js—realizan operaciones que pueden tomar un tiempo asincrónicamente, en lugar de confiar en hilos. Dado que la programación con hilos es notoria-

mente difícil (entender lo que hace un programa es mucho más difícil cuando está haciendo varias cosas a la vez), esto es generalmente considerado una buena cosa.

Tecnología cuervo

La mayoría de las personas son conscientes del hecho de que los cuervos son pájaros muy inteligentes. Pueden usar herramientas, planear con anticipación, recordar cosas e incluso comunicarse estas cosas entre ellos.

Lo que la mayoría de la gente no sabe, es que son capaces de hacer muchas cosas que mantienen bien escondidas de nosotros. Personas de buena reputación (un tanto excéntricas) expertas en córvidos, me han dicho que la tecnología cuervo no esta muy por detrás de la tecnología humana, y que nos estan alcanzando.

Por ejemplo, muchas culturas cuervo tienen la capacidad de construir dispositivos informáticos. Estos no son electrónicos, como lo son los dispositivos informáticos humanos, pero operan a través de las acciones de pequeños insectos, una especie estrechamente relacionada con las termitas, que ha desarrollado una relación simbiótica con los cuervos. Los pájaros les proporcionan comida, y a cambio los insectos construyen y operan sus complejas colonias que, con la ayuda de las criaturas vivientes dentro de ellos, realizan computaciones.

Tales colonias generalmente se encuentran en nidos grandes de larga vida. Las aves e insectos trabajan juntos para construir una red de estructuras bulbosas hechas de arcilla, escondidas entre las ramitas del nido, en el que los insectos viven y trabajan.

Para comunicarse con otros dispositivos, estas máquinas usan señales de luz. Los cuervos incrustan piezas de material reflectante en tallos de comunicación especial, y los insectos apuntan estos para reflejar la luz hacia otro nido, codificando los datos como una secuencia de flashes rápidos. Esto significa que solo los nidos que tienen una conexión visual ininterrumpida pueden comunicarse entre ellos.

Nuestro amigo, el experto en córvidos, ha mapeado la red de nidos de cuervo en el pueblo de Hières-sur-Amby, a orillas del río Ródano. Este mapa muestra los nidos y sus conexiones.

En un ejemplo asombroso de evolución convergente, las computadoras cuervo ejecutan JavaScript. En este capítulo vamos a escribir algunas funciones de redes básicas para ellos.

Devolución de llamadas

Un enfoque para la programación asincrónica es hacer que las funciones que realizan una acción lenta, tomen un argumento adicional, una *función de devolución de llamada*. La acción se inicia y, cuando esta finaliza, la función de devolución es llamada con el resultado.

Como ejemplo, la función `setTimeout`, disponible tanto en Node.js como en navegadores, espera una cantidad determinada de milisegundos (un segundo son mil milisegundos) y luego llama una función.

```
setTimeout(() => console.log("Tick"), 500);
```

Esperar no es generalmente un tipo de trabajo muy importante, pero puede ser útil cuando se hace algo como actualizar una animación o verificar si algo está tardando más que una cantidad dada de tiempo.

La realización de múltiples acciones asíncronas en una fila utilizando devoluciones de llamada significa que debes seguir pasando nuevas funciones para manejar la continuación de la computación después de las acciones.

La mayoría de las computadoras en los nidos de los cuervos tienen un bulbo de almacenamiento de datos a largo plazo, donde las piezas de información se graban en ramitas para que estas puedan ser recuperadas más tarde. Grabar o encontrar un fragmento de información requiere un momento, por lo que la interfaz para el almacenamiento a largo plazo es asíncrona y utiliza funciones de devolución de llamada.

Los bulbos de almacenamiento almacenan piezas de JSON-datos codificables

bajo nombres. Un cuervo podría almacenar información sobre los lugares donde hay comida escondida bajo el nombre `"caches de alimentos"`, que podría contener un array de nombres que apuntan a otros datos, que describen el caché real. Para buscar un caché de alimento en los bulbos de almacenamiento del nido *Gran Roble*, un cuervo podría ejecutar código como este:

```
import {granRoble} from "./tecnologia-cuervo";

granRoble.leerAlmacenamiento("caches de alimentos", caches => {
  let primerCache = caches[0];
  granRoble.leerAlmacenamiento(primerCache, informacion => {
    console.log(informacion);
  });
});
```

(Todos los nombres de las vinculaciones y los strings se han traducido del lenguaje cuervo a Español.)

Este estilo de programación es viable, pero el nivel de indentación aumenta con cada acción asincrónica, ya que terminas en otra función. Hacer cosas más complicadas, como ejecutar múltiples acciones al mismo tiempo, puede ser un poco incómodo.

Las computadoras cuervo están construidas para comunicarse usando pares de solicitud-respuesta. Eso significa que un nido envía un mensaje a otro nido, el cual inmediatamente envía un mensaje de vuelta, confirmando el recibo y, posiblemente, incluyendo una respuesta a una pregunta formulada en el mensaje.

Cada mensaje está etiquetado con un *tipo*, que determina cómo este es manejado. Nuestro código puede definir manejadores para tipos de solicitud específicos, y cuando se recibe una solicitud de este tipo, se llama al controlador para que este produzca una respuesta.

La interfaz exportada por el módulo `"./tecnologia-cuervo"` proporciona funciones de devolución de llamada para la comunicación. Los nidos tienen un método `enviar` que envía una solicitud. Este espera el nombre del nido objetivo, el tipo de solicitud y el contenido de la solicitud como sus primeros tres argumentos, y una función a llamar cuando llega una respuesta como su cuarto y último argumento.

```
granRoble.send("Pastura de Vacas", "nota", "Vamos a graznar fuerte a
    las 7PM",
            () => console.log("Nota entregada."));
```

Pero para hacer nidos capaces de recibir esa solicitud, primero tenemos que definir un tipo de solicitud llamado `"nota"`. El código que maneja las solicitudes debe ejecutarse no solo en este nido-computadora, sino en todos los nidos que puedan recibir mensajes de este tipo. Asumiremos que un cuervo sobrevuela e instala nuestro código controlador en todos los nidos.

```
import {definirTipoSolicitud} from "./tecnologia-cuervo";

definirTipoSolicitud("nota", (nido, contenido, fuente, listo) => {
  console.log(`${nido.nombre} recibio nota: ${contenido}`);
  listo();
});
```

La función `definirTipoSolicitud` define un nuevo tipo de solicitud. El ejemplo agrega soporte para solicitudes de tipo `"nota"`, que simplemente envían una nota a un nido dado. Nuestra implementación llama a `console.log` para que podamos verificar que la solicitud llegó. Los nidos tienen una propiedad `nombre` que contiene su nombre.

El cuarto argumento dado al controlador, `listo`, es una función de devolución de llamada que debe ser llamada cuando se finaliza con la solicitud. Si hubiesemos utilizado el valor de retorno del controlador como el valor de respuesta, eso significaria que un controlador de solicitud no puede realizar acciones asincrónicas por sí mismo. Una función que realiza trabajos asíncronos normalmente retorna antes de que el trabajo este hecho, habiendo arreglado que se llame una devolución de llamada cuando este completada. Entonces, necesitamos algún mecanismo asíncrono, en este caso, otra función de devolución de llamada—para indicar cuándo hay una respuesta disponible.

En cierto modo, la asincronía es *contagiosa*. Cualquier función que llame a una función que funcione asincrónicamente debe ser asíncrona en si misma, utilizando una devolución de llamada o algun mecanismo similar para entregar su resultado. Llamar devoluciones de llamada es algo más involucrado y propenso a errores que simplemente retornar un valor, por lo que necesitar estructurar grandes partes de tu programa de esa manera no es algo muy bueno.

Promesas

Trabajar con conceptos abstractos es a menudo más fácil cuando esos conceptos pueden ser representados por valores. En el caso de acciones asíncronas, podrías, en lugar de organizar a una función para que esta sea llamada en algún momento en el futuro, retornar un objeto que represente este evento en el

futuro.

Esto es para lo que es la clase estándar `Promise` ("Promesa"). Una *promesa* es una acción asíncrona que puede completarse en algún punto y producir un valor. Esta puede notificar a cualquier persona que esté interesada cuando su valor este disponible.

La forma más fácil de crear una promesa es llamando a `Promise.resolve` ("Promesa.resolver"). Esta función se asegura de que el valor que le des, sea envuelto en una promesa. Si ya es una promesa, simplemente es retornada—de lo contrario, obtienes una nueva promesa que termina de inmediato con tu valor como su resultado.

```
let quince = Promise.resolve(15);
quince.then(valor => console.log(`Obtuve ${valor}`));
// → Obtuve 15
```

Para obtener el resultado de una promesa, puede usar su método `then` ("entonces"). Este registra una (función de devolución de llamada) para que sea llamada cuando la promesa resuelva y produzca un valor. Puedes agregar múltiples devoluciones de llamada a una única promesa, y serán llamadas, incluso si las agregas después de que la promesa ya haya sido *resuelta* (terminada).

Pero eso no es todo lo que hace el método `then`. Este retorna otra promesa, que resuelve al valor que retorna la función del controlador o, si esa retorna una promesa, espera por esa promesa y luego resuelve su resultado.

Es útil pensar acerca de las promesas como dispositivos para mover valores a una realidad asincrónica. Un valor normal simplemente esta allí. Un valor prometido es un valor que *podría* ya estar allí o podría aparecer en algún momento en el futuro. Las computaciones definidas en términos de promesas actúan en tales valores envueltos y se ejecutan de forma asíncrona a medida los valores se vuelven disponibles.

Para crear una promesa, puedes usar `Promise` como un constructor. Tiene una interfaz algo extraña—el constructor espera una función como argumento, a la cual llama inmediatamente, pasando una función que puede usar para resolver la promesa. Funciona de esta manera, en lugar de, por ejemplo, con un método `resolve`, de modo que solo el código que creó la promesa pueda resolverla.

Así es como crearía una interfaz basada en promesas para la función `leerAlmacenamiento`
.

```
function almacenamiento(nido, nombre) {
  return new Promise(resolve => {
```

```
    nido.leerAlmacenamiento(nombre, resultado => resolve(resultado))
        ;
  });
}

almacenamiento(granRoble, "enemigos")
  .then(valor => console.log("Obtuve", valor));
```

Esta función asíncrona retorna un valor significativo. Esta es la principal ventaja de las promesas—simplifican el uso de funciones asincrónicas. En lugar de tener que pasar devoluciones de llamadas, las funciones basadas en promesas son similares a las normales: toman entradas como argumentos y retornan su resultado. La única diferencia es que la salida puede que no este disponible inmediatamente.

Fracaso

Las computaciones regulares en JavaScript pueden fallar lanzando una excepción. Las computaciones asincrónicas a menudo necesitan algo así. Una solicitud de red puede fallar, o algún código que sea parte de la computación asincrónica puede arrojar una excepción.

Uno de los problemas más urgentes con el estilo de devolución de llamadas en la programación asíncrona es que hace que sea extremadamente difícil asegurarte de que las fallas sean reportadas correctamente a las devoluciones de llamada.

Una convención ampliamente utilizada es que el primer argumento para la devolución de llamada es usado para indicar que la acción falló, y el segundo contiene el valor producido por la acción cuando tuvo éxito. Tales funciones de devolución de llamadas siempre deben verificar si recibieron una excepción, y asegurarse de que cualquier problema que causen, incluidas las excepciones lanzadas por las funciones que estas llaman, sean atrapadas y entregadas a la función correcta.

Las promesas hacen esto más fácil. Estas pueden ser resueltas (la acción termino con éxito) o rechazadas (esta falló). Los controladores de resolución (registrados con `then`) solo se llaman cuando la acción es exitosa, y los rechazos se propagan automáticamente a la nueva promesa que es retornada por `then`. Y cuando un controlador arroje una excepción, esto automáticamente hace que la promesa producida por su llamada `then` sea rechazada. Entonces, si cualquier elemento en una cadena de acciones asíncronas falla, el resultado de toda la

cadena se marca como rechazado, y no se llaman más manejadores despues del punto en donde falló.

Al igual que resolver una promesa proporciona un valor, rechazar una también proporciona uno, generalmente llamado la *razón* el rechazo. Cuando una excepción en una función de controlador provoca el rechazo, el valor de la excepción se usa como la razón. Del mismo modo, cuando un controlador retorna una promesa que es rechazada, ese rechazo fluye hacia la próxima promesa. Hay una función `Promise.reject` que crea una nueva promesa inmediatamente rechazada.

Para manejar explícitamente tales rechazos, las promesas tienen un método `catch` ("atraoar") que registra un controlador para que sea llamado cuando se rechaze la promesa, similar a cómo los manejadores `then` manejan la resolución normal. También es muy parecido a `then` en que retorna una nueva promesa, que se resuelve en el valor de la promesa original si esta se resuelve normalmente, y al resultado del controlador `catch` de lo contrario. Si un controlador `catch` lanza un error, la nueva promesa también es rechazada.

Como una abreviatura, `then` también acepta un manejador de rechazo como segundo argumento, por lo que puedes instalar ambos tipos de controladores en un solo método de llamada.

Una función que se pasa al constructor `Promise` recibe un segundo argumento, junto con la función de resolución, que puede usar para rechazar la nueva promesa.

Las cadenas de promesas creadas por llamadas a `then` y `catch` puede verse como una tubería a través de la cual los valores asíncronicos o las fallas se mueven. Dado que tales cadenas se crean mediante el registro de controladores, cada enlace tiene un controlador de éxito o un controlador de rechazo (o ambos) asociados a ello. Controladores que no coinciden con ese tipo de resultados (éxito o fracaso) son ignorados. Pero los que sí coinciden son llamados, y su resultado determina qué tipo de valor viene después—éxito cuando retorna un valor que no es una promesa, rechazo cuando arroja una excepción, y el resultado de una promesa cuando retorna una de esas.

Al igual que una excepción no detectada es manejada por el entorno, Los entornos de JavaScript pueden detectar cuándo una promesa rechazada no es manejada, y reportará esto como un error.

Las redes son difíciles

Ocasionalmente, no hay suficiente luz para los sistemas de espejos de los cuervos para transmitir una señal, o algo bloquea el camino de la señal. Es posible que

se envíe una señal, pero que nunca se reciba.

Tal y como es, eso solo causará que la devolución de llamada dada a `send` nunca sea llamada, lo que probablemente hará que el programa se detenga sin siquiera notar que hay un problema. Sería bueno si, después de un determinado período de no obtener una respuesta, una solicitud *expirará* e informara de un fracaso.

A menudo, las fallas de transmisión son accidentes aleatorios, como la luz del faro de un auto interfieriendo con las señales de luz, y simplemente volver a intentar la solicitud puede hacer que esta tenga éxito. Entonces, mientras estamos en eso, hagamos que nuestra función de solicitud automáticamente reintente el envío de la solicitud momentos antes de que se de por vencida.

Y, como hemos establecido que las promesas son algo bueno, tambien haremos que nuestra función de solicitud retorne una promesa. En términos de lo que pueden expresar, las devoluciones de llamada y las promesas son equivalentes. Las funciones basadas en devoluciones de llamadas se pueden envolver para exponer una interfaz basada en promesas, y viceversa.

Incluso cuando una solicitud y su respuesta sean entregadas exitosamente, la respuesta puede indicar un error—por ejemplo, si la solicitud intenta utilizar un tipo de solicitud que no haya sido definida o si el controlador genera un error. Para soportar esto, `send` y `definirTipoSolicitud` siguen la convención mencionada anteriormente, donde el primer argumento pasado a las devoluciones de llamada es el motivo del fallo, si lo hay, y el segundo es el resultado real.

Estos pueden ser traducidos para prometer resolución y rechazo por parte de nuestra envoltura.

```
class TiempoDeEspera extends Error {}

function request(nido, objetivo, tipo, contenido) {
  return new Promise((resolve, reject) => {
    let listo = false;
    function intentar(n) {
      nido.send(objetivo, tipo, contenido, (fallo, value) => {
        listo = true;
        if (fallo) reject(fallo);
        else resolve(value);
      });
      setTimeout(() => {
        if (listo) return;
        else if (n < 3) intentar(n + 1);
        else reject(new TiempoDeEspera("Tiempo de espera agotado"));
      }, 250);
```

```
    }
    intentar(1);
  });
}
```

Debido a que las promesas solo se pueden resolver (o rechazar) una vez, esto funcionara. La primera vez que se llame a `resolve` o `reject` se determinara el resultado de la promesa y cualquier llamada subsecuente, como el tiempo de espera que llega después de que finaliza la solicitud, o una solicitud que regresa después de que otra solicitud es finalizada, es ignorada.

Para construir un ciclo asincrónico, para los reintentos, necesitamos usar un función recursiva—un ciclo regular no nos permite detenernos y esperar por una acción asincrónica. La función `intentar` hace un solo intento de enviar una solicitud. También establece un tiempo de espera que, si no ha regresado una respuesta después de 250 milisegundos, comienza el próximo intento o, si este es el cuarto intento, rechaza la promesa con una instancia de `TiempoDeEspera` como la razón.

Volver a intentar cada cuarto de segundo y rendirse cuando no ha llegado ninguna respuesta después de un segundo es algo definitivamente arbitrario. Es incluso posible, si la solicitud llegó pero el controlador se esta tardando un poco más, que las solicitudes se entreguen varias veces. Escribiremos nuestros manejadores con ese problema en mente—los mensajes duplicados deberían de ser inofensivos.

En general, no construiremos una red robusta de clase mundial hoy. Pero eso esta bien—los cuervos no tienen expectativas muy altas todavía cuando se trata de la computación.

Para aislarnos por completo de las devoluciones de llamadas, seguiremos adelante y también definiremos un contenedor para `definirTipoSolicitud` que permite que la función controlador pueda retornar una promesa o valor normal, y envia eso hasta la devolución de llamada para nosotros.

```
function tipoSolicitud(nombre, manejador) {
  definirTipoSolicitud(nombre, (nido, contenido, fuente,
                                devolucionDeLlamada) => {
    try {
      Promise.resolve(manejador(nido, contenido, fuente))
        .then(response => devolucionDeLlamada(null, response),
              failure => devolucionDeLlamada(failure));
    } catch (exception) {
      devolucionDeLlamada(exception);
    }
```

```
  });
}
```

`Promise.resolve` se usa para convertir el valor retornado por `manejador` a una promesa si no es una ya.

Ten en cuenta que la llamada a `manejador` tenía que estar envuelta en un bloque `try`, para asegurarse de que cualquier excepción que aparezca directamente se le dé a la devolución de llamada. Esto ilustra muy bien la dificultad de manejar adecuadamente los errores con devoluciones de llamada crudas—es muy fácil olvidarse de encaminar correctamente excepciones como esa, y si no lo haces, las fallas no se seran informadas a la devolución de llamada correcta. Las promesas hacen esto casi automático, y por lo tanto, son menos propensas a errores.

Colecciones de promesas

Cada computadora nido mantiene un array de otros nidos dentro de la distancia de transmisión en su propiedad `vecinos`. Para verificar cuáles de esos son actualmente accesibles, puede escribir una función que intente enviar un solicitud `"ping"` (una solicitud que simplemente pregunta por una respuesta) para cada de ellos, y ver cuáles regresan.

Al trabajar con colecciones de promesas que se ejecutan al mismo tiempo, la función `Promise.all` puede ser útil. Esta retorna una promesa que espera a que se resuelvan todas las promesas del array, y luego resuelve un array de los valores que estas promesas produjeron (en el mismo orden que en el array original). Si alguna promesa es rechazada, el el resultado de `Promise.all` es en sí mismo rechazado.

```
tipoSolicitud("ping", () => "pong");

function vecinosDisponibles(nido) {
  let solicitudes = nido.vecinos.map(vecino => {
    return request(nido, vecino, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(solicitudes).then(resultado => {
    return nido.vecinos.filter((_, i) => resultado[i]);
  });
}
```

Cuando un vecino no este disponible, no queremos que todo la promesa combinada falle, dado que entonces no sabríamos nada. Entonces la función que es mappeada en el conjunto de vecinos para convertirlos en promesas de solicitud vincula a los controladores que hacen las solicitudes exitosas produzcan `true` y las rechazadas produzcan `false`.

En el controlador de la promesa combinada, `filter` se usa para eliminar esos elementos de la matriz `vecinos` cuyo valor correspondiente es falso. Esto hace uso del hecho de que `filter` pasa el índice de matriz del elemento actual como segundo argumento para su función de filtrado (`map`,`some`, y métodos similares de orden superior de arrays hacen lo mismo).

Inundación de red

El hecho de que los nidos solo pueden hablar con sus vecinos inhibe en gran cantidad la utilidad de esta red.

Para transmitir información a toda la red, una solución es configurar un tipo de solicitud que sea reenviada automáticamente a los vecinos. Estos vecinos luego la envían a sus vecinos, hasta que toda la red ha recibido el mensaje.

```
import {todosLados} from "./tecnologia-cuervo";

todosLados(nido => {
  nido.estado.chismorreo = [];
});

function enviarChismorreo(nido, mensaje, exceptoPor = null) {
  nido.estado.chismorreo.push(mensaje);
  for (let vecino of nido.vecinos) {
    if (vecino == exceptoPor) continue;
    request(nido, vecino, "chismorreo", mensaje);
  }
}

requestType("chismorreo", (nido, mensaje, fuente) => {
  if (nido.estado.chismorreo.includes(mensaje)) return;
  console.log(`${nido.nombre} recibio chismorreo '${
                mensaje}' de ${fuente}`);
  enviarChismorreo(nido, mensaje, fuente);
});
```

Para evitar enviar el mismo mensaje a traves de la red por siempre, cada nido mantiene un array de strings de chismorreos que ya ha visto. Para definir

este array, usaremos la función `todosLados`—que ejecuta código en todos los nidos—para añadir una propiedad al objeto `estado` del nido, que es donde mantendremos estado local del nido.

Cuando un nido recibe un mensaje de chisme duplicado, lo cual es muy probable que suceda con todo el mundo reenviando estos a ciegas, lo ignora. Pero cuando recibe un mensaje nuevo, emocionadamente le dice a todos sus vecinos a excepción de quien le envió el mensaje.

Esto provocará que una nueva pieza de chismes se propague a través de la red como una mancha de tinta en agua. Incluso cuando algunas conexiones no estan trabajando actualmente, si hay una ruta alternativa a un nido dado, el chisme llegará hasta allí.

Este estilo de comunicación de red se llama *inundamiento*-inunda la red con una pieza de información hasta que todos los nodos la tengan.

Enrutamiento de mensajes

Si un nodo determinado quiere hablar unicamente con otro nodo, la inundación no es un enfoque muy eficiente. Especialmente cuando la red es grande, daría lugar a una gran cantidad de transferencias de datos inútiles.

Un enfoque alternativo es configurar una manera en que los mensajes salten de nodo a nodo, hasta que lleguen a su destino. La dificultad con eso es que requiere de conocimiento sobre el diseño de la red. Para enviar una solicitud hacia la dirección de un nido lejano, es necesario saber qué nido vecino lo acerca más a su destino. Enviar la solicitud en la dirección equivocada no servirá de mucho.

Dado que cada nido solo conoce a sus vecinos directos, no tiene la información que necesita para calcular una ruta. De alguna manera debemos extender la información acerca de estas conexiones a todos los nidos. Preferiblemente en una manera que permita ser cambiada con el tiempo, cuando los nidos son abandonados o nuevos nidos son construidos.

Podemos usar la inundación de nuevo, pero en lugar de verificar si un determinado mensaje ya ha sido recibido, ahora verificamos si el nuevo conjunto de vecinos de un nido determinado coinciden con el conjunto actual que tenemos para él.

```
tipoSolicitud("conexiones", (nido, {nombre, vecinos},
                             fuente) => {
  let conexiones = nido.estado.conexiones;
  if (JSON.stringify(conexiones.get(nombre)) ==
      JSON.stringify(vecinos)) return;
```

```
  conexiones.set(nombre, vecinos);
  difundirConexiones(nido, nombre, fuente);
});

function difundirConexiones(nido, nombre, exceptoPor = null) {
  for (let vecino of nido.vecinos) {
    if (vecino == exceptoPor) continue;
    solicitud(nido, vecino, "conexiones", {
      nombre,
      vecinos: nido.estado.conexiones.get(nombre)
    });
  }
}

todosLados(nido => {
  nido.estado.conexiones = new Map;
  nido.estado.conexiones.set(nido.nombre, nido.vecinos);
  difundirConexiones(nido, nido.nombre);
});
```

La comparación usa `JSON.stringify` porque `==`, en objetos o arrays, solo retornara true cuando los dos tengan exactamente el mismo valor, lo cual no es lo que necesitamos aquí. Comparar los strings JSON es una cruda pero efectiva manera de comparar su contenido.

Los nodos comienzan inmediatamente a transmitir sus conexiones, lo que debería, a menos que algunos nidos sean completamente inalcanzables, dar rápidamente cada nido un mapa del grafo de la red actual.

Una cosa que puedes hacer con grafos es encontrar rutas en ellos, como vimos en el Capítulo 7. Si tenemos una ruta hacia el destino de un mensaje, sabemos en qué dirección enviarlo.

Esta función `encontrarRuta`, que se parece mucho a `encontrarRuta` del Capítulo 7, busca por una forma de llegar a un determinado nodo en la red. Pero en lugar de devolver toda la ruta, simplemente retorna el siguiente paso. Ese próximo nido en si mismo, usando su información actual sobre la red, decididira *hacia* dónde enviar el mensaje.

```
function encontrarRuta(desde, hasta, conexiones) {
  let trabajo = [{donde: desde, via: null}];
  for (let i = 0; i < trabajo.length; i++) {
    let {donde, via} = trabajo[i];
    for (let siguiente of conexiones.get(donde) || []) {
      if (siguiente == hasta) return via;
      if (!trabajo.some(w => w.donde == siguiente)) {
```

```
        trabajo.push({donde: siguiente, via: via || siguiente});
      }
    }
  }
  return null;
}
```

Ahora podemos construir una función que pueda enviar mensajes de larga distancia. Si el mensaje está dirigido a un vecino directo, se entrega normalmente. Si no, se empaqueta en un objeto y se envía a un vecino que este más cerca del objetivo, usando el tipo de solicitud `"ruta"`, que hace que ese vecino repita el mismo comportamiento.

```
function solicitudRuta(nido, objetivo, tipo, contenido) {
  if (nido.vecinos.includes(objetivo)) {
    return solicitud(nido, objetivo, tipo, contenido);
  } else {
    let via = encontrarRuta(nido.nombre, objetivo,
                            nido.estado.conexiones);
    if (!via) throw new Error(`No hay rutas disponibles hacia ${
       objetivo}`);
    return solicitud(nido, via, "ruta",
                     {objetivo, tipo, contenido});
  }
}

tipoSolicitud("ruta", (nido, {objetivo, tipo, contenido}) => {
  return solicitudRuta(nido, objetivo, tipo, contenido);
});
```

Hemos construido varias capas de funcionalidad sobre un sistema de comunicación primitivo para que sea conveniente de usarlo. Este es un buen (aunque simplificado) modelo de cómo las redes de computadoras reales trabajan.

Una propiedad distintiva de las redes de computadoras es que no son confiables—las abstracciones construidas encima de ellas pueden ayudar, pero no se puede abstraer la falla de una falla de red. Entonces la programación de redes es típicamente mucho acerca de anticipar y lidiar con fallas.

Funciones asíncronas

Para almacenar información importante, se sabe que los cuervos la duplican a través de los nidos. De esta forma, cuando un halcón destruye un nido, la información no se pierde.

Para obtener una pieza de información dada que no este en su propia bulbo de almacenamiento, una computadora nido puede consultar otros nidos al azar en la red hasta que encuentre uno que la tenga.

```
tipoSolicitud("almacenamiento", (nido, nombre) => almacenamiento(
   nido, nombre));

function encontrarEnAlmacenamiento(nido, nombre) {
  return almacenamiento(nido, nombre).then(encontrado => {
    if (encontrado != null) return encontrado;
    else return encontrarEnAlmacenamientoRemoto(nido, nombre);
  });
}

function red(nido) {
  return Array.from(nido.estado.conexiones.keys());
}

function encontrarEnAlmacenamientoRemoto(nido, nombre) {
  let fuentes = red(nido).filter(n => n != nido.nombre);
  function siguiente() {
    if (fuentes.length == 0) {
      return Promise.reject(new Error("No encontrado"));
    } else {
      let fuente = fuentes[Math.floor(Math.random() *
                                      fuentes.length)];
      fuentes = fuentes.filter(n => n != fuente);
      return solicitudRuta(nido, fuente, "almacenamiento", nombre)
        .then(valor => valor != null ? valor : siguiente(),
              siguiente);
    }
  }
  return siguiente();
}
```

Como `conexiones` es un `Map`, `Object.keys` no funciona en él. Este tiene un *metódo* `keys`, pero que retorna un iterador en lugar de un array. Un iterador (o valor iterable) se puede convertir a un array con la función `Array.from`.

Incluso con promesas, este es un código bastante incómodo. Múltiples ac-

ciones asincrónicas están encadenadas juntas de maneras no-obvias. Nosotros de nuevo necesitamos una función recursiva (`siguiente`) para modelar ciclos a través de nidos.

Y lo que el código realmente hace es completamente lineal—siempre espera a que se complete la acción anterior antes de comenzar la siguiente. En un modelo de programación sincrónica, sería más simple de expresar.

La buena noticia es que JavaScript te permite escribir código pseudo-sincrónico. Una función `async` es una función que retorna implícitamente una promesa y que puede, en su cuerpo, `await` ("esperar") otras promesas de una manera que *se ve* sincrónica.

Podemos reescribir `encontrarEnAlmacenamiento` de esta manera:

```
async function encontrarEnAlmacenamiento(nido, nombre) {
  let local = await almacenamiento(nido, nombre);
  if (local != null) return local;

  let fuentes = red(nido).filter(n => n != nido.nombre);
  while (fuentes.length > 0) {
    let fuente = fuentes[Math.floor(Math.random() *
                                    fuentes.length)];
    fuentes = fuentes.filter(n => n != fuente);
    try {
      let encontrado = await solicitudRuta(nido, fuente, "
         almacenamiento",
                                     nombre);
      if (encontrado != null) return encontrado;
    } catch (_) {}
  }
  throw new Error("No encontrado");
}
```

Una función `async` está marcada por la palabra `async` antes de la palabra clave `function`. Los métodos también pueden hacerse `async` al escribir `async` antes de su nombre. Cuando se llame a dicha función o método, este retorna una promesa. Tan pronto como el cuerpo retorne algo, esa promesa es resuelta Si arroja una excepción, la promesa es rechazada.

Dentro de una función `async`, la palabra `await` se puede poner delante de una expresión para esperar a que se resuelva una promesa, y solo entonces continua la ejecución de la función.

Tal función ya no se ejecuta, como una función regular de JavaScript de principio a fin de una sola vez. En su lugar, puede ser *congelada* en cualquier punto que tenga un `await`, y se reanuda en un momento posterior.

Para código asincrónico no-trivial, esta notación suele ser más conveniente que usar promesas directamente. Incluso si necesitas hacer algo que no se ajuste al modelo síncrono, como realizar múltiples acciones al mismo tiempo, es fácil combinar `await` con el uso directo de promesas.

Generadores

Esta capacidad de las funciones para pausar y luego reanudarse nuevamente no es exclusiva para las funciones `async`. JavaScript también tiene una caracteristica llamada funciones *generador*. Estss son similares, pero sin las promesas.

Cuando defines una función con `function*` (colocando un asterisco después de la palabra `function`), se convierte en un generador. Cuando llamas un generador, este retorna un iterador, que ya vimos en el Capítulo 6.

```
function* potenciacion(n) {
  for (let actual = n;; actual *= n) {
    yield actual;
  }
}

for (let potencia of potenciacion(3)) {
  if (potencia > 50) break;
  console.log(potencia);
}
// → 3
// → 9
// → 27
```

Inicialmente, cuando llamas a `potenciacion`, la función se congela en su comienzo. Cada vez que llames `next` en el iterador, la función se ejecuta hasta que encuentre una expresión `yield` ("arrojar"), que la pausa y causa que el valor arrojado se convierta en el siguiente valor producido por el iterador. Cuando la función retorne (la del ejemplo nunca lo hace), el iterador está completo.

Escribir iteradores es a menudo mucho más fácil cuando usas funciones generadoras. El iterador para la clase grupal (del ejercicio en el Capítulo 6) se puede escribir con este generador:

```
Conjunto.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.miembros.length; i++) {
    yield this.miembros[i];
  }
};
```

Ya no es necesario crear un objeto para mantener el estado de la iteración—los generadores guardan automáticamente su estado local cada vez ellos arrojen.

Dichas expresiones `yield` solo pueden ocurrir directamente en la función generadora en sí y no en una función interna que definas dentro de ella. El estado que ahorra un generador, cuando arroja, es solo su entorno *local* y la posición en la que fue arrojada.

Una función `async` es un tipo especial de generador. Produce una promesa cuando se llama, que se resuelve cuando vuelve (termina) y rechaza cuando arroja una excepción. Cuando cede (espera) por una promesa, el resultado de esa promesa (valor o excepción lanzada) es el resultado de la expresión `await`.

El ciclo de evento

Los programas asincrónicos son ejecutados pieza por pieza. Cada pieza puede iniciar algunas acciones y programar código para que se ejecute cuando la acción termine o falle. Entre estas piezas, el programa permanece inactivo, esperando por la siguiente acción.

Por lo tanto, las devoluciones de llamada no son llamadas directamente por el código que las programó. Si llamo a `setTimeout` desde adentro de una función, esa función habra retornado para el momento en que se llame a la función de devolución de llamada. Y cuando la devolución de llamada retorne, el control no volvera a la función que la programo.

El comportamiento asincrónico ocurre en su propia función de llamada de pila vacía. Esta es una de las razones por las cuales, sin promesas, la gestión de excepciones en el código asincrónico es dificil. Como cada devolución de llamada comienza con una pila en su mayoría vacía, tus manejadores `catch` no estarán en la pila cuando lanzen una excepción.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (_) {
  // Esto no se va a ejecutar
  console.log("Atrapado!");
}
```

No importa que tan cerca los eventos—como tiempos de espera o solicitudes entrantes—sucedan, un entorno de JavaScript solo ejecutará un programa a

la vez. Puedes pensar en esto como un gran ciclo *alrededor* de tu programa, llamado *ciclo de evento*. Cuando no hay nada que hacer, ese bucle está detenido. Pero a medida que los eventos entran, se agregan a una cola, y su código se ejecuta uno después del otro. Porque no hay dos cosas que se ejecuten al mismo tiempo, código de ejecución lenta puede retrasar el manejo de otros eventos.

Este ejemplo establece un tiempo de espera, pero luego se retrasa hasta después del tiempo de espera previsto, lo que hace que el tiempo de espera este tarde.

```
let comienzo = Date.now();
setTimeout(() => {
  console.log("Tiempo de espera corrio al ", Date.now() - comienzo);
}, 20);
while (Date.now() < comienzo + 50) {}
console.log("Se desperdicio tiempo hasta el ", Date.now() - comienzo
   );
// → Se desperdicio tiempo hasta el 50
// → Tiempo de espera corrio al 55
```

Las promesas siempre se resuelven o rechazan como un nuevo evento. Incluso si una promesa ya ha sido resuelta, esperar por ella hará que la devolución de llamada se ejecute después de que el script actual termine, en lugar de hacerlo inmediatamente.

```
Promise.resolve("Listo").then(console.log);
console.log("Yo primero!");
// → Yo primero!
// → Listo
```

En capítulos posteriores, veremos otros tipos de eventos que se ejecutan en el ciclo de eventos.

Errores asincrónicos

Cuando tu programa se ejecuta de forma síncrona, de una sola vez, no hay cambios de estado sucediendo aparte de aquellos que el mismo programa realiza. Para los programas asíncronos, esto es diferente—estos pueden tener *brechas* en su ejecución durante las cuales se podria ejecutar otro código.

Veamos un ejemplo. Uno de los pasatiempos de nuestros cuervos es contar la cantidad de polluelos que nacen en el pueblo cada año. Los nidos guardan este recuento en sus bulbos de almacenamiento. El siguiente código intenta

enumerar los recuentos de todos los nidos para un año determinado.

```
function cualquierAlmacenamiento(nido, fuente, nombre) {
  if (fuente == nido.nombre) return almacenamiento(nido, nombre);
  else return solicitudRuta(nido, fuente, "almacenamiento", nombre);
}

async function polluelos(nido, años) {
  let lista = "";
  await Promise.all(red(nido).map(async nombre => {
    lista += `${nombre}: ${
      await cualquierAlmacenamiento(nido, nombre, `polluelos en ${
         años}`)
    }\n`;
  }));
  return lista;
}
```

La parte `async nombre =>` muestra que las funciones de flecha también pueden ser `async` al poner la palabra `async` delante de ellas.

El código no parece sospechoso de inmediato... mapea la función de flecha `async` sobre el conjunto de nidos, creando una serie de promesas, y luego usa `Promise.all` para esperar a todos estas antes de retornar la lista que estas construyen.

Pero está seriamente roto. Siempre devolverá solo una línea de salida, enumerando al nido que fue más lento en responder.

Puedes averiguar por qué?

El problema radica en el operador `+=`, que toma el valor *actual* de `lista` en el momento en que la instrucción comienza a ejecutarse, y luego, cuando el `await` termina, establece que la vinculaciòn `lista` sea ese valor más el string agregado.

Pero entre el momento en el que la declaración comienza a ejecutarse y el momento donde termina hay una brecha asincrónica. La expresión `map` se ejecuta antes de que se haya agregado algo a la lista, por lo que cada uno de los operadores `+=` comienza desde un string vacío y termina cuando su recuperación de almacenamiento finaliza, estableciendo `lista` como una lista de una sola línea—el resultado de agregar su línea al string vacío.

Esto podría haberse evitado fácilmente retornando las líneas de las promesas mapeadas y llamando a `join` en el resultado de `Promise.all`, en lugar de construir la lista cambiando una vinculación. Como siempre, calcular nuevos valores es menos propenso a errores que cambiar valores existentes.

```
async function polluelos(nido, año) {
  let lineas = red(nido).map(async nombre => {
    return nombre + ": " +
      await cualquierAlmacenamiento(nido, nombre, `polluelos en ${
         año}`);
  });
  return (await Promise.all(lineas)).join("\n");
}
```

Errores como este son fáciles de hacer, especialmente cuando se usa `await`, y debes tener en cuenta dónde se producen las brechas en tu código. Una ventaja de la asincronicidad *explicita* de JavaScript (ya sea a través de devoluciones de llamada, promesas, o `await`) es que detectar estas brechas es relativamente fácil.

Resumen

La programación asincrónica permite expresar la espera de acciones de larga duración sin congelar el programa durante estas acciones. Los entornos de JavaScript suelen implementar este estilo de programación usando devoluciones de llamada, funciones que son llaman cuando las acciones son completadas. Un ciclo de eventos planifica que dichas devoluciones de llamadas sean llamadas cuando sea apropiado, una después de la otra, para que sus ejecuciones no se superpongan.

La programación asíncrona se hace más fácil mediante promesas, objetos que representar acciones que podrían completarse en el futuro, y funciones `async`, que te permiten escribir un programa asíncrono como si fuera sincrónico.

Ejercicios

Siguiendo el bisturí

Los cuervos del pueblo poseen un viejo bisturí que ocasionalmente usan en misiones especiales—por ejemplo, para cortar puertas de malla o embalar cosas. Para ser capaces de rastrearlo rápidamente, cada vez que se mueve el bisturí a otro nido, una entrada se agrega al almacenamiento tanto del nido que lo tenía como al nido que lo tomó, bajo el nombre `"bisturí"`, con su nueva ubicación como su valor.

Esto significa que encontrar el bisturí es una cuestión de seguir la ruta de navegación de las entradas de almacenamiento, hasta que encuentres un nido

que apunte a el nido en si mismo.

Escribe una función `async`, `localizarBisturi` que haga esto, comenzando en el nido en el que se ejecute. Puede usar la función `cualquierAlmacenamiento` definida anteriormente para acceder al almacenamiento en nidos arbitrarios. El bisturí ha estado dando vueltas el tiempo suficiente como para que puedas suponer que cada nido tiene una entrada `bisturí` en su almacenamiento de datos.

Luego, vuelve a escribir la misma función sin usar `async` y `await`.

Las fallas de solicitud se muestran correctamente como rechazos de la promesa devuelta en ambas versiones? Cómo?

Construyendo Promise.all

Dado un array de promesas, `Promise.all` retorna una promesa que espera a que finalicen todas las promesas del array. Entonces tiene éxito, produciendo un array de valores de resultados. Si una promesa en el array falla, la promesa retornada por `all` también falla, con la razón de la falla proveniente de la promesa fallida.

Implemente algo como esto tu mismo como una función regular llamada `Promise_all`.

Recuerda que una vez que una promesa ha tenido éxito o ha fallado, no puede tener éxito o fallar de nuevo, y llamadas subsecuentes a las funciones que resuelven son ignoradas. Esto puede simplificar la forma en que manejas la falla de tu promesa.

"El evaluador, que determina el significado de las expresiones en un lenguaje de programación, es solo otro programa."

—Hal Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs

Chapter 12

Proyecto: Un Lenguaje de Programación

Construir tu propio lenguaje de programación es sorprendentemente fácil (siempre y cuando no apuntes demasiado alto) y muy esclarecedor.

Lo principal que quiero mostrar en este capítulo es que no hay magia involucrada en la construcción de tu propio lenguaje. A menudo he sentido que algunos inventos humanos eran tan inmensamente inteligentes y complicados que nunca podría llegar a entenderlos. Pero con un poco de lectura y experimentación, a menudo resultan ser bastante mundanos.

Construiremos un lenguaje de programación llamado Egg. Será un lenguaje pequeño y simple—pero lo suficientemente poderoso como para expresar cualquier computación que puedes pensar. Permitirá una abstracción simple basada en funciones.

Análisis

La parte más visible de un lenguaje de programación es su *sintaxis*, o notación. Un *analizador* es un programa que lee una pieza de texto y produce una estructura de datos que refleja la estructura del programa contenido en ese texto. Si el texto no forma un programa válido, el analizador debe señalar el error.

Nuestro lenguaje tendrá una sintaxis simple y uniforme. Todo en Egg es una expresión. Una expresión puede ser el nombre de una vinculación (binding), un número, una cadena de texto o una *aplicación*. Las aplicaciones son usadas para llamadas de función pero también para constructos como `if` o `while`.

Para mantener el analizador simple, las cadenas en Egg no soportarán nada parecido a escapes de barra invertida. Una cadena es simplemente una secuencia de caracteres que no sean comillas dobles, envueltas en comillas dobles. Un número es un secuencia de dígitos. Los nombres de vinculaciones pueden consistir en cualquier carácter no que sea un espacio en blanco y eso no tiene un significado especial en la sintaxis.

Las aplicaciones se escriben tal y como están en JavaScript, poniendo paréntesis después de una expresión y teniendo cualquier cantidad de argumentos

entre esos paréntesis, separados por comas.

```
hacer(definir(x, 10),
   si(>(x, 5),
      imprimir("grande"),
      imprimir("pequeño")))
```

La uniformidad del lenguaje Egg significa que las cosas que son operadores en JavaScript (como `>`) son vinculaciones normales en este lenguaje, aplicadas como cualquier función. Y dado que la sintaxis no tiene un concepto de bloque, necesitamos una construcción `hacer` para representar el hecho de realizar múltiples cosas en secuencia.

La estructura de datos que el analizador usará para describir un programa consta de objetos de expresión, cada uno de los cuales tiene una propiedad `tipo` que indica el tipo de expresión que este es y otras propiedades que describen su contenido.

Las expresiones de tipo `"valor"` representan strings o números literales. Su propiedad `valor` contiene el string o valor numérico que estos representan. Las expresiones de tipo `"palabra"` se usan para identificadores (nombres). Dichos objetos tienen una propiedad `nombre` que contienen el nombre del identificador como un string. Finalmente, las expresiones `"aplicar"` representan aplicaciones. Tienen una propiedad `operador` que se refiere a la expresión que está siendo aplicada, y una propiedad `argumentos` que contiene un array de expresiones de argumentos.

La parte `>(x, 5)` del programa anterior se representaría de esta manera:

```
{
  tipo: "aplicar",
  operador: {tipo: "palabra", nombre: ">"},
  argumentos: [
    {tipo: "palabra", nombre: "x"},
    {tipo: "valor", valor: 5}
  ]
}
```

Tal estructura de datos se llama *árbol de sintaxis*. Si tu imaginas los objetos como puntos y los enlaces entre ellos como líneas entre esos puntos, tiene una forma similar a un árbol. El hecho de que las expresiones contienen otras expresiones, que a su vez pueden contener más expresiones, es similar a la forma en que las ramas de los árboles se dividen y dividen una y otra vez.

Compara esto con el analizador que escribimos para el formato de archivo de configuración en el Capítulo 9, que tenía una estructura simple: dividía la entrada en líneas y manejaba esas líneas una a la vez. Habían solo unas pocas formas simples que se le permitía tener a una línea.

Aquí debemos encontrar un enfoque diferente. Las expresiones no están separadas en líneas, y tienen una estructura recursiva. Las expresiones de aplicaciones *contienen* otras expresiones.

Afortunadamente, este problema se puede resolver muy bien escribiendo una función analizadora que es recursiva en una manera que refleje la naturaleza recursiva del lenguaje.

Definimos una función `analizarExpresion`, que toma un string como entrada y retorna un objeto que contiene la estructura de datos para la expresión al comienzo del string, junto con la parte del string que queda después de analizar esta expresión Al analizar subexpresiones (el argumento para una aplicación, por ejemplo), esta función puede ser llamada de nuevo, produciendo la expresión del argumento, así como al texto que permanece. A su vez, este texto puede contener más argumentos o puede ser el paréntesis de cierre que finaliza la lista de argumentos.

Esta es la primera parte del analizador:

```
function analizarExpresion(programa) {
  programa = saltarEspacio(programa);
  let emparejamiento, expresion;
  if (emparejamiento = /^"([^"]*)"/.exec(programa)) {
    expresion = {tipo: "valor", valor: emparejamiento[1]};
  } else if (emparejamiento = /^\d+\b/.exec(programa)) {
    expresion = {tipo: "valor", valor: Number(emparejamiento[0])};
```

```
  } else if (emparejamiento = /^[^\s(),"]+/.exec(programa)) {
    expresion = {tipo: "palabra", nombre: emparejamiento[0]};
  } else {
    throw new SyntaxError("Sintaxis inesperada: " + programa);
  }

  return aplicarAnalisis(expresion, programa.slice(emparejamiento
     [0].length));
}

function saltarEspacio(string) {
  let primero = string.search(/\S/);
  if (primero == -1) return "";
  return string.slice(primero);
}
```

Dado que Egg, al igual que JavaScript, permite cualquier cantidad de espacios en blanco entre sus elementos, tenemos que remover repetidamente los espacios en blanco del inicio del string del programa. Para esto nos ayuda la función `saltarEspacio`.

Después de saltar cualquier espacio en blanco, `analizarExpresion` usa tres expresiones regulares para detectar los tres elementos atómicos que Egg soporta: strings, números y palabras. El analizador construye un tipo diferente de estructura de datos dependiendo de cuál coincida. Si la entrada no coincide con ninguna de estas tres formas, la expresión no es válida, y el analizador arroja un error. Usamos `SyntaxError` en lugar de `Error` como constructor de excepción, el cual es otro tipo de error estándar, dado que es un poco más específico—también es el tipo de error lanzado cuando se intenta ejecutar un programa de JavaScript no válido.

Luego cortamos la parte que coincidio del string del programa y pasamos eso, junto con el objeto para la expresión, a `aplicarAnalisis`, el cual verifica si la expresión es una aplicación. Si es así, analiza una lista de los argumentos entre paréntesis.

```
function aplicarAnalisis(expresion, programa) {
  programa = saltarEspacio(programa);
  if (programa[0] != "(") {
    return {expresion: expresion, resto: programa};
  }

  programa = saltarEspacio(programa.slice(1));
  expresion = {tipo: "aplicar", operador: expresion, argumentos:
```

```
    []};
  while (programa[0] != ")") {
    let argumento = analizarExpresion(programa);
    expresion.argumentos.push(argumento.expresion);
    programa = saltarEspacio(argumento.resto);
    if (programa[0] == ",") {
      programa = saltarEspacio(programa.slice(1));
    } else if (programa[0] != ")") {
      throw new SyntaxError("Experaba ',' o ')'");
    }
  }
  return aplicarAnalisis(expresion, programa.slice(1));
}
```

Si el siguiente carácter en el programa no es un paréntesis de apertura, esto no es una aplicación, y `aplicarAnalisis` retorna la expresión que se le dio.

De lo contrario, salta el paréntesis de apertura y crea el objeto de árbol de sintaxis para esta expresión de aplicación. Entonces, recursivamente llama a `analizarExpresion` para analizar cada argumento hasta que se encuentre el paréntesis de cierre. La recursión es indirecta, a través de `aplicarAnalisis` y `analizarExpresion` llamando una a la otra.

Dado que una expresión de aplicación puede ser aplicada a sí misma (como en `multiplicador(2)(1)`), `aplicarAnalisis` debe, después de haber analizado una aplicación, llamarse asi misma de nuevo para verificar si otro par de paréntesis sigue a continuación.

Esto es todo lo que necesitamos para analizar Egg. Envolvemos esto en una conveniente función `analizar` que verifica que ha llegado al final del string de entrada después de analizar la expresión (un programa Egg es una sola expresión), y eso nos da la estructura de datos del programa.

```
function analizar(programa) {
  let {expresion, resto} = analizarExpresion(programa);
  if (saltarEspacio(resto).length > 0) {
    throw new SyntaxError("Texto inesperado despues de programa");
  }
  return expresion;
}

console.log(analizar("+(a, 10)"));
// → {tipo: "aplicar",
//    operador: {tipo: "palabra", nombre: "+"},
//    argumentos: [{tipo: "palabra", nombre: "a"},
//           {tipo: "valor", valor: 10}]}
```

Funciona! No nos da información muy útil cuando falla y no almacena la línea y la columna en que comienza cada expresión, lo que podría ser útil al informar errores más tarde, pero es lo suficientemente bueno para nuestros propósitos.

The evaluator

What can we do with the syntax tree for a program? Run it, of course! And that is what the evaluator does. You give it a syntax tree and a scope object that associates names with values, and it will evaluate the expression that the tree represents and return the value that this produces.

```
const specialForms = Object.create(null);

function evaluate(expresion, scope) {
  if (expresion.type == "value") {
    return expresion.value;
  } else if (expresion.type == "word") {
    if (expresion.name in scope) {
      return scope[expresion.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expresion.name}`);
    }
  } else if (expresion.type == "apply") {
    let {operator, args} = expresion;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expresion.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

The evaluator has code for each of the expression types. A literal value expression produces its value. (For example, the expression `100` just evaluates

to the number 100.) For a binding, we must check whether it is actually defined in the scope and, if it is, fetch the binding's value.

Applications are more involved. If they are a special form, like `if`, we do not evaluate anything and pass the argument expressions, along with the scope, to the function that handles this form. If it is a normal call, we evaluate the operator, verify that it is a function, and call it with the evaluated arguments.

We use plain JavaScript function values to represent Egg's function values. We will come back to this later, when the special form called `fun` is defined.

The recursive structure of `evaluate` resembles the similar structure of the parser, and both mirror the structure of the language itself. It would also be possible to integrate the parser with the evaluator and evaluate during parsing, but splitting them up this way makes the program clearer.

This is really all that is needed to interpret Egg. It is that simple. But without defining a few special forms and adding some useful values to the environment, you can't do much with this language yet.

Special forms

The `specialForms` object is used to define special syntax in Egg. It associates words with functions that evaluate such forms. It is currently empty. Let's add `if`.

```
specialForms.si = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to si");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Egg's `si` construct expects exactly three arguments. It will evaluate the first, and if the result isn't the value `false`, it will evaluate the second. Otherwise, the third gets evaluated. This `if` form is more similar to JavaScript's ternary `?:` operator than to JavaScript's `if`. It is an expression, not a statement, and it produces a value, namely the result of the second or third argument.

Egg also differs from JavaScript in how it handles the condition value to `if`. It will not treat things like zero or the empty string as false, only the precise value `false`.

The reason we need to represent `if` as a special form, rather than a regular function, is that all arguments to functions are evaluated before the function is called, whereas `if` should evaluate only *either* its second or its third argument, depending on the value of the first.

The `while` form is similar.

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
  return false;
};
```

Another basic building block is `hacer`, which executes all its arguments from top to bottom. Its value is the value produced by the last argument.

```
specialForms.hacer = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};
```

To be able to create bindings and give them new values, we also create a form called `definir`. It expects a word as its first argument and an expression producing the value to assign to that word as its second argument. Since `definir`, like everything, is an expression, it must return a value. We'll make it return the value that was assigned (just like JavaScript's `=` operator).

```
specialForms.definir = (args, scope) => {
  if (args.length != 2 || args[0].type != "word") {
    throw new SyntaxError("Incorrect use of definir");
  }
  let value = evaluate(args[1], scope);
  scope[args[0].name] = value;
  return value;
```

```
};
```

The environment

The scope accepted by `evaluate` is an object with properties whose names correspond to binding names and whose values correspond to the values those bindings are bound to. Let's define an object to represent the global scope.

To be able to use the `if` construct we just defined, we must have access to Boolean values. Since there are only two Boolean values, we do not need special syntax for them. We simply bind two names to the values `true` and `false` and use those.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

We can now evaluate a simple expression that negates a Boolean value.

```
let prog = parse(`si(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

To supply basic arithmetic and comparison operators, we will also add some function values to the scope. In the interest of keeping the code short, we'll use `Function` to synthesize a bunch of operator functions in a loop, instead of defining them individually.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

A way to output values is also very useful, so we'll wrap `console.log` in a function and call it `imprimir`.

```
topScope.imprimir = value => {
  console.log(value);
  return value;
};
```

That gives us enough elementary tools to write simple programs. The following function provides a convenient way to parse a program and run it in a fresh scope.

```
function run(programa) {
  return evaluate(parse(programa), Object.create(topScope));
}
```

We'll use object prototype chains to represent nested scopes, so that the program can add bindings to its local scope without changing the top-level scope.

```
run(`
hacer(definir(total, 0),
   definir(count, 1),
   while(<(count, 11),
         hacer(definir(total, +(total, count)),
            definir(count, +(count, 1)))),
   imprimir(total))
`);
// → 55
```

This is the program we've seen several times before, which computes the sum of the numbers 1 to 10, expressed in Egg. It is clearly uglier than the equivalent JavaScript program—but not bad for a language implemented in less than 150 lines of code.

Functions

A programming language without functions is a poor programming language indeed.

Fortunately, it isn't hard to add a `fun` construct, which treats its last argument as the function's body and uses all arguments before that as the names of the function's parameters.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
```

```
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
    return expr.name;
  });

  return function() {
    if (arguments.length != params.length) {
      throw new TypeError("Wrong number of arguments");
    }
    let localScope = Object.create(scope);
    for (let i = 0; i < arguments.length; i++) {
      localScope[params[i]] = arguments[i];
    }
    return evaluate(body, localScope);
  };
};
```

Functions in Egg get their own local scope. The function produced by the `fun` form creates this local scope and adds the argument bindings to it. It then evaluates the function body in this scope and returns the result.

```
run(`
hacer(definir(plusOne, fun(a, +(a, 1))),
   imprimir(plusOne(10)))
`);
// → 11

run(`
hacer(definir(pow, fun(base, exp,
     si(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1)))))),
   imprimir(pow(2, 10)))
`);
// → 1024
```

Compilation

What we have built is an interpreter. During evaluation, it acts directly on the representation of the program produced by the parser.

Compilation is the process of adding another step between the parsing and

the running of a program, which transforms the program into something that can be evaluated more efficiently by doing as much work as possible in advance. For example, in well-designed languages it is obvious, for each use of a binding, which binding is being referred to, without actually running the program. This can be used to avoid looking up the binding by name every time it is accessed, instead directly fetching it from some predetermined memory location.

Traditionally, compilation involves converting the program to machine code, the raw format that a computer's processor can execute. But any process that converts a program to a different representation can be thought of as compilation.

It would be possible to write an alternative evaluation strategy for Egg, one that first converts the program to a JavaScript program, uses `Function` to invoke the JavaScript compiler on it, and then runs the result. When done right, this would make Egg run very fast while still being quite simple to implement.

If you are interested in this topic and willing to spend some time on it, I encourage you to try to implement such a compiler as an exercise.

Cheating

When we defined `if` and `while`, you probably noticed that they were more or less trivial wrappers around JavaScript's own `if` and `while`. Similarly, the values in Egg are just regular old JavaScript values.

If you compare the implementation of Egg, built on top of JavaScript, with the amount of work and complexity required to build a programming language directly on the raw functionality provided by a machine, the difference is huge. Regardless, this example hopefully gave you an impression of the way programming languages work.

And when it comes to getting something done, cheating is more effective than doing everything yourself. Though the toy language in this chapter doesn't do anything that couldn't be done better in JavaScript, there *are* situations where writing small languages helps get real work done.

Such a language does not have to resemble a typical programming language. If JavaScript didn't come equipped with regular expressions, for example, you could write your own parser and evaluator for regular expressions.

Or imagine you are building a giant robotic dinosaur and need to program its behavior. JavaScript might not be the most effective way to do this. You might instead opt for a language that looks like this:

```
behavior walk
```

```
  perform when
    destination ahead
  actions
    move left-foot
    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets
```

This is what is usually called a *domain-specific language*, a language tailored to express a narrow domain of knowledge. Such a language can be more expressive than a general-purpose language because it is designed to describe exactly the things that need to be described in its domain, and nothing else.

Exercises

Arrays

Add support for arrays to Egg by adding the following three functions to the top scope: `array(...values)` to construct an array containing the argument values, `length(array)` to get an array's length, and `element(array, n)` to fetch the nth element from an array.

Closure

The way we have defined `fun` allows functions in Egg to reference the surrounding scope, allowing the function's body to use local values that were visible at the time the function was defined, just like JavaScript functions do.

The following program illustrates this: function `f` returns a function that adds its argument to `f`'s argument, meaning that it needs access to the local scope inside `f` to be able to use binding `a`.

```
run(`
hacer(definir(f, fun(a, fun(b, +(a, b)))),
   imprimir(f(4)(5)))
`);
// → 9
```

Go back to the definition of the `fun` form and explain which mechanism causes this to work.

Comments

It would be nice if we could write comments in Egg. For example, whenever we find a hash sign (`#`), we could treat the rest of the line as a comment and ignore it, similar to `//` in JavaScript.

We do not have to make any big changes to the parser to support this. We can simply change `skipSpace` to skip comments as if they are whitespace so that all the points where `skipSpace` is called will now also skip comments. Make this change.

Fixing scope

Currently, the only way to assign a binding a value is `definir`. This construct acts as a way both to define new bindings and to give existing ones a new value.

This ambiguity causes a problem. When you try to give a nonlocal binding a new value, you will end up defining a local one with the same name instead. Some languages work like this by design, but I've always found it an awkward way to handle scope.

Add a special form `set`, similar to `definir`, which gives a binding a new value, updating the binding in an outer scope if it doesn't already exist in the inner scope. If the binding is not defined at all, throw a `ReferenceError` (another standard error type).

The technique of representing scopes as simple objects, which has made things convenient so far, will get in your way a little at this point. You might want to use the `Object.getPrototypeOf` function, which returns the prototype of an object. Also remember that scopes do not derive from `Object.prototype`, so if you want to call `hasOwnProperty` on them, you have to use this clumsy expression:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

"¡Tanto peor! ¡Otra vez la vieja historia! Cuando uno ha acabado de construir su casa advierte que, mientras la construía, ha aprendido, sin darse cuenta, algo que tendría que haber sabido absolutamente antes de comenzar a construir."

—Friedrich Nietzsche, Más allá del bien y del mal

Chapter 13

El Modelo de Objeto del Documento

Cuando abres una página web en tu navegador, el navegador obtiene el texto de la página HTML y lo analiza, de una manera bastante similar a la manera en que nuestro analizador del Capítulo 12 analizaba los programas. El navegador construye un modelo de la estructura del documento y utiliza este modelo para dibujar la página en la pantalla.

Esta representación del documento es uno de los juguetes que un programa de JavaScript tiene disponible en su caja de arena. Es una estructura de datos que puedes leer o modificar. Y actúa como una estructura en *tiempo real*: cuando se modifica, la página en la pantalla es actualizada para reflejar los cambios.

La estructura del documento

Te puedes imaginar a un documento HTML como un conjunto anidado de cajas. Las etiquetas como `<body>` y `</body>` encierran otras etiquetas, que a su vez, contienen otras etiquetas o texto. Este es el documento de ejemplo del capítulo anterior:

```
<!doctype html>
<html>
  <head>
    <title>Mi página de inicio</title>
  </head>
  <body>
    <h1>Mi página de inicio</h1>
    <p>Hola, mi nombre es Marijn y esta es mi página de inicio.</p>
    <p>También escribí un libro! Léelo
      <a href="http://eloquentjavascript.net">aquí</a>.</p>
  </body>
</html>
```

Esta página tiene la siguiente estructura:

La estructura de datos que el navegador utiliza para representar el documento sigue esta figura. Para cada caja, hay un objeto, con el que podemos interactuar para descubrir cosas como que etiqueta de HTML representa y que cajas y texto contiene. Esta representación es llamada *Modelo de Objeto del Documento* o DOM *(por sus siglas en inglés "Document Object Model")*.

El objeto de enlace global `document` nos da acceso a esos objetos. Su propiedad `documentElement` hace referencia al objeto que representa a la etiqueta `<html>`. Dado que cada documento HTML tiene una cabecera y un cuerpo, también tiene propiedades `head` y `body` que apuntan a esos elementos.

Árboles

Pensemos en los árboles sintácticos del Capítulo 12 por un momento. Sus estructuras son sorprendentemente similares a la estructura de un documento del navegador. Cada *nodo* puede referirse a otros nodos *hijos* que, a su vez, pueden tener hijos propios. Esta forma es típica de las estructuras anidadas donde los elementos pueden contener sub elementos que son similares a ellos mismos.

Le damos el nombre de *árbol* a una estructura de datos cuando tiene una estructura de ramificación, no tiene ciclos (un nodo no puede contenerse a sí mismo, directa o indirectamente) y tiene una única *raíz* bien definida. En el caso del *DOM*, `document.documentElement` hace la función de raíz.

Los árboles aparecen constantemente en las ciencias de la computación *(computer sience)*. Además de representar estructuras recursivas como los docu-

mentos HTML o programas, también son comúnmente usados para mantener conjuntos ordenados de datos debido a que los elementos generalmente pueden ser encontrados o agregados más eficientemente en un árbol que en un arreglo plano.

Un árbol típico tiene diferentes tipos de nodos. El árbol sintáctico del lenguaje *Egg* tenía identificadores, valores y nodos de aplicación. Los nodos de aplicación pueden tener hijos, mientras que los identificadores y valores son *hojas*, o nodos sin hijos.

Lo mismo sucede para el DOM, los nodos para los *elementos*, los cuales representan etiquetas HTML, determinan la estructura del documento. Estos pueden tener nodos hijos. Un ejemplo de estos nodos es `document.body`. Algunos de estos hijos pueden ser nodos hoja, como los fragmentos de texto o los nodos comentario.

Cada objeto *nodo DOM* tiene una propiedad `nodeType`, la cual contiene un código (numérico) que identifica el tipo de nodo. Los *Elementos* tienen el código 1, que también es definido por la propiedad constante `Node.ELEMENT_NODE`. Los nodos de texto, representando una sección de texto en el documento, obtienen el código 3 (`Node.TEXT_NODE`). Los comentarios obtienen el código 8 (`Node.COMMENT_NODE`).

Otra forma de visualizar nuestro árbol de documento es la siguiente:

Las hojas son nodos de texto, y las flechas nos indican las relaciones padre-hijo entre los nodos.

El estándar

Usar códigos numéricos crípticos para representar a los tipos de nodos no es algo que se parezca al estilo de JavaScript para hacer las cosas. Más adelante en este capítulo, veremos cómo otras partes de la interfaz DOM también se sienten engorrosas y alienígenas. La razón de esto es que DOM no fue diseñado

solamente para JavaScript. Más bien, intenta ser una interfaz *independiente del lenguaje* que puede ser usada en otros sistemas, no solamente para HTML pero también para XML, que es un formato de datos genérico con una sintaxis similar a la de HTML.

Esto es desafortunado. Usualmente los estándares son bastante útiles. Pero en este caso, la ventaja (consistencia entre lenguajes) no es tan conveniente. Tener una interfaz que está propiamente integrada con el lenguaje que estás utilizando te ahorrará más tiempo que tener una interfaz familiar en distintos lenguajes.

A manera de ejemplo de esta pobre integración, considera la propiedad `childNodes` que los nodos elemento en el DOM tienen. Esta propiedad almacena un objeto parecido a un arreglo, con una propiedad `length` y propiedades etiquetadas por números para acceder a los nodos hijos. Pero es una instancia de tipo `NodeList`, no un arreglo real, por lo que no tiene métodos como `slice` o `map`.

Luego, hay problemas que son simplemente un pobre diseño. Por ejemplo, no hay una manera de crear un nuevo nodo e inmediatamente agregar hijos o attributos. En vez de eso, tienes que crearlo primero y luego agregar los hijos y atributos uno por uno, usando efectos secundarios. El código que interactúa mucho con el DOM tiende a ser largo, repetitivo y feo.

Pero estos defectos no son fatales. Dado que JavaScript nos permite crear nuestra propias abstracciones, es posible diseñar formas mejoradas para expresar las operaciones que estás realizando. Muchas bibliotecas destinadas a la programación del navegador vienen con esas herramientas.

Moviéndose a través del árbol

Los nodos del DOM contienen una amplia cantidad de enlaces a otros nodos cercanos. El siguiente diagrama los ilustra:

A pesar de que el diagrama muestra un solo enlace por cada tipo, cada nodo tiene una propiedad `parentNode` que apunta al nodo al que pertenece, si es que hay alguno. Igualmente, cada nodo *elemento* (nodo tipo 1) tiene una propiedad `childNodes` que apunta a un objeto similar a un arreglo que almacena a sus hijos.

En teoría, te deberías poder mover donde quieras en el árbol utilizando únicamente estos enlaces entre padre e hijo. Pero JavaScript también te otorga acceso a un número de enlaces adicionales convenientes. Las propiedades `firstChild` y `lastChild` apuntan al primer y último elementos hijo, o tiene el valor `null` para nodos sin hijos. De manera similar, las propiedades `previousSibling` y `nextSibling` apuntan a los nodos adyacentes, los cuales, son nodos con el mismo padre que aparecen inmediatamente antes o después del nodo. Para el primer hijo `previousSibling` será `null` y para el último hijo, `nextSibling` será `null`.

También existe una propiedad `children`, que es parecida a `childNodes` pero contiene únicamente hijos de tipo *elemento* (tipo 1), excluyendo otros tipos de nodos. Esto puede ser útil cuando no estás interesando en nodos de tipo texto.

Cuando estás tratando con estructuras de datos anidadas como esta, las funciones recursivas son generalmente útiles. La siguiente función escanea un documento por nodos de texto que contengan una cadena dada y regresan `true` en caso de que encuentren una:

```
function hablaSobre(nodo, cadena) {
  if (nodo.nodeType == Node.ELEMENT_NODE) {
    for (let i = 0; i < nodo.childNodes.length; i++) {
      if (hablaSobre(nodo.childNodes[i], cadena)) {
        return true;
      }
    }
    return false;
  } else if (nodo.nodeType == Node.TEXT_NODE) {
    return nodo.nodeValue.indexOf(cadena) > -1;
  }
}

console.log(hablaSobre(document.body, "libro"));
// → true
```

Debido a que `childNodes` no es un arreglo real, no podemos iterar sobre el usando `for/of` por lo que tenemos que iterar sobre el rango del índice usando un `for` regular o usando `Array.from`.

La propiedad `nodeValue` de un nodo de texto almacena la cadena de texto que representa.

Buscar elementos

Navegar por estos enlaces entre padres, hijos y hermanos suele ser útil. Pero si queremos encontrar un nodo específico en el documento, alcanzarlo comenzando en `document.body` y siguiendo un camino fijo de propiedades es una mala idea. Hacerlo genera suposiciones en nuestro programa sobre la estructura precisa del documento que tal vez quieras cambiar después. Otro factor complicado es que los nodos de texto son creados incluso para los espacios en blanco entre nodos. La etiqueta `<body>` en el documento de ejemplo no tiene solamente tres hijos (`<h1>` y dos elementos `<p>`), en realidad tiene siete: esos tres, más los espacios posteriores y anteriores entre ellos.

Por lo que si queremos obtener el atributo `href` del enlace en ese documento, no queremos decir algo como "Obten el segundo hijo del sexto hijo del elemento *body* del documento". Sería mejor si pudiéramos decir "Obten el primer enlace en el documento". Y de hecho podemos.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Todos los nodos *elemento* tienen un método `getElementsByTagName`, el cual recolecta a todos los elementos con un nombre de etiqueta dado que son descendientes (hijos directos o indirectos) de ese nodo y los regresa como un objeto parecido a un arreglo.

Para encontrar un *único* nodo en específico, puedes otorgarle un atributo `id` y usar `document.getElementById`.

```
<p>Mi avestruz Gertrudiz:</p>
<p><img id="gertrudiz" src="img/ostrich.png"></p>

<script>
  let avestruz = document.getElementById("gertrudiz");
  console.log(avestruz.src);
</script>
```

Un tercer método similar es `getElementsByClassName`, el cual, de manera similar a `getElementsByTagName` busca a través de los contenidos de un nodo elemento y obtiene todos los elementos que tienen una cadena dada en su

attributo `class`.

Actualizar el documento

Prácticamente todo sobre la estructura de datos DOM puede ser cambiado. La forma del árbol de documento puede ser modificada cambiando las relaciones padre-hijo. Los nodos tienen un método `remove` para ser removidos de su nodo padre actual. Para agregar un nodo hijo a un nodo elemento, podemos usar `appendChild`, que lo pondrá al final de la lista de hijos, o `insertBefore`, que insertará el nodo en el primer argumento antes del nodo en el segundo argumento.

```
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>

<script>
  let parrafos = document.body.getElementsByTagName("p");
  document.body.insertBefore(parrafos[2], parrafos[0]);
</script>
```

Un nodo puede existir en el documento solamente en un lugar. En consecuencia, insertar el párrafo *Tres* enfrente del párrafo *Uno* primero lo removerá del final del documento y luego lo insertará en la parte delantera, resultando en *Tres/Uno/Dos*. Todas las operaciones que insertan un nodo en alguna parte causarán, a modo de efecto secundario, que el nodo sea removido de su posición actual (si es que tiene una).

El método `replaceChild` es usado para reemplazar a un nodo hijo con otro. Toma dos nodos como argumentos: un nuevo nodo y el nodo que será reemplazado. El nodo reemplazado debe ser un nodo hijo del elemento desde donde se está llamando el método. Nótese que tanto `replaceChild` como `insertBefore` esperan que el *nuevo* nodo sea el primer argumento.

Crear nodos

Digamos que queremos escribir un *script* que reemplace todas las imagenes (etiquetas `<img>`) en el documento con el texto contenido en sus atributos `alt`, los cuales especifican una representación textual alternativa de la imagen.

Esto no solamente involucra remover las imágenes, si no que también involucra agregar un nuevo nodo texto que las reemplace. Los nodos texto son

creados con el método `document.createTextNode`.

```
<p>El <img src="img/cat.png" alt="Gato"> en el
  <img src="img/hat.png" alt="Sombrero">.</p>

<p><button onclick="sustituirImagenes()">Sustituir</button></p>

<script>
  function sustituirImagenes() {
    let imagenes = document.body.getElementsByTagName("img");
    for (let i = imagenes.length - 1; i >= 0; i--) {
      let imagen = imagenes[i];
      if (imagen.alt) {
        let texto = document.createTextNode(imagen.alt);
        imagen.parentNode.replaceChild(texto, imagen);
      }
    }
  }
</script>
```

Dada una cadena, `createTextNode` nos da un nodo texto que podemos insertar en el documento para hacer que aparezca en la pantalla.

El ciclo que recorre las imágenes empieza al final de la lista. Esto es necesario dado que la lista de nodos regresada por un método como `getElementsByTagName` (o una propiedad como `childNodes`) se actualiza en tiempo real. Esto es, que se actualiza conforme el documento cambia. Si empezáramos desde el frente, remover la primer imagen causaría que la lista perdiera su primer elemento de tal manera que la segunda ocasión que el ciclo se repitiera, donde `i` es 1, se detendría dado que la longitud de la colección ahora es también 1.

Si quieres una colección de nodos *sólida*, a diferencia de una en tiempo real, puedes convertir la colección a un arreglo real llamando `Array.from`.

```
let casi_arreglo = {0: "uno", 1: "dos", length: 2};
let arreglo = Array.from(casi_arreglo);
console.log(arreglo.map(s => s.toUpperCase()));
// → ["UNO", "DOS"]
```

Para crear nodos elemento, puedes utilizar el método `document.createElement`. Este método toma un nombre de etiqueta y regresa un nuevo nodo vacío del tipo dado.

El siguiente ejemplo define una utilidad `elt`, la cual crea un elemento nodo y trata el resto de sus argumentos como hijos de ese nodo. Luego, esta función

es utilizada para agregar una atribución a una cita.

```
<blockquote id="cita">
  Ningún libro puede terminarse jamás. Mientras se trabaja en
  él aprendemos solo lo suficiente para encontrar inmaduro
  el momento en el que nos alejamos de él.
</blockquote>

<script>
  function elt(tipo, ...hijos) {
    let nodo = document.createElement(tipo);
    for (let hijo of hijos) {
      if (typeof hijo != "string") nodo.appendChild(hijo);
      else nodo.appendChild(document.createTextNode(hijo));
    }
    return nodo;
  }

  document.getElementById("cita").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", prefacio de la segunda edición de ",
        elt("em", "La sociedad abierta y sus enemigos"),
        ", 1950"));
</script>
```

Así es como se ve el documento resultante:

> No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it.
> —**Karl Popper**, preface to the second editon of *The Open Society and Its Enemies*, 1950

Atributos

Los atributos de algunos elementos, como `href` para los enlaces, pueden ser accedidos a través de una propiedad con el mismo nombre en el objeto DOM del elemento. Este es el caso para los atributos estándar más comúnmente utilizados.

Pero HTML te permite establecer cualquier atributo que quieras en los nodos. Esto puede ser útil debido a que te permite almacenar información extra en un documento. Sin embargo, si creas tus propios nombres de atributo, dichos atributos no estarán presentes como propiedades en el nodo del elemento. En

vez de eso, tendrás que utilizar los métodos `getAttribute` y `setAttribute` para poder trabajar con ellos.

```
<p data-classified="secreto">El código de lanzamiento es:
   00000000.</p>
<p data-classified="no-classificado">Yo tengo dos pies.</p>

<script>
  let parrafos = document.body.getElementsByTagName("p");
  for (let parrafo of Array.from(parrafos)) {
    if (parrafo.getAttribute("data-classified") == "secreto") {
      parrafo.remove();
    }
  }
</script>
```

Se recomienda anteponer los nombres de dichos atributos inventados con `data-` para asegurarse de que no conflictúan con ningún otro atributo.

Existe un atributo comúnmente usado, `class`, que es una palabra clave en el lenguaje JavaScript. Por motivos históricos, algunas implementaciones antiguas de JavaScript podrían no manejar nombres de propiedades que coincidan con las palabras clave, la propiedad utilizada para acceder a este atributo tiene por nombre `className`. También puedes acceder a ella bajo su nombre real, `"class"`, utilizando los métodos `getAttribute` y `setAttribute`.

Layout

Tal vez hayas notado que diferentes tipos de elementos se exponen de manera distinta. Algunos, como en el caso de los párrafos (`<p>`) o encabezados (`<h1>`), ocupan todo el ancho del documento y se renderizan en líneas separadas. A estos se les conoce como elementos *block* (o bloque). Otros, como los enlaces (`<a>`) o el elemento `<strong>`, se renderizan en la misma línea con su texto circundante. Dichos elementos se les conoce como elementos *inline* (o en línea).

Para cualquier documento dado, los navegadores son capaces de calcular una estructura *(layout)*, que le da a cada elemento un tamaño y una posición basada en el tipo y el contenido. Luego, esta estructura se utiliza para trazar el documento.

Se puede acceder al tamaño y la posición de un elemento desde JavaScript. Las propiedades `offsetWidth` y `offsetHeight` te dan el espacio que el elemento utiliza en pixeles. Un píxel es la unidad básica de las medidas del navegador. Tradicionalmente correspondía al punto más pequeño que la pantalla podía

trazar, pero en los monitores modernos, que pueden trazar puntos *muy* pequeños, este puede no ser más el caso, por lo que un píxel del navegador puede abarcar varios puntos en la pantalla.

De manera similar, `clientWidth` y `clientHeight` te dan el tamaño del espacio *dentro* del elemento, ignorando la anchura del borde.

```
<p style="border: 3px solid red">
  Estoy dentro de una caja
</p>

<script>
  let parrafo = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", parrafo.clientHeight);
  console.log("offsetHeight:", parrafo.offsetHeight);
</script>
```

Darle un borde a un párrafo hace que se dibuje un rectángulo a su alrededor.

I'm boxed in

La manera más efectiva de encontrar la posición precisa de un elemento en la pantalla es el método `getBoundingClientRect`. Este devuelve un objeto con las propiedades `top`, `bottom`, `left`, y `right`, indicando las posiciones de pixeles de los lados el elemento en relación con la parte superior izquierda de la pantalla. Si los quieres en relación a todo el documento, deberás agregar la posición actual del scroll, la cual puedes obtener en los *bindings* `pageXOffset` y `pageYOffset`.

Estructurar un documento puede requerir mucho trabajo. En los intereses de velocidad, los motores de los navegadores no reestructuran inmediatamente un documento cada vez que lo cambias, en cambio, se espera lo más que se pueda. Cuando un programa de JavaScript que modifica el documento termina de ejecutarse, el navegador tendrá que calcular una nueva estructura para trazar el documento actualizado en la pantalla. Cuando un programa *solicita* la posición o el tamaño de algo, leyendo propiedades como `offsetHeight` o llamando a `getBoundingClientRect`, proveer la información correcta también requiere que se calcule una nueva estructura.

A un programa que alterna repetidamente entre leer la información de la estructura DOM y cambiar el DOM, fuerza a que haya bastantes cálculos de estructura, y por consecuencia se ejecutará lentamente. El siguiente código es un ejemplo de esto. Contiene dos programas diferentes que construyen una línea de X caracteres con 2,000 pixeles de ancho y que mide el tiempo que toma

cada uno.

```
<p><span id="uno"></span></p>
<p><span id="dos"></span></p>

<script>
  function tiempo(nombre, accion) {
    let inicio = Date.now(); // Tiempo actual en milisegundos
    accion();
    console.log(nombre, "utilizo", Date.now() - inicio, "ms");
  }

  tiempo("inocente", () => {
    let objetivo = document.getElementById("uno");
    while (objetivo.offsetWidth < 2000) {
      objetivo.appendChild(document.createTextNode("X"));
    }
  });
  // → inocente utilizo 32 ms

  tiempo("ingenioso", function() {
    let objetivo = document.getElementById("dos");
    objetivo.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (objetivo.offsetWidth / 5));
    objetivo.firstChild.nodeValue = "X".repeat(total);
  });
  // → ingenioso utilizo 1 ms
</script>
```

Estilización

Hemos visto que diferentes elementos HTML se trazan de manera diferente. Algunos son desplegados como bloques, otros en línea. Algunos agregan estilos, por ejemplo `<strong>` hace que su contenido esté en negritas y `<a>` lo hace azul y lo subraya.

La forma en la que una etiqueta `<img>` muestra una imagen o una etiqueta `<a>` hace que un enlace sea seguido cuando se hace click en el, está fuertemente atado al tipo del elemento. Pero podemos cambiar los estilos asociados a un elemento, tales como el color o si está subrayado. Este es un ejemplo que utiliza la propiedad `style`:

```
<p><a href=".">Enlace normal</a></p>
```

```
<p><a href="." style="color: green">Enlace verde</a></p>
```

El segundo enlace será verde en vez del color por defecto.

Normal link

Green link

Un atributo *style* puede llegar a contener una o más declaraciones, que consisten en una propiedad (como `color`) seguido del símbolo de dos puntos y un valor (como `green`). Cuando hay más de una declaración, estas deben ser separadas por punto y coma, como en `"color: red; border: none"`.

Muchos de los aspectos del documento pueden ser influenciados por la estilización. Por ejemplo, la propiedad `display` controla si un elemento es desplegado como un bloque o como un elemento en línea.

```
El texto es desplegado <strong>en línea</strong>,
<strong style="display: block">como bloque</strong>, y
<strong style="display: none">no se despliega</strong>.
```

La etiqueta `block` terminará en su propia línea dado que los elementos *bloque* no son desplegados en línea con el texto que los rodea. La última etiqueta no se despliega—`display: none` previene que el elemento sea mostrado en la pantalla. Esta es una manera de ocultar elementos. A menudo esto es preferido sobre removerlos completamente del documento debido a que hace más fácil mostrarlos nuevamente en el futuro.

This text is displayed **inline**,
as a block
, and .

El código de JavaScript puede manipular directamente el estilo de un elemento a través de la propiedad `style` del elemento. Esta propiedad almacena un objeto que tiene propiedades por todas las posibles propiedades de estilo. Los valores de esas propiedades son cadenas, que podemos escribir para cambiar un aspecto en particular del estilo del elemento.

```
<p id="parrafo" style="color: purple">
  Texto mejorado
</p>

<script>
  let parrafo = document.getElementById("parrafo");
```

```
  console.log(parrafo.style.color);
  parrafo.style.color = "magenta";
</script>
```

Algunos nombres de propiedades pueden contener guiones, como es el caso de `font-family`. Dado que estos nombres de propiedades son incómodos para trabajar con ellos en JavaScript (tendrías que decir `style["font-family"]`), los nombres de propiedades en el objeto `style` para tales propiedades no tendrán guiones y las letra después del guion estará en mayúsculas (`style.fontFamily`).

Estilos en Cascada

El sistema de estilos para HTML es llamado CSS por sus siglas en ingles *Cascading Style Sheets* (Hojas de estilo en cascada). Una *hoja de estilos* es un conjunto de reglas sobre cómo estilizar a los elementos en un documento. Puede estar declarado dentro de una etiqueta `<style>`.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Ahora <strong>el texto en negritas</strong> esta en italicas y es
    gris.</p>
```

La sección *cascada* en el nombre se refiere al hecho de que varias reglas son combinadas para producir el estilo final para un elemento. En el ejemplo, el estilo por defecto para las etiquetas `<strong>`, que les da `font-weight: bold`, es superpuesto por la regla en la etiqueta `<style>`, que le agrega `font-style` y `color`.

Cuando varias reglas definen un valor para una misma propiedad, la regla leída más recientemente obtiene una mayor precedencia y gana. Por lo que si la regla en la etiqueta `<style>` incluyera `font-weight: normal`, contradiciendo la regla por defecto de `font-weight`, el texto se vería normal, *no* en negritas. Los estilos en un atributo `style` aplicados directamente al nodo tienen la mayor precedencia y siempre ganan.

Es posible apuntar a otras cosas que no sean nombres de etiqueta en las reglas CSS. Una regla para `.abc` aplica a todos los elementos con `"abc"` en su atributo `class`. Una regla para `#xyz` aplica a todos los elementos con un

atributo `id` con valor `"xyz"` (que debería ser único en el documento).

```
.sutil {
  color: gray;
  font-size: 80%;
}
#cabecera {
  background: blue;
  color: white;
}
/* Elementos p con un id principal y clases a y b */
p#principal.a.b {
  margin-bottom: 20px;
}
```

La regla de precedencia que favorece a las reglas más recientemente definidas aplican solamente cuando las reglas tienen la misma *especificidad*. La especificidad de una regla es una medida de que tan precisamente describe a los elementos que coinciden con la regla, determinado por el número y tipo (etiqueta, clase o ID) de los aspectos del elemento que requiera. Por ejemplo, una regla que apunta a `p.a` es más específica que las reglas que apuntan a `p` o solamente a `.a` y tendrá precedencia sobre ellas.

La notación `p > a …{}` aplica los estilos dados a todas las etiquetas `<a>` que son hijas directas de las etiquetas `<p>`. De manera similar, `p a …{}` aplica a todas las etiquetas `<a>` dentro de etiquetas `<p>`, sin importar que sean hijas directas o indirectas.

Selectores de consulta

No utilizaremos mucho las hojas de estilo en este libro. Entenderlas es útil cuando se programa en el navegador, pero son lo suficientemente complicadas para justificar un libro por separado.

La principal razón por la que introduje la sintaxis de selección—la notación usada en las hojas de estilo para determinar a cuales elementos aplicar un conjunto de estilos—es que podemos utilizar el mismo mini-lenguaje como una manera efectiva de encontrar elementos DOM.

El método `querySelectorAll`, que se encuentra definido tanto en el objeto `document` como en los nodos elemento, toma la cadena de un selector y regresa una `NodeList` que contiene todos los elementos que coinciden con la consulta.

```
<p>And if you go chasing
```

```
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="caracter">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function contar(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(contar("p"));           // Todos los elementos <p>
  // → 4
  console.log(contar(".animal"));     // Clase animal
  // → 2
  console.log(contar("p .animal"));   // Animales dentro de <p>
  // → 2
  console.log(contar("p > .animal")); // Hijos directos de <p>
  // → 1
</script>
```

A diferencia de métodos como `getElementsByTagName`, el objeto que regresa `querySelectorAll` no es un objeto en tiempo real. No cambiará cuando cambies el documento. Sin embargo, sigue sin ser un arreglo real, por lo que aún necesitas llamar a `Array.from` si lo quieres tratar como uno real.

El método `querySelector` (sin la parte de `All`) trabaja de una manera similar. Este es útil si quieres un único elemento en específico. Regresará únicamente el primer elemento que coincida o *null* en el caso que ningún elemento coincida.

Posicionamiento y animaciones

La propiedad de estilo `position` influye de un manera poderosa sobre la estructura. Por defecto, tiene el valor de `static`, eso significa que el elemento se coloca en su lugar normal en el documento. Cuando se establece como `relative`, el elemento sigue utilizando espacio en el documento pero ahora las propiedades `top` y `left` pueden ser utilizadas para moverlo relativamente a ese espacio normal. Cuando `position` se establece como `absolute`, el elemento es removido del flujo normal del documento—esto es, deja de tomar espacio y puede encimarse con otros elementos. Además, sus propiedades `top` y `left` pueden ser utilizadas para posicionarlo absolutamente con relación a la esquina superior izquierda del elemento envolvente más cercano cuya propiedad `position` no sea `static`, o con relación al documento si dicho elemento envolvente no existe.

Podemos utilizar esto para crear una animación. El siguiente documento despliega una imagen de un gato que se mueve alrededor de una elipse:

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let gato = document.querySelector("img");
  let angulo = Math.PI / 2;
  function animar(tiempo, ultimoTiempo) {
    if (ultimoTiempo != null) {
      angulo += (tiempo - ultimoTiempo) * 0.001;
    }
    gato.style.top = (Math.sin(angulo) * 20) + "px";
    gato.style.left = (Math.cos(angulo) * 200) + "px";
    requestAnimationFrame(nuevoTiempo => animar(nuevoTiempo, tiempo)
       );
  }
  requestAnimationFrame(animar);
</script>
```

La flecha gris indica el camino por el que se mueve la imagen.

Nuestra imagen se centra en la página y se le da un valor para `position` de `relative`. Actualizaremos repetidamente los estilos `top` y `left` de la imagen para moverla.

El script utiliza `requestAnimationFrame` para programar la función `animar` para ejecutarse en el momento en el que el navegador está listo para volver a pintar la pantalla. La misma función `animar` llama a `requestAnimationFrame` otra vez para programar la siguiente actualización. Cuando la ventana del navegador (o pestaña) está activa, esto causará que sucedan actualizaciones a un rango de aproximadamente 60 actualizaciones por segundo, lo que tiende a producir una animación agradable a la vista.

Si únicamente actualizáramos el DOM en un ciclo, la página se congelaría, y no se mostraría nada en la pantalla. Los navegadores no actualizan la pantalla si un programa de JavaScript se encuentra en ejecución, tampoco permiten ninguna interacción con la página. Es por esto que necesitamos a

`requestAnimationFrame`—le permite al navegador saber que hemos terminado por el momento, y que puede empezar a hacer las cosas que le navegador hace, cómo actualizar la pantalla y responder a las acciones del usuario.

A la función `animar` se le pasa el tiempo actual como un argumento. Para asegurarse de que el movimiento del gato por milisegundo es estable, basa la velocidad a la que cambia el ángulo en la diferencia entre el tiempo actual y la última vez que la función se ejecutó. Si solamente movieramos el ángulo una cierta cantidad por paso, la animación tartamudearía si, por ejemplo, otra tarea pesada se encontrara ejecutándose en la misma computadora que pudiera prevenir que la función se ejecutará por una fracción de segundo.

Moverse en círculos se logra a través de las funciones `Math.cos` y `Math.sin`. Para aquellos que no estén familiarizados con estas, las introduciré brevemente dado que las usaremos ocasionalmente en este libro.

Las funciones `Math.cos` y `Math.sin` son útiles para encontrar puntos que recaen en un círculo alrededor del punto (0,0) con un radio de uno. Ambas funciones interpretan sus argumentos como las posiciones en el círculo, con cero denotando el punto en la parte más alejada del lado derecho del círculo, moviéndose en el sentido de las manecillas del reloj hasta que 2π (cerca de 6.28) nos halla tomado alrededor de todo el círculo. `Math.cos` indica la coordenada x del punto que corresponde con la posición dada, y `Math.sin` indica la coordenada y. Las posiciones (o ángulos) mayores que 2π o menores que 0 son válidas—la rotación se repite por lo que $a+2\pi$ se refiere al mismo ángulo que a.

Esta unidad para medir ángulos se conoce como radianes-un círculo completo corresponde a 2π radianes, de manera similar a 360 grados cuando se utilizan grados. La constante π está disponible como `Math.PI` en JavaScript.

El código de animación del gato mantiene un contador, `angulo`, para el ángulo actual de la animación y lo incrementa cada vez que la función `animar` es llamada. Luego, se puede utilizar este ángulo para calcular la posición actual del elemento imagen. El estilo `top` es calculado con `Math.sin` y multiplicado por 20, que es el radio vertical de nuestra elipse. El estilo `left` se basa en

`Math.cos` multiplicado por 200 por lo que la elipse es mucho más ancha que su altura.

Nótese que los estilos usualmente necesitan *unidades*. En este caso, tuvimos que agregar `"px"` al número para informarle al navegador que estábamos contando en pixeles (al contrario de centímetros, "ems", u otras unidades). Esto es sencillo de olvidar. Usar números sin unidades resultará en estilos que son ignorados—a menos que el número sea 0, que siempre indica la misma cosa, independientemente de su unidad.

Resumen

Los programas de JavaScript pueden inspeccionar e interferir con el documento que el navegador está desplegando a través de una estructura de datos llamada el DOM. Esta estructura de datos representa el modelo del navegador del documento, y un programa de JavaScript puede modificarlo para cambiar el documento visible.

El DOM está organizado como un árbol, en el cual los elementos están ordenados jerárquicamente de acuerdo a la estructura del documento. Estos objetos que representan a los elementos tienen propiedades como `parentNode` y `childNodes`, que pueden ser usadas para navegar a través de este árbol.

La forma en que un documento es desplegado puede ser influenciada por la *estilización*, tanto agregando estilos directamente a los nodos cómo definiendo reglas que coincidan con ciertos nodos. Hay muchas propiedades de estilo diferentes, tales como `color` o `display`. El código de JavaScript puede manipular el estilo de un elemento directamente a través de su propiedad de `style`.

Ejercicios

Construye una tabla

Una tabla HTML se construye con la siguiente estructura de etiquetas:

```
<table>
  <tr>
    <th>nombre</th>
    <th>altura</th>
    <th>ubicacion</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
```

```
    <td>Tanzania</td>
  </tr>
</table>
```

Para cada *fila*, la etiqueta `<table>` contiene una etiqueta `<tr>`. Dentro de estas etiquetas `<tr>`, podemos poner ciertos elementos: ya sean celdas cabecera (`<th>`) o celdas regulares (`<td>`).

Dado un conjunto de datos de montañas, un arreglo de objetos con propiedades `nombre`, `altura` y `lugar`, genera la estructura DOM para una tabla que enlista esos objetos. Deberá tener una columna por llave y una fila por objeto, además de una fila cabecera con elementos `<th>` en la parte superior, listando los nombres de las columnas.

Escribe esto de manera que las columnas se deriven automáticamente de los objetos, tomando los nombres de propiedad del primer objeto en los datos.

Agrega la tabla resultante al elemento con el atributo `id` de `"montañas"` de manera que se vuelva visible en el documento.

Una vez que lo tengas funcionando, alinea a la derecha las celdas que contienen valores numéricos, estableciendo su propiedad `style.textAlign` cómo `"right"`.

Elementos por nombre de tag

El método `document.getElementsByTagName` regresa todos los elementos hijo con un nombre de etiqueta dado. Implementa tu propia versión de esto como una función que toma un nodo y una cadena (el nombre de la etiqueta) como argumentos y regresa un arreglo que contiene todos los nodos elemento descendientes con el nombre del tag dado.

Para encontrar el nombre del tag de un elemento, utiliza su propiedad `nodeName`. Pero considera que esto regresará el nombre de la etiqueta todo en mayúsculas. Utiliza las funciones de las cadenas (`string`), `toLowerCase` o `toUpperCase` para compensar esta situación.

El sombrero del gato

Extiende la animación del gato definida anteriormente de manera de que tanto el gato como su sombrero (`<img src="img/hat.png">`) orbiten en lados opuestos de la elipse.

O haz que el sombrero circule alrededor del gato. O altera la animación en alguna otra forma interesante.

Para hacer el posicionamiento de múltiples objetos más sencillo, probablemente sea buena idea intercambiar a un posicionamiento absoluto. Esto significa que `top` y `left` serán contados con relación a la parte superior izquierda del documento. Para evitar usar coordenadas negativas, que pueden causar que la imagen se mueva fuera de la página visible, puedes agregar un número fijo de pixeles a los valores de las posiciones.

"Tienes poder sobre tu mente, no sobre los acontecimientos. Date cuenta de esto, y encontrarás la fuerza."

—Marco Aurelio, Meditaciones

Chapter 14

Manejo de Eventos

Algunos programas funcionan con la entrada directa del usuario, como las acciones del mouse y el teclado. Ese tipo de entrada no está disponible como una estructura de datos bien organizada, viene pieza por pieza, en tiempo real, y se espera que el programa responda a ella a medida que sucede.

Manejador de eventos

Imagina una interfaz en donde la única forma de saber si una tecla del teclado está siendo presionada es leer el estado actual de esa tecla. Para poder reaccionar a las pulsaciones de teclas, tendrías que leer constantemente el estado de la tecla para poder detectarla antes de que se vuelva a soltar. Esto sería peligroso al realizar otros cálculos que requieran mucho tiempo, ya que se podría perder una pulsación de tecla.

Algunas máquinas antiguas manejan las entradas de esa forma. Un paso adelante de esto sería que el hardware o el sistema operativo detectaran la pulsación de la tecla y lo pusieran en una cola. Luego, un programa puede verificar periódicamente la cola por nuevos eventos y reaccionar a lo que encuentre allí.

Por supuesto, este tiene que recordar de mirar la cola, y hacerlo con frecuencia, porque en cualquier momento entre que se presione la tecla y que el programa se de cuenta del evento causará que que el programa no responda. Este enfoque es llamado *sondeo*. La mayororía de los programadores prefieren evitarlo.

Un mejor mecanismo es que el sistema notifique activamente a nuestro código cuando un evento ocurre. Los navegadores hacen esto permitiéndonos registrar funciones como *manejadores* (*manejadores*) para eventos específicos.

```
<p>Da clic en este documento para activar el manejador.</p>
<script>
  window.addEventListener("click", () => {
    console.log("¿Tocaste?");
  });
```

```
</script>
```

La *vinculación* `window` se refiere a un objeto integrado proporcionado por el navegador. Este representa la ventana del navegador que contiene el documento. Llamando a su método `addEventListener` se registra el segundo argumento que se llamará siempre que ocurra el evento descrito por su primer argumento.

Eventos y nodos DOM

Cada manejador de eventos del navegador es registrado dentro de un contexto. En el ejemplo anterior llamamos a `addEventListener` en el objeto `window` para registrar un manejador para toda la ventana. Este método puede también ser encontrado en elementos DOM y en algunos otros tipos de objetos. Los controladores de eventos son llamados únicamente cuando el evento ocurra en el contexto del objeto en que están registrados.

```
<button>Presióname</button>
<p>No hay manejadores aquí.</p>
<script>
  let boton = document.querySelector("button");
  boton.addEventListener("click", () => {
    console.log("Botón presionado.");
  });
</script>
```

Este ejemplo adjunta un manejador al nodo del botón. Los clics sobre el botón hacen que se ejecute ese manejador, pero los clics sobre el resto del documento no.

Dar a un nodo un atributo `onclick` tiene un efecto similar. Esto funciona para la mayoría de tipos de eventos—se puede adjuntar un manejador a través del atributo cuyo nombre es el nombre del evento con `on` en frente de este.

Pero un nodo puede tener únicamente un atributo `onclick`, por lo que se puede registrar únicamente un manejador por nodo de esa manera. El método `addEventListener` permite agregar cualquier número de manejadores siendo seguro agregar manejadores incluso si ya hay otro manejador en el elemento.

El método `removeEventListener`, llamado con argumentos similares a `addEventListener` , remueve un manejador:

```
<button>Botón de única acción</button>
```

```
<script>
  let boton = document.querySelector("button");
  function unaVez() {
    console.log("Hecho.");
    boton.removeEventListener("click", unaVez);
  }
  boton.addEventListener("click", unaVez);
</script>
```

La función dada a `removeEventListener` tiene que ser el mismo valor de función que se le dio a `addEventListener`. Entonces, para desregistrar un manejador, se le tiene que dar un nombre a la función (`unaVez`, en el ejemplo) para poder pasar el mismo valor de función a ambos métodos.

Objetos de evento

Aunque lo hemos ignorado hasta ahora, las funciones del manejador de eventos reciben un argumento: el *objeto de evento*. Este objeto contiene información adicional acerca del evento. Por ejemplo, si queremos saber *cuál* botón del mouse fue presionado, se puede ver la propiedad `button` del objeto de evento.

```
<button>Haz clic en mí de la forma que quieras</button>
<script>
  let boton = document.querySelector("button");
  boton.addEventListener("mousedown", evento => {
    if (evento.button == 0) {
      console.log("Botón izquierdo");
    } else if (evento.button == 1) {
      console.log("Botón central");
    } else if (evento.button == 2) {
      console.log("Botón derecho");
    }
  });
</script>
```

La información almacenada en un objeto de evento es diferente por cada tipo de evento. Se discutirán los distintos tipos de eventos más adelante en el capítulo. La propiedad `type` del objeto siempre contiene una cadena que identifica al evento (como `"click"` o `"mousedown"`)

Propagación

Para la mayoría de tipos de eventos, los manejadores registrados en nodos con hijos también recibirán los eventos que sucedan en los hijos. Si se hace clic a un botón dentro de un párrafo, los manejadores de eventos del párrafo también verán el evento clic.

Pero si tanto el párrafo como el botón tienen un manejador, el manejador más específico—el del botón—es el primero en lanzarse. Se dice que el evento se *propaga* hacia afuera, desde el nodo donde este sucedió hasta el nodo padre del nodo y hasta la raíz del documento. Finalmente, después de que todos los manejadores registrados en un nodo específico hayan tenido su turno, los manejadores registrados en general ventana tienen la oportunidad de responder al evento.

En cualquier momento, un manejador de eventos puede llamar al método `stopPropagation` en el objeto de evento para evitar que los manejadores que se encuentran más arriba reciban el evento. Esto puede ser útil cuando, por ejemplo, si tienes un botón dentro de otro elemento en el que se puede hacer clic y que no se quiere que los clics sobre el botón activen el comportamiento de clic del elemento exterior.

El siguiente ejemplo registra manejadores `"mousedown"` tanto en un botón como el párrafo que lo rodea. Cuando se hace clic con el botón derecho del mouse, el manejador del botón llama a `stopPropagation`, lo que evitará que se ejecute el manejador del párrafo. Cuando se hace clic en el botón con otro botón del mouse, ambos manejadores se ejecutarán.

```
<p>Un párrafo con un <button>botón</button>.</p>
<script>
  let parrafo = document.querySelector("p");
  let boton = document.querySelector("button");
  parrafo.addEventListener("mousedown", () => {
    console.log("Manejador del párrafo.");
  });
  boton.addEventListener("mousedown", evento => {
    console.log("Manejador del botón.");
    if (evento.button == 2) evento.stopPropagation();
  });
</script>
```

La mayoría de objetos de eventos tienen una propiedad `target` que se refiere al nodo donde se originaron. Se puede usar esta propiedad para asegurar de que no se está manejando accidentalmente algo que se propagó desde un nodo

que no se desea manejar.

También es posible utilizar la propiedad `target` para lanzar una red amplia para un evento específico. Por ejemplo, si tienes un nodo que contiene una gran cantidad de botones, puede ser más conveniente el registrar un manejador en un solo clic en el nodo externo y hacer que use la propiedad `target` para averiguar si se hizo clic en un botón, en lugar de registrar manejadores individuales en todos los botones.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", evento => {
    if (evento.target.nodeName == "BUTTON") {
      console.log("Presionado", evento.target.textContent);
    }
  });
</script>
```

Acciones por defecto

La mayoría de eventos tienen una acción por defecto asociada a ellos. Si haces clic en un enlace, se te dirigirá al destino del enlace. Si presionas la flecha hacia abajo, el navegador desplazará la página hacia abajo. Si das clic derecho, se obtendrá un menú contextual. Y así.

Para la mayoría de los tipos de eventos, los manejadores de eventos de JavaScript se llamarán *antes* de que el comportamiento por defecto se produzca. Si el manejador no quiere que suceda este comportamiento por defecto, normalmente porque ya se ha encargado de manejar el evento, se puede llamar al método `preventDefault` en el objeto de evento.

Esto puede ser utilizado para implementar un atajo de teclado propio o menú contextual. Esto también puede ser utilizado para interferir de forma desagradable el comportamiento que los usuarios esperan. Por ejemplo, aquí hay un enlace que no se puede seguir:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let enlace = document.querySelector("a");
  enlace.addEventListener("click", evento => {
    console.log("Nope.");
    evento.preventDefault();
```

```
  });
</script>
```

Trata de no hacer tales cosas a menos que tengas una buena razón para hacerlo. Será desagradable para las personas que usan tu página cuando el comportamiento esperado no funcione.

Dependiendo del navegador, algunos eventos no pueden ser interceptados en lo absoluto. En Chrome, por ejemplo, el atajo de teclado para cerrar la pestaña actual (CONTROL-W o COMMAND-W) no se puede manejar con JavaScript.

Eventos de teclado

Cuando una tecla del teclado es presionado, el navegador lanza un evento "`keydown`". Cuando este es liberado, se obtiene un evento `"keyup"`.

```
<p>Esta página se pone violenta cuando se mantiene presionado la
   tecla V.</p>
<script>
  window.addEventListener("keydown", evento => {
    if (evento.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", evento => {
    if (evento.key == "v") {
      document.body.style.background = "";
    }
  });
</script>
```

A pesar de su nombre, `"keydown"` se lanza no solamente cuando la tecla es físicamente presionada. Cuando una tecla se presiona y se mantiene presionada, el evento se lanza una vez más cada que la tecla se *repite*. Algunas veces se debe tener cuidado con esto. Por ejemplo, si tienes un botón al DOM cuando el botón es presionado y removido cuando la tecla es liberada, puedes agregar accidentalmente cientos de botones cuando la tecla se mantiene presionada por más tiempo.

El ejemplo analizó la propiedad `key` del objeto de evento para ver de qué tecla se trata el evento. Esta propiedad contiene una cadena que, para la mayoría de las teclas, corresponde a lo que escribiría al presionar esa tecla. Para teclas especiales como ENTER, este contiene una cadena que nombre la tecla {`"Enter`

", en este caso}. Si mantienes presionado SHIFT mientras que presionas una tecla, esto también puede influir en el nombre de la tecla-`"v"` se convierte en `"V"` y `"1"` puede convertirse en `"!"`, es lo que se produce al presionar SHIFT-1 en tu teclado.

La teclas modificadoras como SHIFT, CONTROL, ALT y META (COMMAND en Mac) generan eventos de teclado justamente como las teclas normales. Pero cuando se busque combinaciones de teclas, también se puede averiguar si estas teclas se mantienen presionadas viendo las propiedades `shiftKey`, `ctrlKey`, `altKey` y `metaKey` de los eventos de teclado y mouse.

```
<p>Presiona Control-Espacio para continuar.</p>
<script>
  window.addEventListener("keydown", evento => {
    if (evento.key == " " && evento.ctrlKey) {
      console.log("¡Continuando!");
    }
  });
</script>
```

El nodo DOM donde un evento de teclado se origina depende en el elemento que tiene el foco cuando la tecla es presionada. La mayoría de los nodos no pueden tener el foco a menos que se les de un atributo `tabindex`, pero elementos como enlaces, botones y campos de formularios sí pueden. Volveremos a los campos de formularios en el capítulo Chapter?. Cuando nadie en particular tiene el foco, `document.body` actua como el nodo objetivo de los eventos de teclado.

Cuando el usuario está escribiendo texto, usando los eventos de teclado para averiguar qué se está escribiendo es problematico. Algunas plataformas, sobre todo el teclado virtual en teléfonos Android, no lanzan eventos de teclado. Pero incluso cuando se tiene un teclado antiguo, algunos tipos de entradas de texto no coinciden con las pulsaciones de teclas de forma sencilla, como el software *editor de métodos de entrada* (IME) usado por personas cuyos *guiones* (*script*) no encajan en un teclado, donde se combinan varias teclas para crear caracteres.

Para notar cuando se escribió algo, los elementos en los que se puede escribir, como las etiquetas `<input>` y `<textarea>`, lanzan eventos `"input"` cada vez que el usuario cambia su contenido. Para obtener el contenido real que fue escrito, es mejor leerlo directamente desde el campo seleccionado. El Chapter ? mostrará cómo.

Eventos de puntero

Actualmente, hay dos formas muy utilizadas para señalar en una pantalla: mouse (incluyendo dispositivos que actuan como mouse, como páneles táctiles y *trackballs*) y pantallas táctiles. Estos producen diferentes tipos de eventos.

Clics de mouse

Al presionar un botón del mouse se lanzan varios eventos. Los eventos "`mousedown`" y "`mouseup`" son similares a "`keydown`" y "`keyup`" y se lanzan cuando el boton es presionado y liberado. Estos ocurren en los nodos del DOM que están inmediatamente debajo del puntero del mouse cuando ocurre el evento.

Después del evento "`mouseup`", un evento "`click`" es lanzado en el nodo más específico que contiene la pulsación y la liberación del botón. Por ejemplo, si presiono el botón del mouse sobre un párrafo y entonces muevo el puntero a otro párrafo y suelto el botón, el evento "`click`" ocurrirá en el elemento que contiene ambos párrafos.

Si se producen dos clics juntos, un evento "`dblclick`" (doble-clic) también se lanza, después del segundo evento de clic.

Para obtener la información precisa sobre el lugar en donde un evento del mouse ocurrió, se puede ver en las propiedades `clientX` y `clientY`, los cuales contienen las coordenadas (en pixeles) relativas a la esquina superior izquierda de la ventana o `pageX` y `pageY`, las cuales son relativas a la esquina superior izquierda de todo el documento (el cual puede ser diferente cuando la ventana ha sido desplazada).

Lo siguiente implementa un programa de dibujo primitivo. Cada vez que se hace clic en el documento, agrega un punto debajo del puntero del mouse. Ver Chapter ? para un programa de dibujo menos primitivo.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .punto {
    height: 8px; width: 8px;
    border-radius: 4px; /* redondea las esquinas */
    background: blue;
    position: absolute;
  }
</style>
<script>
```

```
  window.addEventListener("click", evento => {
    let punto = document.createElement("div");
    punto.className = "punto";
    punto.style.left = (evento.pageX - 4) + "px";
    punto.style.top = (evento.pageY - 4) + "px";
    document.body.appendChild(punto);
  });
</script>
```

Movimiento del mouse

Cada vez que el puntero del mouse se mueve, un evento `"mousemove"` es lanzado. Este evento puede ser utilizado para rastrear la posición del mouse. Una situación común en la cual es útil es cuando se implementa una funcionalidad de mouse-arrastrar.

Como un ejemplo, el siguiente programa muestra una barra y configura los manejadores de eventos para que cuando se arrastre hacia la izquierda o a la derecha en esta barra la haga más estrecha o más ancha:

```
<p>Dibuja la barra para cambiar su anchura:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let ultimoX; // Rastrea la última posición de X del mouse
     observado
  let barra = document.querySelector("div");
  barra.addEventListener("mousedown", evento => {
    if (evento.button == 0) {
      ultimoX = evento.clientX;
      window.addEventListener("mousemove", movido);
      evento.preventDefault(); // Evitar la selección
    }
  });

  function movido(evento) {
    if (evento.buttons == 0) {
      window.removeEventListener("mousemove", movido);
    } else {
      let distancia = event.clientX - lastX;
      let nuevaAnchura = Math.max(10, barra.offsetWidth + distancia)
         ;
      barra.style.width = nuevaAnchura + "px";
      ultimoX = evento.clientX;
```

```
    }
  }
</script>
```

La página resultante se ve así:

{{figure {url: “img/drag-bar.png”, alt: “Una barra arrastrable”, width: “5.3cm”}}}

Tener en cuenta que el manejador `"mousemove"` es registrado sobre toda la ventana. Incluso si el mouse se sale de la barra durante el cambio de tamaño, siempre que se mantenga presionado el botón, su tamaño se actualizará.

Se debe dejar de cambiar el tamaño de la barra cuando se suelta el botón del mouse. Para eso, podemos usar la propiedad `buttons` (nótese el plural), que informa sobre los botones que se mantienen presionados actualmente. Cuando este es cero, no hay botones presionados. Cuando los botones se mantienen presionados, su valor es la suma de los códigos de esos botones—el botón izquierdo tiene el código 1, el botón derecho 2 y el central 4. De esta manera, puedes verificar si un botón dado es presionado tomando el resto del valor de `buttons` y su código.

Tener en cuenta que el orden de los códigos es diferente del utilizado por `button`, donde el botón central se encuentra que el derecho. Como se mencionó, la coherencia no es realmente un punto fuerte de la interfaz de programación del navegador.

Eventos de toques

El estilo del navegador gráfico que usamos fue diseñado con la interfaz de mouse en mente, en un tiempo en el cual las pantallas táctiles eran raras. Para hacer que la Web “funcionara” en los primeros teléfonos con pantalla táctil, los navegadores de esos dispositivos pretendían, hasta cierto punto, que los eventos táctiles fueran eventos del mouse. Si se toca la pantalla, se obtendrán los eventos `"mousedown"`, `"mouseup"` y `"click"`.

Pero esta ilusión no es muy robusta. Una pantalla táctil funciona de manera diferente a un mouse: no tiene multiples botones, no puedes rastrear el dedo cuando no está en la pantalla (para simular `"mousemove"`) y permite que multiples dedos estén en la pantalla al mismo tiempo.

Los eventos del mouse cubren las interacciones solamente en casos sencillos—si se agrega un manejador `"click"` a un butón, los usuarios táctiles aún podrán usarlo. Pero algo como la barra redimensionable del ejemplo anterior no funciona en una pantalla táctil.

Hay tipos de eventos específicos lanzados por la interacción táctil. Cuando un dedo inicia tocando la pantalla, se obtiene el evento `"touchstart"`. Cuando este es movido mientras se toca, se lanza el evento `"touchmove"`. Finalmente, cuando se deja de tocar la pantalla, se lanzará un evento `"touchend"`.

Debido a que muchas pantallas táctiles pueden detectar multiples dedos al mismo tiempo, estos eventos no tienen un solo conjunto de coordenadas asociados a ellos. Más bien, sus objetos de evento tienen una propiedad `touches`, el cual contiene un objeto tipo matriz de puntos, cada uno de ellos tiene sus propias propiedades `clientX`, `clientY`, `pageX` y `pageY`.

Se puede hacer algo como esto para mostrar circulos rojos alrededor de cada que toca:

```
<style>
  punto { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Toca esta página</p>
<script>
  function actualizar(event) {
    for (let punto; punto = document.querySelector("punto");) {
      punto.remove();
    }
    for (let i = 0; i < evento.touches.length; i++) {
      let {paginaX, paginaY} = evento.touches[i];
      let punto = document.createElement("punto");
      punto.style.left = (paginaX - 50) + "px";
      punto.style.top = (paginaY - 50) + "px";
      document.body.appendChild(punto);
    }
  }
  window.addEventListener("touchstart", actualizar);
  window.addEventListener("touchmove", actualizar);
  window.addEventListener("touchend", actualizar);
</script>
```

A menudo se querrá llamar a `preventDefault` en un manejador de eventos táctiles para sobreescribir el comportamiento por defecto del navegador (que puede incluir desplazarse por la pagina al deslizar el dedo) y para evitar que los eventos del mouse se lancen, para lo cual se puede tener *también* un manejador.

Eventos de desplazamiento

Siempre que un elemento es desplazado, un evento `"scroll"` es lanzado. Esto tiene varios usos, como son saber qué está mirando el usuario actualmente (para deshabilitar la animación fuera de pantalla o enviar informes espías a su sede maligna) o mostrar alguna indicación de progreso (resaltando parte de una tabla de contenido o mostrando un número de página).

El siguiente ejemplo dibuja una barra de progreso sobre el documento y lo actualiza para que se llene a medida que se desplza hacia abajo:

```
<style>
  #progreso {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progreso"></div>
<script>
  // Crear algo de contenido
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticoespialidoso ".repeat(1000)));

  let barra = document.querySelector("#progreso");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    barra.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>
```

Darle a un elemento una `position` de `fixed` actua como una posición `absolute` pero también evita que se desplace junto con el resto del documento. El efecto es hacer que la barra de progreso permanezca en la parte superior. Su ancho cambia para indicar el progreso actual. Se usa `%`, en lugar de `px`, como una unidad cuando se configura el ancho para que el tamaño del elemento sea relativo al ancho de la página.

La vincualación global `innerHeight` proporciona la altura de la ventana, que se tiene que restar de la altura total desplazable—no se puede seguir desplazando cuando se llega al final del documento. También hay un `innerWidth` para el ancho de la ventana. Al dividir `pageYOffset`, la posición de desplazamiento actual, por la posición de desplazamiento máximo y mulplicado por

100, se obtiene el porcentaje de la barra de progreso.

Al llamar `preventDefault` en un evento de desplazamiento no evita que el desplazamiento se produzca. De hecho, el manejador de eventos es llamado unicamente *después* de que se realiza el desplazamiento.

Eventos de foco

Cuando un elemento gana el foco, el navegador lanza un evento de `"focus"` en él. Cuando este pierde el foco, el elemento obtiene un evento `"blur"`.

A diferencia de los eventos discutidos con anterioridad, estos dos eventos no se propagan. Un manejador en un elemento padre no es notificado cuando un elemento hijo gana o pierde el foco.

El siguiente ejemplo muestra un texto de ayuda para campo de texto que actualmente tiene el foco:

```
<p>Nombre: <input type="text" dato-ayuda="Tu nombre"></p>
<p>Edad: <input type="text" dato-ayuda="Tu edad en años"></p>
<p id="ayuda"></p>

<script>
  let ayuda = document.querySelector("#ayuda");
  let campos = document.querySelectorAll("input");
  for (let campo of Array.from(campos)) {
    campo.addEventListener("focus", evento => {
      let texto = evento.target.getAttribute("dato-ayuda");
      ayuda.textContent = texto;
    });
    field.addEventListener("blur", evento => {
      ayuda.textContent = "";
    });
  }
</script>
```

Esta captura de pantalla muestra el texto de ayuda para el campo edad.

{{figure {url: “img/help-field.png”, alt: “Brindar ayuda cuando un campo está enfocado”, width: “4.4cm”}}}

El objeto window recibirá eventos de `"focus"` y `"blur"` cuando el usuario se mueva desde o hacia la pestaña o ventana del navegador en la que se muestra el documento.

Evento de carga

Cuando una página termina de cargarse, el evento `"load"` es lanzado en la ventana y en los objetos del cuerpo del documento. Esto es usado a menudo para programar acciones de inicialización que requieren que todo el documento haya sido construido. Recordar que el contenido de las etiquetas `<script>` se ejecutan inmediatamente cuando la etiqueta es encontrada. Esto puede ser demasiado pronto, por ejemplo cuando el guión necesita hacer algo con las partes del documento que aparecen después de la etiqueta `<script>`.

Elementos como imagenes y etiquetas de guiones que cargan un archivo externo también tienen un evento `"load"` que indica que se cargaron los archivos a los que hacen referencia. Al igual que los eventos relacionado con el foco, los eventos de carga no se propagan.

Cuando una página se cierra o se navega fuera de ella (por ejemplo, siguiendo un enlace), un evento `"beforeunload"` es lanzado. El principal uso de este evento es evitar que el usuario pierda su trabajo accidentalmente al cerrar un documento. Si se evita el comportamiento por defecto en este evento *y* se establece la propiedad `returnValue` en el objeto de evento a una cadena, el navegador le mostrará al usuario un diálogo preguntándole si realmente quiere salir de la página. Ese cuadro de diálogo puede incluir una cadena de texto, pero debido a que algunos sitios intentan usar estos cuadros de diálogo para confundir a las personas para que permanezcan en su página para ver anuncios poco fiables sobre la pérdida de peso, la mayoría de los navegadores ya no lo muestran.

Eventos y el ciclo de eventos

En el contexto del ciclo de eventos, como se explica en el Chapter 11, los manejadores de eventos del navegador se comportan como otras notificaciones asíncronas. Se programan cuando el evento ocurre pero deben esperar a que finalicen otros guiones que se están ejecutando antes de que tengan la oportunidad de ejecutarse.

El hecho de que los eventos puedan ser procesados solo cuando no se está ejecutando nada más que, si el ciclo de eventos está vinculado con otro trabajo, cualquier interacción con la página (que ocurre a través de eventos) se retrasará hasta que haya tiempo para procesarlo. Por lo tanto, si programas demasiado trabajo, ya sea con manejadores de eventos de ejecución prolongada o con muchos de breve ejecución, la página se volverá lenta y engorrosa de usar.

Para casos en donde *realmente* se quiera hacer algo que requiera mucho

tiempo en segundo plano sin congelar la página, los navegadores proveeen algo llamado **web worker**. Un worker es un proceso de JavaScript que se jecuta junto con el guión principal, en su propia línea de tiempo.

Imaginar que se eleva un número al cuadrado es un cálculo pesado y de larga duración que se quiere realizar en un hilo separado. Se podría escribir un archivo llamado `code/cuadradoworker.js` que responde a los mensajes calculando un cuadrado y enviando de vuelta un mensaje.

```
addEventListener("message", evento => {
  enviarMensaje(evento.data * evento.data);
});
```

Para evitar el problema de tener multiples hilos tocando los mismos datos, los workers no comparten su alcance global o cualquier otro tipo de dato con el entorno del guión principal. En cambio, tienes que comunicarte con ellos enviando mensajes de un lado a otro.

Este código genera un worker que ejecuta ese guión, le envía algunos mensajes y genera las respuestas.

```
let cuadradoWorker = new Worker("code/cuadradoworker.js");
cuadradoWorker.addEventListener("message", evento => {
  console.log("El respondió:", evento.data);
});
cuadradoWorker.enviarMensaje(10);
cuadradoWorker.enviarMensaje(24);
```

La función `enviarMensaje` envía un mensaje, que provocará que se lance un evento `"message"`en el receptor. El guión que creó al worker envía y recibe mensajes a través del objeto `Worker`, mientras que el worker habla con el guión que lo creó enviando y escuchando directamente en su alcance global. Solo los valores que pueden ser representados como un JSON pueden ser enviados como un mensaje—el otro lado recibirá una *copia* de ellos, en lugar del valor en sí.

Temporizadores

Se mostró la función `establecerTiempoEspera` en el Chapter 11. Este programa otra función para que se llame más tarde, después de un número determinado de milisegundos.

Algunas veces se necesita cancelar una función que se haya programado. Esto se hace almacenando el valor retornado por `establecerTiempoEspera` y

llamando a `reinicarTiempoEspera` en él.

```
let temporizadorBomba = setTimeout(() => {
  console.log("¡BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% de cambio
  console.log("Desactivada.");
  reinicarTiempoEspera(temporizadorBomba);
}
```

La función `cancelAnimationFrame` funciona de la misma forma que `reinicarTiempoEspera` —llamarla en un valor devuelto por `requestAnimationFrame` cancelará ese marco (asumiendo que aún no ha sido llamado).

Un conjunto similar de funciones, `setInterval` y `clearInterval`, son usadas para restablecer los temporizadores que deberían *repetirse* cada X milisegundos.

```
let tictac = 0;
let reloj = setInterval(() => {
  console.log("tictac", tictac++);
  if (tictac == 10) {
    clearInterval(reloj);
    console.log("Detener.");
  }
}, 200);
```

Antirrebote

Algunos tipos de eventos tienen el potencial de ser lanzados rápidamente, muchas veces seguidos (los eventos `"mousemove"` y `"scroll"`, por ejmplo). Cuando se menejan tales eventos, se debe tener cuidado de no hacer nada que consuma demasiado tiempo o su manejador tomará tanto tiempo que la interacción con el documento comenzará a sentirse lenta.

Si necesitas hacer algo no trivial en algún manejador de este tipo, se puede usar `setTimeout` para asegurarse de que no se está haciendo con demasiada frecuencia. Esto generalmente se llama *antirrebote* del evento. Hay varios enfoques ligeramente diferentes para esto.

En el primer ejemplo, se quiere reaccionar cuando el usuario ha escrito algo, pero no se quiere hacer inmediatamente por cada evento de entrada. Cuando se está escribiendo rápidamente, se requiere esperar hasta que se produzca una

pausa. En lugar de realizar inmediatamente una acción en el manejador de eventos, se establece un tiempo de espera. También se borra el tiempo de espera anterior (si lo hay) para que cuando los eventos ocurran muy juntos (más cerca que el tiempo de espera), el tiempo de espera del evento anterior será cancelado.

```
<textarea>Escribe algo aquí...</textarea>
<script>
  let areaTexto = document.querySelector("textarea");
  let tiempoEspera;
  areaTexto.addEventListener("input", () => {
    clearTimeout(tiempoEspera);
    tiempoEspera = setTimeout(() => console.log("¡Escribió!"), 500);
  });
</script>
```

Dar un valor indefinido a `clearTimeout` o llamándolo en un tiempo de espera que ya ha se ha lanzado no tiene ningún efecto. Por lo tanto, no debemos tener cuidado sobre cuándo llamarlo, y simplemente se hace para cada evento.

Podemos utilizar un patrón ligeramente diferente si se quiere espaciar las respuestas de modo que estén separadas por al menos una cierta longitud de tiempo pero se quiere lanzar *durante* una serie de eventos, no solo después. Por ejemplo, se podría querer responder a los eventos `"mousemove"` mostrando las coordenadas actuales del mourse pero solo cada 250 milisegundos.

```
<script>
  let programado = null;
  window.addEventListener("mousemove", evento => {
    if (!programado) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse en ${programado.pageX}, ${programado.pageY}`;
        programado = null;
      }, 250);
    }
    programado = evento;
  });
</script>
```

Resumen

Los manejadores de eventos hacen posible detectar y reaccionar a eventos que suceden en nuestra página web. El método `addEventListener` es usado para registrar un manejador de eventos.

Cada evento tiene un tipo (`"keydown"`, `"focus"`, etc.) que lo identifica. La mayoría de eventos son llamados en un elemento DOM específico y luego se *propagan* a los ancentros de ese elemento, lo que permite que los manejadores asociados con esos elementos los manejen.

Cuando un manejador de evento es llamado, se le pasa un objeto evento con información adicional acerca del evento. Este objeto también tiene métodos que permiten detener una mayor propagación (`stopPropagation`) y evitar que el navegador maneje el evento por defecto (`preventDefault`).

Al presiosar una tecla se lanza los eventos `"keydown"` y `"keyup"`. Al presionar un botón del mouse se lanzan los eventos `"mousedown"`, `"mouseup"` y `"click"` . Al mover el mouse se lanzan los eventos `"mousemove"`. Las interacción de la pantalla táctil darán como resultado eventos `"touchstart"`, `"touchmove"` y `"touchend"`.

El desplazamiento puede ser detectado con el evento `"scroll"` y los cambios de foco pueden ser detactados con los eventos `"focus"` y `"blur"`. Cuando el documento termina de cargarse, se lanza el evento `"load"` en la ventana.

Ejercicios

Globo

Escribe una página que muestre un globo (usando el globo emoji, 🎈). Cuando se presione la flecha hacia arriba, debe inflarse (crecer) un 10 por cierto, y cuando se presiona la flecha hacia abajo, debe desinflarse (contraerse) un 10 por cierto)

Puedes controlar el tamaño del texto (los emojis son texto) configurando la propiedad CSS `font-size` (`style.fontSize`) en su elemento padre. Recuerda incluir una unidad en el valor, por ejemplo pixeles (`10px`).

Los nombres de las teclas de flecha son `"ArrowUp"` y `"ArrowDown"`. Asegúratede que las teclas cambien solo al globo, sin desplazar la página.

Cuando eso funcione, agrega una nueva función en la que, si infla el globo más allá de cierto tamaño, explote. En este caso, explotar significa que se reemplaza con un emoji 💥, y se elimina el manejador de eventos (para que no se pueda inflar o desinflar la explosión).

Mouse trail

En los primeros días de JavaScript, que era el momento de páginas de inicio llamativas con muchas imágenes, a la gente se le ocurrieron formas realmente inspiradoras de usar el lenguaje.

Uno de estos fue el *rastro del mouse*, una serie de elementos que seguirían el puntero del mouse mientras lo movías por la página.

En este ejercicio, quiero que implementes un rastro del mouse. Utiliza elementos `<div>` con un tamaño fijo y un color de fondo (consulta a code en la sección "Clics del mouse" por un ejemplo). Crea un montón de estos elementos y, cuando el mouse se mueva, muestralos después del puntero del mouse.

Hay varios enfoques posibles aquí. Puedes hacer tu solución tan simple o tan compleja como desees. Una solución simple para comenzar es mantener un número de elementos de seguimiento fijos y recorrerlos, moviendo el sigueinte a la posición actual del mouse cada vez que ocurra un evento `"mousemove"`.

Pestañas

Los paneles con pestañas son utilizados ampliamente en las interfaces de usuario. Te permiten seleccionar un panel de interfaz eligiendo entre una serie de pestañas "que sobresalen" sobre un elemento.

En este ejercicio debes implementar una interfaz con pestañas simple. Escribe una función, `asTabs`, que tome un nodo DOM y cree una interfaz con pestañas que muestre los elementos secundarios de ese nodo. Se debe insertar una lista de elementos `<button>` en la parte superior del nodo, uno para cada elemento hijo, que contenga texto recuperado del atributo `data-tabname` del hijo. Todos menos uno de los elementos secundarios originales deben estar ocultos (dado un estilo `display` de`none`). El nodo visible actualmente se puede seleccionar haciendo clic en los botones.

Cuando eso funcione, extiéndelo para diseñar el botón de la pestaña seleccionada actualmente de manera diferente para que sea obvio qué pestaña está seleccionada.

"Toda realidad es un juego."

—Iain Banks, El Jugador de Juegos

Chapter 15

Proyecto: Un Juego de Plataforma

Mucha de mi fascinación inicial con computadoras, como la de muchos niños nerd, tiene que ver con juegos de computadora. Fui atraído a los pequeños mundos simulados que podía manipular y en los que las historias (algo así) revelaban más, yo supongo, por la manera en que yo proyectaba mi imaginación en ellas más que por las posibilidades que realmente ofrecían.

No le deseo a nadie una carrera en programación de juegos. Tal como la industria musical, la discrepancia entre el número de personas jóvenes ansiosas queriendo trabajar en ella y la actual demanda de tales personas crea un ambiente bastante malsano.

Este capítulo recorrerá la implementación de un juego de plataforma pequeño. Los juegos de plataforma (o juegos de "brincar y correr") son juegos que esperan que el jugador mueva una figura a través de un mundo, el cual usualmente es bidimensional y visto de lado, mientras brinca encima y sobre cosas.

El juego

Nuestro juego estará ligeramente basado en Dark Blue (*www.lessmilk.com/games/10*) de Thomas Palef. Escogí ese juego porque es tanto entretenido como minimalista y porque puede ser construido sin demasiado código. Se ve así:

La caja oscura representa al jugador, cuya tarea es recolectar las cajas amarillas (monedas) mientras evita la cosa roja (lava). Un nivel es completado cuando todas las monedas han sido recolectadas.

El jugador puede caminar alrededor con las teclas de las flechas izquierda y derecha y pueden brincar con la flecha arriba. Brincar es una especilidad de este personaje de juego. Puede alcanzar varias veces su propia altura y puede cambiar de dirección en el aire. Esto puede no ser completamente realista, pero ayuda a darle al jugador el sentimiento de estar en control directo del avatar.

El juego consiste en un fondo estático, expuesto como una cuadrícula, con los elementos móviles superpuestos sobre ese fondo. Cada campo en la cuadrícula está ya sea vacío, sólido o lava. Las posiciones de estos elementos no están restringidas a la cuadrícula-sus coordenadas pueden ser fraccionales, permitiendo un movimiento suave.

La tecnología

Usaremos el DOM del navegador para mostrar el juego, y leeremos la entrada del usuario por medio del manejo los eventos de teclas.

El código relacionado con la pantalla y el teclado es sólo una pequeña parte del trabajo que necesitamos hacer para construir este juego. Ya que todo parece como cajas coloreadas, dibujar es no es complicado: creamos los elementos del DOM y usamos estilos para darlos un color de fondo, tamaño y posición.

Podemos representar el fondo como una tabla ya que es una cuadrícula invariable de cuadrados. Los elementos libres de moverse pueden ser superpuestos usando elementos posicionados absolutamente.

En los juegos y otros programas que deben animar gráficos y responder a la entrada del usuario sin retraso notable, la eficiencia es importante. Aunque el DOM no fue diseñado originalmente para gráficos de alto rendimiento, es realmente mejor en ello de lo que se esperaría. Viste algunas animaciones en Chapter 13. En una máquina moderna, un simple juego como este desempeña bien, aún si no nos preocupamos mucho de la optimización.

En el siguiente capítulo, exploraremos otra tecnología del navegador, la etiqueta `<canvas>`, la cual provee un forma más tradicional de dibujar gráficos, trabajando en término de formas y pixeles más que elementos del DOM.

Niveles

Vamos a querer una forma de especificar niveles que sea fácilmente leíble y editable por humanos. Ya que está bien que todo empiece sobre una cuadrícula,

podríamos usar cadenas de caracteres largas en las que cada caracter representa un elemento-ya sea una parte de la cuadrícula de fondo o un elemento móvil.

El plano para un nivel pequeño podría lucir como esto:

```
let planoDeNivel = `
......................
..#................#..
..#..............=.#..
..#.........o.o....#..
..#.@......#####...#..
..#####............#..
......#+++++++++++++#..
......##############..
......................`;
```

Los puntos son espacios vacíos, los signos de número (`#`) son muros y los signos de suma son lava. La posición inicial del jugador es el arroba (`@`). Cada caracter O es una moneda y el signo de igual (`=`) en la parte superior es un bloque de lava que se mueve de un lado a otro horizontalmente.

Vamos a soportar dos tipos adicionales de lava en movimento: el caracter de la barra vertical (`|`) crea gotas que mueven verticalmente y la `v` indica lava *goteando*-la lava que se mueve verticalmente que no rebota de un lado a otro sino que sólo se mueve hacia abajo, brincando a su posición inicial cuando toca el suelo.

Un juego completo consiste en múltiples niveles que el jugador debe completar. Un nivel es completado cuando todas las monedass han sido recolectadas. Si el jugador toca la lava, el nivel actual se restaura a su posición inicial y el juego puede intentar de nuevo.

Leyendo un nivel

La siguiente clase guarda un objeto de nivel. Su argumento debe ser la cadena de carateres que define el nivel.

```
class Nivel {
  constructor(plano) {
    let filas = plano.trim().split("\n").map(l => [...l]);
    this.height = filas.length;
    this.width = filas[0].length;
    this.iniciarActores = [];

    this.filas = filas.map((fila, y) => {
```

```
      return fila.map((car, x) => {
        let tipo = caracteresDelNivel[car];
        if (typeof tipo == "string") return tipo;
        this.iniciarActores.push(
          tipo.create(new Vector(x, y), car));
        return "vacío";
      });
    });
  }
}
```

El método `trim` es usado para remover el espacio en blanco al inicio y al final de la cadena de caracteres del plano. Esto permite a nuestro plano de ejemplo empezar con una nueva línea de manera que todas las líneas esten directamente una debajo de la otra. La cadena restante es dividida en caracteres de nueva línea y cada línea es propagada en un arreglo, produciendo un arreglo de caracteres.

Entonces `filas` guarda un arreglo de arreglos de caracteres, las filas del plano. Podemos derivar el ancho y alto del nivel de ellas. Pero aún debemos separar los elementos móviles de la cuadrícula de fondo. Llamaremos a los elementos móviles *actores*. Ellos se guardaran en un arreglo de objetos. El fondo será un arreglo de arreglos de caracteres, que tendrán tipos de campos como `"vacío"`, `"muro"` o `"lava"`.

Para crear estos arreglos, mapearemos sobre las filas y luego sobre su contenido. Recuerda que `map` pasa el índice del arreglo como un segundo argumento a la función de mapeo, que nos dice las coordenadas x y las coordenadas y de un caracter dado. Las posiciones en el juego serán guardadas como pares de coordenadas, con la izquierda superior siendo 0,0 y cada cuadrado del fondo siendo 1 unidad de alto y ancho.

Para interpretar los caracteres en el plano, el constructor del `Nivel` usa el objeto de `caracteresDelNivel`, el cual mapea los elementos a cadenas de caracteres y caracteres de actores a clases. Cuando `tipo` está en la clase actor, su método estático `create` es usado para crear un objeto, el cual es agregado a `iniciarActores` y la función de mapeo regresa `"vacío"` para este cuadrado de fondo.

La posición del actor es guardada como un objeto `Vector`. Este es un vector bidimensional, un objeto con propiedades `x` y `y`, como se vió en los ejercicios de Chapter 6.

Mientras el juego se ejecuta, los actores terminarán en diferentes lugares o incluso desaparecerán completamente (como las monedas lo hacen cuando son

recolectadas). Usaremos una clase `Estado` para dar seguimiento al estado de un juego que en ejecución.

```
class Estado {
  constructor(nivel, actores, estatus) {
    this.nivel = nivel;
    this.actores = actores;
    this.estatus = estatus;
  }

  static start(nivel) {
    return new Estado(nivel, nivel.iniciarActores, "jugando");
  }

  get jugador() {
    return this.actores.find(a => a.tipo == "jugador");
  }
}
```

La propiedad `estatus` cambiará a `"perdido"` or `"ganado"` cuando el juego haya terminado.

Esto es de nuevo una estructura de datos persistente-actualizar el estado del juego crea un nuevo estado y deja el anterior intacto.

Actors

Actor objects represent the current position and state of a given moving element in our game. All actor objects conform to the same interface. Their `pos` property holds the coordinates of the element's top-left corner, and their `size` property holds its size.

Then they have an `update` method, which is used to compute their new state and position after a given time step. It simulates the thing the actor does—moving in response to the arrow keys for the player and bouncing back and forth for the lava—and returns a new, updated actor object.

A `type` property contains a string that identifies the type of the actor—`"player"`, `"coin"`, or `"lava"`. This is useful when drawing the game—the look of the rectangle drawn for an actor is based on its type.

Actor classes have a static `create` method that is used by the `Level` constructor to create an actor from a character in the level plan. It is given the coordinates of the character and the character itself, which is needed because the `Lava` class handles several different characters.

This is the `Vector` class that we'll use for our two-dimensional values, such as the position and size of actors.

```
class Vector {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vector(this.x + other.x, this.y + other.y);
  }
  times(factor) {
    return new Vector(this.x * factor, this.y * factor);
  }
}
```

The `times` method scales a vector by a given number. It will be useful when we need to multiply a speed vector by a time interval to get the distance traveled during that time.

The different types of actors get their own classes since their behavior is very different. Let's define these classes. We'll get to their `update` methods later.

The player class has a property `speed` that stores its current speed to simulate momentum and gravity.

```
class Player {
  constructor(pos, speed) {
    this.pos = pos;
    this.speed = speed;
  }

  get type() { return "player"; }

  static create(pos) {
    return new Player(pos.plus(new Vector(0, -0.5)),
                      new Vector(0, 0));
  }
}

Player.prototype.size = new Vector(0.8, 1.5);
```

Because a player is one-and-a-half squares high, its initial position is set to be half a square above the position where the `@` character appeared. This way, its bottom aligns with the bottom of the square it appeared in.

The `size` property is the same for all instances of `Player`, so we store it on

the prototype rather than on the instances themselves. We could have used a getter like `type`, but that would create and return a new `Vector` object every time the property is read, which would be wasteful. (Strings, being immutable, don't have to be re-created every time they are evaluated.)

When constructing a `Lava` actor, we need to initialize the object differently depending on the character it is based on. Dynamic lava moves along at its current speed until it hits an obstacle. At that point, if it has a `reset` property, it will jump back to its start position (dripping). If it does not, it will invert its speed and continue in the other direction (bouncing).

The `create` method looks at the character that the `Level` constructor passes and creates the appropriate lava actor.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vector(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vector(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vector(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vector(1, 1);
```

`Coin` actors are relatively simple. They mostly just sit in their place. But to liven up the game a little, they are given a "wobble", a slight vertical back-and-forth motion. To track this, a coin object stores a base position as well as a `wobble` property that tracks the phase of the bouncing motion. Together, these determine the coin's actual position (stored in the `pos` property).

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
```

```
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vector(0.2, 0.1));
    return new Coin(basePos, basePos,
                    Math.random() * Math.PI * 2);
  }
}

Coin.prototype.size = new Vector(0.6, 0.6);
```

In Chapter 13, we saw that `Math.sin` gives us the y-coordinate of a point on a circle. That coordinate goes back and forth in a smooth waveform as we move along the circle, which makes the sine function useful for modeling a wavy motion.

To avoid a situation where all coins move up and down synchronously, the starting phase of each coin is randomized. The *phase* of `Math.sin`'s wave, the width of a wave it produces, is 2π. We multiply the value returned by `Math.random` by that number to give the coin a random starting position on the wave.

We can now define the `levelChars` object that maps plan characters to either background grid types or actor classes.

```
const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};
```

That gives us all the parts needed to create a `Level` instance.

```
let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9
```

The task ahead is to display such levels on the screen and to model time and motion inside them.

Encapsulation as a burden

Most of the code in this chapter does not worry about encapsulation very much for two reasons. First, encapsulation takes extra effort. It makes programs bigger and requires additional concepts and interfaces to be introduced. Since there is only so much code you can throw at a reader before their eyes glaze over, I've made an effort to keep the program small.

Second, the various elements in this game are so closely tied together that if the behavior of one of them changed, it is unlikely that any of the others would be able to stay the same. Interfaces between the elements would end up encoding a lot of assumptions about the way the game works. This makes them a lot less effective—whenever you change one part of the system, you still have to worry about the way it impacts the other parts because their interfaces wouldn't cover the new situation.

Some *cutting points* in a system lend themselves well to separation through rigorous interfaces, but others don't. Trying to encapsulate something that isn't a suitable boundary is a sure way to waste a lot of energy. When you are making this mistake, you'll usually notice that your interfaces are getting awkwardly large and detailed and that they need to be changed often, as the program evolves.

There is one thing that we *will* encapsulate, and that is the drawing subsystem. The reason for this is that we'll display the same game in a different way in the next chapter. By putting the drawing behind an interface, we can load the same game program there and plug in a new display module.

Drawing

The encapsulation of the drawing code is done by defining a *display* object, which displays a given level and state. The display type we define in this chapter is called `DOMDisplay` because it uses DOM elements to show the level.

We'll be using a style sheet to set the actual colors and other fixed properties of the elements that make up the game. It would also be possible to directly assign to the elements' `style` property when we create them, but that would produce more verbose programs.

The following helper function provides a succinct way to create an element and give it some attributes and child nodes:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
```

```
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

A display is created by giving it a parent element to which it should append itself and a level object.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

The level's background grid, which never changes, is drawn once. Actors are redrawn every time the display is updated with a given state. The `actorLayer` property will be used to track the element that holds the actors so that they can be easily removed and replaced.

Our coordinates and sizes are tracked in grid units, where a size or distance of 1 means one grid block. When setting pixel sizes, we will have to scale these coordinates up—everything in the game would be ridiculously small at a single pixel per square. The `scale` constant gives the number of pixels that a single unit takes up on the screen.

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
        ...row.map(type => elt("td", {class: type})))
  ));
}
```

As mentioned, the background is drawn as a `<table>` element. This nicely corresponds to the structure of the `rows` property of the level—each row of the grid is turned into a table row (`<tr>` element). The strings in the grid are used as class names for the table cell (`<td>`) elements. The spread (triple dot) operator is used to pass arrays of child nodes to `elt` as separate arguments.

The following CSS makes the table look like the background we want:

```
.background    { background: rgb(52, 166, 251);
                 table-layout: fixed;
                 border-spacing: 0;              }
.background td { padding: 0;                     }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;              }
```

Some of these (`table-layout`, `border-spacing`, and `padding`) are used to suppress unwanted default behavior. We don't want the layout of the table to depend upon the contents of its cells, and we don't want space between the table cells or padding inside them.

The `background` rule sets the background color. CSS allows colors to be specified both as words (`white`) or with a format such as `rgb(R, G, B)`, where the red, green, and blue components of the color are separated into three numbers from 0 to 255. So, in `rgb(52, 166, 251)`, the red component is 52, green is 166, and blue is 251. Since the blue component is the largest, the resulting color will be bluish. You can see that in the `.lava` rule, the first number (red) is the largest.

We draw each actor by creating a DOM element for it and setting that element's position and size based on the actor's properties. The values have to be multiplied by `scale` to go from game units to pixels.

```
function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}
```

To give an element more than one class, we separate the class names by spaces. In the CSS code shown next, the `actor` class gives the actors their absolute position. Their type name is used as an extra class to give them a color. We don't have to define the `lava` class again because we're reusing the class for the lava grid squares we defined earlier.

```
.actor  { position: absolute;            }
.coin   { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64);   }
```

The `syncState` method is used to make the display show a given state. It first removes the old actor graphics, if any, and then redraws the actors in their new positions. It may be tempting to try to reuse the DOM elements for actors, but to make that work, we would need a lot of additional bookkeeping to associate actors with DOM elements and to make sure we remove elements when their actors vanish. Since there will typically be only a handful of actors in the game, redrawing all of them is not expensive.

```
DOMDisplay.prototype.syncState = function(state) {
  if (this.actorLayer) this.actorLayer.remove();
  this.actorLayer = drawActors(state.actors);
  this.dom.appendChild(this.actorLayer);
  this.dom.className = `game ${state.status}`;
  this.scrollPlayerIntoView(state);
};
```

By adding the level's current status as a class name to the wrapper, we can style the player actor slightly differently when the game is won or lost by adding a CSS rule that takes effect only when the player has an ancestor element with a given class.

```
.lost .player {
  background: rgb(160, 64, 64);
}
.won .player {
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;
}
```

After touching lava, the player's color turns dark red, suggesting scorching. When the last coin has been collected, we add two blurred white shadows—one to the top left and one to the top right—to create a white halo effect.

We can't assume that the level always fits in the *viewport*—the element into which we draw the game. That is why the `scrollPlayerIntoView` call is needed. It ensures that if the level is protruding outside the viewport, we scroll that viewport to make sure the player is near its center. The following CSS gives the game's wrapping DOM element a maximum size and ensures that anything that sticks out of the element's box is not visible. We also give it a relative position so that the actors inside it are positioned relative to the level's top-left corner.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

In the `scrollPlayerIntoView` method, we find the player's position and update the wrapping element's scroll position. We change the scroll position by manipulating that element's `scrollLeft` and `scrollTop` properties when the player is too close to the edge.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // The viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
                         .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
```

```
};
```

The way the player's center is found shows how the methods on our `Vector` type allow computations with objects to be written in a relatively readable way. To find the actor's center, we add its position (its top-left corner) and half its size. That is the center in level coordinates, but we need it in pixel coordinates, so we then multiply the resulting vector by our display scale.

Next, a series of checks verifies that the player position isn't outside of the allowed range. Note that sometimes this will set nonsense scroll coordinates that are below zero or beyond the element's scrollable area. This is okay—the DOM will constrain them to acceptable values. Setting `scrollLeft` to -10 will cause it to become 0.

It would have been slightly simpler to always try to scroll the player to the center of the viewport. But this creates a rather jarring effect. As you are jumping, the view will constantly shift up and down. It is more pleasant to have a "neutral" area in the middle of the screen where you can move around without causing any scrolling.

We are now able to display our tiny level.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```

The `<link>` tag, when used with `rel="stylesheet"`, is a way to load a CSS file into a page. The file `game.css` contains the styles necessary for our game.

Motion and collision

Now we're at the point where we can start adding motion—the most interesting aspect of the game. The basic approach, taken by most games like this, is to split time into small steps and, for each step, move the actors by a distance corresponding to their speed multiplied by the size of the time step. We'll measure time in seconds, so speeds are expressed in units per second.

Moving things is easy. The difficult part is dealing with the interactions between the elements. When the player hits a wall or floor, they should not simply move through it. The game must notice when a given motion causes an object to hit another object and respond accordingly. For walls, the motion must be stopped. When hitting a coin, it must be collected. When touching lava, the game should be lost.

Solving this for the general case is a big task. You can find libraries, usually called *physics engines*, that simulate interaction between physical objects in two or three dimensions. We'll take a more modest approach in this chapter, handling only collisions between rectangular objects and handling them in a rather simplistic way.

Before moving the player or a block of lava, we test whether the motion would take it inside of a wall. If it does, we simply cancel the motion altogether. The response to such a collision depends on the type of actor—the player will stop, whereas a lava block will bounce back.

This approach requires our time steps to be rather small since it will cause motion to stop before the objects actually touch. If the time steps (and thus the motion steps) are too big, the player would end up hovering a noticeable distance above the ground. Another approach, arguably better but more complicated, would be to find the exact collision spot and move there. We will take the simple approach and hide its problems by ensuring the animation proceeds in small steps.

This method tells us whether a rectangle (specified by a position and a size) touches a grid element of the given type.

```
Level.prototype.touches = function(pos, size, type) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
```

```
                    y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

The method computes the set of grid squares that the body overlaps with by using `Math.floor` and `Math.ceil` on its coordinates. Remember that grid squares are 1 by 1 units in size. By rounding the sides of a box up and down, we get the range of background squares that the box touches.

We loop over the block of grid squares found by rounding the coordinates and return `true` when a matching square is found. Squares outside of the level are always treated as `"wall"` to ensure that the player can't leave the world and that we won't accidentally try to read outside of the bounds of our `rows` array.

The state `update` method uses `touches` to figure out whether the player is touching lava.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status != "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
```

```
  return newState;
};
```

The method is passed a time step and a data structure that tells it which keys are being held down. The first thing it does is call the `update` method on all actors, producing an array of updated actors. The actors also get the time step, the keys, and the state, so that they can base their update on those. Only the player will actually read keys, since that's the only actor that's controlled by the keyboard.

If the game is already over, no further processing has to be done (the game can't be won after being lost, or vice versa). Otherwise, the method tests whether the player is touching background lava. If so, the game is lost, and we're done. Finally, if the game really is still going on, it sees whether any other actors overlap the player.

Overlap between actors is detected with the `overlap` function. It takes two actor objects and returns true when they touch—which is the case when they overlap both along the x-axis and along the y-axis.

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

If any actor does overlap, its `collide` method gets a chance to update the state. Touching a lava actor sets the game status to `"lost"`. Coins vanish when you touch them and set the status to `"won"` when they are the last coin of the level.

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

Actor updates

Actor objects' `update` methods take as arguments the time step, the state object, and a `keys` object. The one for the `Lava` actor type ignores the `keys` object.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

This `update` method computes a new position by adding the product of the time step and the current speed to its old position. If no obstacle blocks that new position, it moves there. If there is an obstacle, the behavior depends on the type of the lava block—dripping lava has a `reset` position, to which it jumps back when it hits something. Bouncing lava inverts its speed by multiplying it by -1 so that it starts moving in the opposite direction.

Coins use their `update` method to wobble. They ignore collisions with the grid since they are simply wobbling around inside of their own square.

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vector(0, wobblePos)),
                  this.basePos, wobble);
};
```

The `wobble` property is incremented to track time and then used as an argument to `Math.sin` to find the new position on the wave. The coin's current position is then computed from its base position and an offset based on this wave.

That leaves the player itself. Player motion is handled separately per axis because hitting the floor should not prevent horizontal motion, and hitting a wall should not stop falling or jumping motion.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vector(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
    pos = movedX;
  }

  let ySpeed = this.speed.y + time * gravity;
  let movedY = pos.plus(new Vector(0, ySpeed * time));
  if (!state.level.touches(movedY, this.size, "wall")) {
    pos = movedY;
  } else if (keys.ArrowUp && ySpeed > 0) {
    ySpeed = -jumpSpeed;
  } else {
    ySpeed = 0;
  }
  return new Player(pos, new Vector(xSpeed, ySpeed));
};
```

The horizontal motion is computed based on the state of the left and right arrow keys. When there's no wall blocking the new position created by this motion, it is used. Otherwise, the old position is kept.

Vertical motion works in a similar way but has to simulate jumping and gravity. The player's vertical speed (`ySpeed`) is first accelerated to account for gravity.

We check for walls again. If we don't hit any, the new position is used. If there *is* a wall, there are two possible outcomes. When the up arrow is pressed *and* we are moving down (meaning the thing we hit is below us), the speed is set to a relatively large, negative value. This causes the player to jump. If that is not the case, the player simply bumped into something, and the speed is set to zero.

The gravity strength, jumping speed, and pretty much all other constants in this game have been set by trial and error. I tested values until I found a combination I liked.

Tracking keys

For a game like this, we do not want keys to take effect once per keypress. Rather, we want their effect (moving the player figure) to stay active as long as they are held.

We need to set up a key handler that stores the current state of the left, right, and up arrow keys. We will also want to call `preventDefault` for those keys so that they don't end up scrolling the page.

The following function, when given an array of key names, will return an object that tracks the current position of those keys. It registers event handlers for `"keydown"` and `"keyup"` events and, when the key code in the event is present in the set of codes that it is tracking, updates the object.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

The same handler function is used for both event types. It looks at the event object's `type` property to determine whether the key state should be updated to true (`"keydown"`) or false (`"keyup"`).

Running the game

The `requestAnimationFrame` function, which we saw in Chapter 13, provides a good way to animate a game. But its interface is quite primitive—using it requires us to track the time at which our function was called the last time around and call `requestAnimationFrame` again after every frame.

Let's define a helper function that wraps those boring parts in a convenient interface and allows us to simply call `runAnimation`, giving it a function that

expects a time difference as an argument and draws a single frame. When the frame function returns the value `false`, the animation stops.

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
  }
  requestAnimationFrame(frame);
}
```

I have set a maximum frame step of 100 milliseconds (one-tenth of a second). When the browser tab or window with our page is hidden, `requestAnimationFrame` calls will be suspended until the tab or window is shown again. In this case, the difference between `lastTime` and `time` will be the entire time in which the page was hidden. Advancing the game by that much in a single step would look silly and might cause weird side effects, such as the player falling through the floor.

The function also converts the time steps to seconds, which are an easier quantity to think about than milliseconds.

The `runLevel` function takes a `Level` object and a display constructor and returns a promise. It displays the level (in `document.body`) and lets the user play through it. When the level is finished (lost or won), `runLevel` waits one more second (to let the user see what happens) and then clears the display, stops the animation, and resolves the promise to the game's end status.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
```

```
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

A game is a sequence of levels. Whenever the player dies, the current level is restarted. When a level is completed, we move on to the next level. This can be expressed by the following function, which takes an array of level plans (strings) and a display constructor:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

Because we made `runLevel` return a promise, `runGame` can be written using an `async` function, as shown in Chapter 11. It returns another promise, which resolves when the player finishes the game.

There is a set of level plans available in the `GAME_LEVELS` binding in this chapter's sandbox (*https://eloquentjavascript.net/code#16*). This page feeds them to `runGame`, starting an actual game.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Exercises

Game over

It's traditional for platform games to have the player start with a limited number of *lives* and subtract one life each time they die. When the player is out of lives, the game restarts from the beginning.

Adjust `runGame` to implement lives. Have the player start with three. Output the current number of lives (using `console.log`) every time a level starts.

Pausing the game

Make it possible to pause (suspend) and unpause the game by pressing the Esc key.

This can be done by changing the `runLevel` function to use another keyboard event handler and interrupting or resuming the animation whenever the Esc key is hit.

The `runAnimation` interface may not look like it is suitable for this at first glance, but it is if you rearrange the way `runLevel` calls it.

When you have that working, there is something else you could try. The way we have been registering keyboard event handlers is somewhat problematic. The `arrowKeys` object is currently a global binding, and its event handlers are kept around even when no game is running. You could say they *leak* out of our system. Extend `trackKeys` to provide a way to unregister its handlers and then change `runLevel` to register its handlers when it starts and unregister them again when it is finished.

A monster

It is traditional for platform games to have enemies that you can jump on top of to defeat. This exercise asks you to add such an actor type to the game.

We'll call it a monster. Monsters move only horizontally. You can make them move in the direction of the player, bounce back and forth like horizontal lava, or have any movement pattern you want. The class doesn't have to handle falling, but it should make sure the monster doesn't walk through walls.

When a monster touches the player, the effect depends on whether the player is jumping on top of them or not. You can approximate this by checking whether the player's bottom is near the monster's top. If this is the case, the monster disappears. If not, the game is lost.

"Un estudiante preguntó: 'Los programadores de antaño sólo usaban máquinas simples y ningún lenguaje de programación, pero hicieron programas hermosos. ¿Por qué usamos máquinas y lenguajes de programación complicados?'. Fu-Tzu respondió, Los constructores de antaño sólo usaban palos y arcilla, pero hicieron bellas chozas."'

—Master Yuan-Ma, The Book of Programming

Chapter 16

Node.js

Hasta ahora, hemos usado el lenguaje JavaScript en un solo entorno: el navegador. Este capítulo y el siguiente presentarán brevemente Node.js, un programa que te permite aplicar tus habilidades de JavaScript fuera del navegador. Con él, puedes crear cualquier cosa, desde pequeñas herramientas de línea de comandos hasta servidores HTTP que potencian los sitios web dinámicos.

Estos capítulos tienen como objetivo enseñarte los principales conceptos que Node.js utiliza y darte suficiente información para escribir programas útiles de este. No intentan ser un tratamiento completo, ni siquiera minucioso, de la plataforma.

Si quieres seguir y ejecutar el código de este capítulo, necesitarás instalar Node.js versión 10.1 o superior. Para ello, dirigete a *https://nodejs.org* y sigue las instrucciones de instalación para tu sistema operativo. También podrás encontrar más documentación para Node.js allí.

Antecedentes

Uno de los problemas más difíciles de los sistemas de escritura que se comunican a través de la red es la gestión de entrada y salida, es decir, la lectura y la escritura de datos hacia y desde la red y el disco duro. Mover los datos toma tiempo, y programarlos con habilidad puede hacer una gran diferencia en que tan rápido un sistema responde al usuario o a las peticiones de red.

En tales programas, la programación asíncrona suele ser útil. Permite que el programa envíe y reciba datos desde y hacia múltiples dispositivos al mismo tiempo sin necesidad de una complicada administración y sincronización de hilos.

Node fue concebido inicialmente con el propósito de hacer la programación asíncrona fácil y conveniente. JavaScript se presta bien a un sistema como Node. Es uno de los pocos lenguajes de programación que no tiene una forma incorporada de hacer entrada- y salida. Por lo tanto, JavaScript podría encajar en un enfoque bastante excéntrico de Node para hacer entrada y salida sin

terminar con dos interfaces inconsistentes. En 2009, cuando se diseñó Node, la gente ya estaba haciendo programación basada en llamadas en el navegador, por lo que la comunidad en torno al lenguaje estaba acostumbrada a un estilo de programación asíncrono.

El comando Node

Cuando Node.js se instala en un sistema, proporciona un programa llamado `node`, que se utiliza para ejecutar archivos JavaScript. Digamos que tienes un archivo `hello.js`, que contiene este código:

```
let message = "Hello world";
console.log(message);
```

Entonces podrás lanzar `node` desde la línea de comandos de esta manera para ejecutar el programa:

```
$ node hello.js
Hello world
```

El método `console.log` en Node hace algo similar a lo que realiza en el navegador. Imprime una porción de texto. Pero en Node, el texto irá al flujo de la salida estándar del proceso, en lugar de la consola de JavaScript del navegador. Cuando ejecutas Node desde la línea de comandos, eso significa que verás los valores escritos en tu terminal.

Si ejecutas `node` sin indicar un archivo, te proporciona un prompt en el cual podrás escribir código JavaScript e inmediatamente ver el resultado.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

La referencia `process`, al igual que la referencia `console`, está disponible globalmente en Node. Esta proporciona varias formas de inspeccionar y manipular el programa actual. El método `exit` finaliza el proceso y se le puede pasar un código de estado de salida, esto le dice al programa que inició node (en este

caso, el shell de la línea de comandos) si el programa finalizó con éxito (código cero) o si se encontró con un error (cualquier otro código).

Para obtener los argumentos de la línea de comandos proporcionados a tu script, puede leer `process.argv`, que es un conjunto de cadenas. Ten en cuenta que también incluye el nombre del comando `node` y el nombre de tu script, por lo que los argumentos reales comienzan en el índice 2. Si `showargv.js` contiene la sentencia console.log(process.argv), podrías ejecutarlo así:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

Todas las referencias globales JavaScript estándar, como Array, Math y JSON, también están presentes en el entorno de Node. La funcionalidad relacionada con el navegador, como el documento o el prompt, no lo son.

Módulos

Más allá de las referencias que mencioné, como `console` y `process`, Node pone unas cuantas referencias adicionales en el ámbito global. Si quieres acceder a la funcionalidad integrada, tienes que pedírsela al sistema de módulos.

El sistema de módulos CommonJS, basado en la función `require`, se describió en el Capítulo 10. Este sistema está incorporado en Node y se usa para cargar cualquier cosa desde módulos incorporados hasta paquetes descargados y archivos que forman parte de tu propio programa.

Cuando se llama a `require`, Node tiene que resolver la cadena dada a un archivo real que pueda cargar. Las rutas que empiezan por `/`, `./`, o `../` se resuelven relativamente a la ruta del módulo actual, donde `.` se refiere al directorio actual, `../` a un directorio superior, y `/` a la raíz del sistema de archivos. Así que si pide `"./graph"` del archivo `/tmp/robot/robot.js`, Node intentará cargar el archivo `/tmp/robot/graph.js`.

La extension `.js` puede ser omitida, y Node la añadirá si existe tal archivo. Si la ruta requerida se refiere a un directorio, Node intentará cargar el archivo llamado `index.js` en ese directorio.

Cuando una cadena que no parece una ruta relativa o absoluta es dada a `require`, se asume que se refiere a un módulo incorporado o a un módulo instalado en un directorio `node_modules`. Por ejemplo, `require("fs")` te dará el módulo incorporado del sistema de archivos de Node. Y `require("robot")` podría intentar cargar la biblioteca que se encuentra en `node_modules/robot/`. Una forma común de instalar tales librerías es usando NPM, al cual volveremos

en un momento.

Vamos a crear un pequeño proyecto que consiste en dos archivos. El primero, llamado `main.js`, define un script que puede ser llamado desde la línea de comandos para invertir una cadena.

```
const {reverse} = require("./reverse");

// El índice 2 contiene el primer argumento de la línea de comandos
let argument = process.argv[2];

console.log(reverse(argument));
```

El archivo `reverse.js` define una biblioteca para invertir cadenas, el cual puede ser utilizado tanto por la herramienta de línea de comandos como por otros scripts que necesiten acceso directo a una función de inversión de cadenas.

```
exports.reverse = function(string) {
  return Array.from(string).reverse().join("");
};
```

Recuerde que al añadir propiedades a `exports` estas se agregan a la interfaz del módulo. Dado que Node.js trata a los archivos como módulos CommonJS, `main.js` puede tomar la función exportada `reverse` desde `reverse.js`.

Ahora podemos llamar a nuestra herramienta así:

```
$ node main.js JavaScript
tpircSavaJ
```

Instalación con NPM

NPM, el cual fue introducido en el Capítulo 10, es un repositorio en línea de módulos JavaScript, muchos de los cuales están escritos específicamente para Node. Cuando instalas Node en tu ordenador, también obtienes el comando `npm`, que puedes usar para interactuar con este repositorio.

El uso principal de NPM es descargar paquetes. Vimos el paquete `ini` en el Capítulo 10. nosotros podemos usar NPM para buscar e instalar ese paquete en nuestro ordenador.

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
```

```
         open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

Después de ejecutar `npm install`, NPM habrá creado un directorio llamado `node_modules`. Dentro de ese directorio habrá un directorio `ini` que contiene la biblioteca. Puedes abrirlo y mirar el código. Cuando llamamos a `require ("ini")`, esta librería se carga, y podemos llamar a la propiedad `parse` para analizar un archivo de configuración.

Por defecto, NPM instala los paquetes bajo el directorio actual, en lugar de una ubicación central. Si está acostumbrado a otros gestores de paquetes, esto puede parecer inusual, pero tiene ventajas, pone a cada aplicación en control total de los paquetes que instala y facilita la gestión de versiones y la limpieza al eliminar una aplicación.

Archivos de paquetes

En el ejemplo `npm install`, se podía ver una advertencia sobre el hecho de que el archivo `package.json` no existía. Se recomienda crear dicho archivo para cada proyecto, ya sea manualmente o ejecutando `npm init`. Este contiene alguna información sobre el proyecto, como lo son su nombre y versión, y enumera sus dependencias.

La simulación del robot del Capítulo 7, tal como fue modularizada en el ejercicio del Capítulo 10, podría tener un archivo `package.json` como este:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulation of a package-delivery robot",
  "version": "1.0.0",
  "main": "run.js",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

Cuando ejecutas `npm install` sin nombrar un paquete a instalar, NPM instalará las dependencias listadas en `package.json`. Cuando instale un paquete específico que no esté ya listado como una dependencia, NPM lo añadirá al `package.json`.

Versiones

Un archivo `package.json` lista tanto la versión propia del programa como las versiones de sus dependencias. Las versiones son una forma de lidiar con el hecho de que los paquetes evolucionan por separado, y el código escrito para trabajar con un paquete tal y como existía en un determinado momento pueda no funcionar con una versión posterior y modificada del paquete.

NPM exige que sus paquetes sigan un esquema llamado *versionado semántico*, el cual codifica cierta información sobre qué versiones son *compatibles* (no rompan la vieja interfaz) en el número de versión. Una versión semántica consiste en tres números, separados por puntos, como lo es `2.3.0`. Cada vez que una nueva funcionalidad es agregada, el número intermedio tiene que ser incrementado. Cada vez que se rompe la compatibilidad, de modo que el código existente que utiliza el paquete podría no funcionar con esa nueva versión, el primer número tiene que ser incrementado.

Un carácter cuña (`^`) delante del número de versión para una dependencia en el `package.json` indica que cualquier versión compatible con el número dado puede ser instalada. Así, por ejemplo, `"^2.3.0"` significaría que cualquier versión mayor o igual a 2.3.0 y menor a 3.0.0 está permitida.

El comando `npm` también se usa para publicar nuevos paquetes o nuevas versiones de paquetes. Si ejecutas `npm publish` en un directorio que tiene un archivo `package.json`, esto publicará un paquete con el nombre y la versión que aparece en el archivo JSON. Cualquiera puede publicar paquetes en NPM, aunque sólo bajo un nombre de paquete que no esté en uso todavía, ya que sería algo aterrador si personas al azar pudieran actualizar los paquetes existentes.

Dado que el programa `npm` es una pieza de software que habla con un sistema abierto—El registro de paquetes— no es el único que lo hace. Otro programa, `yarn`, el cual puede ser instalado desde el registro NPM, cumple el mismo papel que `npm` usando una interfaz y una estrategia de instalación algo diferente.

Este libro no profundizará en los detalles del uso del NPM. Consulte *https://npmjs.org* para obtener más documentación y formas para buscar paquetes.

El módulo del sistema de archivos

Uno de los módulos incorporados más utilizados en Node es el módulo `fs`, que significa *sistema de archivos*. Este exporta funciones para trabajar con archivos y directorios.

Por ejemplo, la función llamada `readFile` lee un archivo y luego llama un callback con el contenido del archivo.

```
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
  if (error) throw error;
  console.log("The file contains:", text);
});
```

El segundo argumento para `readFile` indica la *codificación de caracteres* utilizada para decodificar el archivo en una cadena. Hay varias formas de codificar texto en datos binarios, pero la mayoría de los sistemas modernos utilizan UTF-8. Por lo tanto, a menos que tenga razones para creer que se utiliza otra codificación, pase `"utf8"` cuando lea un archivo de texto. Si no pasas una codificación, Node asumirá que estás interesado en los datos binarios y te dará un objeto `Buffer` en lugar de una cadena. Este es un objeto tipo array que contiene números que representan los bytes (trozos de 8 bits de datos) en los archivos.

```
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

Una función similar, `writeFile`, es usada para escribir un archivo en el disco.

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

Aquí no fue necesario especificar la codificación—`writeFile` asumirá que cuando se le da una cadena a escribir, en lugar de un objeto `Buffer`, debe

escribirlo como texto usando su codificación de caracteres por defecto, que es UTF-8.

El módulo `fs` contiene muchas otras funciones útiles: `readdir` devolverá los archivos de un directorio como un array de cadenas, `stat` recuperará información sobre un archivo, `rename` renombrará un archivo, `unlink` eliminará uno, y así sucesivamente. Ve la documentación en *https://nodejs.org* para más detalle.

La mayoría de estos toman una función de devolución como último parámetro, el cual lo llaman ya sea con un error (el primer argumento) o con un resultado exitoso (el segundo). Como vimos en el Capítulo 11, este estilo de programación tiene sus desventajas , la mayor de las cuales es que el manejo de los errores se vuelve enredado y propenso a errores.

Aunque las promesas han sido parte de JavaScript por un tiempo, hasta el momento de escribir este documento su integración en Node.js es aun un trabajo en progreso. Hay un objeto `promises` exportado del paquete `fs` desde la versión 10.1 que contiene la mayoría de las mismas funciones que `fs` pero que utiliza promesas en lugar de funciones de devolución de llamada.

```
const {readFile} = require("fs").promises;
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text));
```

Algunas veces no se necesita asincronicidad, y esto sólo obstaculiza el camino. Muchas de las funciones en `fs` también tienen una variante síncrona, que tiene el mismo nombre con `Sync` añadido al final. Por ejemplo, la versión síncrona de `readFile` es llamada `readFileSync`.

```
const {readFileSync} = require("fs");
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

Ten en cuenta que mientras se realiza esta operación sincrónica, tu programa se detiene por completo. Si debe responder al usuario o a otras máquinas en la red, quedarse atascado en una acción síncrona puede producir retrasos molestos.

The HTTP module

Another central module is called `http`. It provides functionality for running HTTP servers and making HTTP requests.

This is all it takes to start an HTTP server:

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

If you run this script on your own machine, you can point your web browser at *http://localhost:8000/hello* to make a request to your server. It will respond with a small HTML page.

The function passed as argument to `createServer` is called every time a client connects to the server. The `request` and `response` bindings are objects representing the incoming and outgoing data. The first contains information about the request, such as its `url` property, which tells us to what URL the request was made.

So, when you open that page in your browser, it sends a request to your own computer. This causes the server function to run and send back a response, which you can then see in the browser.

To send something back, you call methods on the `response` object. The first, `writeHead`, will write out the response headers (see Chapter ?). You give it the status code (200 for "OK" in this case) and an object that contains header values. The example sets the `Content-Type` header to inform the client that we'll be sending back an HTML document.

Next, the actual response body (the document itself) is sent with `response.write`. You are allowed to call this method multiple times if you want to send the response piece by piece, for example to stream data to the client as it becomes available. Finally, `response.end` signals the end of the response.

The call to `server.listen` causes the server to start waiting for connections on port 8000. This is why you have to connect to *localhost:8000* to speak to this server, rather than just *localhost*, which would use the default port 80.

When you run this script, the process just sits there and waits. When a script is listening for events—in this case, network connections—`node` will not automatically exit when it reaches the end of the script. To close it, press CONTROL-C.

A real web server usually does more than the one in the example—it looks at the request's method (the `method` property) to see what action the client is

trying to perform and looks at the request's URL to find out which resource this action is being performed on. We'll see a more advanced server later in this chapter.

To act as an HTTP *client*, we can use the `request` function in the `http` module.

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code",
              response.statusCode);
});
requestStream.end();
```

The first argument to `request` configures the request, telling Node what server to talk to, what path to request from that server, which method to use, and so on. The second argument is the function that should be called when a response comes in. It is given an object that allows us to inspect the response, for example to find out its status code.

Just like the `response` object we saw in the server, the object returned by `request` allows us to stream data into the request with the `write` method and finish the request with the `end` method. The example does not use `write` because `GET` requests should not contain data in their request body.

There's a similar `request` function in the `https` module that can be used to make requests to `https:` URLs.

Making requests with Node's raw functionality is rather verbose. There are much more convenient wrapper packages available on NPM. For example, `node-fetch` provides the promise-based `fetch` interface that we know from the browser.

Streams

We have seen two instances of writable streams in the HTTP examples—namely, the response object that the server could write to and the request object that was returned from `request`.

Writable streams are a widely used concept in Node. Such objects have a `write` method that can be passed a string or a `Buffer` object to write something

to the stream. Their `end` method closes the stream and optionally takes a value to write to the stream before closing. Both of these methods can also be given a callback as an additional argument, which they will call when the writing or closing has finished.

It is possible to create a writable stream that points at a file with the `createWriteStream` function from the `fs` module. Then you can use the `write` method on the resulting object to write the file one piece at a time, rather than in one shot as with `writeFile`.

Readable streams are a little more involved. Both the `request` binding that was passed to the HTTP server's callback and the `response` binding passed to the HTTP client's callback are readable streams—a server reads requests and then writes responses, whereas a client first writes a request and then reads a response. Reading from a stream is done using event handlers, rather than methods.

Objects that emit events in Node have a method called `on` that is similar to the `addEventListener` method in the browser. You give it an event name and then a function, and it will register that function to be called whenever the given event occurs.

Readable streams have `"data"` and `"end"` events. The first is fired every time data comes in, and the second is called whenever the stream is at its end. This model is most suited for *streaming* data that can be immediately processed, even when the whole document isn't available yet. A file can be read as a readable stream by using the `createReadStream` function from `fs`.

This code creates a server that reads request bodies and streams them back to the client as all-uppercase text:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

The `chunk` value passed to the data handler will be a binary `Buffer`. We can convert this to a string by decoding it as UTF-8 encoded characters with its `toString` method.

The following piece of code, when run with the uppercasing server active, will send a request to that server and write out the response it gets:

```
const {request} = require("http");
```

```
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

The example writes to `process.stdout` (the process's standard output, which is a writable stream) instead of using `console.log`. We can't use `console.log` because it adds an extra newline character after each piece of text that it writes, which isn't appropriate here since the response may come in as multiple chunks.

A FILE SERVER

Let's combine our newfound knowledge about HTTP servers and working with the file system to create a bridge between the two: an HTTP server that allows remote access to a file system. Such a server has all kinds of uses—it allows web applications to store and share data, or it can give a group of people shared access to a bunch of files.

When we treat files as HTTP resources, the HTTP methods `GET`, `PUT`, and `DELETE` can be used to read, write, and delete the files, respectively. We will interpret the path in the request as the path of the file that the request refers to.

We probably don't want to share our whole file system, so we'll interpret these paths as starting in the server's working directory, which is the directory in which it was started. If I ran the server from `/tmp/public/` (or `C:\tmp\public\` on Windows), then a request for `/file.txt` should refer to `/tmp/public/file.txt` (or `C:\tmp\public\file.txt`).

We'll build the program piece by piece, using an object called `methods` to store the functions that handle the various HTTP methods. Method handlers are `async` functions that get the request object as argument and return a promise that resolves to an object that describes the response.

```
const {createServer} = require("http");

const methods = Object.create(null);
```

```
createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
       response.writeHead(status, {"Content-Type": type});
       if (body && body.pipe) body.pipe(response);
       else response.end(body);
    });
}).listen(8000);

async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

This starts a server that just returns 405 error responses, which is the code used to indicate that the server refuses to handle a given method.

When a request handler's promise is rejected, the `catch` call translates the error into a response object, if it isn't one already, so that the server can send back an error response to inform the client that it failed to handle the request.

The `status` field of the response description may be omitted, in which case it defaults to 200 (OK). The content type, in the `type` property, can also be left off, in which case the response is assumed to be plain text.

When the value of `body` is a readable stream, it will have a `pipe` method that is used to forward all content from a readable stream to a writable stream. If not, it is assumed to be either `null` (no body), a string, or a buffer, and it is passed directly to the response's `end` method.

To figure out which file path corresponds to a request URL, the `urlPath` function uses Node's built-in `url` module to parse the URL. It takes its pathname, which will be something like `"/file.txt"`, decodes that to get rid of the `%20`-style escape codes, and resolves it relative to the program's working directory.

```
const {parse} = require("url");
const {resolve, sep} = require("path");
```

```
const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = parse(url);
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

As soon as you set up a program to accept network requests, you have to start worrying about security. In this case, if we aren't careful, it is likely that we'll accidentally expose our whole file system to the network.

File paths are strings in Node. To map such a string to an actual file, there is a nontrivial amount of interpretation going on. Paths may, for example, include `../` to refer to a parent directory. So one obvious source of problems would be requests for paths like `/../secret_file`.

To avoid such problems, `urlPath` uses the `resolve` function from the `path` module, which resolves relative paths. It then verifies that the result is *below* the working directory. The `process.cwd` function (where `cwd` stands for "current working directory") can be used to find this working directory. The `sep` binding from the `path` package is the system's path separator—a backslash on Windows and a forward slash on most other systems. When the path doesn't start with the base directory, the function throws an error response object, using the HTTP status code indicating that access to the resource is forbidden.

We'll set up the `GET` method to return a list of files when reading a directory and to return the file's content when reading a regular file.

One tricky question is what kind of `Content-Type` header we should set when returning a file's content. Since these files could be anything, our server can't simply return the same content type for all of them. NPM can help us again here. The `mime` package (content type indicators like `text/plain` are also called *MIME types*) knows the correct type for a large number of file extensions.

The following `npm` command, in the directory where the server script lives, installs a specific version of `mime`:

```
$ npm install mime@2.2.0
```

When a requested file does not exist, the correct HTTP status code to return

is 404. We'll use the `stat` function, which looks up information about a file, to find out both whether the file exists and whether it is a directory.

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Because it has to touch the disk and thus might take a while, `stat` is asynchronous. Since we're using promises rather than callback style, it has to be imported from `promises` instead of directly from `fs`.

When the file does not exist, `stat` will throw an error object with a `code` property of `"ENOENT"`. These somewhat obscure, Unix-inspired codes are how you recognize error types in Node.

The `stats` object returned by `stat` tells us a number of things about a file, such as its size (`size` property) and its modification date (`mtime` property). Here we are interested in the question of whether it is a directory or a regular file, which the `isDirectory` method tells us.

We use `readdir` to read the array of files in a directory and return it to the client. For normal files, we create a readable stream with `createReadStream` and return that as the body, along with the content type that the `mime` package gives us for the file's name.

The code to handle `DELETE` requests is slightly simpler.

```
const {rmdir, unlink} = require("fs").promises;

methods.DELETE = async function(request) {
```

```
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

When an HTTP response does not contain any data, the status code 204 ("no content") can be used to indicate this. Since the response to deletion doesn't need to transmit any information beyond whether the operation succeeded, that is a sensible thing to return here.

You may be wondering why trying to delete a nonexistent file returns a success status code, rather than an error. When the file that is being deleted is not there, you could say that the request's objective is already fulfilled. The HTTP standard encourages us to make requests *idempotent*, which means that making the same request multiple times produces the same result as making it once. In a way, if you try to delete something that's already gone, the effect you were trying to do has been achieved—the thing is no longer there.

This is the handler for `PUT` requests:

```
const {createWriteStream} = require("fs");

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

We don't need to check whether the file exists this time—if it does, we'll just overwrite it. We again use `pipe` to move data from a readable stream to a writable one, in this case from the request to the file. But since `pipe` isn't written to return a promise, we have to write a wrapper, `pipeStream`, that creates a promise around the outcome of calling `pipe`.

When something goes wrong when opening the file, `createWriteStream` will still return a stream, but that stream will fire an `"error"` event. The output stream to the request may also fail, for example if the network goes down. So we wire up both streams' `"error"` events to reject the promise. When `pipe` is done, it will close the output stream, which causes it to fire a `"finish"` event. That's the point where we can successfully resolve the promise (returning nothing).

The full script for the server is available at *https://eloquentjavascript.net/code/file_server.js*. You can download that and, after installing its dependencies, run it with Node to start your own file server. And, of course, you can modify and extend it to solve this chapter's exercises or to experiment.

The command line tool `curl`, widely available on Unix-like systems (such as macOS and Linux), can be used to make HTTP requests. The following session briefly tests our server. The `-X` option is used to set the request's method, and `-d` is used to include a request body.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

The first request for `file.txt` fails since the file does not exist yet. The `PUT` request creates the file, and behold, the next request successfully retrieves it. After deleting it with a `DELETE` request, the file is again missing.

Summary

Node is a nice, small system that lets us run JavaScript in a nonbrowser context. It was originally designed for network tasks to play the role of a *node* in a network. But it lends itself to all kinds of scripting tasks, and if writing JavaScript is something you enjoy, automating tasks with Node works well.

NPM provides packages for everything you can think of (and quite a few things you'd probably never think of), and it allows you to fetch and install those packages with the `npm` program. Node comes with a number of built-in modules, including the `fs` module for working with the file system and the `http` module for running HTTP servers and making HTTP requests.

All input and output in Node is done asynchronously, unless you explicitly use a synchronous variant of a function, such as `readFileSync`. When calling such asynchronous functions, you provide callback functions, and Node will call them with an error value and (if available) a result when it is ready.

Exercises

Search tool

On Unix systems, there is a command line tool called `grep` that can be used to quickly search files for a regular expression.

Write a Node script that can be run from the command line and acts somewhat like `grep`. It treats its first command line argument as a regular expression and treats any further arguments as files to search. It should output the names of any file whose content matches the regular expression.

When that works, extend it so that when one of the arguments is a directory, it searches through all files in that directory and its subdirectories.

Use asynchronous or synchronous file system functions as you see fit. Setting things up so that multiple asynchronous actions are requested at the same time might speed things up a little, but not a huge amount, since most file systems can read only one thing at a time.

Directory creation

Though the `DELETE` method in our file server is able to delete directories (using `rmdir`), the server currently does not provide any way to *create* a directory.

Add support for the `MKCOL` method ("make collection"), which should create a directory by calling `mkdir` from the `fs` module. `MKCOL` is not a widely used HTTP method, but it does exist for this same purpose in the *WebDAV* standard, which specifies a set of conventions on top of HTTP that make it suitable for creating documents.

A public space on the web

Since the file server serves up any kind of file and even includes the right `Content-Type` header, you can use it to serve a website. Since it allows everybody to delete and replace files, it would be an interesting kind of website: one that can be modified, improved, and vandalized by everybody who takes the time to create the right HTTP request.

Write a basic HTML page that includes a simple JavaScript file. Put the files in a directory served by the file server and open them in your browser.

Next, as an advanced exercise or even a weekend project, combine all the knowledge you gained from this book to build a more user-friendly interface for modifying the website—from *inside* the website.

Use an HTML form to edit the content of the files that make up the website, allowing the user to update them on the server by using HTTP requests, as described in Chapter ?.

Start by making only a single file editable. Then make it so that the user can select which file to edit. Use the fact that our file server returns lists of files when reading a directory.

Don't work directly in the code exposed by the file server since if you make a mistake, you are likely to damage the files there. Instead, keep your work outside of the publicly accessible directory and copy it there when testing.

"Si tienes conocimiento, deja a otros encender sus velas allí."
—Margaret Fuller

Chapter 17

Proyecto: Sitio web para compartir habilidades

Una reunión para *compartir habilidades* es un evento en donde personas con intereses comunes se juntan para dar pequeñas presentaciones informales acerca de cosas que saben. En un reunión de jardinería alguien explicaría como cultivar apio. O en un grupo de programación, podrías presentarte y hablarle a las personas acerca de Node.js.

En estas reuniones, también llamadas *grupos de usuarios* cuando se tratan de computadoras, son una forma genial de ampliar tus horizontes, aprender acerca de nuevos desarrollos, o simplemente conoce gente con intereses similares. Muchas ciudades grandes tienen reuniones sobre JavaScript. Típicamente la entrada es gratis, y las que he visitado han sido amistosas y cálidas.

En este capítulo de proyecto final, nuestra meta es construir un sitio web para administrar las pláticas dadas en una reunión para compartir habilidades. Imagina un pequeño grupo de gente que se reúne regularmente en las oficinas de uno de ellos para hablar de monociclismo. El organizador previo se fue a otra ciudad, y nadie se postuló para tomar esta tarea. Queremos un sistema que deje a los participantes proponer y discutir pláticas entre ellos, sin un organizador central.

El proyecto completo puede ser descargado de *https://eloquentjavascript.net/code/skillsharing.zip* (En inglés).

Diseño

El proyecto tiene un parte de *servidor*, escrita para Node.js, y una parte *cliente*, escrita para el navegador. La parte del servidor guarda la información del sistema y se lo pasa al cliente. Además, sirve los archivos que implementan la parte cliente.

El servidor mantiene la lista de exposiciones propuestas para la próxima charla, y el cliente muestra la lista. Cada charla tiene el nombre del presentados, un título, un resumen, y una lista de comentarios asociados. el cliente permite

proponer nuevas charlas, (agregándolas a la lista), borrar charlas y comentar en las existentes. Cuando un usuario hace un cambio, el cliente hace la petición HTTP para notificar al servidor.

{{index "vista en vivo", "experiencia de usuario", "empujar datos", conexión}}

La aplicación será construida para mostrar una vista *en vivo* de las charlas propuestas y sus comentarios. Cuando alguien en algún lugar manda una nueva charla o agrega un comentario, todas las personas que tienen la página abierta en sus navegadores deberían ver inmediatamente el cambio. Esto nos reta un poco: no hay forma de que el servidor abra una conexión a un cliente, ni una buena forma de saber qué clientes están viendo un sitio web.

Una solución común a este problema es llamada *sondeo largo* (le llamaremos *long polling*), que de casualidad es una de las motivaciones para el diseño de Node.

Long Polling

{{index firewall, notificación, "long polling", red, [navegador, seguridad]}}

Para ser capaces de notificar inmediatamente a un cliente que algo ha cambiado, necesitamos una conexión con ese cliente. Como los navegadores normalmente no aceptan conexiones y los clientes están detrás de routers que blo-

quearían la conexión de todas maneras, hacer que el servidor inicie la conexión no es práctico.

Podemos hacer que el cliente abra la conexión y la mantenga de tal manera que el servidor pueda usarla para mandar información cunado lo necesite.

Pero una petición HTTP permite sólamente un flujo simple de información: el cliente manda una petición, el servidor responde con una sola respuesta, y eso es todo. Existe una tecnología moderna llamada *WebSockets*, soportada por los principales navegadores, que hace posible abrir conexiones para intercambio arbitrario de datos. Pero usarla correctamente es un poco complicado.

En este capítulo usamos una técnica más simple, el *long polling* en donde los clientes continuamente le piden al servidor nueva información usando las peticiones HTTP regulares, y el server detiene su respuesta cuando no hay nada nuevo que reportar.

Mientras el cliente se asegure de tener constantemente abierta una petición de sondeo, recibirá nueva información del servidor muy poco tiempo después de que esté disponible. Por ejemplo, si Fatma tiene nuestra aplicación abierta in su navegador, ese navegador ya habrá hecho una petición de actualizaciones y estará esperando por una respuesta a esa petición. Cuando Iman manda una charla acerca de Bajada Extrema en Monociclo, el servidor se dará cuenta de que Fatma está esperando actualizaciones y mandará una respuesta conteniendo la nueva charla, respondiendo a la petición pendiente. El navegador de Fatma recibirá los datos y actualizará la pantalla.

Para evitar que las conexiones se venzan (que sean abortadas por falta de actividad), las técnicas de *long polling* usualmente ponen el tiempo máximo para cada petición después de lo cuál el servidor responderá de todos modos aunque no tenga nada que reportar, después de lo cuál el cliente iniciará otra petición. Reiniciar periódicamente la petición hace además la técnica más robusta, permitiendo a los clientes recuperarse de fallas temporales en la conexión o problemas del servidor.

Un servidor ocupado que esté usando *long polling* podría tener miles de peticiones esperando, y por lo tanto miles de conexiones TCP abiertas. Node, que hace fácil de manejar muchas conexiones sin crear un hilo de control para cada una, es un buen elemento para nuestro sistema.

HTTP INTERFACE

Before we start designing either the server or the client, let's think about the point where they touch: the HTTP interface over which they communicate.

We will use JSON as the format of our request and response body. Like in

the file server from Chapter 17, we'll try to make good use of HTTP methods and headers. The interface is centered around the `/talks` path. Paths that do not start with `/talks` will be used for serving static files—the HTML and JavaScript code for the client-side system.

A `GET` request to `/talks` returns a JSON document like this:

```
[{"title": "Unituning",
  "presenter": "Jamal",
  "summary": "Modifying your cycle for extra style",
  "comments": []}]
```

Creating a new talk is done by making a `PUT` request to a URL like `/talks/Unituning`, where the part after the second slash is the title of the talk. The `PUT` request's body should contain a JSON object that has `presenter` and `summary` properties.

Since talk titles may contain spaces and other characters that may not appear normally in a URL, title strings must be encoded with the `encodeURIComponent` function when building up such a URL.

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

A request to create a talk about idling might look something like this:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92

{"presenter": "Maureen",
 "summary": "Standing still on a unicycle"}
```

Such URLs also support `GET` requests to retrieve the JSON representation of a talk and `DELETE` requests to delete a talk.

Adding a comment to a talk is done with a `POST` request to a URL like `/talks/Unituning/comments`, with a JSON body that has `author` and `message` properties.

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72

{"author": "Iman",
```

```
"message": "Will you talk about raising a cycle?"}
```

To support long polling, `GET` requests to `/talks` may include extra headers that inform the server to delay the response if no new information is available. We'll use a pair of headers normally intended to manage caching: `ETag` and `If-None-Match`.

Servers may include an `ETag` ("entity tag") header in a response. Its value is a string that identifies the current version of the resource. Clients, when they later request that resource again, may make a *conditional request* by including an `If-None-Match` header whose value holds that same string. If the resource hasn't changed, the server will respond with status code 304, which means "not modified", telling the client that its cached version is still current. When the tag does not match, the server responds as normal.

We need something like this, where the client can tell the server which version of the list of talks it has, and the server responds only when that list has changed. But instead of immediately returning a 304 response, the server should stall the response and return only when something new is available or a given amount of time has elapsed. To distinguish long polling requests from normal conditional requests, we give them another header, `Prefer: wait=90`, which tells the server that the client is willing to wait up to 90 seconds for the response.

The server will keep a version number that it updates every time the talks change and will use that as the `ETag` value. Clients can make requests like this to be notified when the talks change:

```
GET /talks HTTP/1.1
If-None-Match: "4"
Prefer: wait=90

(time passes)

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "5"
Content-Length: 295

[....]
```

The protocol described here does not do any access control. Everybody can comment, modify talks, and even delete them. (Since the Internet is full of

hooligans, putting such a system online without further protection probably wouldn't end well.)

The server

Let's start by building the server-side part of the program. The code in this section runs on Node.js.

Routing

Our server will use `createServer` to start an HTTP server. In the function that handles a new request, we must distinguish between the various kinds of requests (as determined by the method and the path) that we support. This can be done with a long chain of `if` statements, but there is a nicer way.

A *router* is a component that helps dispatch a request to the function that can handle it. You can tell the router, for example, that `PUT` requests with a path that matches the regular expression `/^\/talks\/([^\/]+)$/` (`/talks/` followed by a talk title) can be handled by a given function. In addition, it can help extract the meaningful parts of the path (in this case the talk title), wrapped in parentheses in the regular expression, and pass them to the handler function.

There are a number of good router packages on NPM, but here we'll write one ourselves to illustrate the principle.

This is `router.js`, which we will later `require` from our server module:

```
const {parse} = require("url");

module.exports = class Router {
  constructor() {
    this.routes = [];
  }
  add(method, url, handler) {
    this.routes.push({method, url, handler});
  }
  resolve(context, request) {
    let path = parse(request.url).pathname;

    for (let {method, url, handler} of this.routes) {
      let match = url.exec(path);
      if (!match || request.method != method) continue;
      let urlParts = match.slice(1).map(decodeURIComponent);
      return handler(context, ...urlParts, request);
```

```
    }
    return null;
  }
};
```

The module exports the `Router` class. A router object allows new handlers to be registered with the `add` method and can resolve requests with its `resolve` method.

The latter will return a response when a handler was found, and `null` otherwise. It tries the routes one at a time (in the order in which they were defined) until a matching one is found.

The handler functions are called with the `context` value (which will be the server instance in our case), match strings for any groups they defined in their regular expression, and the request object. The strings have to be URL-decoded since the raw URL may contain `%20`-style codes.

Serving files

When a request matches none of the request types defined in our router, the server must interpret it as a request for a file in the `public` directory. It would be possible to use the file server defined in Chapter 17 to serve such files, but we neither need nor want to support `PUT` and `DELETE` requests on files, and we would like to have advanced features such as support for caching. So let's use a solid, well-tested static file server from NPM instead.

I opted for `ecstatic`. This isn't the only such server on NPM, but it works well and fits our purposes. The `ecstatic` package exports a function that can be called with a configuration object to produce a request handler function. We use the `root` option to tell the server where it should look for files. The handler function accepts `request` and `response` parameters and can be passed directly to `createServer` to create a server that serves *only* files. We want to first check for requests that we should handle specially, though, so we wrap it in another function.

```
const {createServer} = require("http");
const Router = require("./router");
const ecstatic = require("ecstatic");

const router = new Router();
const defaultHeaders = {"Content-Type": "text/plain"};

class SkillShareServer {
```

```
  constructor(talks) {
    this.talks = talks;
    this.version = 0;
    this.waiting = [];

    let fileServer = ecstatic({root: "./public"});
    this.server = createServer((request, response) => {
      let resolved = router.resolve(this, request);
      if (resolved) {
        resolved.catch(error => {
          if (error.status != null) return error;
          return {body: String(error), status: 500};
        }).then(({body,
                  status = 200,
                  headers = defaultHeaders}) => {
          response.writeHead(status, headers);
          response.end(body);
        });
      } else {
        fileServer(request, response);
      }
    });
  }
  start(port) {
    this.server.listen(port);
  }
  stop() {
    this.server.close();
  }
}
```

This uses a similar convention as the file server from the previous chapter for responses—handlers return promises that resolve to objects describing the response. It wraps the server in an object that also holds its state.

Talks as resources

The talks that have been proposed are stored in the `talks` property of the server, an object whose property names are the talk titles. These will be exposed as HTTP resources under `/talks/[title]`, so we need to add handlers to our router that implement the various methods that clients can use to work with them.

The handler for requests that `GET` a single talk must look up the talk and

respond either with the talk's JSON data or with a 404 error response.

```
const talkPath = /^\/talks\/([^\/]+)$/;

router.add("GET", talkPath, async (server, title) => {
  if (title in server.talks) {
    return {body: JSON.stringify(server.talks[title]),
            headers: {"Content-Type": "application/json"}};
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Deleting a talk is done by removing it from the `talks` object.

```
router.add("DELETE", talkPath, async (server, title) => {
  if (title in server.talks) {
    delete server.talks[title];
    server.updated();
  }
  return {status: 204};
});
```

The `updated` method, which we will define later, notifies waiting long polling requests about the change.

To retrieve the content of a request body, we define a function called `readStream` , which reads all content from a readable stream and returns a promise that resolves to a string.

```
function readStream(stream) {
  return new Promise((resolve, reject) => {
    let data = "";
    stream.on("error", reject);
    stream.on("data", chunk => data += chunk.toString());
    stream.on("end", () => resolve(data));
  });
}
```

One handler that needs to read request bodies is the `PUT` handler, which is used to create new talks. It has to check whether the data it was given has `presenter` and `summary` properties, which are strings. Any data coming from outside the system might be nonsense, and we don't want to corrupt our

internal data model or crash when bad requests come in.

If the data looks valid, the handler stores an object that represents the new talk in the `talks` object, possibly overwriting an existing talk with this title, and again calls `updated`.

```
router.add("PUT", talkPath,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let talk;
  try { talk = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
    return {status: 400, body: "Bad talk data"};
  }
  server.talks[title] = {title,
                         presenter: talk.presenter,
                         summary: talk.summary,
                         comments: []};
  server.updated();
  return {status: 204};
});
```

Adding a comment to a talk works similarly. We use `readStream` to get the content of the request, validate the resulting data, and store it as a comment when it looks valid.

```
router.add("POST", /^\/talks\/([^\/]+)\/comments$/,
           async (server, title, request) => {
  let requestBody = await readStream(request);
  let comment;
  try { comment = JSON.parse(requestBody); }
  catch (_) { return {status: 400, body: "Invalid JSON"}; }

  if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
    return {status: 400, body: "Bad comment data"};
  } else if (title in server.talks) {
    server.talks[title].comments.push(comment);
    server.updated();
    return {status: 204};
```

```
  } else {
    return {status: 404, body: `No talk '${title}' found`};
  }
});
```

Trying to add a comment to a nonexistent talk returns a 404 error.

Long polling support

The most interesting aspect of the server is the part that handles long polling. When a `GET` request comes in for `/talks`, it may be either a regular request or a long polling request.

There will be multiple places in which we have to send an array of talks to the client, so we first define a helper method that builds up such an array and includes an `ETag` header in the response.

```
SkillShareServer.prototype.talkResponse = function() {
  let talks = [];
  for (let title of Object.keys(this.talks)) {
    talks.push(this.talks[title]);
  }
  return {
    body: JSON.stringify(talks),
    headers: {"Content-Type": "application/json",
              "ETag": `"${this.version}"`}
  };
};
```

The handler itself needs to look at the request headers to see whether `If-None-Match` and `Prefer` headers are present. Node stores headers, whose names are specified to be case insensitive, under their lowercase names.

```
router.add("GET", /^\/talks$/, async (server, request) => {
  let tag = /"(.*)"/.exec(request.headers["if-none-match"]);
  let wait = /\bwait=(\d+)/.exec(request.headers["prefer"]);
  if (!tag || tag[1] != server.version) {
    return server.talkResponse();
  } else if (!wait) {
    return {status: 304};
  } else {
    return server.waitForChanges(Number(wait[1]));
  }
});
```

If no tag was given or a tag was given that doesn't match the server's current version, the handler responds with the list of talks. If the request is conditional and the talks did not change, we consult the `Prefer` header to see whether we should delay the response or respond right away.

Callback functions for delayed requests are stored in the server's `waiting` array so that they can be notified when something happens. The `waitForChanges` method also immediately sets a timer to respond with a 304 status when the request has waited long enough.

```
SkillShareServer.prototype.waitForChanges = function(time) {
  return new Promise(resolve => {
    this.waiting.push(resolve);
    setTimeout(() => {
      if (!this.waiting.includes(resolve)) return;
      this.waiting = this.waiting.filter(r => r != resolve);
      resolve({status: 304});
    }, time * 1000);
  });
};
```

Registering a change with `updated` increases the `version` property and wakes up all waiting requests.

```
SkillShareServer.prototype.updated = function() {
  this.version++;
  let response = this.talkResponse();
  this.waiting.forEach(resolve => resolve(response));
  this.waiting = [];
};
```

That concludes the server code. If we create an instance of `SkillShareServer` and start it on port 8000, the resulting HTTP server serves files from the `public` subdirectory alongside a talk-managing interface under the `/talks` URL.

```
new SkillShareServer(Object.create(null)).start(8000);
```

The client

The client-side part of the skill-sharing website consists of three files: a tiny HTML page, a style sheet, and a JavaScript file.

HTML

It is a widely used convention for web servers to try to serve a file named `index.html` when a request is made directly to a path that corresponds to a directory. The file server module we use, `ecstatic`, supports this convention. When a request is made to the path `/`, the server looks for the file `./public/index.html` (`./public` being the root we gave it) and returns that file if found.

Thus, if we want a page to show up when a browser is pointed at our server, we should put it in `public/index.html`. This is our index file:

```
<!doctype html>
<meta charset="utf-8">
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">

<h1>Skill Sharing</h1>

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

It defines the document title and includes a style sheet, which defines a few styles to, among other things, make sure there is some space between talks.

At the bottom, it adds a heading at the top of the page and loads the script that contains the client-side application.

Actions

The application state consists of the list of talks and the name of the user, and we'll store it in a `{talks, user}` object. We don't allow the user interface to directly manipulate the state or send off HTTP requests. Rather, it may emit *actions* that describe what the user is trying to do.

The `handleAction` function takes such an action and makes it happen. Because our state updates are so simple, state changes are handled in the same function.

```
function handleAction(state, action) {
  if (action.type == "setUser") {
    localStorage.setItem("userName", action.user);
```

```
    return Object.assign({}, state, {user: action.user});
  } else if (action.type == "setTalks") {
    return Object.assign({}, state, {talks: action.talks});
  } else if (action.type == "newTalk") {
    fetchOK(talkURL(action.title), {
      method: "PUT",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        presenter: state.user,
        summary: action.summary
      })
    }).catch(reportError);
  } else if (action.type == "deleteTalk") {
    fetchOK(talkURL(action.talk), {method: "DELETE"})
      .catch(reportError);
  } else if (action.type == "newComment") {
    fetchOK(talkURL(action.talk) + "/comments", {
      method: "POST",
      headers: {"Content-Type": "application/json"},
      body: JSON.stringify({
        author: state.user,
        message: action.message
      })
    }).catch(reportError);
  }
  return state;
}
```

We'll store the user's name in `localStorage` so that it can be restored when the page is loaded.

The actions that need to involve the server make network requests, using `fetch`, to the HTTP interface described earlier. We use a wrapper function, `fetchOK`, which makes sure the returned promise is rejected when the server returns an error code.

```
function fetchOK(url, options) {
  return fetch(url, options).then(response => {
    if (response.status < 400) return response;
    else throw new Error(response.statusText);
  });
}
```

This helper function is used to build up a URL for a talk with a given title.

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

When the request fails, we don't want to have our page just sit there, doing nothing without explanation. So we define a function called `reportError`, which at least shows the user a dialog that tells them something went wrong.

```
function reportError(error) {
  alert(String(error));
}
```

Rendering components

We'll use an approach similar to the one we saw in Chapter ?, splitting the application into components. But since some of the components either never need to update or are always fully redrawn when updated, we'll define those not as classes but as functions that directly return a DOM node. For example, here is a component that shows the field where the user can enter their name:

```
function renderUserField(name, dispatch) {
  return elt("label", {}, "Your name: ", elt("input", {
    type: "text",
    value: name,
    onchange(event) {
      dispatch({type: "setUser", user: event.target.value});
    }
  }));
}
```

The `elt` function used to construct DOM elements is the one we used in Chapter ?.

A similar function is used to render talks, which include a list of comments and a form for adding a new comment.

```
function renderTalk(talk, dispatch) {
  return elt(
    "section", {className: "talk"},
    elt("h2", null, talk.title, " ", elt("button", {
      type: "button",
      onclick() {
```

```
        dispatch({type: "deleteTalk", talk: talk.title});
      }
    }, "Delete")),
    elt("div", null, "by ",
        elt("strong", null, talk.presenter)),
    elt("p", null, talk.summary),
    ...talk.comments.map(renderComment),
    elt("form", {
      onsubmit(event) {
        event.preventDefault();
        let form = event.target;
        dispatch({type: "newComment",
                  talk: talk.title,
                  message: form.elements.comment.value});
        form.reset();
      }
    }, elt("input", {type: "text", name: "comment"}), " ",
       elt("button", {type: "submit"}, "Add comment")));
}
```

The `"submit"` event handler calls `form.reset` to clear the form's content after creating a `"newComment"` action.

When creating moderately complex pieces of DOM, this style of programming starts to look rather messy. There's a widely used (non-standard) JavaScript extension called *JSX* that lets you write HTML directly in your scripts, which can make such code prettier (depending on what you consider pretty). Before you can actually run such code, you have to run a program on your script to convert the pseudo-HTML into JavaScript function calls much like the ones we use here.

Comments are simpler to render.

```
function renderComment(comment) {
  return elt("p", {className: "comment"},
             elt("strong", null, comment.author),
             ": ", comment.message);
}
```

Finally, the form that the user can use to create a new talk is rendered like this:

```
function renderTalkForm(dispatch) {
  let title = elt("input", {type: "text"});
  let summary = elt("input", {type: "text"});
```

```
  return elt("form", {
    onsubmit(event) {
      event.preventDefault();
      dispatch({type: "newTalk",
                title: title.value,
                summary: summary.value});
      event.target.reset();
    }
  }, elt("h3", null, "Submit a Talk"),
     elt("label", null, "Title: ", title),
     elt("label", null, "Summary: ", summary),
     elt("button", {type: "submit"}, "Submit"));
}
```

Polling

To start the app we need the current list of talks. Since the initial load is closely related to the long polling process—the `ETag` from the load must be used when polling—we'll write a function that keeps polling the server for `/talks` and calls a callback function when a new set of talks is available.

```
async function pollTalks(update) {
  let tag = undefined;
  for (;;) {
    let response;
    try {
      response = await fetchOK("/talks", {
        headers: tag && {"If-None-Match": tag,
                         "Prefer": "wait=90"}
      });
    } catch (e) {
      console.log("Request failed: " + e);
      await new Promise(resolve => setTimeout(resolve, 500));
      continue;
    }
    if (response.status == 304) continue;
    tag = response.headers.get("ETag");
    update(await response.json());
  }
}
```

This is an `async` function so that looping and waiting for the request is easier.

It runs an infinite loop that, on each iteration, retrieves the list of talks—either normally or, if this isn't the first request, with the headers included that make it a long polling request.

When a request fails, the function waits a moment and then tries again. This way, if your network connection goes away for a while and then comes back, the application can recover and continue updating. The promise resolved via `setTimeout` is a way to force the `async` function to wait.

When the server gives back a 304 response, that means a long polling request timed out, so the function should just immediately start the next request. If the response is a normal 200 response, its body is read as JSON and passed to the callback, and its `ETag` header value is stored for the next iteration.

The application

The following component ties the whole user interface together:

```
class SkillShareApp {
  constructor(state, dispatch) {
    this.dispatch = dispatch;
    this.talkDOM = elt("div", {className: "talks"});
    this.dom = elt("div", null,
                   renderUserField(state.user, dispatch),
                   this.talkDOM,
                   renderTalkForm(dispatch));
    this.syncState(state);
  }

  syncState(state) {
    if (state.talks != this.talks) {
      this.talkDOM.textContent = "";
      for (let talk of state.talks) {
        this.talkDOM.appendChild(
          renderTalk(talk, this.dispatch));
      }
      this.talks = state.talks;
    }
  }
}
```

When the talks change, this component redraws all of them. This is simple but also wasteful. We'll get back to that in the exercises.

We can start the application like this:

```
function runApp() {
  let user = localStorage.getItem("userName") || "Anon";
  let state, app;
  function dispatch(action) {
    state = handleAction(state, action);
    app.syncState(state);
  }

  pollTalks(talks => {
    if (!app) {
      state = {user, talks};
      app = new SkillShareApp(state, dispatch);
      document.body.appendChild(app.dom);
    } else {
      dispatch({type: "setTalks", talks});
    }
  }).catch(reportError);
}

runApp();
```

If you run the server and open two browser windows for *http://localhost:8000* next to each other, you can see that the actions you perform in one window are immediately visible in the other.

Exercises

The following exercises will involve modifying the system defined in this chapter. To work on them, make sure you download the code first (*https://eloquentjavascript.net/code/skillsharing.zip*), have Node installed *https://nodejs.org*, and have installed the project's dependency with `npm install`.

Disk persistence

The skill-sharing server keeps its data purely in memory. This means that when it crashes or is restarted for any reason, all talks and comments are lost.

Extend the server so that it stores the talk data to disk and automatically reloads the data when it is restarted. Do not worry about efficiency—do the simplest thing that works.

Comment field resets

The wholesale redrawing of talks works pretty well because you usually can't tell the difference between a DOM node and its identical replacement. But there are exceptions. If you start typing something in the comment field for a talk in one browser window and then, in another, add a comment to that talk, the field in the first window will be redrawn, removing both its content and its focus.

In a heated discussion, where multiple people are adding comments at the same time, this would be annoying. Can you come up with a way to solve it?

Exercise Hints

The hints below might help when you are stuck with one of the exercises in this book. They don't give away the entire solution, but rather try to help you find it yourself.

Estructura de Programa

Ciclo de un triángulo

Puedes comenzar con un programa que imprima los números del 1 al 7, al que puedes derivar haciendo algunas modificaciones al ejemplo de impresión de números pares dado anteriormente en el capítulo, donde se introdujo el ciclo `for`.

Ahora considera la equivalencia entre números y strings de caracteres de numeral. Puedes ir de 1 a 2 agregando 1 (`+= 1`). Puedes ir de `"#"` a `"##"` agregando un caracter (`+= "#"`). Por lo tanto, tu solución puede seguir de cerca el programa de impresión de números.

FizzBuzz

Ir a traves de los números es claramente el trabajo de un ciclo y seleccionar qué imprimir es una cuestión de ejecución condicional. Recuerda el truco de usar el operador restante (`%`) para verificar si un número es divisible por otro número (tiene un residuo de cero).

En la primera versión, hay tres resultados posibles para cada número, por lo que tendrás que crear una cadena `if`/`else if`/`else`.

La segunda versión del programa tiene una solución directa y una inteligente. La manera simple es agregar otra "rama" condicional para probar precisamente la condición dada. Para el método inteligente, crea un string que contenga la palabra o palabras a imprimir e imprimir ya sea esta palabra o el número si no hay una palabra, posiblemente haciendo un buen uso del operador `||`.

Tablero de ajedrez

El string se puede construir comenzando con un string vacío (`""`) y repetidamente agregando caracteres. Un carácter de nueva línea se escribe `"\n"`.

Para trabajar con dos dimensiones, necesitarás un ciclo dentro de un ciclo. Coloca llaves alrededor de los cuerpos de ambos ciclos para hacer fácil de ver dónde comienzan y terminan. Intenta indentar adecuadamente estos cuerpos. El orden de los ciclos debe seguir el orden en el que construimos el string (línea por línea, izquierda a derecha, arriba a abajo). Entonces el ciclo externo maneja las líneas y el ciclo interno maneja los caracteres en una sola linea.

Necesitará dos vinculaciones para seguir tu progreso. Para saber si debes poner un espacio o un signo de numeral en una posición determinada, podrías probar si la suma de los dos contadores es par (`% 2`).

Terminar una línea al agregar un carácter de nueva línea debe suceder después de que la línea ha sido creada, entonces haz esto después del ciclo interno pero dentro del bucle externo.

Funciones

Mínimo

Si tienes problemas para poner llaves y paréntesis en los lugares correctos para obtener una definición válida de función, comienza copiando uno de los ejemplos en este capítulo y modificándolo.

Una función puede contener múltiples declaraciones de `return`.

Recursión

Es probable que tu función se vea algo similar a la función interna `encontrar` en la función recursiva `encontrarSolucion` de ejemplo en este capítulo, con una cadena `if/else if/else` que prueba cuál de los tres casos aplica. El `else` final, correspondiente al tercer caso, hace la llamada recursiva. Cada una de las ramas debe contener una declaración de `return` u organizarse de alguna otra manera para que un valor específico sea retornado.

Cuando se le dé un número negativo, la función volverá a repetirse una y otra vez, pasándose a si misma un número cada vez más negativo, quedando así más y más lejos de devolver un resultado. Eventualmente quedandose sin espacio en la pila y abortando el programa.

Conteo de frijoles

TU función necesitará de un ciclo que examine cada carácter en el string. Puede correr desde un índice de cero a uno por debajo de su longitud (`< string.length`). Si el carácter en la posición actual es el mismo al que se está buscando en la función, agrega 1 a una variable contador. Una vez que el ciclo haya terminado, puedes retornat el contador.

Ten cuidado de hacer que todos las vinculaciones utilizadas en la función sean *locales* a la función usando la palabra clave `let` o `const`.

Estructuras de Datos: Objetos y Arrays

La suma de un rango

Construir un array se realiza más fácilmente al inicializar primero una vinculación a `[]` (un array nuevo y vacío) y llamando repetidamente a su método `push` para agregar un valor. No te olvides de retornar el array al final de la función.

Dado que el límite final es inclusivo, deberias usar el operador `<=` en lugar de `<` para verificar el final de tu ciclo.

El parámetro de paso puede ser un parámetro opcional que por defecto (usando el operador `=`) tenga el valor 1.

Hacer que `rango` entienda valores de paso negativos es probablemente mas facil de realizar al escribir dos ciclos por separado—uno para contar hacia arriba y otro para contar hacia abajo—ya que la comparación que verifica si el ciclo está terminado necesita ser `>=` en lugar de `<=` cuando se cuenta hacia abajo.

También puede que valga la pena utilizar un paso predeterminado diferente, es decir -1, cuando el final del rango sea menor que el inicio. De esa manera, `rango(5, 2)` retornaria algo significativo, en lugar de quedarse atascado en un ciclo infinito. Es posible referirse a parámetros anteriores en el valor predeterminado de un parámetro.

Revirtiendo un array

Hay dos maneras obvias de implementar `revertirArray`. La primera es simplemente pasar a traves del array de entrada de adelante hacia atrás y usar el metodo `unshift` en el nuevo array para insertar cada elemento en su inicio. La segundo es hacer un ciclo sobre el array de entrada de atrás hacia adelante y usar el método `push`. Iterar sobre un array al revés requiere de una especificación (algo incómoda) del ciclo `for`, como `(let i = array.length - 1; i >= 0; i--)`.

Revertir al array en su lugar es más difícil. Tienes que tener cuidado de no sobrescribir elementos que necesitarás luego. Usar `revertirArray` o de lo contrario, copiar toda el array (`array.slice(0)` es una buena forma de copiar un array) funciona pero estás haciendo trampa.

El truco consiste en *intercambiar* el primer y el último elemento, luego el segundo y el penúltimo, y así sucesivamente. Puedes hacer esto haciendo un ciclo basandote en la mitad de la longitud del array (use `Math.floor` para redondear—no necesitas tocar el elemento del medio en un array con un número impar de elementos) e intercambiar el elemento en la posición `i` con el de la posición `array.length - 1 - i`. Puedes usar una vinculación local para aferrarse brevemente a uno de los elementos, sobrescribirlo con su imagen espejo, y luego poner el valor de la vinculación local en el lugar donde solía estar la imagen espejo.

Una lista

Crear una lista es más fácil cuando se hace de atrás hacia adelante. Entonces `arrayAList`a podría iterar sobre el array hacia atrás (ver ejercicio anterior) y, para cada elemento, agregar un objeto a la lista. Puedes usar una vinculación local para mantener la parte de la lista que se construyó hasta el momento y usar una asignación como `lista = {valor: X, resto: lista}` para agregar un elemento.

Para correr a traves de una lista (en `listaAArray` y `posicion`), una especificación del ciclo `for` como esta se puede utilizar:

```
for (let nodo = lista; nodo; nodo = nodo.resto) {}
```

Puedes ver cómo eso funciona? En cada iteración del ciclo, `nodo` apunta a la sublista actual, y el cuerpo puede leer su propiedad `valor` para obtener el elemento actual. Al final de una iteración, `nodo` se mueve a la siguiente sublista. Cuando eso es nulo, hemos llegado al final de la lista y el ciclo termina.

La versión recursiva de `posición`, de manera similar, mirará a una parte más pequeña de la "cola" de la lista y, al mismo tiempo, contara atrás el índice hasta que llegue a cero, en cuyo punto puede retornar la propiedad `valor` del nodo que está mirando. Para obtener el elemento cero de una lista, simplemente toma la propiedad `valor` de su nodo frontal. Para obtener el elemento $N + 1$, toma el elemento N de la lista que este en la propiedad `resto` de esta lista.

Comparación profunda

Tu prueba de si estás tratando con un objeto real se verá algo así como `typeof x == "object" && x != null`. Ten cuidado de comparar propiedades solo cuando *ambos* argumentos sean objetos. En todo los otros casos, puede retornar inmediatamente el resultado de aplicar `===`.

Usa `Object.keys` para revisar las propiedades. Necesitas probar si ambos objetos tienen el mismo conjunto de nombres de propiedad y si esos propiedades tienen valores idénticos. Una forma de hacerlo es garantizar que ambos objetos tengan el mismo número de propiedades (las longitudes de las listas de propiedades son las mismas). Y luego, al hacer un ciclo sobre una de las propiedades del objeto para compararlos, siempre asegúrate primero de que el otro realmente tenga una propiedad con ese mismo nombre. Si tienen el mismo número de propiedades, y todas las propiedades en uno también existen en el otro, tienen el mismo conjunto de nombres de propiedad.

Retornar el valor correcto de la función se realiza mejor al inmediatamente retornar falso cuando se encuentre una discrepancia y retornar verdadero al final de la función.

Funciones de Orden Superior

Cada

Al igual que el operador `&&`, el método `every` puede dejar de evaluar más elementos tan pronto como haya encontrado uno que no coincida. Entonces la versión basada en un ciclo puede saltar fuera del ciclo—con `break` o `return`—tan pronto como se encuentre con un elemento para el cual la función predicado retorne falso. Si el ciclo corre hasta su final sin encontrar tal elemento, sabemos que todos los elementos coinciden y debemos retornar verdadero.

Para construir `cada` usando `some`, podemos aplicar las *leyes De Morgan*, que establecen que `a && b` es igual a `!(!a ||! b)`. Esto puede ser generalizado a arrays, donde todos los elementos del array coinciden si no hay elemento en el array que no coincida.

Dirección de Escritura Dominante

Tu solución puede parecerse mucho a la primera mitad del ejemplo `codigosTexto`. De nuevo debes contar los caracteres por el criterio basado en `codigoCaracter`, y luego filtrar hacia afuera la parte del resultado que se refiere a caracteres sin interés (que no tengan codigos).

Encontrar la dirección con la mayor cantidad de caracteres se puede hacer con `reduce`. Si no está claro cómo, refiérate al ejemplo anterior en el capítulo, donde se usa `reduce` para encontrar el código con la mayoría de los caracteres.

La Vida Secreta de los Objetos

Un tipo vector

Mira de nuevo al ejemplo de la clase `Conejo` si no recuerdas muy bien como se ven las declaraciones de clases.

Agregar una propiedad getter al constructor se puede hacer al poner la palabra `get` antes del nombre del método. Para calcular la distancia desde (0, 0) a (x, y), puedes usar el teorema de Pitágoras, que dice que el cuadrado de la distancia que estamos buscando es igual al cuadrado de la coordenada x más el cuadrado de la coordenada y. Por lo tanto, $\sqrt{x^2 + y^2}$ es el número que quieres, y `Math.sqrt` es la forma en que calculas una raíz cuadrada en JavaScript.

Conjuntos

La forma más fácil de hacer esto es almacenar un array con los miembros del conjunto en una propiedad de instancia. Los métodos `includes` o `indexOf` pueden ser usados para verificar si un valor dado está en el array.

El constructor de clase puede establecer la colección de miembros como un array vacio. Cuando se llama a `añadir`, debes verificar si el valor dado esta en el conjunto y agregarlo, por ejemplo con `push`, de lo contrario.

Eliminar un elemento de un array, en `eliminar`, es menos sencillo, pero puedes usar `filter` para crear un nuevo array sin el valor. No te olvides de sobrescribir la propiedad que sostiene los miembros del conjunto con la versión recién filtrada del array.

El método `desde` puede usar un bucle `for/of` para obtener los valores de el objeto iterable y llamar a `añadir` para ponerlos en un conjunto recien creado.

Conjuntos Iterables

Probablemente valga la pena definir una nueva clase `IteradorConjunto`. Las instancias de Iterador deberian tener una propiedad que rastree la posición actual en el conjunto. Cada vez que se invoque a `next`, este comprueba si está hecho, y si no, se mueve más allá del valor actual y lo retorna.

La clase `Conjunto` recibe un método llamado por `Symbol.iterator` que, cuando se llama, retorna una nueva instancia de la clase de iterador para ese grupo.

Tomando un método prestado

Recuerda que los métodos que existen en objetos simples provienen de `Object.prototype`.

Y que puedes llamar a una función con una vinculación `this` específica al usar su método `call`.

Proyecto: Un Robot

Midiendo un robot

Tendrás que escribir una variante de la función `correrRobot` que, en lugar de registrar los eventos en la consola, retorne el número de pasos que le tomó al robot completar la tarea.

Tu función de medición puede, en un ciclo, generar nuevos estados y contar los pasos que lleva cada uno de los robots. Cuando has generado suficientes mediciones, puedes usar `console.log` para mostrar el promedio de cada robot, que es la cantidad total de pasos tomados dividido por el número de mediciones

Eficiencia del robot

La principal limitación de `robotOrientadoAMetas` es que solo considera un paquete a la vez. A menudo caminará de ida y vuelta por el pueblo porque el paquete que resulta estar mirando sucede que esta en el otro lado del mapa, incluso si hay otros mucho más cerca.

Una posible solución sería calcular rutas para todos los paquetes, y luego tomar la más corta. Se pueden obtener incluso mejores resultados, si hay múltiples rutas más cortas, al ir prefiriendo las que van a recoger un paquete en lugar de entregar un paquete.

Conjunto persistente

La forma más conveniente de representar el conjunto de valores de miembro sigue siendo un array, ya que son fáciles de copiar.

Cuando se agrega un valor al grupo, puedes crear un nuevo grupo con una copia del array original que tiene el valor agregado (por ejemplo, usando `concat`). Cuando se borra un valor, lo filtra afuera del array.

El constructor de la clase puede tomar un array como argumento, y almacenarlo como la (única) propiedad de la instancia. Este array nunca es actualizado.

Para agregar una propiedad (`vacio`) a un constructor que no sea un método, tienes que agregarlo al constructor después de la definición de la clase, como una propiedad regular.

Solo necesita una instancia `vacio` porque todos los conjuntos vacíos son iguales y las instancias de la clase no cambian. Puedes crear muchos conjuntos diferentes de ese único conjunto vacío sin afectarlo.

Bugs y Errores

Reintentar

La llamada a `multiplicacionPrimitiva` definitivamente debería suceder en un bloquear `try`. El bloque `catch` correspondiente debe volver a lanzar la excepción cuando no esta no sea una instancia de `FalloUnidadMultiplicadora` y asegurar que la llamada sea reintentada cuando lo es.

Para reintentar, puedes usar un ciclo que solo se rompa cuando la llamada tenga éxito, como en el ejemplo de `mirar` anteriormente en este capítulo—o usar recursión y esperar que no obtengas una cadena de fallas tan largas que desborde la pila (lo cual es una apuesta bastante segura).

La caja bloqueada

Este ejercicio requiere de un bloque `finally`. Tu función deberia primero desbloquear la caja y luego llamar a la función argumento desde dentro de cuerpo `try`. El bloque `finally` después de el debería bloquear la caja nuevamente.

Para asegurarte de que no bloqueemos la caja cuando no estaba ya bloqueada, comprueba su bloqueo al comienzo de la función y desbloquea y bloquea solo cuando la caja comenzó bloqueada.

Expresiones Regulares

Estilo entre comillas

La solución más obvia es solo reemplazar las citas con una palabra no personaje en al menos un lado. Algo como `/\W'|'\W/`. Pero también debes tener en cuenta el inicio y el final de la línea.

Además, debes asegurarte de que el reemplazo también incluya los caracteres que coincidieron con el patrón `\W` para que estos no sean dejados. Esto se puede hacer envolviéndolos en paréntesis e incluyendo sus grupos en la cadena de

reemplazo (`$1`,`$2`). Los grupos que no están emparejados serán reemplazados por nada.

Números otra vez

Primero, no te olvides de la barra invertida delante del punto.

Coincidir el signo opcional delante de el número, así como delante del exponente, se puede hacer con `[+\-]?` o `(\+|-|)` (más, menos o nada).

La parte más complicada del ejercicio es el problema hacer coincidir ambos `"5."` y `".5"` sin tambien coincidir coincidir con `"."`. Para esto, una buena solución es usar el operador `|` para separar los dos casos—ya sea uno o más dígitos seguidos opcionalmente por un punto y cero o más dígitos *o* un punto seguido de uno o más dígitos.

Finalmente, para hacer que la *e* pueda ser mayuscula o minuscula, agrega una opción `i` a la expresión regular o usa `[eE]`.

Módulos

Un robot modular

Aqui esta lo que habría hecho (pero, una vez más, no hay una sola forma *correcta* de diseñar un módulo dado):

El código usado para construir el camino de grafo vive en el módulo `grafo`. Ya que prefiero usar `dijkstrajs` de NPM en lugar de nuestro propio código de busqueda de rutas, haremos que este construya el tipo de datos de grafos que `dijkstajs` espera. Este módulo exporta una sola función, `construirGrafo` . Haria que `construirGrafo` acepte un array de arrays de dos elementos, en lugar de strings que contengan guiones, para hacer que el módulo sea menos dependiente del formato de entrada.

El módulo `caminos` contiene los datos en bruto del camino (el array `caminos`) y la vinculación `grafoCamino`. Este módulo depende de `./grafo` y exporta el grafo del camino.

La clase `EstadoPueblo` vive en el módulo `estado`. Depende del módulo `./caminos`, porque necesita poder verificar que un camino dado existe. También necesita `eleccionAleatoria`. Dado que eso es una función de tres líneas, podríamos simplemente ponerla en el módulo `estado` como una función auxiliar interna. Pero `robotAleatorio` también la necesita. Entonces tendriamos que duplicarla o ponerla en su propio módulo. Dado que esta función existe en NPM en el paquete `random-item`, una buena solución es hacer que ambos módulos

dependan de el. Podemos agregar la función `correrRobot` a este módulo también, ya que es pequeña y estrechamente relacionada con la gestión de estado. El módulo exporta tanto la clase `EstadoPueblo` como la función `correrRobot`.

Finalmente, los robots, junto con los valores de los que dependen, como `mailRoute`, podrían ir en un módulo `robots-ejemplo`, que depende de `./caminos` y exporta las funciones de robot. Para que sea posible que el `robotOrientadoAMetas` haga busqueda de rutas, este módulo también depende de `dijkstrajs`.

Al descargar algo de trabajo a los módulos de NPM, el código se volvió un poco mas pequeño. Cada módulo individual hace algo bastante simple, y puede ser leído por sí mismo. La división del código en módulos también sugiere a menudo otras mejoras para el diseño del programa. En este caso, parece un poco extraño que `EstadoPueblo` y los robots dependan de un grafo de caminos. Podría ser una mejor idea hacer del grafo un argumento para el constructor del estado y hacer que los robots lo lean del objeto estado—esto reduce las dependencias (lo que siempre es bueno) y hace posible ejecutar simulaciones en diferentes mapas (lo cual es aún mejor).

Es una buena idea usar módulos de NPM para cosas que podríamos haber escrito nosotros mismos? En principio, sí—para cosas no triviales como la función de busqueda de rutas es probable que cometas errores y pierdas el tiempo escribiendola tú mismo. Para pequeñas funciones como `eleccionAleatoria`, escribirla por ti mismo es lo suficiente fácil. Pero agregarlas donde las necesites tiende a desordenar tus módulos.

Sin embargo, tampoco debes subestimar el trabajo involucrado en *encontrar* un paquete apropiado de NPM. E incluso si encuentras uno, este podría no funcionar bien o faltarle alguna característica que necesitas. Ademas de eso, depender de los paquetes de NPM, significa que debes asegurarte de que están instalados, tienes que distribuirlos con tu programa, y podrías tener que actualizarlos periódicamente.

Entonces, de nuevo, esta es una solución con compromisos, y tu puedes decidir de una u otra manera dependiendo sobre cuánto te ayuden los paquetes.

Módulo de Caminos

Como este es un módulo CommonJS, debes usar `require` para importar el módulo grafo. Eso fue descrito como exportar una función `construirGrafo`, que puedes sacar de su objeto de interfaz con una declaración `const` de desestructuración.

Para exportar `grafoCamino`, agrega una propiedad al objeto `exports`. Ya que `construirGrafo` toma una estructura de datos que no empareja precisamente `caminos`, la división de los strings de los caminis debe ocurrir en tu módulo.

Dependencias circulares

El truco es que `require` agrega módulos a su caché *antes* de comenzar a cargar el módulo. De esa forma, si se realiza una llamada `require` mientras está ejecutando el intento de cargarlo, ya es conocido y la interfaz actual sera retornada, en lugar de comenzar a cargar el módulo una vez más (lo que eventualmente desbordaría la pila).

Si un módulo sobrescribe su valor `module.exports`, cualquier otro módulo que haya recibido su valor de interfaz antes de que termine de cargarse ha conseguido el objeto de interfaz predeterminado (que es probable que este vacío), en lugar del valor de interfaz previsto.

Programación Asincrónica

Siguiendo el bisturí

Esto se puede realizar con un solo ciclo que busca a través de los nidos, avanzando hacia el siguiente cuando encuentre un valor que no coincida con el nombre del nido actual, y retornando el nombre cuando esta encuentra un valor que coincida. En la función `async`, un ciclo regular `for` o `while` puede ser utilizado.

Para hacer lo mismo con una función simple, tendrás que construir tu ciclo usando una función recursiva. La manera más fácil de hacer esto es hacer que esa función retorne una promesa al llamar a `then` en la promesa que recupera el valor de almacenamiento. Dependiendo de si ese valor coincide con el nombre del nido actual, el controlador devuelve ese valor o una promesa adicional creada llamando a la función de ciclo nuevamente.

No olvides iniciar el ciclo llamando a la función recursiva una vez desde la función principal.

En la función `async`, las promesas rechazadas se convierten en excepciones por `await` Cuando una función `async` arroja una excepción, su promesa es rechazada. Entonces eso funciona.

Si implementaste la función no-`async` como se describe anteriormente, la forma en que `then` funciona también provoca automáticamente que una falla termine en la promesa devuelta. Si una solicitud falla, el manejador pasado a `then` no se llama, y la promesa que devuelve se rechaza con la misma razón.

Construyendo Promise.all

La función pasada al constructor `Promise` tendrá que llamar `then` en cada una de las promesas del array dado. Cuando una de ellas tenga éxito, dos cosas

deben suceder. El valor resultante debe ser almacenado en la posición correcta de un array de resultados, y debemos verificar si esta fue la última promesa pendiente y terminar nuestra promesa si asi fue.

Esto último se puede hacer con un contador que se inicializa con la longitud del array de entrada y del que restamos 1 cada vez que una promesa tenga éxito. Cuando llega a 0, hemos terminado. Asegúrate de tener en cuenta la situación en la que el array de entrada este vacío (y por lo tanto ninguna promesa nunca se resolverá).

El manejo de la falla requiere pensar un poco, pero resulta ser extremadamente sencillo. Solo pasa la función `reject` de la promesa de envoltura a cada una de las promesas en el array como manejador `catch` o como segundo argumento a `then` para que una falla en una de ellos desencadene el rechazo de la promesa de envoltura completa.

Proyecto: Un Lenguaje de Programación

Arrays

The easiest way to do this is to represent Egg arrays with JavaScript arrays.

The values added to the top scope must be functions. By using a rest argument (with triple-dot notation), the definition of `array` can be *very* simple.

Closure

Again, we are riding along on a JavaScript mechanism to get the equivalent feature in Egg. Special forms are passed the local scope in which they are evaluated so that they can evaluate their subforms in that scope. The function returned by `fun` has access to the `scope` argument given to its enclosing function and uses that to create the function's local scope when it is called.

This means that the prototype of the local scope will be the scope in which the function was created, which makes it possible to access bindings in that scope from the function. This is all there is to implementing closure (though to compile it in a way that is actually efficient, you'd need to do some more work).

Comments

Make sure your solution handles multiple comments in a row, with potentially whitespace between or after them.

A regular expression is probably the easiest way to solve this. Write something that matches "whitespace or a comment, zero or more times". Use the `exec` or `match` method and look at the length of the first element in the returned array (the whole match) to find out how many characters to slice off.

Fixing scope

You will have to loop through one scope at a time, using `Object.getPrototypeOf` to go the next outer scope. For each scope, use `hasOwnProperty` to find out whether the binding, indicated by the `name` property of the first argument to `set`, exists in that scope. If it does, set it to the result of evaluating the second argument to `set` and then return that value.

If the outermost scope is reached (`Object.getPrototypeOf` returns null) and we haven't found the binding yet, it doesn't exist, and an error should be thrown.

El Modelo de Objeto del Documento

Construye una tabla

Puedes utilizar `document.createElement` para crear nuevos nodos elemento, `document.createTextNode` para crear nuevos nodos de texto, y el método `appendChild` para poner nodos dentro de otros nodos.

Querrás recorrer los nombres de las llaves una vez para llenar la fila superior y luego nuevamente para cada objeto en el arreglo para construir las filas de datos. Para obtener un arreglo con los nombres de las llaves proveniente del primer objeto, la función `Object.keys` será de utilidad.

Para agregar la tabla al nodo padre correcto, puedes utilizar `document.getElementById` o `document.querySelector` para encontrar el nodo con el atributo `id` adecuado.

Elementos por nombre de tag

La solución es expresada de manera más sencilla con una función recursiva, similar a la función `hablaSobre` definida anteriormente en este capítulo.

Puedes llamar a `byTagname` recursivamente, concatenando los arreglos resultantes para producir la salida. O puedes crear una función interna que se llama a sí misma recursivamente y que tiene acceso a un arreglo definido en la función exterior, al cual se puede agregar los elementos coincidentes que encuentre. No

olvides llamar a la función interior una vez desde la función exterior para iniciar el proceso.

La función recursiva debe revisar el tipo de nodo. En este caso solamente estamos interesados por los nodos de tipo 1 (`Node.ELEMENT_NODE`). Para tales nodos, debemos de iterar sobre sus hijos y, para cada hijo, observar si los hijos coinciden con la consulta mientras también se realiza una llamada recursiva en él para inspeccionar a sus propios hijos.

El sombrero del gato

Las funciones `Math.cos` y `Math.sin` miden los ángulos en radianes, donde un círculo completo es 2π. Para un ángulo dado, puedes obtener el ángulo inverso agregando la mitad de esto, que es `Math.PI`. Esto puede ser útil para colocar el sombrero en el lado opuesto de la órbita.

Manejo de Eventos

Globo

Querrás registrar un manejador para el evento `"keydown"` y mirar `event.key` para averiguar si se presionó la teclas de flecha hacia arra o hacia abajo.

El tamaño actual se puede mantener en una vinculación para que puedas basar el nuevo tamaño en él. Será útil definir una función que actualice el tamaño, tanto el enlace como el estilo del globo en el DOM, para que pueda llamar desde su manejador de eventos, y posiblemente también una vez al comenzar, para establecer el tamaño inicial.

Puedes cambiar el globo a una explosión reemplazando el texto del nodo con otro (usando `replaceChild`) o estableciendo la propiedad `textContent` de su no padre a una nueva cadena.

Mouse trail

La creación de los elementos se realiza de mjor manera con un ciclo. Añadelos al documento para que aparezcan. Para poder acceder a ellos más tarde para cambiar su posición, querrás alamacenar los elementos en una matriz.

Se puede hacer un ciclo a través de ellos manteniendo una variable contador y agregando 1 cada vez que se activa el evento `"mousemove"`. El operador restante (`% elements.length`) pueden ser usado para obtener un índice de matriz válido para elegir el elemento que se desaea colocar durante un evento determinado.

Otro efecto interesante se puede lograr modelando un sistema simple físico. Utiliza el evento `"mousemove"` para actualizar un par de enlaces que rastrean la posición del mouse. Luego usa `requestAnimationFrame` para simular que los elementos finales son atraidos a la posición a la posición del puntero del mouse. En cada paso de la animación, actualiza la posición en función de su posición relativa al puntero (y, opcionalmente, una velocidad que se alamacena para cada elemento). Averiguar una buena forma de hacerlo depende de ti.

Pestañas

Un error con el que puedes encontrarte es que no puede usar directamente la propiedad `childNodes` del nodo como una colección de nodos de pestañas. Por un lado, cuando agregas los botones, también se convertirán en nodos secundarios y terminarán en este objeto porque es una estructura de datos en vivo. Por otro lado, los nodos de texto creados para el espacio en blanco entre los nodos también están en `childNodes` pero no deberían tener sus propias pestañas. Puedes usar `children` en lugar de`childNodes` para ignorar los nodos con texto.

Puedes comenzar creando una colección de pestañas para que tengas fácil acceso a ellas. Para implementar el estilo de los botones, puedes almacenar objetos que contengan tanto el panel de pestañas como su botón.

Yo recomiendo escribir una función aparte para cambiar las pestañas. Puedes almacenar la pestaña seleccionada anteriormente y cambiar solo los estilos necesarios para ocultarla y mostrar la nueva, o simplemente puedes actualizar el estilo de todas las pestañas cada vez que se selecciona una nueva pestaña.

Es posible que desees llamar a esta función inmediatamente para que la interfaz comience con la primera pestaña visible.

Proyecto: Un Juego de Plataforma

Pausing the game

An animation can be interrupted by returning `false` from the function given to `runAnimation`. It can be continued by calling `runAnimation` again.

So we need to communicate the fact that we are pausing the game to the function given to `runAnimation`. For that, you can use a binding that both the event handler and that function have access to.

When finding a way to unregister the handlers registered by `trackKeys`, remember that the *exact* same function value that was passed to `addEventListener` must be passed to `removeEventListener` to successfully remove a handler.

Thus, the `handler` function value created in `trackKeys` must be available to the code that unregisters the handlers.

You can add a property to the object returned by `trackKeys`, containing either that function value or a method that handles the unregistering directly.

A monster

If you want to implement a type of motion that is stateful, such as bouncing, make sure you store the necessary state in the actor object—include it as constructor argument and add it as a property.

Remember that `update` returns a *new* object, rather than changing the old one.

When handling collision, find the player in `state.actors` and compare its position to the monster's position. To get the *bottom* of the player, you have to add its vertical size to its vertical position. The creation of an updated state will resemble either `Coin`'s `collide` method (removing the actor) or `Lava` 's (changing the status to `"lost"`), depending on the player position.

Node.js

Search tool

Your first command line argument, the regular expression, can be found in `process.argv[2]`. The input files come after that. You can use the `RegExp` constructor to go from a string to a regular expression object.

Doing this synchronously, with `readFileSync`, is more straightforward, but if you use `fs.promises` again to get promise-returning functions and write an `async` function, the code looks similar.

To figure out whether something is a directory, you can again use `stat` (or `statSync`) and the stats object's `isDirectory` method.

Exploring a directory is a branching process. You can do it either by using a recursive function or by keeping an array of work (files that still need to be explored). To find the files in a directory, you can call `readdir` or `readdirSync`. The strange capitalization—Node's file system function naming is loosely based on standard Unix functions, such as `readdir`, that are all lowercase, but then it adds `Sync` with a capital letter.

To go from a filename read with `readdir` to a full path name, you have to combine it with the name of the directory, putting a slash character (`/`) between them.

Directory creation

You can use the function that implements the `DELETE` method as a blueprint for the `MKCOL` method. When no file is found, try to create a directory with `mkdir`. When a directory exists at that path, you can return a 204 response so that directory creation requests are idempotent. If a nondirectory file exists here, return an error code. Code 400 ("bad request") would be appropriate.

A public space on the web

You can create a `<textarea>` element to hold the content of the file that is being edited. A `GET` request, using `fetch`, can retrieve the current content of the file. You can use relative URLs like *index.html*, instead of *http://localhost:8000/index.html*, to refer to files on the same server as the running script.

Then, when the user clicks a button (you can use a `<form>` element and `"submit"` event), make a `PUT` request to the same URL, with the content of the `<textarea>` as request body, to save the file.

You can then add a `<select>` element that contains all the files in the server's top directory by adding `<option>` elements containing the lines returned by a `GET` request to the URL `/`. When the user selects another file (a `"change"` event on the field), the script must fetch and display that file. When saving a file, use the currently selected filename.

Proyecto: Sitio web para compartir habilidades

Disk persistence

The simplest solution I can come up with is to encode the whole `talks` object as JSON and dump it to a file with `writeFile`. There is already a method (`updated`) that is called every time the server's data changes. It can be extended to write the new data to disk.

Pick a filename, for example `./talks.json`. When the server starts, it can try to read that file with `readFile`, and if that succeeds, the server can use the file's contents as its starting data.

Beware, though. The `talks` object started as a prototype-less object so that the `in` operator could reliably be used. `JSON.parse` will return regular objects with `Object.prototype` as their prototype. If you use JSON as your file format, you'll have to copy the properties of the object returned by `JSON.parse` into a new, prototype-less object.

Comment field resets

The best way to do this is probably to make talks component objects, with a `syncState` method, so that they can be updated to show a modified version of the talk. During normal operation, the only way a talk can be changed is by adding more comments, so the `syncState` method can be relatively simple.

The difficult part is that, when a changed list of talks comes in, we have to reconcile the existing list of DOM components with the talks on the new list—deleting components whose talk was deleted and updating components whose talk changed.

To do this, it might be helpful to keep a data structure that stores the talk components under the talk titles so that you can easily figure out whether a component exists for a given talk. You can then loop over the new array of talks, and for each of them, either synchronize an existing component or create a new one. To delete components for deleted talks, you'll have to also loop over the components and check whether the corresponding talks still exist.

Index

! operator, 19, 33
!= operator, 18
!== operator, 21
* operator, 13, 20, 153
= operator, 35
+ operator, 13, 16, 20, 153
++ operator, 36
+= operator, 35, 210
− operator, 14, 17, 20
−− operator, 36
−= operator, 35
/ operator, 14
/= operator, 35
< operator, 17
<= operator, 18
= operator, 25, 49, 65, 167, 170, 220
== operator, 18, 21, 68, 85, 203
=== operator, 21, 85, 120, 333
> operator, 17
>= operator, 18
?: operator, 19, 22, 219
[] (array), 61
[] (subscript), 61, 62
[network, stack], 204
% operator, 14, 35, 329, 330, 342
&& operator, 18, 22, 100
| | operator, 19, 21, 53, 100, 329
{} (block), 30
{} (object), 64, 68
200 (HTTP status code), 298, 302
204 (HTTP status code), 305, 306
304 (HTTP status code), 313, 320, 326
400 (HTTP status code), 345
403 (HTTP status code), 303
404 (HTTP status code), 303, 316, 319
405 (HTTP status code), 302
500 (HTTP status code), 302

a (HTML tag), 236, 238
Abelson, Hal, 213
absolute positioning, 242, 246, 255, 259, 266
absolute value, 80
abstraccion, 230
abstraccione, 87
abstracción, 213
abstract data type, 101
abstract syntax tree, *see* syntax tree
abstraction, 41, 86, 87, 89
acceleration, 285
access control, 101, 149, 313
actor, 271, 277, 284
add method, 120
addEntry function, 68
addEventListener method, 249, 286, 300
addition, 13, 120

adoption, 150
advertencia, 294
ages example, 109
aislamiento, 178
alcace, 42
alcance, 43, 178, 179, 181, 182
alcance global, 178, 262
alcance léxico, 44
alphanumeric character, 152
alt attribute, 233
alt key, 254
altKey property, 254
ambiguity, 226
American English, 153
analysis, 134, 139
ancestor element, 278
Android, 254
animaciones, 268
animación, 259
animation, 243, 266, 273
  platform game, 281, 285–287, 343
  spinning cat, 242, 246
antirrebote, 263
anyStorage function, 210, 212
apio, 309
aplicación, 1, 310
appendChild method, 233, 341
application (of functions), *see* function application
aprender, 7
aprendizaje, 8, 309
archivo, 176, 183, 292
archivos, 181
argument, 48, 162
argumento, 27, 53, 77, 213
arguments object, 331
argv property, 292
arithmetic, 13, 20, 221
arrastrar, 256
array, 62–64, 66, 78, 81, 84, 88, 98, 100, 113, 128, 147, 154, 185, 225, 334, 335
  as matrix, 270
  as table, 70
  creation, 61, 331
  filtering, 91
  indexing, 61, 71, 75, 331, 342
  iteration, 71, 90
  length of, 63
  methods, 74, 83, 90–93, 96, 99, 100
  representation, 81
  searching, 71, 75
  traversal, 88
Array prototype, 104, 108
array-like object, 230, 231
Array.from function, 205, 231, 293
arrays, 96
arrays in egg (exercise), 225, 340
arroba, 269
arrow function, 46, 103
arrow key, 265
artificial intelligence, 122, 224
artificial life, 268
asignación, 167
assert function, 147
assertion, 147
assignment, 25, 35, 170, 226, 341
assumption, 147
asterisk, 13, 153
async function, 206, 207, 210, 212, 325
asynchronous programming, 189, 190, 194, 195, 206, 208, 209, 288
  in Node.js, 290, 297, 299, 304, 307
atributo, 235
attribute, 235
attributo, 230
automatic semicolon insertion, 24

automation, 132, 137
autómata, 122
avatar, 268
average function, 95
await keyword, 206, 208, 210
axis, 284

Babbage, Charles, 60
background, 276, 282
background (CSS), 266, 268, 277
backslash character, 15, 150, 152, 165, 337
    as path separator, 303
backtick, 15, 16
backtracking, 159, 160, 163
balloon (exercise), 265, 342
banking example, 143
Banks, Ian, 267
barra de progreso, 259
bean counting (exercise), 59, 331
beforeunload event, 261
behavior, 224
benchmark, 237
best practices, 3
biblioteca, 294
big ball of mud, 175
binary data, 3, 11
binary number, 11, 12, 138, 159
binary operator, 13, 17
binding, 220, 221, 224, 226
    assignment, 25, 45
    definition, 24, 226, 341
    from parameter, 42, 51
    global, 42, 135, 289, 291, 292
    local, 43
    model of, 25, 67
    naming, 26, 37, 54, 79, 136
    scope of, 42
    visibility, 43
bit, 4, 11, 12, 17
bitfield, 257
bits, 70
block, 41, 142, 144, 214
block comment, 38, 163
block element, 236, 238
blocking, 190, 243, 263, 297
bloque, 30, 43, 46, 65, 239
bloques, 33
blur event, 260
body (HTML tag), 228
body (HTTP), 298, 305, 306, 317
body property, 228, 229, 232
Book of Programming, 11, 175, 290
Boolean, 17, 29, 32, 66, 219, 221
    conversion to, 21, 29, 33
Boolean function, 29
Booleano, 151
border (CSS), 236, 239
border-radius (CSS), 255
botón del mouse, 250, 251, 255
bouncing, 269, 273, 281, 284
boundary, 157, 169, 173, 336
box shadow (CSS), 278
braces, *see* curly braces
branching, 158, 160
branching recursion, 52
break keyword, 35, 37
British English, 153
broadcastConnections function, 202
browser, 6, 249
    window, 249
bubbling, *see* event propagation
Buffer class, 296, 299, 300
bug, 134, 166, 173, 176
building Promise.all (exercise), 212, 339
button, 248
button (HTML tag), 249, 254, 266
button property, 250, 257
buttons property, 257

cache, 181
caché, 193
caja, 149, 227, 268
caja de arena, 60, 227
call method, 103, 108
call stack, 47, 50, 53, 141, 142, 145
callback function, 190, 196, 198, 200, 248, 286, 287, 296, 297, 299, 320, 325
calling (of functions), *see* function application
camel case, 37, 240
campo, 254
campo de texto, 260
cancelAnimationFrame function, 263
canvas, 268
capas, 204
capitalization, 37, 106, 154, 240, 246, 300
capture group, 155, 157, 162, 315
capítulo de proyecto, 309
caracteres chinos, 97
caracteres de nueva línea, 270
caret character, 152, 157, 169, 295
carrera, 267
carriage return, 168
carácter, 96
carácter de tubería, 158
carácteres de punto, 28
carácteres de tabulación, 33
Cascading Style Sheets, *see* CSS
case conversion, 63
case keyword, 37
case sensitivity, 154, 337
casual computing, 1
cat's hat (exercise), 246
catch keyword, 141, 142, 145, 146, 148, 208, 336
catch method, 197
CD, 11
Celsius, 116
center, 280
centering, 243
change event, 345
character, 15, 16, 97
character category, 171
character encoding, 296
characterCount function, 94
characterScript function, 99, 100, 333
charCodeAt method, 97
chess board (exercise), 40, 330
chicks function, 210
child node, 231, 232
childNodes property, 231, 234, 343
children property, 231
Chinese characters, 98
ciclo, 31, 39, 88, 95, 199, 228, 330, 331
ciclo infinito, 35, 331
cierre, 50
circo, 74
circular dependency, 187, 339
clase, 105, 106, 120, 124, 269
class attribute, 232, 236, 240, 275, 277, 278
class declaration
  properties, 107
class hierarchy, 118
className property, 236
cleaning up, 143
clearInterval function, 263
clearTimeout function, 262, 263
click event, 249, 251, 255, 257
client, 299, 321
cliente, 309
clientHeight property, 236
clientWidth property, 236
clientX property, 255, 258
clientY property, 255, 258
closure, 225, 340, 341, 343

closure in egg (exercise), 225, 340
code
    structure of, 23
code golf, 173
code structure, 33, 41, 175, 184
codePointAt method, 97
codiciosos, 164
codificación de caracteres, 296
coercion de tipo, 20
coin, 268, 284
Coin class, 273, 284
colección, 6, 61
collection, 63, 66, 84
collision detection, 281, 284, 285, 344
colon character, 19, 36, 64, 239
color (CSS), 239
comentario, 38, 163, 229, 309
comillas, 174
comma character, 213
command key, 254
command line, 177, 290, 292, 307
comment, 81, 168, 226, 312, 318, 323, 340
comment field reset (exercise), 328, 346
COMMENT_NODE code, 229
comments in egg (exercise), 226, 340
CommonJS, 292, 293
CommonJS modules, 179, 180
compareRobots function, 132
comparison, 17, 21, 32, 37, 85, 221, 331
    of NaN, 18
    of numbers, 17, 28
    of objects, 68
    of strings, 18
    of undefined values, 21
compartir habilidades, 309
compatibilidad, 7
compatibility, 295
compilation, 183, 223, 224, 340
complejidad, 3, 86
complexity, 4, 118, 160, 241, 275
comportamiento, 173
composability, 6, 95, 184
computadora, 1, 3
computed property, 62
concat method, 75, 100, 335, 341
concatenation, 16, 75, 341
conditional execution, 19, 36, 40, 219
conditional operator, 19, 22, 219
conditional request, 313
conexiones, 311
conexión, 310, 311
configuración, 168
conjunto, 151, 229
conjunto de datos, 71, 90, 91
connected graph, 131
connections binding, 202
consistencia, 37
consistency, 230
consola de JavaScript, 28, 291
console.log, 6, 10, 17, 28, 47, 49, 57, 139, 291, 301
const keyword, 26, 43, 67, 79, 81
constant, 285
constante, 26
constantes, 79
constructor, 37, 106, 118, 133, 142, 156, 164, 334, 335
constructora, 105, 135
constructoras, 106
Content-Type header, 298, 302, 303, 308
continuación, 192
continue keyword, 35
control flow, 29, 31, 33, 34, 47, 141, 190, 206
control key, 254
convención, 37

coordenadas, 255
coordinates, 120, 244, 276, 280, 282
copy-paste programming, 55, 176
copyright, 177
corchete, 112
corchetes, 61, 81, 152
corredores de pruebas, 138
correlaciones, 73
correlación, 73
correlaciónes, 71
correlation, 69, 70
cosine, 79, 244
countBy function, 98, 100
counter variable, 32, 34, 244, 330, 331, 340
CPU, 190
crash, 145, 147, 318, 327
createElement method, 234, 341
createReadStream function, 300, 304
createServer function, 297, 298, 300, 314, 315
createTextNode method, 233
createWriteStream function, 300, 306
crow-tech module, 193
crying, 154
CSS, 239–241, 275, 277–280, 321
ctrlKey property, 254
cuadrado, 29
cuadro de diálogo, 27
cuadrícula, 268
cuervo, 191, 193, 205
curl program, 306
curly braces, 5, 41, 46, 64, 68, 81, 89, 153
cutting point, 275
cwd function, 303
código, 8, 163, 267
córvidos, 191

Dark Blue (game), 267
dash character, 14, 152
data, 2, 11
data attribute, 235, 266
data event, 300
data format, 81
data loss, 327
data structure, 60, 61, 84, 109, 126
  tree, 228
Date class, 156, 178, 179
date-names package, 179
Date.now function, 156
datos, 60
datos binarios, 296
dblclick event, 255
debugger statement, 139
debugging, 7, 134, 136, 138, 139, 142, 146, 147, 173
decimal number, 11, 138, 159
declaración, 24, 29, 32, 42, 65
declaración de clase, 107
declaraciónes, 34
decodeURIComponent function, 302, 315
deep comparison, 68, 85
deep comparison (exercise), 85, 333
default behavior, 238, 252
default export, 182
default keyword, 37
default value, 22, 49
defineProperty function, 334
definirTipoSolicitud function, 194, 199
DELETE method, 301, 304, 317
delete method, 120
delete operator, 65
dependencia, 69
dependency, 175, 176, 178, 182, 187, 294, 295
depuración, 135
desbordar, 12
descarga, 309

descargar, 293
desenrollando la pila, 141
deserialization, 82
design, 176
destructuring, 157
destructuring binding, 80, 180, 338
developer tools, 8, 28, 145
diagrama de flujo, 158
dialecto, 183
diario, 68
Dijkstra's algorithm, 185
Dijkstra, Edsger, 122, 185
dijkstrajs package, 185, 337
dimensiones, 330
dimensions, 120, 236, 267, 268, 281
dinosaur, 224
dirección, 81
direct child node, 241
direction (writing), 100
directorio, 292, 295, 297
directory, 292, 296, 301, 303, 304, 307, 345
directory creation (exercise), 307, 345
disco duro, 189
discretization, 268, 281, 287
dispatch, 314
dispatching, 36
display, 275, 287, 288
display (CSS), 239, 266
division, 14
division by zero, 14
do loop, 33, 128
Document Object Model, *see* DOM
documentElement property, 228
documento, 227, 261
dollar sign, 26, 157, 162, 169
DOM, 228, 235
  attributes, 235
  construction, 230, 233, 234
  events, 249, 254
  graphics, 268, 275, 277, 278
  interface, 229
  modification, 233
  querying, 232, 241
  tree, 228
domain-specific language, 86, 138, 150, 225, 241
DOMDisplay class, 275, 276
dominant direction (exercise), 100, 333
dot character, *see* period character
double click, 255
double-quote character, 15, 174, 213
download, 8, 177, 306, 327
draggable bar example, 256
drawing, 227, 236, 237, 243, 275
drawing program example, 255
duplication, 176
dígito, 138, 151–153, 155
dígitos, 154

ECMAScript, 7, 182
ECMAScript 6, 7
ecstatic package, 315
editores, 34
efecto secundario, 24, 28, 42, 233
efficiency, 52, 83, 95, 202, 223, 237, 278
eficiencia, 268
Egg language, 213, 217–219, 221, 222, 224–226, 229
ejecución condicional, 29
ejemplo de la granja, 57
ejercicios, 2, 8, 138
elección, 158
electronic life, 268
elegance, 215
elegancia, 52
element, 229, 231
ELEMENT_NODE code, 229, 342

elemento, 229, 234
elipse, 243
ellipse, 244
else keyword, 30
elt function, 234, 323
emoji, 16, 97, 171, 265
empaquetadores, 184
empty set, 163
encapsulación, 102, 118
encapsulation, 101, 110, 249, 275
encodeURIComponent function, 312, 322
end event, 300
end method, 298, 299, 302
enemies example, 168
enlace, 230, 232, 252, 254
ENOENT (status code), 304
entorno, 27
entrada, 268
enum (reserved word), 26
environment, 219
equality, 18
error, 97, 134, 135, 138, 140, 141, 145, 146, 196, 198, 204
error de sintaxis, 26
error event, 306
error handling, 134, 141, 145, 297, 302, 304, 323, 326
error message, 218
error recovery, 140
error response, 302, 305
Error type, 142, 145, 146, 304
errores, 164
ES modules, 182
escape key, 289
escaping
    in regexps, 150, 152, 165
    in strings, 15, 213
    in URLs, 302, 312, 315
escribiendo, 263
espacio en blanco, 213
espacios en blanco, 216
especificidad, 241
espías, 259
estado, 24, 124, 202, 209
estructura, 227
estructura de datos, 64, 186, 214, 227
estándar, 7, 27, 37, 92, 170, 292
ETag header, 313, 319, 326
etiqueta, 227, 240
eval, 178
evaluación de corto circuito, 22
evaluate function, 218, 219, 221
evaluation, 178, 218, 224
even number, 58
event handling, 248–253, 259–261, 268, 286, 288, 289, 300, 343
event loop, 208
event object, 255
event propagation, 251, 260, 261
event type, 250
every method, 100
everything (exercise), 100, 333
everywhere function, 201
evolución convergente, 192
evolution, 150, 295
exception handling, 142, 143, 145–149, 196, 197, 206, 208, 212, 339
exception safety, 144
exec method, 154, 155, 166, 167
execution order, 29, 45, 47
exercises, 39
exit method, 291
expectation, 252
experiment, 3
experimentar, 8, 173
exponent, 13, 174, 337
exponente, 337

exponentiation, 32, 34
export keyword, 182
exports object, 179, 181, 182, 293, 338, 339
exposiciones, 309
expresion, 45
expresiones regular, 216
expresiones regulare, 150
expresiones regulares, 164, 168
expresión, 23, 24, 28, 32, 34, 213, 214
expresión regular, 151, 173
expression, 218
expressivity, 225
extension, 292
extiende, 78
extraction, 155

factorial function, 9
Fahrenheit, 116
fallthrough, 37
false, 17
farm example, 55, 158
fecha, 152, 154–156
fetch function, 299, 322, 325, 345
field, 328
fila, 246
file, 304, 345
  access, 296
  resource, 301, 303
  stream, 300
file extension, 303
file format, 168
file server, 321
file server example, 301, 303–308, 345
file size, 184
file system, 296, 301, 303, 345
filter method, 91, 92, 95, 99, 125, 201, 333–335
finally keyword, 144, 149, 336
findIndex method, 98
findInStorage function, 205, 206
findRoute function, 130, 203
finish event, 306
firstChild property, 231
fixed positioning, 259
fixing scope (exercise), 226, 341
FizzBuzz (exercise), 40, 329
flattening (exercise), 100
flexibility, 7
floating-point number, 13
flooding, 202
flow diagram, 159
flujo de control, 90, 143
foco, 254, 260
focus, 328
focus event, 260
fold, *see* reduce method
fondo, 268
font-family (CSS), 240
font-size (CSS), 265
font-weight (CSS), 240
for loop, 34, 35, 71, 88, 100, 146, 331, 332
for/of loop, 72, 97, 111, 113, 115, 334
forEach method, 90
form, 308
form (HTML tag), 324, 345
formatDate module, 179, 182
formato de datos, 230
fractional number, 13, 174, 268
framework, 57
fs package, 296, 297
funcion, 213
funciones de flecha, 210
función, 27, 214
función de devolución de llamada, 192, 194
función de predicado, 99

función interior, 342
function, 27, 41, 46, 135, 222
  application, 27, 28, 42, 47, 48, 51, 52, 92, 145, 213, 219
  as property, 63
  as value, 41, 45, 50, 88, 89, 92, 250, 286, 343
  body, 41, 46
  callback, *see* callback function
  declaration, 45
  definition, 41, 45, 54
  higher-order, 45, 88, 89, 91–93, 95, 162, 286
  model of, 51
  naming, 54, 56
  purity, 57
  scope, 44, 178, 225
function application, 78
Function constructor, 179, 181, 221, 224
function keyword, 41, 45
Function prototype, 104, 108
futuras, 26
future, 7, 45
físico, 343

game, 286, 288, 289
  screenshot, 280
GAME_LEVELS data set, 288
garbage collection, 12
garble example, 293
generador, 207
GET method, 299, 301, 303, 312, 316
get method, 110
getAttribute method, 235, 236
getBoundingClientRect method, 237
getDate function, 157
getDate method, 156
getElementById method, 232, 341
getElementsByClassName method, 232
getElementsByTagName method, 232, 234, 246, 341
getFullYear method, 156
getHours method, 156
getMinutes method, 156
getMonth method, 156
getPrototypeOf function, 104, 106, 226, 341
getSeconds method, 156
getter, 115, 120, 273
getTime method, 156
getYear method, 156
global object, 135
global scope, 42, 221, 291, 292, 341
globo, 265
goalOrientedRobot function, 131
gossip property, 201
grafo, 123, 130, 185, 203
grammar, 23, 168
gramática, 134
gran bola de barro, 175
graphics, 268, 275, 278
grave accent, *see* backtick
gravity, 285
greater than, 17
greed, 163
grep, 307
grid, 276, 282
Group class, 120, 132, 207, 334
groupBy function, 100
grouping, 14, 30, 154, 155, 162, 336
groups (exercise), 120, 334
grupo de usuarios, 309
gráficos, 268

h1 (HTML tag), 236
handleAction function, 321
hard disk, 184, 192
hard drive, 11, 327

hard-coding, 232
has method, 110, 120
hash character, 226
hasOwnProperty method, 110, 226, 341
head (HTML tag), 228
head property, 228
header, 298, 312
help text example, 260
herencia, 117, 118
herramienta, 150
herramientas, 150, 172
herramientas de desarrollador, 140
hexadecimal number, 159
hidden element, 239, 266
higher-order function, *see* function, higher-order
hilo, 190, 262
historia, 6
Hières-sur-Amby, 191
hooligan, 314
hora, 152, 154, 156
href attribute, 232, 235
HTML, 227, 308
  structure, 227, 229
html (HTML tag), 228
HTTP, 298, 305, 307, 311
  client, 299, 306
  cliente, 309
  server, 297, 301, 320
http package, 297, 299
HTTPS, 299
https package, 299
human language, 23
hyphen character, 240

id attribute, 232, 240
idempotence, 199, 305
idempotency, 345
identifier, 214
identity, 67
if keyword, 29, 170
  chaining, 30, 36, 329, 330
If-None-Match header, 313, 320, 325
imagen, 233
imagenes, 261
imaginación, 267
IME, 254
img (HTML tag), 233, 238, 261
immutable, 273
implements (reserved word), 26
import keyword, 182
in operator, 65, 110
includes method, 71, 72, 334
indefinido, 134
indentación, 33
index property, 155
index.html, 321
index.js, 292
indexOf method, 75, 76, 98, 120, 151, 165, 334
infinite loop, 48, 146
infinity, 14
infraestructura, 177
inheritance, 104, 117, 119, 146, 304
INI file, 168
ini package, 177, 181, 184, 293
inicialización, 261
inline element, 236, 238
inmutables, 66, 126
inner function, 44
inner loop, 161
innerHeight property, 259
innerWidth property, 259
input, 140, 248, 290, 317
input (HTML tag), 260
insertBefore method, 233
installation, 177
instanceof operator, 118, 146
instancia, 105

integer, 13
integration, 150, 230
interface, 110, 115, 120, 184, 198
    design, 56, 150, 156, 162, 166, 229, 230, 275
    HTTP, 311
    module, 293
    object, 271
interface (reserved word), 26
interfaces, 101, 178
interfaz, 111, 150, 175, 178, 179, 181, 184
internationalization, 170
Internet, 168
interpolation, 16
interpretation, 8, 178, 218, 219, 223
inversion, 152
invoking (of functions), *see* function application
isDirectory method, 304, 344
isEven (exercise), 58, 330
isolation, 101, 175, 178
iterable interface, 113, 334
iterador, 207
iterator interface, 111, 113, 120

Jacques, 60
jardinería, 309
Java, 6
JavaScript, 6
    availability of, 1
    flexibility of, 7
    history of, 6
    puntos débiles, 7
    syntax, 23
    uses of, 8
    versions of, 7
JavaScript console, 8, 17, 28, 139, 145
JavaScript Object Notation, *see* JSON
join method, 99, 108, 293
journal, 61, 64, 66, 72
JOURNAL data set, 71
journalEvents function, 72
JSON, 81, 184, 192, 203, 311, 312, 326, 345
JSON.parse function, 82, 345
JSON.stringify function, 82
JSX, 324
juego, 267–269
juego de plataforma, 267
jugador, 267–269
jump, 5
jump-and-run game, 267
jumping, 268, 285

Kernighan, Brian, 134
key code, 286
key property, 253, 342
keyboard, 253, 268, 285, 286, 289
keydown event, 253, 263, 286, 342
keyup event, 253, 286
kill process, 298
Knuth, Donald, 41

la salida estándar, 291
landscape example, 44
Laozi, 189
lastChild property, 231
lastIndex property, 166, 167
lastIndex property*, 166
lastIndexOf method, 75
latency, 183
lava, 268, 269, 278, 281, 284
Lava class, 273, 284
layout, 236, 239
laziness, 237
Le Guin, Ursula K., 2
leaf node, 229
leak, 289

learning, 2
leerAlmacenamiento function, 193
left (CSS), 242–244, 246
legibilidad, 4
LEGOS, 175
length property
    for array, 63
    for string, 55, 59, 62, 77, 331
lenguaje de programación, 213
lenguaje Egg, 214
lenguajes de programación, 1
less than, 17
let keyword, 24, 25, 43, 67, 79, 81, 135
level, 269, 275, 276, 278, 288
Level class, 269
lexical scoping, 44
leyes De Morgan, 333
library, 230, 293
licencia, 177
limite, 91
line break, 15, 168
line comment, 38, 163
lines of code, 222
link (HTML tag), 280
linked list, 84, 332
linter, 182
Liskov, Barbara, 101
list (exercise), 84, 332
lista de trabajo, 131
listen method, 297, 298
listening (TCP), 297
literal expression, 23, 150, 216, 218
live data structure, 227, 234, 242, 343
live view, 326, 346
lives (exercise), 289
llamada de pila, 208
llaves, 30, 65, 154, 330
load event, 261
local binding, 50, 226, 331
local scope, 43, 223
localhost, 298
localStorage object, 322
locked box (exercise), 149, 336
logging, 139
logical and, 18
logical operators, 18
logical or, 19
long polling, 310, 311, 313, 317, 319, 320, 325
loop, 5, 34, 40, 51, 71, 88, 94, 167
    termination of, 35
loop body, 33, 89
lycanthropy, 60, 68
límite, 159, 165
línea, 24, 169
línea de comandos, 291, 293
líneas, 33

machine code, 3, 224
magia, 213
magic, 103
mailRoute array, 129
maintenance, 177
manejadores, 248
manejo de excepciones, 141
map, 274
map (data structure), 109
Map class, 110, 115, 205
map method, 92, 95, 99, 103, 109, 125, 201, 270
Marcus Aurelius, 248
match method, 155, 167
matching, 151, 157, 158, 166, 173
    algorithm, 158–160
matemáticas, 89
Math object, 58, 62, 78
Math.abs function, 80
Math.acos function, 79

Math.asin function, 79
Math.atan function, 79
Math.ceil function, 80, 282
Math.cos function, 79, 244
Math.floor function, 80, 128, 282
Math.max function, 28, 62, 77, 78
Math.min function, 28, 58, 78
Math.PI constant, 79
Math.random function, 79, 128, 274
Math.round function, 80
Math.sin function, 79, 244, 274, 284
Math.sqrt function, 70, 78, 334
Math.tan function, 79
mathematics, 51
Matrix class, 113
matrix example, 113, 117
MatrixIterator class, 114
max-height (CSS), 278
max-width (CSS), 278
maximum, 28, 78, 94
measuring a robot (exercise), 132, 335
media type, 303
meetup, 309
memoria, 61, 81, 189
memory, 11, 24, 47, 67, 84, 224
  persistence, 327
menú contextual, 252
message event, 262
meta key, 254
metaKey property, 254
method, 63, 74, 102, 298
  HTTP, 298, 306, 311, 314
method call, 102
methods object, 301
mime package, 303
MIME type, 303
minificadores, 184
minimalism, 267
minimum, 28, 58, 78
minimum (exercise), 58, 330
minus, 14, 174
MKCOL method, 307, 345
mkdir function, 307, 345
modification date, 304
modifier key, 254
modular robot (exercise), 187, 337
modularity, 101
module, 187, 275, 314
  design, 184
module loader, 292
module object, 181
modulo operator, 14
modulos CommonJS, 187
monedas, 269
monociclismo, 309
monster (exercise), 289, 344
mouse, 27
mouse cursor, 255
mouse trail (exercise), 266, 342
mousedown event, 251, 255, 257
mousemove event, 256, 257, 263, 264, 266, 342
mouseup event, 255, 257
movimiento, 268
multiplication, 13, 272, 284
multiplier function, 50
mundo, 267
mundo virtual, 122
musical, 267
mutability, 65–67, 126
método, 101, 105, 135
módulo, 175, 292, 293
módulo CommonJS, 187, 338
módulos, 178
módulos CommonJS, 182
módulos ES, 182

namespace, 78
namespace pollution, 78

naming, 26
NaN, 14, 18, 20, 134
navegador, 2, 27, 268, 309
navegadore, 28, 190
navegadores, 9, 183
negation, 17, 19
negritas, 238
neighbors property, 200
nerd, 165
nesting
    in regexps, 161
    of arrays, 70
    of expressions, 23, 215
    of functions, 44
    of loops, 40, 330
    of objects, 228, 231
    of scope, 44
Netscape, 6
network, 197
    speed, 290
network function, 205
new operator, 105
newline character, 15, 40, 152, 163, 168
next method, 113, 207, 334
nextSibling property, 231
nivel, 268, 269
node program, 291
node-fetch package, 299
Node.js, 8, 9, 28, 179, 190, 290–293, 296–299, 301, 303–306, 309–311, 314, 327
node_modules directory, 292, 294
NodeList type, 230, 241
nodeName property, 246
nodeType property, 229, 342, 343
nodeValue property, 231
nodo, 228, 229
nodos de texto, 231
nodos hijos, 229
nodos hoja, 229
nombrado, 4
nombre, 6
not a number, 14
notación, 182
NPM, 177, 179, 181, 183, 185, 187, 292–295, 303, 314, 315, 327, 338
npm program, 293–295, 303
nueces, 73, 74
null, 20, 21, 53, 62, 81, 85, 140
number, 12, 66
    conversion to, 20, 29
    notation, 12, 13
    precision of, 13
    representation, 12
    special values, 14
Number function, 29, 37
number puzzle example, 52
Number.isNaN function, 30
número, 151, 174, 337
número binario, 70
números pare, 31

object, 28, 60, 65, 66, 78, 81, 84, 103, 118, 333
    as map, 274
    creation, 105
    identity, 67
    property, 62
    representation, 81
Object prototype, 104
object-oriented programming, 105, 111, 117, 185
Object.create function, 104, 109, 222
Object.keys function, 65, 85, 205, 333, 341
Object.prototype, 109
objeto, 64, 101, 178
objeto de evento, 250

objeto tipo array, 296
objeto tipo matriz, 258
objetos de evento, 258
obstacle, 281
offsetHeight property, 236, 237
offsetWidth property, 236
on method, 300
onclick attribute, 249
opcional, 153
operador, 214
operador binario, 23
operador unario, 23
operator, 13, 17, 21, 221
    application, 13
optimización, 268
optimization, 52, 57, 237, 263, 278, 297
option (HTML tag), 345
optional argument, 49, 83
ordinal package, 179, 181
organic growth, 175
organization, 175
output, 17, 27, 28, 139, 140, 221
overflow (CSS), 278
overlap, 282
overlay, 240
overriding, 107, 111, 117, 339
overwriting, 306, 308, 318

p (HTML tag), 236
package (reserved word), 26
package manager, 177
package.json, 294, 295
padding (CSS), 277
page reload, 261
pageX property, 255, 258
pageXOffset property, 237
pageY property, 255, 258
pageYOffset property, 237, 259
palabra caracter, 170
palabra clave, 24, 236
palabras clave, 26
Palef, Thomas, 267
paquete, 176, 179, 292, 295
parallelism, 190
parameter, 27, 42, 49, 80, 136
parametros, 181
parent node, 251
parentheses, 14, 158
parentNode property, 231
parse function, 217
parseApply function, 216
parseExpression function, 215
parseINI function, 169, 176
parsing, 82, 134, 169, 213–215, 217, 219, 222, 227, 302, 319
parámetro, 41, 43, 46, 48, 77, 103
parámetro restante, 78
paréntesis, 23, 27, 30, 32, 34, 46, 89, 154, 157, 170, 213, 336
path
    file system, 292, 301
    URL, 301, 302, 311, 314
path package, 303
pathfinding, 129, 185, 203
patrones, 150, 151
patrón, 150, 152, 164
pattern, 151
pausing (exercise), 289, 343
pea soup, 87
percentage, 259
performance, 160, 183, 223, 237, 268, 297
period character, 62, *see* max example, 152, 163, 174
persistence, 327, 345
persistencia, 309
persistent data structure, 124, 126, 132, 138, 344
persistent group (exercise), 132

persistent map (exercise), 335
petición, 310
PGroup class, 132, 335
phase, 273, 274, 284
phi coefficient, 69, 70
phi function, 70, 80
physics, 281, 285
physics engine, 281
pi, 13, 79, 244, 274
PI constant, 79, 244
pila, 64
pila de llamadas, 48
ping request, 200
pipe character, 337
pipe method, 302, 306
pipeline, 184
pixel, 236, 245, 255, 268, 276
pizza, 69, 70
plantilla, 179
plantillas literales, 16
platform game, 289
Plauger, P.J., 134
player, 278, 281, 284, 288
Player class, 272, 284
plus character, 13, 153, 174
Poignant Guide, 23
pointer, 230
pointer event, 251
polling, 248
pollTalks function, 325
polymorphism, 111
pop method, 63, 74
Popper, Karl, 234
porcentaje, 99
port, 298
position, 237
position (CSS), 242, 246, 259, 268, 277, 278
POST method, 312
postMessage method, 262
power example, 42, 49, 51
precedence, 14, 19
precedencia, 14, 240
predicate function, 91, 96, 100
Prefer header, 313, 320, 325
pregunta de entrevista, 40
premature optimization, 52
preventDefault method, 252, 258, 260, 261, 286
previousSibling property, 231
primitiveMultiply (exercise), 148, 336
private (reserved word), 26
private properties, 101
private property, 149
problema de búsqueda, 130
procesador, 189
process object, 291, 303
profiling, 52
program, 23, 29
  nature of, 2
program size, 86, 173, 275
programación, 1
programación asincrónica, 192
programación orientada a objetos, 101, 124
programming
  difficulty of, 2
  history of, 4
  joy of, 1, 3
programming language, 3, 224, 229, 290
  power of, 6
programming style, 3, 24, 33, 37, 275
project chapter, 122, 213, 267
promesa, 212, 340
Promise class, 195, 197–199, 205, 208, 209, 212, 297, 299, 301, 326, 339
Promise.all function, 200, 210, 212, 339

Promise.reject function, 197
Promise.resolve function, 195, 200
promises package, 297
promptDirection function, 145, 146
promptInteger function, 140
propagation, *see* event propagation
property, 62, 64, 102, 103, 111, 115
    assignment, 65
    deletion, 65
    model of, 65
    testing for, 65
propiedad, 28, 62, 68, 105, 107, 112, 134
propiedades, 110
protected (reserved word), 26
prototipo, 104, 106–108
prototipos, 105
prototype, 109, 117, 222, 226, 340, 345
    diagram, 108
prototype property, 106
proyecto para compartir habilidades, 309
pseudorandom number, 79
public (reserved word), 26
public properties, 101
public space (exercise), 308, 345
publishing, 295
punch card, 4
punto de interrupción, 139
punto y coma, 34, 239
pure function, 56, 57, 83, 92, 185
push method, 63, 72, 74, 334
PUT method, 301, 305, 312, 317, 345
Pythagoras, 334
página web, 183
páginas de inicio llamativas, 266

query string, 313, 319
querySelector method, 242, 341
querySelectorAll method, 241
question mark, 19, 153
queue, 208
quotation mark, 15
quoting
    in JSON, 81
    of object properties, 64
quoting style (exercise), 174, 336

rabbit example, 102, 104, 106, 107
radian, 244
radix, 11
raising (exception), 141
random number, 79, 274
random-item package, 337
randomPick function, 127
randomRobot function, 127
range function, 83, 331
rango, 91, 152, 153
rangos, 154
raíz, 228
read-eval-print loop, 291
readability, 6, 38, 51, 56, 141, 175, 219, 280
readable stream, 299, 300, 302, 317
readdir function, 297, 304, 344
readdirSync function, 344
readFile function, 180, 296, 345
readFileSync function, 297, 344
reading code, 8, 122
readStream function, 317, 318
real-time, 248
reasoning, 18
recipe analogy, 87
record, 64
rectangle, 268, 281
recursion, 48, 51, 52, 58, 84, 199, 205, 215, 217, 219, 231, 246, 330, 332, 339, 341

recursión, 336
red, 183, 189
reduce method, 93, 95, 99, 100, 333
ReferenceError type, 226
RegExp class, 150, 164, 344
regexp golf (exercise), 173
regular expression, 151, 152, 161, 163, 166, 307, 314, 315, 341, 344
  alternatives, 158
  backtracking, 159
  boundary, 157
  creation, 150, 164
  escaping, 150, 165, 337
  flags, 154, 161, 165, 337
  global, 161, 166, 167
  grouping, 154, 162
  internationalization, 170
  matching, 158, 166
  methods, 151, 155, 165
  repetition, 153
rejecting (a promise), 196, 199, 209
relación simbiótica, 191
relative path, 181, 292, 301, 345
relative positioning, 242, 243
remainder operator, 14, 35, 329, 330, 342
remote access, 301
removeChild method, 233
removeEventListener method, 249, 343
rename function, 297
renderTalk function, 323
renderTalkForm function, 324
renderUserField function, 323
repeat method, 77, 259
repeating key, 253
repetición, 160, 164
repetition, 54, 153, 263
replace method, 161, 162, 174, 336
replaceChild method, 233, 342
reportError function, 323
request, 297–299, 306
request function, 198, 299, 300
requestAnimationFrame function, 243, 261, 263, 286, 342
requestType function, 199
require function, 179, 180, 187, 292–294, 303, 314
reserved word, 26
resolution, 181, 292
resolve function, 303
resolving (a promise), 195, 196, 199, 209
resource, 301, 316
response, 298, 302, 305
responsiveness, 248, 290
respuesta, 193, 198
retry, 198
return keyword, 42, 47, 105, 206, 330, 333
return value, 28, 42, 140, 333
reuse, 57, 118, 175–177, 293
reverse method, 83
reversing (exercise), 83, 331
rgb (CSS), 277
right-aligning, 246
rmdir function, 304, 307
roadGraph object, 123
roads array, 122
roads module (exercise), 187, 338
robot, 122, 124, 127, 129, 132, 187
robot efficiency (exercise), 132, 335
robustez, 311
rounding, 80, 139, 282
router, 310, 314
Router class, 314, 315
routeRequest function, 204
routeRobot function, 129
routing, 202
rule (CSS), 240, 241
run function, 222

run-time error, 137, 138, 140, 147, 341
runAnimation function, 286, 289
runGame function, 288, 289
runLevel function, 287, 289
running code, 8
runRobot function, 127, 335

sandbox, 8
scaling, 276
scalpel (exercise), 211, 339
scheduling, 208
scientific notation, 13, 174
scope, 44, 50, 218, 221, 225, 226, 340, 341
script, 254
script (HTML tag), 261
SCRIPTS data set, 91, 93, 96, 98, 100
scroll event, 259, 263
scrolling, 252, 259, 278, 279, 286
search method, 165
search problem, 232, 307
search tool (exercise), 307, 344
searching, 158, 160, 165
sección, 168
secuencia, 153
security, 303, 313
seguimiento de la pila, 142
selección, 241
select (HTML tag), 345
semicolon, 23, 24
send method, 193, 198
sendGossip function, 201
sep binding, 303
Separador de vocales Mongol, 170
serialization, 81, 82
server, 297, 298, 300, 301, 314
servidor, 290, 309
set, 152
set (data structure), 120, 132
Set class, 120, 132, 335
set method, 110
setAttribute method, 235, 236
setInterval function, 263
setter, 116
setTimeout function, 192, 208, 262, 263, 320, 326
shared property, 105, 107, 108
shift key, 254
shift method, 74
shiftKey property, 254
short-circuit evaluation, 53, 219, 333
SICP, 213
side effect, 24, 35, 57, 66, 83, 92, 166, 184, 210, 230, 233, 237
sign, 13, 174
sign bit, 13
signal, 11
signo, 337
signo de interrogación, 164
simplicity, 224
simulation, 124, 127, 267, 272, 342
sine, 79, 244, 274, 284
single-quote character, 15, 174
singleton, 133
sintaxis, 213, 214
sintáctico, 228
sistema de archivos, 296
sistema de escritura, 90
sistema de módulos, 178
sitio web, 309
sitios web, 290
skill-sharing project, 311, 314, 321
SkillShareApp class, 326
skipSpace function, 216, 226
slash character, 14, 38, 150, 163, 303, 344
slice method, 75, 76, 91, 234, 331, 340

sloppy programming, 264
smooth animation, 244
socket, 311
solicitud, 193, 198
some method, 96, 100, 201, 315
sondeo, 248
sondeo largo, 310
sorting, 228
source property, 166
special form, 213, 219
special return value, 140, 141
specialForms object, 219
speed, 1
split method, 124, 270
spread operator, 277
square brackets, 62, 78, 152, 331
square example, 41, 45, 46
square root, 70, 78, 334
stack, *see* call stack
stack overflow, 48, 51, 59, 330
standard, 290
standard environment, 27
standard output, 301
startsWith method, 302
stat function, 297, 303, 304, 344
state, 32, 34, 35
    in
        objects, 270
    of application, 278, 328
statement, 23
static (reserved word), 26
static file, 312, 315
static method, 116, 120, 270, 335
Stats type, 304
statSync function, 344
status code, 291
status property, 322
stdout property, 301
stoicism, 248
stopPropagation method, 251
storage function, 195
stream, 298–300, 302, 306, 317
strict mode, 135
string, 15, 61, 63, 66
    indexing, 59, 75, 77, 97, 155
    length, 39, 97
    methods, 76, 155
    notation, 15
    properties, 76
    representation, 16
    searching, 76
String function, 29, 111
strings, 96
strong (HTML tag), 236, 238
structure, 176
structure sharing, 84
style, 238
style (HTML tag), 240
style attribute, 238–240, 275
style sheet, *see* CSS
subclase, 117
submit event, 324, 345
substitution, 57
subtraction, 14, 120
suites de prueba, 138
sum function, 83
summing (exercise), 83, 331
summing example, 5, 86, 93, 222
superclase, 117
suposición, 145
Sussman, Gerald, 213
switch keyword, 36
Symbol function, 111
Symbol.iterator symbol, 113
SymmetricMatrix class, 117
synchronization, 326, 346
synchronous programming, 189, 206, 297, 307
syncState method, 346
syntax, 12, 13, 15, 23, 24, 26, 29, 32,

34, 36, 41, 45, 64, 134, 135, 141, 145, 174
syntax tree, 215, 217, 218, 228
SyntaxError type, 216
símbolo, 112

tabbed interface (exercise), 266, 343
tabindex attribute, 254
tabla, 70, 71
tabla de frecuencias, 69
table, 277
table (HTML tag), 245, 268, 277
table example, 341
tableFor function, 71
talk, 316–318
talkResponse method, 319
talksAbout function, 231
talkURL function, 322
Tamil, 90
tangent, 79
target property, 251
task management example, 74
TCP, 311
td (HTML tag), 246, 277
teclado, 27, 248, 252, 253
teclado virtual, 254
teléfono, 254
temperature example, 116
template, 328, 346
tentacle (analogy), 25, 65, 67
teoría, 139
terminal, 291
termitas, 191
ternary operator, 19, 22, 219
test method, 151
test suite, 137
testing, 132, 137
text, 15
text node, 229, 231, 234, 343
text-align (CSS), 246
TEXT_NODE code, 229, 343
textarea (HTML tag), 263, 345
textContent property, 342
texto, 227, 229, 296
textScripts function, 98, 333
th (HTML tag), 246
then method, 195–197, 200, 339
this, 63, 102, 103, 105, 135
thread, 208
throw keyword, 141, 142, 146, 148, 336
tiempo, 156, 192, 244, 264
Tiempo Unix, 156
time, 281, 284, 287
timeline, 190, 208, 243, 248, 261
timeout, 198, 262, 313, 320
Timeout class, 198
times method, 272
tipo, 12
tipo de solicitud, 194
tipo variable, 137
title, 321
toLowerCase method, 63, 246
tool, 184, 295
top (CSS), 242–244, 246
top-level scope, *see* global scope
toString method, 103, 104, 108, 109, 111, 300
touch, 257
touchend event, 257
touches method, 281
touches property, 258
touchmove event, 257
touchstart event, 257
toUpperCase method, 63, 137, 246, 300
tr (HTML tag), 246, 277
trackKeys function, 286, 289
transpilation, 224
traversal, 159

trial and error, 139, 285
triangle (exercise), 39, 329
trigonometry, 244
trigonometría, 79
trim method, 76, 270
truco, 182
true, 17
try keyword, 142, 144, 200, 336
type, 17, 118
type checking, 137, 183
type coercion, 20, 21, 29
type property, 214, 250
typeof operator, 17, 85, 333
TypeScript, 137

unary operator, 17
uncaught exception, 145, 197
undefined, 20, 21, 25, 42, 48, 62, 65, 81, 135, 140
underline, 238
underscore character, 26, 37, 101, 157, 164
Unicode, 16, 18, 90, 96, 152, 170, 171
  property, 171
unidades de código, 96
uniformidad, 214
uniqueness, 240
unit (CSS), 245, 259
Unix, 304, 306, 307
unlink function, 297, 304
unshift method, 74
upcasing server example, 300
updated method, 317, 318, 320, 345
upgrading, 177
URL, 299, 312, 322
url package, 302, 319
urlToPath function, 302
usability, 253
use strict, *see* strict mode
user experience, 248, 323
user interface, 145
UTF16, 16, 96
UTF8, 296, 297

validation, 140, 147, 213, 280, 317, 318
valor de retorno, 194
valores, 11, 194
var keyword, 26, 42, 43, 81
variable, 4, *see* binding
variable contador, 342
Vec class, 120, 270, 271, 284
vector (exercise), 120, 334
velocidad, 3
vencimiento, 311
ventana, 251, 257
verbosity, 47, 190
versionado semántico, 295
versión, 177, 294, 295
viewport, 278, 280
VillageState class, 124
VillaPradera, 122
vinculaciones, 45, 65, 81, 145, 182
vinculación, 32, 34, 40–42, 67, 168
virtual world, 124, 127
vista en vivo, 311
vocabulario, 41, 86
vocabulary, 41, 87
void operator, 26
volatile data storage, 11

waitForChanges method, 320
waiting, 192
wave, 274, 284
Web, *see* World Wide Web
web application, 6
web browser, *see* browser
web worker, 262
WebDAV, 307

website, 308
WebSockets, 311
weekDay module, 178
weekend project, 308
weresquirrel example, 60, 64, 66, 68, 72, 74
while loop, 5, 32, 34, 55, 167
whitespace, 33, 37, 76, 152, 170, 226, 340
  in HTML, 232, 343
  in URLs, 312
  trimming, 270
why, 23
window, 260
window object, 249
with statement, 136
wizard (mighty), 4
word boundary, 157
word character, 157
World Wide Web, 6, 81
writable stream, 298, 299, 301, 302
write method, 298, 299
writeFile function, 296, 300, 345
writeHead method, 298
writing code, 8, 122
WWW, *see* World Wide Web

XML, 230

yield (reserved word), 26
yield keyword, 207
your own loop (example), 100
Yuan-Ma, 11, 175, 290

Zawinski, Jamie, 150
zero-based counting, 59, 61, 156
zeroPad function, 56
zona horaria, 156

ángulo, 244
árbol, 104, 214, 217, 228, 229
árbol de sintaxis, 214
índice, 61