

PIENSA EN JAVA

PIENSA EN JAVA

Cuarta Edición

BRUCE ECKEL

President, MindView, Inc.

Traducción

Vuelapluma



Madrid • México • Santa Fe de Bogotá • Buenos Aires • Caracas • Lima
Montevideo • San Juan • San José • Santiago • São Paulo • White Plains •

EDUCACIÓN GENERAL
UNIVERSIDAD NACIONAL
NUEVA YORK

| Datos de catalogación bibliográfica |
|---|
| <p>PIENSA EN JAVA Bruce Eckel PEARSON EDUCACIÓN S.A., Madrid, 2007 ISBN: 978-84-8966-034-2 Materia: Informática, 004 Formato: 215 x 270 mm. Páginas: 1004</p> |

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

DERECHOS RESERVADOS

© 2007 por PEARSON EDUCACIÓN S.A.
Ribera del Loira, 28
28042 Madrid

PIENSA EN JAVA

Bruce Eckel

ISBN: 978-84-8966-034-2

Deposito Legal: M-4.753-2007

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Authorized translation from the English language edition, entitled THINKING IN JAVA, 4th Edition by ECKEL BRUCE, published by Pearson Education Inc, publishing as Prentice Hall, Copyright © 2006

EQUIPO EDITORIAL

Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

EQUIPO DE PRODUCCIÓN:

Director: José A. Clares
Técnico: María Alvear

Diseño de Cubierta: Equipo de diseño de Pearson Educación S.A.

Impreso por: Gráficas Rógar, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

Dedicatoria

A Dawn

Prefacio

Originalmente, me enfrenté a Java como si fuera “simplemente otro lenguaje más de programación”, lo cual es cierto en muchos sentidos.

Pero, a medida que fue pasando el tiempo y lo fui estudiando con mayor detalle, comencé a ver que el objetivo fundamental de este lenguaje era distinto de los demás lenguajes que había visto hasta el momento.

La tarea de programación se basa en gestionar la complejidad: la complejidad del problema que se quiere resolver, sumada a la complejidad de la máquina en la cual hay que resolverlo. Debido a esta complejidad, la mayoría de los proyectos de programación terminan fallando. A pesar de lo cual, de todos los lenguajes de programación que conozco, casi ninguno de ellos había adoptado como *principal* objetivo de diseño resolver la complejidad inherente al desarrollo y el mantenimiento de los programas.¹ Por supuesto, muchas decisiones del diseño de lenguajes se realizan teniendo presente esa complejidad, pero siempre termina por considerarse esencial introducir otros problemas dentro del conjunto de los objetivos. Inevitablemente, son estos otros problemas los que hacen que los programadores terminen no pudiendo cumplir el objetivo principalmente con esos lenguajes. Por ejemplo, C++ tenía que ser compatible en sentido descendente con C (para permitir una fácil migración a los programadores de C), además de ser un lenguaje eficiente. Ambos objetivos resultan muy útiles y explican parte del éxito de C++, pero también añaden un grado adicional de complejidad que impide a algunos proyectos finalizar (por supuesto, podemos echar la culpa a los programadores y a los gestores, pero si un lenguaje puede servir de ayuda detectando los errores que cometemos, ¿por qué no utilizar esa posibilidad?). Otro ejemplo, Visual BASIC (VB) estaba ligado a BASIC, que no había sido diseñado para ser un lenguaje extensible, por lo que todas las extensiones añadidas a VB han dado como resultado una sintaxis verdaderamente inmanejable. Perl es compatible en sentido descendente con awk, sed, grep y otras herramientas Unix a las que pretendía sustituir, y como resultado, se le acusa a menudo de generar “código de sólo escritura” (es decir, después de pasado un tiempo se vuelve completamente ilegible). Por otro lado, C++, VB, Perl y otros lenguajes como Smalltalk han centrado *algo* de esfuerzo de diseño en la cuestión de la complejidad, y como resultado, ha tenido un gran éxito a la hora de resolver ciertos tipos de problemas.

Lo que más me ha impresionado cuando he llegado a entender el lenguaje Java es que dentro del conjunto de objetivos de diseño establecido por Sun, parece que se hubiera decidido tratar de reducir la complejidad *para el programador*. Como si quienes marcaron esos objetivos hubieran dicho: “Tratemos de reducir el tiempo y la dificultad necesarios para generar código robusto”. Al principio, este objetivo daba como resultado un código que no se ejecutaba especialmente rápido (aunque esto ha mejorado a lo largo del tiempo), pero desde luego ha permitido reducir considerablemente el tiempo de desarrollo, que es inferior en un 50 por ciento o incluso más al tiempo necesario para crear un programa en C++ equivalente. Simplemente por esto, ya podemos ahorrar cantidades enormes de tiempo y de dinero, pero Java no se detiene ahí, sino que trata de hacer transparentes muchas de las complejas tareas que han llegado a ser importantes en el mundo de la programación, como la utilización de múltiples hebras o la programación de red, empleando para conseguir esa transparencia una serie de características del lenguaje y de bibliotecas preprogramadas que pueden hacer que muchas tareas lleguen a resultar sencillas. Finalmente, Java aborda algunos problemas realmente complejos: programas interplataforma, cambios de código dinámicos e incluso cuestiones de seguridad, todos los cuales representan problemas de una complejidad tal que pueden hacer fracasar proyectos completos de programación. Por tanto, a pesar de los problemas de prestaciones, las oportunidades que Java nos proporciona son inmensas, ya que puede incrementar significativamente nuestra productividad como programadores.

Java incrementa el ancho de banda de comunicación *entre las personas* en todos los sentidos: a la hora de crear los programas, a la hora de trabajar en grupo, a la hora de construir interfaces para comunicarse con los usuarios, a la hora de

¹ Sin embargo, creo que el lenguaje Python es el que más se acerca a ese objetivo. Consulte www.Python.org.

ejecutar los programas en diferentes tipos de máquinas y a la hora de escribir con sencillez aplicaciones que se comuniquen a través de Internet.

En mi opinión, los resultados de la revolución de las comunicaciones no se percibirán a partir de los efectos de desplazar grandes cantidades de bits de un sitio a otro, sino que seremos conscientes de la verdadera revolución a medida que veamos cómo podemos comunicarnos con los demás de manera más sencilla, tanto en comunicaciones de persona a persona, como en grupos repartidos por todo el mundo. Algunos sugieren que la siguiente revolución será la formación de una especie de mente global derivada de la interconexión de un número suficiente de personas. No sé si Java llegará a ser la herramienta que fomente dicha revolución, pero esa posibilidad me ha hecho sentir, al menos, que estoy haciendo algo importante al tratar de enseñar este lenguaje.

Java SE5 y SE6

Esta edición del libro aprovecha en buena medida las mejoras realizadas al lenguaje Java en lo que Sun originalmente denominó JDK 1.5 y cambió posteriormente a JDK5 o J2SE5. Posteriormente, la empresa eliminó el obsoleto “2” y cambió el nombre a Java SE5. Muchos de los cambios en el lenguaje Java SE5 fueron decididos para mejorar la experiencia de uso del programador. Como veremos, los diseñadores del lenguaje Java no obtuvieron un completo éxito en esta tarea, pero en general dieron pasos muy significativos en la dirección correcta.

Uno de los objetivos más importantes de esta edición es absorber de manera completa las mejoras introducidas por Java SE5/6, presentarlas y emplearlas a lo largo de todo el texto. Esto quiere decir que en esta edición se ha tomado la dura decisión de hacer el texto únicamente compatible con Java SE5/6, por lo que buena parte del código del libro no puede compilarse con las versiones anteriores de Java; el sistema de generación de código dará errores y se detendrá si se intenta efectuar esa compilación. A pesar de todo, creo que los beneficios de este enfoque compensan el riesgo asociado a dicha decisión.

Si el lector prefiere por algún motivo las versiones anteriores de Java, se puede descargar el texto de las versiones anteriores de este libro (en inglés) en la dirección www.MindView.net. Por diversas razones, la edición actual del libro no está en formato electrónico gratuito, sino que sólo pueden descargarse las ediciones anteriores.

Java SE6

La redacción de este libro ha sido, en sí misma, un proyecto de proporciones colosales y al que ha habido que dedicar muchísimo tiempo. Y antes de que el libro fuera publicado, la versión Java SE6 (cuyo nombre en clave es *mustang*) apareció en versión beta. Aunque hay unos cuantos cambios de menor importancia en Java SE6 que mejoran algunos de los ejemplos incluidos en el libro, el tratamiento de Java SE6 no ha afectado en gran medida al contenido del texto; las principales mejoras de la nueva versión se centran en el aumento de la velocidad y en determinadas funcionalidades de biblioteca que caían fuera del alcance del texto.

El código incluido en el libro ha sido comprobado con una de las primeras versiones comerciales de Java SE6, por lo que no creo que vayan a producirse cambios que afecten al contenido del texto. Si hubiera algún cambio importante a la hora de lanzar oficialmente JavaSE6, ese cambio se verá reflejado en el código fuente del libro, que puede descargarse desde www.MindView.net.

En la portada del libro se indica que este texto es para “Java SE5/6”, lo que significa “escrito para Java SE5 teniendo en cuenta los significativos cambios que dicha versión ha introducido en el lenguaje, pero siendo el texto igualmente aplicable a Java SE6”.

La cuarta edición

La principal satisfacción a la hora de realizar una nueva edición de un libro es la de poder “corregir” el texto, aplicando todo aquello que he podido aprender desde que la última edición viera la luz. A menudo, estas lecciones son derivadas de esa frase que dice: “Aprender es aquello que conseguimos cuando no conseguimos lo que queremos”, y escribir una nueva edición del libro constituye siempre una oportunidad de corregir errores o hacer más amena la lectura. Asimismo, a la hora de abordar una nueva edición vienen a la mente nuevas ideas fascinantes y la posibilidad de cometer nuevos errores se ve más que compensada por el placer de descubrir nuevas cosas y la capacidad de expresar las ideas de una forma más adecuada.

Asimismo, siempre se tiene presente, en el fondo de la mente, ese desafío de escribir un libro que los poseedores de las ediciones anteriores estén dispuestos a comprar. Ese desafío me anima siempre a mejorar, reescribir y reorganizar todo lo que puedo, con el fin de que el libro constituya una experiencia nueva y valiosa para los lectores más fieles.

Cambios

El CD-ROM que se había incluido tradicionalmente como parte del libro no ha sido incluido en esta edición. La parte esencial de dicho CD, el seminario multimedia *Thinking in C* (creado para MindView por Chuck Allison), está ahora disponible como presentación Flash descargable. El objetivo de dicho seminario consiste en preparar a aquellos que no estén lo suficientemente familiarizados con la sintaxis de C, de manera que puedan comprender mejor el material presentado en este libro. Aunque en dos de los capítulos del libro se cubre en buena medida la sintaxis a un nivel introductorio, puede que no sean suficientes para aquellas personas que carezcan de los conocimientos previos adecuados, y la presentación *Thinking in C* trata de ayudar a dichas personas a alcanzar el nivel necesario.

El capítulo dedicado a la concurrencia, que antes llevaba por título “Programación multihebra”, ha sido completamente reescrito con el fin de adaptarlo a los cambios principales en las bibliotecas de concurrencia de Java SE5, pero sigue proporcionando información básica sobre las ideas fundamentales en las que la concurrencia se apoya. Sin esas ideas fundamentales, resulta difícil comprender otras cuestiones más complejas relacionadas con la programación multihebra. He invertido muchos meses en esta tarea, inmerso en ese mundo denominado “concurrencia” y el resultado final es que el capítulo no sólo proporciona los fundamentos del tema sino que también se aventura en otros territorios más novedosos.

Existe un nuevo capítulo dedicado a cada una de las principales características nuevas del lenguaje Java SE5, y el resto de las nuevas características han sido reflejadas en las modificaciones realizadas sobre el material existente. Debido al estudio continuado que realicé de los patrones de diseño, también se han introducido en todo el libro nuevos patrones.

El libro ha sufrido una considerable reorganización. Buena parte de los cambios se deben a razones pedagógicas, junto con la perfección de que quizás mi concepto de “capítulo” necesitaba ser revisado. Adicionalmente, siempre he tendido a creer que un tema tenía que tener “la suficiente envergadura” para justificar el dedicarle un capítulo. Pero luego he visto, especialmente a la hora de enseñar los patrones de diseño, que las personas que asistían a los seminarios obtenían mejores resultados si se presentaba un único patrón y a continuación se hacía, inmediatamente, un ejercicio, incluso si eso significaba que yo sólo hablara durante un breve periodo de tiempo (asimismo, descubrí que esta nueva estructura era más agradable para el profesor). Por tanto, en esta versión del libro he tratado de descomponer los capítulos según los temas, sin preocuparme de la longitud final de cada capítulo. Creo que el resultado representa una auténtica mejora.

También he llegado a comprender la enorme importancia que tiene el tema de las pruebas de código. Sin un marco de pruebas predefinido, con una serie de pruebas que se ejecuten cada vez que se construya el sistema, no hay forma de saber si el código es fiable o no. Para conseguir este objetivo en el libro, he creado un marco de pruebas que permite mostrar y validar la salida de cada programa (dicho marco está escrito en Python, y puede descargarse en www.MindView.net). El tema de las pruebas, en general, se trata en el suplemento disponible en <http://www.MindView.net/Books/BetterJava>, que presenta lo que creo que son capacidades fundamentales que todos los programadores deberían tener como parte de sus conocimientos básicos.

Además, he repasado cada uno de los ejemplos del libro preguntándome a mí mismo: “¿Por qué lo hice de esta manera?”. En la mayoría de los casos, he realizado algunas modificaciones y mejoras, tanto para hacer los ejemplos más coherentes entre sí, como para demostrar lo que considero que son las reglas prácticas de programación en Java, (al menos dentro de los límites de un texto introductorio). Para muchos de los ejemplos existentes, se ha realizado un rediseño y una nueva implementación con cambios significativos con respecto a las versiones anteriores. Aquellos ejemplos que me parecía que ya no tenían sentido han sido eliminados y se han añadido, asimismo, nuevos ejemplos.

Los lectores de las ediciones anteriores han hecho numerosísimos comentarios muy pertinentes, lo que me llena de satisfacción. Sin embargo, de vez en cuando también me llegan algunas quejas y, por alguna razón, una de las más frecuentes es que “este libro es demasiado voluminoso”. En mi opinión, si la única queja es que este libro tiene “demasiadas páginas”, creo que el resultado global es satisfactorio (se me viene a la mente el comentario de aquel emperador de Austria que se quejaba de la obra de Mozart diciendo que tenía “demasiadas notas”; por supuesto, no trato en absoluto de compararme con Mozart). Además, debo suponer que ese tipo de quejas proceden de personas que todavía no han llegado a familiarizarse con la enorme variedad de características del propio lenguaje Java y que no han tenido ocasión de consultar el resto de libros dedicados a este tema. De todos modos, una de las cosas que he tratado de hacer en esta edición es recortar aquellas partes

que han llegado a ser obsoletas, o al menos, no esenciales. En general, se ha repasado todo el texto eliminando lo que ya había dejado de ser necesario, incluyendo los cambios pertinentes y mejorando el contenido de la mejor manera posible. No me importa demasiado eliminar algunas partes, porque el material original correspondiente continúa estando en el sitio web (www.MindView.net), gracias a la versión descargable de las tres primeras ediciones del libro. Asimismo, el lector tiene a su disposición material adicional en suplementos descargables de esta edición.

En cualquier caso, para aquellos lectores que sigan considerando excesivo el tamaño del libro les pido disculpas. Lo crean o no, he hecho cuanto estaba en mi mano para que ese tamaño fuera el menor posible.

Sobre el diseño de la cubierta

La cubierta del libro está inspirada por el movimiento *American Arts & Crafts Movement* que comenzó poco antes del cambio de siglo y alcanzó su cenit entre 1900 y 1920. Comenzó en Inglaterra como reacción a la producción en masa de la revolución industrial y al estilo altamente ornamental de la época victoriana. *Arts & Crafts* enfatizaba el diseño con formas naturales, como en el movimiento *art nouveau*, como el trabajo manual y la importancia del artesano, sin por ello renunciar al uso de herramientas modernas. Existen muchos ecos con la situación que vivimos hoy en día: el cambio de siglo, la evolución desde los rudimentarios comienzos de la revolución informática hacia algo más refinado y significativo y el énfasis en la artesanía del software en lugar de en su simple manufactura.

La concepción de Java tiene mucho que ver con este punto de vista. Es un intento de elevar al programador por encima del sistema operativo, para transformarlo en un “artesano del software”.

Tanto el autor de este libro como el diseñador de la cubierta (que son amigos desde la infancia) han encontrado inspiración en este movimiento, ambos poseemos muebles, lámparas y otros objetos originales de este periodo o inspirados en el mismo.

El otro tema de la cubierta sugiere una vitrina colecciónista que un naturalista podría emplear para mostrar los especímenes de insectos que ha preservado. Estos insectos son objetos situados dentro de los objetos compartimento. Los objetos compartimento están a su vez, colocados dentro del “objeto cubierta”, lo que ilustra el concepto de agregación dentro de la programación orientada a objetos. Por supuesto, cualquier programador de habla inglesa efectuará enseguida entre los insectos “bugs” y los errores de programación (también *bugs*). Aquí, esos insectos/errores han sido capturados y presumiblemente muertos en un tarro y confinados finalmente dentro de una vitrina, con lo que tratamos de sugerir la habilidad que Java tiene para encontrar, mostrar y corregir los errores (habilidad que constituye uno de sus más potentes atributos).

En esta edición, yo me he encargado de la acuarela que puede verse como fondo de la cubierta.

Agradecimientos

En primer lugar, gracias a todos los colegas que han trabajado conmigo a la hora de impartir seminarios, realizar labores de consultoría y desarrollar proyectos pedagógicos: Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer y Jamie King. Agradezco la paciencia que mostráis mientras continúo tratando de desarrollar el mejor modelo para que una serie de personas independientes como nosotros puedan continuar trabajando juntos.

Recientemente, y gracias sin duda a Internet, he tenido la oportunidad de relacionarme con un número sorprendentemente grande de personas que me ayudan en mi trabajo, usualmente trabajando desde sus propias oficinas. En el pasado, yo tendría que haber adquirido o alquilado una gran oficina para que todas estas personas pudieran trabajar, pero gracias a Internet, a los servicios de mensajeros y al teléfono, ahora puedo contar con su ayuda sin esos costes adicionales. Dentro de mis intentos por aprender a “trabajar eficazmente con los demás”, todos vosotros me habéis ayudado enormemente y espero poder continuar aprendiendo a mejorar mi trabajo gracias a los esfuerzos de otros. La ayuda de Paula Steuer ha sido valiosísima a la hora de tomar mis poco inteligentes prácticas empresariales y transformarlas en algo razonable (gracias por ayudarme cuando no quiero encargarme de algo concreto, Paula). Jonathan Wilcox, Esq., se encargó de revisar la estructura de mi empresa y de eliminar cualquier piedra que pudiera tener un posible escorpión, haciéndonos marchar disciplinadamente a través del proceso de poner todo en orden desde el punto de vista legal, gracias por tu minuciosidad y tu persistencia. Sharlynn Cobaugh ha llegado a convertirse en una auténtica experta en edición de sonido y ha sido una de las personas esenciales a la hora de crear los cursos de formación multimedia, además de ayudar en la resolución de muchos otros problemas. Gracias por la perseverancia que has demostrado a la hora de enfrentarte con problemas informáticos complejos. La gente de Amaio en Praga también ha sido de gran ayuda en numerosos proyectos. Daniel Will-Harris fue mi primera fuen-

te de inspiración en lo que respecta al proyecto de trabajo a través de Internet y también ha sido imprescindible, por supuesto, en todas las soluciones de diseño gráfico que he desarrollado.

A lo largo de los años, a través de sus conferencias y seminarios, Gerald Weinberg se ha convertido en mi entrenador y mentor extraoficial, por lo que le estoy enormemente agradecido.

Ervin Varga ha proporcionado numerosas correcciones técnicas para la cuarta edición, aunque también hay otras personas que han ayudado en esta tarea, con diversos capítulos y ejemplos. Ervin ha sido el revisor técnico principal del libro y también se encargó de escribir la guía de soluciones para la cuarta edición. Los errores detectados por Ervin y las mejoras que él ha introducido en el libro han permitido redondear el texto. Su minuciosidad y su atención al detalle resultan sorprendentes y es, con mucho, el mejor revisor técnico que he tenido. Muchas gracias, Ervin.

Mi weblog en la página www.Artilma.com de Bill Venners también ha resultado de gran ayuda a la hora de verificar determinadas ideas. Gracias a los lectores que me han ayudado a aclarar los conceptos enviando sus comentarios; entre esos lectores debo citar a James Watson, Howard Lovatt, Michael Barker, y a muchos otros que no menciono por falta de espacio, en particular a aquellos que me han ayudado en el tema de los genéricos.

Gracias a Mark Welsh por su ayuda continuada.

Evan Cofsky continúa siendo de una gran ayuda, al conocer de memoria todos los arcanos detalles relativos a la configuración y mantenimiento del servidor web basados en Linux, así como a la hora de mantener optimizado y protegido el servidor MindView.

Gracias especiales a mi nuevo amigo el café, que ha permitido aumentar enormemente el entusiasmo por el proyecto. Camp4 Coffee en Crested Butte, Colorado, ha llegado a ser el lugar de reunión normal cada vez que alguien venía a los seminarios de MindView y proporciona el mejor catering que he visto para los descansos en el seminario. Gracias a mi colega Al Smith por crear ese café y por convertirlo en un lugar tan extraordinario, que ayuda a hacer de Crested Butte un lugar mucho más interesante y placentero. Gracias también a todos los camareros de Camp4, que tan servicialmente atienden a sus clientes.

Gracias a la gente de Prentice Hall por seguir atendiendo a todas mis peticiones, y por facilitarme las cosas en todo momento.

Hay varias herramientas que han resultado de extraordinaria utilidad durante el proceso de desarrollo y me siento en deuda con sus creadores cada vez que las uso. Cygwin (www.cygwin.com) me ha permitido resolver innumerables problemas que Windows no puede resolver y cada día que pasa más me engancha a esta herramienta (me hubiera encantado disponer de ella hace 15 años, cuando tenía mi mente orientada a Gnu Emacs). Eclipse de IBM (www.eclipse.org) representa una maravillosa contribución a la comunidad de desarrolladores y cabe esperar que se puedan obtener grandes cosas con esta herramienta a medida que vaya evolucionando. JetBrains IntelliJ Idea continúa abriendo nuevos y creativos caminos dentro del campo de las herramientas de desarrollo.

Comencé a utilizar Enterprise Architect de Sparxsystems con este libro y se ha convertido rápidamente en mi herramienta UML favorita. El formateador de código Jalopy de Marco Hunsicker (www.triemax.com) también ha resultado muy útil en numerosas ocasiones y Marco me ha ayudado extraordinariamente a la hora de configurarlo para mis necesidades concretas. En mi opinión, la herramienta JEdit de Slava Pestov y sus correspondientes *plug-ins* también resultan útiles en diversos momentos (www.jedit.org); esta herramienta es un editor muy adecuado para todos aquellos que se estén iniciando en el desarrollo de seminarios.

Y por supuesto, por si acaso no lo he dejado claro aún, utilizo constantemente Python (www.Python.org) para resolver problemas, esta herramienta es la criatura de mi colega Guido Van Rossum y de la panda de enloquecidos genios con los que disfruté enormemente haciendo deporte durante unos cuantos días (a Tim Peters me gustaría decirle que he enmarcado ese ratón que tomó prestado, al que le he dado el nombre oficial de “TimBotMouse”). Permitidme tan sólo recomendaros que busquéis otros lugares más sanos para comer. Asimismo, mi agradecimiento a toda la comunidad Python, formada por un conjunto de gente extraordinaria.

Son muchas las personas que me han hecho llegar sus correcciones y estoy en deuda con todas ellas, pero quiero dar las gracias en particular a (por la primera edición): Kevin Raulerson (encontró numerosísimos errores imperdonables), Bob Resendes (simplemente increíble), John Pinto, Joe Dante, Joe Sharp (fabulosos vuestros comentarios), David Combs (numerosas correcciones de clarificación y de gramática), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, y muchos otros. El Profesor Marc Meurrens dedicó

una gran cantidad de esfuerzo a publicitar y difundir la versión electrónica de la primera edición de este libro en Europa.

Gracias a todos aquellos que me han ayudado a reescribir los ejemplos para utilizar la biblioteca Swing (para la segunda edición), así como a los que han proporcionado otros tipos de comentarios: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis y a todos los demás que me han manifestado su apoyo.

En la cuarta edición, Chris Grindstaff resultó de gran ayuda durante el desarrollo de la sección SWT y Sean Neville escribió para mí el primer borrador de la sección dedicada a Flex.

Kraig Brockschmidt y Gen Kiyooka son algunas de esas personas inteligentes que he podido conocer en algún momento de vida y que han llegado a ser auténticos amigos, habiendo tenido una enorme influencia sobre mí. Son personas poco usuales en el sentido de que practican yoga y otras formas de engrandecimiento espiritual, que me resultan particularmente inspiradoras e instructivas.

Me resulta sorprendente que el saber de Delphi me ayudara a comprender Java, ya que ambos lenguajes tienen en común muchos conceptos y decisiones relativas al diseño del lenguaje. Mis amigos de Delphi me ayudaron enormemente a la hora de entender mejor ese maravilloso entorno de programación. Se trata de Marco Cantu (otro italiano, quizás el ser educado en latín mejora las aptitudes de uno para los lenguajes de programación), Neil Rubenking (que solía dedicarse al yoga, la comida vegetariana y el Zen hasta que descubrió las computadoras) y por supuesto Zack Urlocker (el jefe de producto original de Delphi), un antiguo amigo con el que he recorrido el mundo. Todos nosotros estamos en deuda con el magnífico Anders Hejlsberg, que continúa asombrándonos con C# (lenguaje que, como veremos en el libro, fue una de las principales inspiraciones para Java SE5).

Los consejos y el apoyo de mi amigo Richard Hale Shaw (al igual que los de Kim) me han sido de gran ayuda. Richard y yo hemos pasado muchos meses juntos impartiendo seminarios y tratando de mejorar los aspectos pedagógicos con el fin de que los asistentes disfrutaran de una experiencia perfecta.

El diseño del libro, el diseño de la cubierta y la fotografía de la cubierta han sido realizados por mi amigo Daniel Will-Harris, renombrado autor y diseñador (www.Will-Harris.com), que ya solía crear sus propios diseños en el colegio, mientras esperaba a que se inventaran las computadoras y las herramientas de autoedición, y que ya entonces se quejaba de mi forma de resolver los problemas de álgebra. Sin embargo, yo me he encargado de preparar para impresión las páginas, por lo que los errores de fotocomposición son míos. He utilizado Microsoft® Word XP para Windows a la hora de escribir el libro y de preparar las páginas para impresión mediante Adobe Acrobat; este libro fue impreso directamente a partir de los archivos Acrobat PDF. Como tributo a la era electrónica yo me encontraba fuera del país en el momento de producir la primera y la segunda ediciones finales del libro; la primera edición fue enviada desde Ciudad del Cabo (Sudáfrica), mientras que la segunda edición fue enviada desde Praga. La tercera y cuarta ediciones fueron realizadas desde Crested Butte, Colorado. En la versión en inglés del libro se utilizó el tipo de letra *Georgia* para el texto y los títulos están en *Verdana*. La letra de la cubierta original es *ITC Rennie Mackintosh*.

Gracias en especial a todos mis profesores y estudiantes (que también son mis profesores).

Mi gato Molly solía sentarse en mi regazo mientras trabajaba en esta edición, ofreciéndome así mi propio tipo de apoyo peludo y cálido.

Entre los amigos que también me han dado su apoyo, y a los que debo citar (aunque hay muchos otros a los que no cito por falta de espacio), me gustaría destacar a: Patty Gast (extraordinaria masajista), Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates en *Midnight Engineering Magazine*, Larry Constantine y Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris y Laura Strand, los Almquists, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, la familia Robbins, la familia Moelter (y los McMillans), Michael Wilk, Dave Stoner, los Cranstons, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin y Sonda Donovan, Joe Lordi, Dave y Brenda Bartlett, Patti Gast, Blake, Annette & Jade, los Rentschlers, los Sudeks, Dick, Patty, y Lee Eckel, Lynn y Todd, y sus familias. Y por supuesto, a mamá y papá.

Resumen del contenido

| | |
|--|-----|
| Prefacio | xix |
| Introducción | xxv |
| 1 Introducción a los objetos | 1 |
| 2 Todo es un objeto | 23 |
| 3 Operadores | 43 |
| 4 Control de ejecución | 71 |
| 5 Inicialización y limpieza | 85 |
| 6 Control de acceso | 121 |
| 7 Reutilización de clases | 139 |
| 8 Polimorfismo | 165 |
| 9 Interfaces | 189 |
| 10 Clases internas | 211 |
| 11 Almacenamiento de objetos | 241 |
| 12 Tratamiento de errores mediante excepciones | 277 |
| 13 Cadenas de caracteres | 317 |
| 14 Información de tipos | 351 |
| 15 Genéricos | 393 |
| 16 Matrices | 483 |
| 17 Análisis detallado de los contenedores | 513 |
| 18 E/S | 587 |
| 19 Tipos enumerados | 659 |
| 20 Anotaciones | 693 |
| 21 Concurrencia | 727 |
| 22 Interfaces gráficas de usuario | 857 |
| A Suplementos | 955 |
| B Recursos | 959 |
| Índice | 963 |

Contenido

| | |
|---|------------|
| Prefacio | xix |
| Java SE5 y SE6..... | xx |
| Java SE6..... | xx |
| La cuarta edición..... | xx |
| Cambios | xxi |
| Sobre el diseño de la cubierta..... | xxii |
| Agradecimientos..... | xxii |
| Introducción | xxv |
| Prerrequisitos | xxv |
| Aprendiendo Java | xxvi |
| Objetivos | xxvi |
| Enseñar con este libro..... | xxvii |
| Documentación del JDK en HTML..... | xxvii |
| Ejercicios | xxvii |
| Fundamentos para Java..... | xxvii |
| Código fuente | xxviii |
| Estándares de codificación | xxix |
| Errores..... | xxx |
| 1 Introducción a los objetos | 1 |
| El progreso de la abstracción | 1 |
| Todo objeto tiene una interfaz | 3 |
| Un objeto proporciona servicios..... | 4 |
| La implementación oculta..... | 5 |
| Reutilización de la implementación..... | 6 |
| Herencia | 6 |
| Relaciones es-un y es-como-un | 8 |
| Objetos intercambiables con polimorfismo | 9 |
| La jerarquía de raíz única | 11 |
| Contenedores..... | 12 |
| Tipos parametrizados (genéricos) | 13 |
| Creación y vida de los objetos | 13 |
| Tratamiento de excepciones: manejo de errores.. | 15 |
| Programación concurrente | 15 |
| Java e Internet..... | 16 |
| ¿Qué es la Web?..... | 16 |
| Programación del lado del cliente | 18 |
| Programación del lado del servidor..... | 21 |
| Resumen | 22 |
| 2 Todo es un objeto | 23 |
| Los objetos se manipulan mediante referencias .. | 23 |
| Es necesario crear todos los objetos | 24 |
| Los lugares de almacenamiento..... | 24 |
| Caso especial: tipos primitivos | 25 |
| Matrices en Java..... | 26 |
| Nunca es necesario destruir un objeto | 27 |
| Ámbito | 27 |
| Ámbito de los objetos | 27 |
| Creación de nuevos tipos de datos: class | 28 |
| Campos y métodos | 28 |
| Métodos, argumentos y valores de retorno..... | 29 |
| La lista de argumentos | 30 |
| Construcción de un programa Java | 31 |
| Visibilidad de los nombres | 31 |
| Utilización de otros componentes | 31 |
| La palabra clave static | 32 |
| Nuestro primer programa Java | 33 |
| Compilación y ejecución | 35 |
| Comentarios y documentación embebida..... | 35 |
| Documentación mediante comentarios | 36 |
| Sintaxis | 36 |
| HTML embebido | 37 |
| Algunos marcadores de ejemplo | 37 |
| Ejemplo de documentación | 39 |
| Estilo de codificación | 39 |
| Resumen | 40 |
| Ejercicios | 40 |

| | |
|---|------------|
| 3 Operadores | 43 |
| Instrucciones simples de impresión | 43 |
| Utilización de los operadores Java | 44 |
| Precedencia | 44 |
| Asignación | 44 |
| Creación de alias en las llamadas a métodos | 46 |
| Operadores matemáticos | 46 |
| Operadores unarios más y menos | 48 |
| Autoincremento y autodecremento | 48 |
| Operadores relacionales | 49 |
| Comprobación de la equivalencia de objetos | 49 |
| Operadores lógicos | 51 |
| Cortocircuitos | 52 |
| Literales | 53 |
| Notación exponencial | 54 |
| Operadores bit a bit | 55 |
| Operadores de desplazamiento | 55 |
| Operador ternario if-else | 58 |
| Operadores + y += para String | 59 |
| Errores comunes a la hora de utilizar operadores | 60 |
| Operadores de proyección | 60 |
| Truncamiento y redondeo | 61 |
| Promoción | 62 |
| Java no tiene operador “ sizeof ” | 62 |
| Compendio de operadores | 62 |
| Resumen | 70 |
| 4 Control de ejecución | 71 |
| true y false | 71 |
| if-else | 71 |
| Iteración | 72 |
| do-while | 73 |
| for | 73 |
| El operador coma | 74 |
| Sintaxis foreach | 74 |
| return | 76 |
| break y continue | 77 |
| La despreciada instrucción “ goto ” | 78 |
| switch | 81 |
| Resumen | 83 |
| 5 Inicialización y limpieza | 85 |
| Inicialización garantizada con el constructor | 85 |
| Sobrecarga de métodos | 87 |
| Cómo se distingue entre métodos sobrecargados | 88 |
| 6 Control de acceso | 121 |
| package : la unidad de biblioteca | 122 |
| Organización del código | 123 |
| Creación de nombres de paquete únicos | 124 |
| Una biblioteca personalizada de herramientas | 126 |
| Utilización de importaciones para modificar el comportamiento | 128 |
| Un consejo sobre los nombres de paquete | 128 |
| Especificadores de acceso Java | 128 |
| Acceso de paquete | 129 |
| public : acceso de interfaz | 129 |
| private : ¡no lo toque! | 130 |
| protected : acceso de herencia | 131 |
| Interfaz e implementación | 133 |
| Acceso de clase | 134 |
| Resumen | 136 |
| 7 Reutilización de clases | 139 |
| Sintaxis de la composición | 139 |
| Sintaxis de la herencia | 142 |
| Inicialización de la clase base | 143 |

| | | | |
|---|-----|---|-----|
| Delegación | 145 | Ampliación de la interfaz mediante herencia | 201 |
| Combinación de la composición y de la herencia | 147 | Colisiones de nombres al combinar interfaces | 202 |
| Cómo garantizar una limpieza apropiada | 148 | Adaptación a una interfaz | 203 |
| Ocultación de nombres | 151 | Campos en las interfaces | 205 |
| Cómo elegir entre la composición y la herencia | 152 | Inicialización de campos en las interfaces | 205 |
| protected | 153 | Anidamiento de interfaces | 206 |
| <i>Upcasting</i> (generalización)..... | 154 | Interfaces y factorías | 208 |
| ¿Por qué “ <i>upcasting</i> ”?..... | 155 | Resumen | 210 |
| Nueva comparación entre la composición y la herencia..... | 155 | 10 Clases internas | 211 |
| La palabra clave final | 156 | Creación de clases internas | 211 |
| Datos final | 156 | El enlace con la clase externa | 213 |
| Métodos final | 159 | Utilización de this y new | 214 |
| Clases final | 161 | Clases interna y generalización | 216 |
| Una advertencia sobre final | 161 | Clases internas en los métodos y ámbitos | 217 |
| Inicialización y carga de clases..... | 162 | Clases internas anónimas | 219 |
| Inicialización con herencia | 162 | Un nuevo análisis del método factoria | 222 |
| Resumen | 163 | Clase anidadas | 224 |
| 8 Polimorfismo | 165 | Clases dentro de interfaces | 225 |
| Nuevas consideraciones sobre la generalización | 165 | Acceso al exterior desde una clase múltiplemente anidada | 226 |
| Por qué olvidar el tipo de un objeto | 166 | ¿Para qué se usan las clases internas? | 227 |
| El secreto | 168 | Cierres y retrollamada | 229 |
| Acoplamiento de las llamadas a métodos | 168 | Clases internas y marcos de control | 230 |
| Especificación del comportamiento correcto | 168 | Cómo heredar de clases internas | 236 |
| Ampliabilidad..... | 171 | ¿Pueden sustituirse las clases internas? | 236 |
| Error: “sustitución” de métodos private | 173 | Clases internas locales | 238 |
| Error: campos y métodos static | 174 | Identificadores de una clase interna | 239 |
| Constructores y polimorfismo | 175 | Resumen | 239 |
| Orden de las llamadas a los constructores..... | 176 | 11 Almacenamiento de objetos | 241 |
| Herencia y limpieza | 177 | Genéricos y contenedores seguros respecto al tipo | 242 |
| Comportamiento de los métodos polimórficos dentro de los constructores | 181 | Conceptos básicos | 244 |
| Tipos de retorno covariantes | 183 | Adición de grupos de elementos | 245 |
| Diseño de sistemas con herencia | 183 | Impresión de contenedores | 247 |
| Sustitución y extensión | 184 | List | 248 |
| Especialización e información de tipos en tiempo de ejecución | 186 | Iterator | 252 |
| Resumen | 187 | ListIterator | 254 |
| 9 Interfaces | 189 | LinkedList | 255 |
| Clases abstractas y métodos abstractos | 189 | Stack | 256 |
| Interfaces | 192 | Set | 258 |
| Desacoplamiento completo | 195 | Map | 260 |
| “Herencia múltiple” en Java..... | 199 | Queue | 263 |
| | | PriorityQueue | 264 |
| | | Comparación entre Collection e Iterator | 266 |

| | |
|--|------------|
| La estructura <i>foreach</i> y los iteradores | 268 |
| El método basado en adaptadores | 270 |
| Resumen | 273 |
| 12 Tratamiento de errores mediante excepciones | 277 |
| Conceptos | 277 |
| Excepciones básicas | 278 |
| Argumentos de las excepciones | 279 |
| Detección de una excepción | 279 |
| El bloque try | 280 |
| Rutinas de tratamiento de excepciones | 280 |
| Creación de nuestras propias excepciones | 281 |
| Excepciones y registro | 283 |
| La especificación de la excepción | 286 |
| Cómo capturar una excepción | 286 |
| La traza de la pila | 288 |
| Regeneración de una excepción | 289 |
| Encadenamiento de excepciones | 291 |
| Excepciones estándar de Java | 294 |
| Caso especial: RuntimeException | 294 |
| Realización de tareas de limpieza con finally .. | 295 |
| ¿Para qué sirve finally ? | 296 |
| Utilización de finally durante la ejecución de la instrucción return | 299 |
| Un error: la excepción perdida | 300 |
| Restricciones de las excepciones | 301 |
| Constructores | 303 |
| Localización de excepciones | 307 |
| Enfoques alternativos | 308 |
| Historia | 309 |
| Perspectivas | 310 |
| Paso de las excepciones a la consola | 312 |
| Conversión de las excepciones comprobadas en no comprobadas | 313 |
| Directrices relativas a las excepciones | 315 |
| Resumen | 315 |
| 13 Cadenas de caracteres | 317 |
| Cadenas de caracteres inmutables | 317 |
| Comparación entre la sobrecarga de '+' y StringBuilder | 318 |
| Recursión no intencionada | 321 |
| Operaciones con cadenas de caracteres | 322 |
| Formato de la salida | 324 |
| printf() | 324 |
| System.out.format() | 324 |
| La clase Formatter | 325 |
| Especificadores de formato | 326 |
| Conversiones posibles con Formatter | 327 |
| String.format() | 329 |
| Expresiones regulares | 331 |
| Fundamentos básicos | 331 |
| Creación de expresiones regulares | 333 |
| Cuantificadores | 335 |
| Pattern y Matcher | 336 |
| split() | 342 |
| Operaciones de sustitución | 343 |
| reset() | 344 |
| Expresiones regulares y E/S en Java | 345 |
| Análisis de la entrada | 346 |
| Delimitadores de Scanner | 348 |
| Análisis con expresiones regulares | 348 |
| StringTokenizer | 349 |
| Resumen | 350 |
| 14 Información de tipos | 351 |
| La necesidad de RTTI | 351 |
| El objeto Class | 353 |
| Literales de clase | 357 |
| Referencias de clase genéricas | 359 |
| Nueva sintaxis de proyección | 361 |
| Comprobación antes de una proyección | 361 |
| Utilización de literales de clase | 367 |
| Instanceof dinámico | 368 |
| Recuento recursivo | 369 |
| Factorías registradas | 371 |
| instanceof y equivalencia de clases | 373 |
| Reflexión: información de clases en tiempo de ejecución | 374 |
| Un extractor de métodos de clases | 375 |
| Proxies dinámicos | 378 |
| Objetos nulos | 381 |
| Objetos maqueta y stubs | 387 |
| Interfaces e información de tipos | 387 |
| Resumen | 392 |
| 15 Genéricos | 393 |
| Comparación con C++ | 394 |
| Genéricos simples | 394 |
| Una biblioteca de tuplas | 396 |

| | | | |
|--|-----|---|------------|
| Una clase que implementa una pila | 398 | Utilización del patrón Decorador | 461 |
| RandomList | 399 | Mixins con proxies dinámicos | 462 |
| Interfaces genéricas | 399 | Tipos latentes | 464 |
| Métodos genéricos | 403 | Compensación de la carencia de tipos latentes | 467 |
| Aprovechamiento de la inferencia del argumento de tipo | 404 | Reflexión | 467 |
| Varargs y métodos genéricos | 405 | Aplicación de un método a una secuencia | 468 |
| Un método genérico para utilizar con generadores | 406 | ¿Qué pasa cuando no disponemos de la interfaz correcta? | 471 |
| Un generador de propósito general | 407 | Simulación de tipos latentes mediante adaptadores | 472 |
| Simplificación del uso de las tuplas | 408 | Utilizando los objetos de función como estrategias | 474 |
| Una utilidad Set | 409 | Resumen: ¿realmente es tan malo el mecanismo de proyección? | 479 |
| Clases internas anónimas | 412 | Lecturas adicionales | 481 |
| Construcción de modelos complejos | 413 | | |
| El misterio del borrado | 415 | | |
| La técnica usada en C++ | 417 | | |
| Compatibilidad de la migración | 419 | | |
| El problema del borrado de tipos | 419 | | |
| El efecto de los límites | 421 | | |
| Compensación del borrado de tipos | 424 | | |
| Creación de instancias de tipos | 425 | | |
| Matrices de genéricos | 427 | | |
| Límites | 431 | | |
| Comodines | 434 | | |
| ¿Hasta qué punto es inteligente el compilador? | 436 | | |
| Contravarianza | 438 | | |
| Comodines no limitados | 440 | | |
| Conversion de captura | 444 | | |
| Problemas | 445 | | |
| No pueden usarse primitivas como parámetros de tipo | 445 | | |
| Implementación de interfaces parametrizadas | 447 | | |
| Proyecciones de tipos y advertencias | 447 | | |
| Sobrecarga | 449 | | |
| Secuestro de una interfaz por parte de la clase base | 449 | | |
| Tipos autolimitados | 450 | | |
| Genéricos curiosamente recurrentes | 450 | | |
| Autolimitación | 451 | | |
| Covarianza de argumentos | 453 | | |
| Seguridad dinámica de los tipos | 456 | | |
| Excepciones | 457 | | |
| Mixins | 458 | | |
| Mixins en C++ | 459 | | |
| Mezclado de clases utilizando interfaces | 460 | | |
| | | 16 Matrices | 483 |
| | | Por qué las matrices son especiales | 483 |
| | | Las matrices son objetos de primera clase | 484 |
| | | Devolución de una matriz | 487 |
| | | Matrices multidimensionales | 488 |
| | | Matrices y genéricos | 491 |
| | | Creación de datos de prueba | 493 |
| | | Arrays.fill() | 493 |
| | | Generadores de datos | 494 |
| | | Creación de matrices a partir de generadores | 498 |
| | | Utilidades para matrices | 502 |
| | | Copia en una matriz | 502 |
| | | Comparación de matrices | 503 |
| | | Comparaciones de elementos de matriz | 504 |
| | | Ordenación de una matriz | 507 |
| | | Búsquedas en una matriz ordenada | 508 |
| | | Resumen | 509 |
| | | 17 Análisis detallado de los contenedores | 513 |
| | | Taxonomía completa de los contenedores | 513 |
| | | Relleno de contenedores | 514 |
| | | Una solución basada en generador | 515 |
| | | Generadores de mapas | 516 |
| | | Utilización de clases abstractas | 519 |
| | | Funcionalidad de las colecciones | 525 |
| | | Operaciones opcionales | 528 |
| | | Operaciones no soportadas | 529 |
| | | Funcionalidad de List | 530 |
| | | Conjuntos y orden de almacenamiento | 533 |
| | | SortedSet | 536 |
| | | Colas | 537 |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|-----|---|-----|---|-----|---|-----|---|-----|---|-----|--|-----|---|-----|--|-----|---|-----|--|-----|---|-----|--|-----|---|-----|--|-----|--|-----|---|-----|--|-----|---|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|---|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|--|-----|------------------------------------|-----|--|-----|--|-----|--|-----|--|--|---------------------|-----|--------------------------------|-----|---------------|-----|---|--|---------------------------------|-----|-----------------------------|-----|----------------------------------|-----|--|-----|--|--|---------------------|-----|
| Colas con prioridad | 538 | Orígenes y destinos de los datos | 601 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Colas dobles | 539 | Modificación del comportamiento de los flujos | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Mapas | 540 | Rendimiento | 541 | de datos | 601 | SortedMap | 544 | Clases no modificadas | 602 | LinkedHashMap | 545 | RandomAccessFile | 602 | Almacenamiento y códigos <i>hash</i> | 545 | Utilización típica de los flujos de E/S | 603 | Funcionamiento de <i>hashCode()</i> | 548 | Archivo de entrada con <i>buffer</i> | 603 | Mejora de la velocidad con el almacenamiento | | <i>hash</i> | 550 | Entrada desde memoria | 604 | Sustitución de hashCode() | 553 | Entrada de memoria formateada | 604 | Selección de una implementación | 558 | Salida básica a archivo | 605 | Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 |
| Rendimiento | 541 | de datos | 601 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SortedMap | 544 | Clases no modificadas | 602 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| LinkedHashMap | 545 | RandomAccessFile | 602 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Almacenamiento y códigos <i>hash</i> | 545 | Utilización típica de los flujos de E/S | 603 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Funcionamiento de <i>hashCode()</i> | 548 | Archivo de entrada con <i>buffer</i> | 603 | Mejora de la velocidad con el almacenamiento | | <i>hash</i> | 550 | Entrada desde memoria | 604 | Sustitución de hashCode() | 553 | Entrada de memoria formateada | 604 | Selección de una implementación | 558 | Salida básica a archivo | 605 | Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | |
| Archivo de entrada con <i>buffer</i> | 603 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Mejora de la velocidad con el almacenamiento | | <i>hash</i> | 550 | Entrada desde memoria | 604 | Sustitución de hashCode() | 553 | Entrada de memoria formateada | 604 | Selección de una implementación | 558 | Salida básica a archivo | 605 | Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | |
| <i>hash</i> | 550 | Entrada desde memoria | 604 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Sustitución de hashCode() | 553 | Entrada de memoria formateada | 604 | Selección de una implementación | 558 | Salida básica a archivo | 605 | Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Entrada de memoria formateada | 604 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Selección de una implementación | 558 | Salida básica a archivo | 605 | Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Salida básica a archivo | 605 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Un marco de trabajo para pruebas de rendimiento | 558 | Almacenamiento y recuperación de datos | 607 | Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Almacenamiento y recuperación de datos | 607 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Selección entre listas | 561 | Lectura y escritura de archivos de acceso | | Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Lectura y escritura de archivos de acceso | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Peligros asociados a las micropuebas de | | aleatorio | 608 | rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| aleatorio | 608 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| rendimiento | 566 | Flujos de datos canalizados | 609 | Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Flujos de datos canalizados | 609 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Selección de un tipo de conjunto | 567 | Utilidades de lectura y escritura de archivos | 609 | Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Utilidades de lectura y escritura de archivos | 609 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Selección de un tipo de mapa | 569 | Lectura de archivos binarios | 612 | Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Lectura de archivos binarios | 612 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Utilidades | 572 | E/S estándar | 612 | Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| E/S estándar | 612 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Ordenaciones y búsquedas en las listas | 575 | Lectura de la entrada estándar | 613 | Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Lectura de la entrada estándar | 613 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Creación de colecciones o mapas no | | Cambio de System.out a un objeto PrintWriter | 613 | modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Cambio de System.out a un objeto PrintWriter | 613 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| modificables | 576 | Redirecciónamiento de la E/S estándar | 613 | Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Redirecciónamiento de la E/S estándar | 613 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Sincronización de una colección o un mapa | 577 | Control de procesos | 614 | Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Control de procesos | 614 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Almacenamiento de referencias | 578 | Los paquetes new | 616 | WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Los paquetes new | 616 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| WeakHashMap | 580 | Conversión de datos | 618 | Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Conversión de datos | 618 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Contenedores Java 1.0/1.1 | 581 | Extracción de primitivas | 621 | Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Extracción de primitivas | 621 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Vector y Enumeration | 581 | Buffers utilizados como vistas | 622 | Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Buffers utilizados como vistas | 622 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Hashtable | 582 | Manipulación de datos con <i>buffers</i> | 625 | Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Manipulación de datos con <i>buffers</i> | 625 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Stack | 582 | Detalles acerca de los <i>buffers</i> | 625 | BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Detalles acerca de los <i>buffers</i> | 625 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| BitSet | 584 | Archivos mapeados en memoria | 629 | Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Archivos mapeados en memoria | 629 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Resumen | 585 | Bloqueo de archivos | 632 | 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Bloqueo de archivos | 632 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 18 Entrada/salida | 587 | Compresión | 634 | La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Compresión | 634 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| La clase File | 587 | Compresión simple con GZIP | 635 | Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Compresión simple con GZIP | 635 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Una utilidad para listados de directorio | 587 | Almacenamiento de múltiples archivos con Zip | 635 | Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Almacenamiento de múltiples archivos con Zip | 635 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Utilidades de directorio | 590 | Archivos Java (JAR) | 637 | Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Archivos Java (JAR) | 637 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Búsqueda y creación de directorios | 594 | Serialización de objetos | 639 | Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Serialización de objetos | 639 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Entrada y salida | 596 | Localización de la clase | 642 | Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Localización de la clase | 642 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tipos de InputStream | 597 | Control en la serialización | 642 | Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Control en la serialización | 642 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tipos de OutputStream | 598 | Utilización de la persistencia | 649 | Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Utilización de la persistencia | 649 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Adición de atributos e interfaces útiles | 598 | XML | 654 | Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| XML | 654 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Lectura de un flujo InputStream con | | Preferencias | 656 | FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Preferencias | 656 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| FilterInputStream | 599 | Resumen | 658 | Escritura de un flujo OutputStream con | | FilterOutputStream | 599 | Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Resumen | 658 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Escritura de un flujo OutputStream con | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| FilterOutputStream | 599 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Lectores y escritores | 600 | 19 Tipos enumerados | 659 | Características básicas de las enumeraciones | 659 | Utilización de importaciones estáticas con las | | enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 19 Tipos enumerados | 659 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Características básicas de las enumeraciones | 659 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Utilización de importaciones estáticas con las | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| enumeraciones | 660 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

| | | | |
|---|------------|---|-----|
| Adición de métodos a una enumeración | 661 | Mejora del diseño del código | 730 |
| Sustitución de los métodos de una enumeración | 662 | Conceptos básicos sobre hebras | 731 |
| Enumeraciones en las instrucciones switch | 662 | Definición de las tareas | 731 |
| El misterio de values() | 663 | La clase Thread | 732 |
| Implementa, no hereda | 665 | Utilización de Executor | 734 |
| Selección aleatoria | 666 | Producción de valores de retorno de las tareas | 736 |
| Utilización de interfaces con propósitos de organización | 667 | Cómo dormir una tarea | 737 |
| Utilización de EnumSet en lugar de indicadores | 671 | Prioridad | 738 |
| Utilización de EnumMap | 672 | Cesión del control | 740 |
| Métodos específicos de constante | 673 | Hebras demonio | 740 |
| Cadena de responsabilidad en las enumeraciones | 676 | Variaciones de código | 744 |
| Máquinas de estado con enumeraciones | 680 | Terminología | 748 |
| Despacho múltiple | 684 | Absorción de una hebra | 748 |
| Cómo despachar con enumeraciones | 686 | Creación de interfaces de usuario de respuesta rápida | 750 |
| Utilización de métodos específicos de constante | 688 | Grupos de hebras | 751 |
| Cómo despachar con mapas EnumMap | 690 | Captura de excepciones | 751 |
| Utilización de una matriz 2-D | 690 | Compartición de recursos | 753 |
| Resumen | 691 | Acceso inapropiado a los recursos | 754 |
| 20 Anotaciones | 693 | Resolución de la contienda por los recursos compartidos | 756 |
| Sintaxis básica | 694 | Atomicidad y volatilidad | 760 |
| Definición de anotaciones | 694 | Clases atómicas | 764 |
| Meta-anotaciones | 695 | Secciones críticas | 765 |
| Escritura de procesadores de anotaciones | 696 | Sincronización sobre otros objetos | 770 |
| Elementos de anotación | 697 | Almacenamiento local de las hebras | 771 |
| Restricciones de valor predeterminado | 697 | Terminación de tareas | 772 |
| Generación de archivos externos | 697 | El jardín ornamental | 772 |
| Soluciones alternativas | 700 | Terminación durante el bloqueo | 775 |
| Las anotaciones no soportan la herencia | 700 | Interrupción | 776 |
| Implementación del procesador | 700 | Comprobación de la existencia de una interrupción | 782 |
| Utilización de apt para procesar anotaciones | 703 | Cooperación entre tareas | 784 |
| Utilización del patrón de diseño Visitante con apt | 706 | wait() y notifyAll() | 784 |
| Pruebas unitarias basadas en anotaciones | 709 | notify() y notifyAll() | 788 |
| Utilización de @Unit con genéricos | 716 | Productores y consumidores | 791 |
| No hace falta ningún “agrupamiento” | 717 | Productores-consumidores y colas | 796 |
| Implementación de @Unit | 717 | Utilización de canalizaciones para la E/S entre tareas | 800 |
| Eliminación del código de prueba | 723 | Interbloqueo | 801 |
| Resumen | 724 | Nuevos componentes de biblioteca | 805 |
| 21 Conurrencia | 727 | CountDownLatch | 805 |
| Las múltiples caras de la concurrencia | 728 | CyclicBarrier | 807 |
| Ejecución más rápida | 728 | DelayQueue | 809 |
| | | PriorityBlockingQueue | 811 |

| | |
|---|------------|
| El controlador de invernadero implementado con <i>ScheduledExecutor</i> | 814 |
| <i>Semaphore</i> | 817 |
| <i>Exchanger</i> | 820 |
| Simulación | 821 |
| Simulación de un cajero | 821 |
| Simulación de un restaurante | 826 |
| Distribución de trabajo | 829 |
| Optimización del rendimiento | 834 |
| Comparación de las tecnologías mutex | 834 |
| Contenedores libres de bloqueos | 841 |
| Bloqueo optimista | 847 |
| <i>ReadWriteLocks</i> | 848 |
| Objetos activos | 850 |
| Resumen | 853 |
| Lecturas adicionales | 855 |
| 22 Interfaces gráficas de usuario | 857 |
| <i>Applets</i> | 858 |
| Fundamentos de Swing | 859 |
| Un entorno de visualización | 861 |
| Definición de un botón | 862 |
| Captura de un suceso | 862 |
| Áreas de texto | 864 |
| Control de la disposición | 865 |
| <i>BorderLayout</i> | 866 |
| <i>FlowLayout</i> | 867 |
| <i>GridLayout</i> | 867 |
| <i>GridBagLayout</i> | 868 |
| Posicionamiento absoluto | 868 |
| <i>BoxLayout</i> | 868 |
| ¿Cuál es la mejor solución? | 868 |
| El modelo de sucesos de Swing | 868 |
| Tipos de sucesos y de escuchas | 869 |
| Control de múltiples sucesos | 874 |
| Una selección de componentes Swing | 876 |
| Botones | 876 |
| Iconos | 878 |
| Sugerencias | 880 |
| Campos de texto | 880 |
| Bordes | 881 |
| Un mini-editor | 882 |
| Casillas de verificación | 883 |
| Botones de opción | 884 |
| Cuadros combinados (listas desplegables) | 885 |
| Cuadros de lista | 886 |
| Tableros con fichas | 887 |
| Recuadros de mensaje | 888 |
| Menús | 890 |
| Menús emergentes | 894 |
| Dibujo | 895 |
| Cuadros de diálogo | 898 |
| Cuadros de diálogo de archivos | 901 |
| HTML en los componentes Swing | 902 |
| Deslizadores y barras de progreso | 903 |
| Selección del aspecto y del estilo | 904 |
| Árboles, tablas y portapapeles | 906 |
| JNLP y Java Web Start | 906 |
| Concurrencia y Swing | 910 |
| Tareas de larga duración | 910 |
| Hebras visuales | 916 |
| Programación visual y componentes JavaBean | 918 |
| ¿Qué es un componente JavaBean? | 919 |
| Extracción de la información BeanInfo con Introspectar | 920 |
| Una Bean más sofisticada | 924 |
| Sincronización en JavaBeans | 927 |
| Empaquetado de una Bean | 930 |
| Soporte avanzado de componentes Bean | 931 |
| Más información sobre componentes Bean | 932 |
| Alternativas a Swing | 932 |
| Construcción de clientes web Flash con Flex | 932 |
| Hello, Flex | 932 |
| Compilación de MXML | 933 |
| MXML y ActionScript | 934 |
| Contenedores y controles | 935 |
| Efectos y estilos | 936 |
| Sucesos | 937 |
| Conexión con Java | 937 |
| Modelos de datos y acoplamiento de datos | 939 |
| Construcción e implantación de aplicaciones | 940 |
| Creación de aplicaciones SWT | 941 |
| Instalación de SWT | 941 |
| Hello, SWT | 941 |
| Eliminación del código redundante | 944 |
| Menús | 945 |
| Paneles con fichas, botones y sucesos events | 946 |
| Gráficos | 949 |
| Concurrencia en SWT | 950 |
| ¿SWT o Swing? | 952 |
| Resumen | 952 |
| Recursos | 953 |

| | |
|--|------------|
| A Suplementos | 955 |
| Suplementos descargables | 955 |
| <i>Thinking in C</i> : fundamentos para Java | 955 |
| Seminario <i>Thinking in Java</i> | 955 |
| Seminario <i>CD Hands-On Java</i> | 956 |
| Seminario <i>Thinking in Objects</i> | 956 |
| <i>Thinking in Enterprise Java</i> | 956 |
| <i>Thinking in Patterns</i> (con Java) | 957 |
| Seminario <i>Thinking in Patterns</i> | 957 |
| Consultoría y revisión de diseño | 957 |
| B Recursos | 959 |
| Software | 959 |
| Editores y entornos IDE | 959 |
| Libros | 959 |
| Análisis y diseño | 960 |
| Python | 962 |
| Mi propia lista de libros | 962 |
| Índice | 963 |

Introducción

“El dio al hombre la capacidad de hablar, y de esa capacidad surgió el pensamiento. Que es la medida del Universo” *Prometeo desencadenado*, Shelley

Los seres humanos ... estamos, en buena medida, a merced del lenguaje concreto que nuestra sociedad haya elegido como medio de expresión. Resulta completamente ilusorio creer que nos ajustamos a la realidad esencialmente sin utilizar el lenguaje y que el lenguaje es meramente un medio incidental de resolver problemas específicos de comunicación y reflexión. Lo cierto es que el “mundo real” está en gran parte construido, de manera inconsciente, sobre los hábitos lingüísticos del grupo.

El estado de la Lingüística como ciencia, 1929, Edward Sapir

Como cualquier lenguaje humano, Java proporciona una forma de expresar conceptos. Si tiene éxito, esta forma de expresión será significativamente más fácil y flexible que las alternativas a medida que los problemas crecen en tamaño y en complejidad.

No podemos ver Java sólo como una colección de características, ya que algunas de ellas no tienen sentido aisladas. Sólo se puede emplear la suma de las partes si se está pensando en el *diseño* y no simplemente en la codificación. Y para entender Java así, hay que comprender los problemas del lenguaje y de la programación en general. Este libro se ocupa de los problemas de la programación, porque son problemas, y del método que emplea Java para resolverlos. En consecuencia, el conjunto de características que el autor explica en cada capítulo se basa en la forma en que él ve cómo puede resolverse un tipo de problema en particular con este lenguaje. De este modo, el autor pretende conducir, poco a poco, al lector hasta el punto en que Java se convierta en su lengua materna.

La actitud del autor a lo largo del libro es la de conseguir que el lector construya un modelo mental que le permita desarrollar un conocimiento profundo del lenguaje; si se enfrenta a un puzzle, podrá fijarse en el modelo para tratar de deducir la respuesta.

Prerrequisitos

Este libro supone que el lector está familiarizado con la programación: sabe que un programa es una colección de instrucciones, qué es una subrutina, una función o una macro, conoce las instrucciones de control como “if” y las estructuras de bucle como “while”, etc. Sin embargo, es posible que el lector haya aprendido estos conceptos en muchos sitios, tales como la programación con un lenguaje de macros o trabajando con una herramienta como Perl. Cuando programe sintiéndose cómodo con las ideas básicas de la programación, podrá abordar este libro. Por supuesto, el libro será *más fácil* para los programadores de C y más todavía para los de C++, pero tampoco debe autoexcluirse si no tiene experiencia con estos lenguajes (aunque tendrá que trabajar duro). Puede descargarse en www.MindView.net el seminario multimedia *Thinking in C*, el cual le ayudará a aprender más rápidamente los fundamentos necesarios para estudiar Java. No obstante, en el libro se abordan los conceptos de programación orientada a objetos (POO) y los mecanismos de control básicos de Java.

Aunque a menudo se hacen referencias a las características de los lenguajes C y C++ no es necesario profundizar en ellos, aunque sí ayudarán a todos los programadores a poner a Java en perspectiva con respecto a dichos lenguajes, de los que al fin y al cabo desciende. Se ha intentado que estas referencias sean simples y sirvan para explicar cualquier cosa con la que una persona que nunca haya programado en C/C++ no esté familiarizado.

Aprendiendo Java

Casi al mismo tiempo que se publicó mi primer libro, *Using C++* (Osborne/McGraw-Hill, 1989), comencé a enseñar dicho lenguaje. Enseñar lenguajes de programación se convirtió en mi profesión; desde 1987 he visto en auditorios de todo el mundo ver dudar a los asistentes, he visto asimismo caras sorprendidas y expresiones de incredulidad. Cuando empecé a impartir cursos de formación a grupos pequeños, descubrí algo mientras se hacían ejercicios. Incluso aquellos que sonreían se quedaban con dudas sobre muchos aspectos. Comprendí al dirigir durante una serie de años la sesión de C++ en la *Software Development Conference* (y más tarde la sesión sobre Java), que tanto yo como otros oradores tocábamos demasiados temas muy rápidamente. Por ello, tanto debido a la variedad en el nivel de la audiencia como a la forma de presentar el material, se termina perdiendo audiencia. Quizá es pedir demasiado pero dado que soy uno de esos que se resisten a las conferencias tradicionales (y en la mayoría de los casos, creo que esa resistencia proviene del aburrimiento), quería intentar algo que permitiera tener a todo el mundo enganchado.

Durante algún tiempo, creé varias presentaciones diferentes en poco tiempo, por lo que terminé aprendiendo según el método de la experimentación e iteración (una técnica que también funciona en el diseño de programas). Desarrollé un curso utilizando todo lo que había aprendido de mi experiencia en la enseñanza. Mi empresa, MindView, Inc., ahora imparte el seminario *Thinking in Java* (piensa en Java); que es nuestro principal seminario de introducción que proporciona los fundamentos para nuestros restantes seminarios más avanzados. Puede encontrar información detallada en www.MindView.net. El seminario de introducción también está disponible en el CD-ROM *Hands-On Java*. La información se encuentra disponible en el mismo sitio web.

La respuesta que voy obteniendo en cada seminario me ayuda a cambiar y reenfocar el material hasta que creo que funciona bien como método de enseñanza. Pero este libro no son sólo las notas del seminario; he intentado recopilar el máximo de información posible en estas páginas y estructurarla de manera que cada tema lleve al siguiente. Más que cualquier otra cosa, el libro está diseñado para servir al lector solitario que se está enfrentando a un nuevo lenguaje de programación.

Objetivos

Como mi anterior libro, *Thinking in C++*, este libro se ha diseñado con una idea en mente: la forma en que las personas aprenden un lenguaje. Cuando pienso en un capítulo del libro, pienso en términos de qué hizo que fuera una lección durante un seminario. La información que me proporcionan las personas que asisten a un seminario me ayuda a comprender cuáles son las partes complicadas que precisan una mayor explicación. En las áreas en las que fui ambicioso e incluí demasiadas características a un mismo tiempo, pude comprobar que si incluía muchas características nuevas, tenía que explicarlas y eso contribuía fácilmente a la confusión del estudiante.

En cada capítulo he intentado enseñar una sola característica o un pequeño grupo de características asociadas, sin que sean necesarios conceptos que todavía no se hayan presentado. De esta manera, el lector puede asimilar cada pieza en el contexto de sus actuales conocimientos.

Mis objetivos en este libro son los siguientes:

1. Presentar el material paso a paso de modo que cada idea pueda entenderse fácilmente antes de pasar a la siguiente. Secuenciar cuidadosamente la presentación de las características, de modo que se haya explicado antes de que se vea en un ejemplo. Por supuesto, esto no siempre es posible, por lo que en dichas situaciones, se proporciona una breve descripción introductoria.
2. Utilizar ejemplos que sean tan simples y cortos como sea posible. Esto evita en ocasiones acometer problemas del “mundo real”, pero he descubierto que los principiantes suelen estar más contentos cuando pueden comprender todos los detalles de un ejemplo que cuando se ven impresionados por el ámbito del problema que resuelve. También, existe una seria limitación en cuanto a la cantidad de código que se puede absorber en el aula. Por esta razón, no dudaré en recibir críticas acerca del uso de “ejemplos de juguete”, sino que estoy deseando recibirlas en aras de lograr algo pedagógicamente útil.
3. Dar lo que yo creo que es importante para que se comprenda el lenguaje, en lugar de contar todo lo que yo sé. Pienso que hay una jerarquía de importancia de la información y que existen hechos que el 95% de los programadores nunca conocerán, detalles que sólo sirven para confundir a las personas y que incrementan su percepción de la complejidad del lenguaje. Tomemos un ejemplo de C, si se memoriza la tabla de precedencia de los

operadores (yo nunca lo he hecho), se puede escribir código inteligente. Pero si se piensa en ello, también confundirá la lectura y el mantenimiento de dicho código, por tanto, hay que olvidarse de la precedencia y emplear paréntesis cuando las cosas no estén claras.

4. Mantener cada sección enfocada de manera que el tiempo de lectura y el tiempo entre ejercicios, sea pequeño. Esto no sólo mantiene las mentes de los alumnos más activas cuando se está en un seminario, sino que también proporciona al lector una mayor sensación de estar avanzando.
5. Proporcionar al alumno una base sólida de modo que pueda comprender los temas lo suficientemente bien como para que desee acudir a cursos o libros más avanzados.

Enseñar con este libro

La edición original de este libro ha evolucionado a partir de un seminario de una semana que era, cuando Java se encontraba en su infancia, suficiente tiempo para cubrir el lenguaje. A medida que Java fue creciendo y añadiendo más y más funcionalidades y bibliotecas, yo tenazmente trataba de enseñarlo todo en una semana. En una ocasión, un cliente me sugirió que enseñara “sólo los fundamentos” y al hacerlo descubrí que tratar de memorizar todo en una única semana era angustioso tanto para mí como para las personas que asistían al seminario. Java ya no era un lenguaje “simple” que se podía aprender en una semana.

Dicha experiencia me llevó a reorganizar este libro, el cual ahora está diseñado como material de apoyo para un seminario de dos semanas o un curso escolar de dos trimestres. La parte de introducción termina con el Capítulo 12, *Tratamiento de errores mediante excepciones*, aunque también puede complementarla con una introducción a JDBC, Servlets y JSP. Esto proporciona las bases y es el núcleo del CD-ROM *Hands-On Java*. El resto del libro se corresponde con un curso de nivel intermedio y es el material cubierto en el CD-ROM *Intermediate Thinking in Java*. Ambos discos CD ROM pueden adquirirse a través de www.MindView.net.

Contacte con Prentice-Hall en www.prenhallprofessional.com para obtener más información acerca del material para el profesor relacionado con este libro.

Documentación del JDK en HTML

El lenguaje Java y las bibliotecas de Sun Microsystems (descarga gratuita en <http://java.sun.com>) se suministran con documentación en formato electrónico, que se puede leer con un explorador web. Muchos de los libros publicados sobre Java proporcionan esta documentación. Por tanto, o ya se tiene o puede descargarse y, a menos que sea necesario, en este libro no se incluye dicha documentación, porque normalmente es mucho más rápido encontrar las descripciones de las clases en el explorador web que buscarlas en un libro (y probablemente la documentación en línea estará más actualizada). Basta con que utilice la referencia “JDK documentation”. En este libro se proporcionan descripciones adicionales de las clases sólo cuando es necesario complementar dicha documentación, con el fin de que se pueda comprender un determinado ejemplo.

Ejercicios

He descubierto que durante las clases los ejercicios sencillos son excepcionalmente útiles para que el alumno termine de comprender el tema, por lo que he incluido al final de cada capítulo una serie de ejercicios.

La mayor parte de los ejercicios son bastante sencillos y están diseñados para que se puedan realizar durante un tiempo razonable de la clase, mientras el profesor observa los progresos, asegurándose de que los estudiantes aprenden el tema. Algunos son algo más complejos, pero ninguno presenta un reto inalcanzable.

Las soluciones a los ejercicios seleccionados se pueden encontrar en el documento electrónico *The Thinking in Java Annotated Solution Guide*, que se puede adquirir en www.MindView.net.

Fundamentos para Java

Otra ventaja que presenta esta edición es el seminario multimedia gratuito que puede descargarse en la dirección www.MindView.net. Se trata del seminario *Thinking in C*, el cual proporciona una introducción a los operadores, funciones

y la sintaxis de C en la que se basa la sintaxis de Java. En las ediciones anteriores del libro se encontraba en el CD *Foundations for Java* que se proporcionaba junto con el libro, pero ahora este seminario puede descargarse gratuitamente.

Originalmente, encargué a Chuck Allison que creara *Thinking in C* como un producto autónomo, pero decidí incluirlo en la segunda edición de *Thinking in C++* y en la segunda y tercera ediciones de *Thinking in Java*, por la experiencia de haber estado con personas que llegan a los seminarios sin tener una adecuada formación en la sintaxis básica de C. El razonamiento suele ser: "Soy un programador inteligente y no quiero aprender C, sino C++ o Java, por tanto, me salto el C y paso directamente a ver el C++/Java". Después de asistir al seminario, lentamente todo el mundo se da cuenta de que el prerequisito de conocer la sintaxis de C tiene sus buenas razones de ser.

Las tecnologías han cambiado y han permitido rehacer *Thinking in C* como una presentación Flash descargable en lugar de tener que proporcionarlo en CD. Al proporcionar este seminario en línea, puedo garantizar que todo el mundo pueda comenzar con una adecuada preparación.

El seminario *Thinking in C* también permite atraer hacia el libro a una audiencia importante. Incluso aunque los capítulos dedicados a operadores y al control de la ejecución cubren las partes fundamentales de Java que proceden de C, el seminario en línea es una buena introducción y precisa del estudiante menos conocimientos previos sobre programación que este libro.

Código fuente

Todo el código fuente de este libro está disponible gratuitamente y sometido a *copyright*, distribuido como un paquete único, visitando el sitio web www.MindView.net. Para asegurarse de que obtiene la versión más actual, éste es el sitio oficial de distribución del código. Puede distribuir el código en las clases y en cualquier otra situación educativa.

El objetivo principal del *copyright* es asegurar que el código fuente se cite apropiadamente y evitar así que otros lo publiquen sin permiso. No obstante, mientras se cite la fuente, no constituye ningún problema en la mayoría de los medios que se empleen los ejemplos del libro.

En cada archivo de código fuente se encontrará una referencia a la siguiente nota de *copyright*:

```
//:! Copyright.txt
This computer source code is Copyright ©2006 MindView, Inc.
All Rights Reserved.
```

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "Thinking in
Java" is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941
Wayne@MindView.net

4. The Source Code and documentation are copyrighted by
MindView, Inc. The Source code is provided without express

or implied warranty of any kind, including any implied warranty of merchantability, fitness for a particular purpose or non-infringement. MindView, Inc. does not warrant that the operation of any program that includes the Source Code will be uninterrupted or error-free. MindView, Inc. makes no representation about the suitability of the Source Code or of any software that includes the Source Code for any purpose. The entire risk as to the quality and performance of any program that includes the Source Code is with the user of the Source Code. The user understands that the Source Code was developed for research and instructional purposes and is advised not to rely exclusively for any reason on the Source Code or any program that includes the Source Code. Should the Source Code or any resulting software prove defective, the user assumes the cost of all necessary servicing, repair, or correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a Web site which is the sole distribution point for electronic copies of the Source Code, <http://www.MindView.net> (and official mirror sites), where it is freely available under the terms stated above.

If you think you've found an error in the Source Code, please submit a correction using the feedback system that you will find at <http://www.MindView.net>.

///:-

Puede utilizar el código en sus proyectos y en la clase (incluyendo su material de presentaciones) siempre y cuando se mantenga la nota de *copyright* en cada uno de los archivos fuente.

Estándares de codificación

En el texto del libro, los identificadores (nombres de métodos, variables y clases) se escriben en **negrita**. La mayoría de las palabras clave se escriben en negrita, excepto aquellas palabras clave que se usan con mucha frecuencia y ponerlas en negrita podría volverse tedioso, como en el caso de la palabra "class".

En este libro, he utilizado un estilo de codificación particular para los ejemplos. Este estilo sigue el que emplea Sun en prácticamente todo el código que encontrará en su sitio (véase <http://java.sun.com/docs/codeconv/index.html>), y que parece que soporta la mayoría de los entornos de desarrollo Java. Si ha leído mis otros libros, observará también que el estilo de codificación de Sun coincide con el mío, lo que me complace, ya que yo no tengo nada que ver con la creación del estilo de

Sun. El tema del estilo de formato es bueno para conseguir horas de intenso debate, por lo que no voy a intentar dictar un estilo correcto a través de mis ejemplos; tengo mis propias motivaciones para usar el estilo que uso. Dado que Java es un lenguaje de programación de formato libre, se puede emplear el estilo con el que uno se encuentre a gusto. Una solución para el tema del estilo de codificación consiste en utilizar una herramienta como *Jalopy* (www.triemax.com), la cual me ha ayudado en el desarrollo de este libro a cambiar el formato al que se adaptaba a mí.

Los archivos de código impresos en el libro se han probado con un sistema automatizado, por lo que deberían ejecutarse sin errores de compilación.

Este libro está basado y se ha comprobado con Java SE5/6. Si necesita obtener información sobre versiones anteriores del lenguaje que no se cubren en esta edición, las ediciones primera y tercera del mismo pueden descargarse gratuitamente en www.MindView.net.

Errores

No importa cuantas herramientas utilice un escritor para detectar los errores, algunos quedan ahí y a menudo son lo que primero ve el lector. Si descubre cualquier cosa que piensa que es un error, por favor utilice el vínculo que encontrará para este libro en www.MindView.net y envíeme el error junto con la corrección que usted crea. Cualquier ayuda siempre es bienvenida.

Introducción a los objetos

1

“Analizamos la Naturaleza, la organizamos en conceptos y vamos asignando significados a medida que lo hacemos, fundamentalmente porque participamos en un acuerdo tácito suscrito por toda nuestra comunidad de hablantes y que está codificado en los propios patrones de nuestro idioma... nos resulta imposible hablar si no utilizamos la organización y clasificación de los datos decretadas por ese acuerdo”.
Benjamin Lee Whorf (1897-1941)

La génesis de la revolución de las computadoras se hallaba en una máquina. La génesis de nuestros lenguajes de programación tiende entonces a parecerse a dicha máquina.

Pero las computadoras, más que máquinas, pueden considerarse como herramientas que permiten ampliar la mente (“bicicletas para la mente”, como se enorgullece en decir Steve Jobs), además de un medio de expresión diferente. Como resultado, las herramientas empiezan a parecerse menos a máquinas y más a partes de nuestras mentes, al igual que ocurre con otras formas de expresión como la escritura, la pintura, la escultura, la animación y la realización de películas. La programación orientada a objetos (POO) es parte de este movimiento dirigido al uso de las computadoras como un medio de expresión.

Este capítulo presenta los conceptos básicos de la programación orientada a objetos, incluyendo una introducción a los métodos de desarrollo. Este capítulo, y este libro, supone que el lector tiene cierta experiencia en programación, aunque no necesariamente en C. Si cree que necesita una mayor preparación antes de abordar este libro, debería trabajar con el seminario multimedia sobre C, *Thinking in C*, que puede descargarse en www.MindView.net.

Este capítulo contiene material de carácter general y supplementario. Muchas personas pueden no sentirse cómodas si se enfrentan a la programación orientada a objetos sin obtener primero una visión general. Por tanto, aquí se presentan muchos conceptos que proporcionarán una sólida introducción a la POO. Sin embargo, otras personas pueden no necesitar tener una visión general hasta haber visto algunos de los mecanismos primero, estas personas suelen perderse si no se les ofrece algo de código que puedan manipular. Si usted forma parte de este último grupo, estará ansioso por ver las especificidades del lenguaje, por lo que puede saltarse este capítulo, esto no le impedirá aprender a escribir programas ni conocer el lenguaje. Sin embargo, podrá volver aquí cuando lo necesite para completar sus conocimientos, con el fin de comprender por qué son importantes los objetos y cómo puede diseñarse con ellos.

El progreso de la abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede argumentarse que la complejidad de los problemas que sea capaz de resolver está directamente relacionada con el tipo (clase) y la calidad de las abstracciones, entendiendo por “clase”, “¿qué es lo que se va a abstraer?”. El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados “imperativos” que le siguieron (como FORTRAN, BASIC y C) fueron abstracciones del lenguaje ensamblador. Estos lenguajes constituyen grandes mejoras sobre el lenguaje ensamblador, pero su principal abstracción requiere que se piense en términos de la estructura de la computadora en lugar de en la estructura del problema que se está intentado resolver. El programador debe establecer la asociación entre el modelo de la máquina (en el “espacio de la solución”, que es donde se va a implementar dicha solución, como puede ser una computadora) y el modelo

del problema que es lo que realmente se quiere resolver (en el “espacio del problema”, que es el lugar donde existe el problema, como por ejemplo en un negocio). El esfuerzo que se requiere para establecer esta correspondencia y el hecho de que sea extrínseco al lenguaje de programación, da lugar a programas que son difíciles de escribir y caros de mantener, además del efecto colateral de toda una industria de “métodos de programación”.

La alternativa a modelar la máquina es modelar el problema que se está intentado solucionar. Los primeros lenguajes como LISP y APL eligen vistas parciales del mundo (“todos los problemas pueden reducirse a listas” o “todos los problemas son algorítmicos”, respectivamente). Prolog convierte todos los problemas en cadenas de decisión. Los lenguajes se han creado para programar basándose en restricciones y para programar de forma exclusiva manipulando símbolos gráficos (aunque se demostró que este caso era demasiado restrictivo). Cada uno de estos métodos puede ser una buena solución para resolver la clase de problema concreto para el que están diseñados, pero cuando se aplican en otro dominio resultan inadecuados.

El enfoque orientado a objetos trata de ir un paso más allá proporcionando herramientas al programador para representar los elementos en el espacio del problema. Esta representación es tan general que el programador no está restringido a ningún tipo de problema en particular. Se hace referencia a los elementos en el espacio del problema denominando “objetos” a sus representaciones en el espacio de la solución (también se necesitarán otros objetos que no tendrán análogos en el espacio del problema). La idea es que el programa pueda adaptarse por sí sólo a la jerga del problema añadiendo nuevos tipos de objetos, de modo que cuando se lea el código que describe la solución, se estén leyendo palabras que también expresen el problema. Ésta es una abstracción del lenguaje más flexible y potente que cualquiera de las que se hayan hecho anteriormente¹. Por tanto, la programación orientada a objetos permite describir el problema en términos del problema en lugar de en términos de la computadora en la que se ejecutará la solución. Pero aún existe una conexión con la computadora, ya que cada objeto es similar a una pequeña computadora (tiene un estado y dispone de operaciones que el programador puede pedirle que realice). Sin embargo, esto no quiere decir que nos encontremos ante una mala analogía de los objetos del mundo real, que tienen características y comportamientos.

Alan Kay resumió las cinco características básicas del Smalltalk, el primer lenguaje orientado a objetos que tuvo éxito y uno de los lenguajes en los que se basa Java. Estas características representan un enfoque puro de la programación orientada a objetos.

- 1. Todo es un objeto.** Piense en un objeto como en una variable: almacena datos, permite que se le “planteen solicitudes”, pidiéndole que realice operaciones sobre sí mismo. En teoría, puede tomarse cualquier componente conceptual del problema que se está intentado resolver (perros, edificios, servicios, etc.) y representarse como un objeto del programa.
- 2. Un programa es un montón de objetos que se dicen entre sí lo que tienen que hacer enviándose mensajes.** Para hacer una solicitud a un objeto, hay que enviar un mensaje a dicho objeto. Más concretamente, puede pensar en que un mensaje es una solicitud para llamar a un método que pertenece a un determinado objeto.
- 3. Cada objeto tiene su propia memoria formada por otros objetos.** Dicho de otra manera, puede crear una nueva clase de objeto definiendo un paquete que contenga objetos existentes. Por tanto, se puede incrementar la complejidad de un programa ocultándola tras la simplicidad de los objetos.
- 4. Todo objeto tiene un tipo asociado.** Como se dice popularmente, cada objeto es una *instancia* de una *clase*, siendo “clase” sinónimo de “tipo”. La característica distintiva más importante de una clase es “el conjunto de mensajes que se le pueden enviar”.
- 5. Todos los objetos de un tipo particular pueden recibir los mismos mensajes.** Como veremos más adelante, esta afirmación es realmente importante. Puesto que un objeto de tipo “círculo” también es un objeto de tipo “forma”, puede garantizarse que un círculo aceptará los mensajes de forma. Esto quiere decir que se puede escribir código para comunicarse con objetos de tipo forma y controlar automáticamente cualquier cosa que se ajuste a la descripción de una forma. Esta capacidad de *suplantación* es uno de los conceptos más importantes de la programación orientada a objetos.

Booch ofrece una descripción aún más sucinta de objeto:

¹Algunos diseñadores de lenguajes han decidido que la programación orientada a objetos por sí misma no es adecuada para resolver fácilmente todos los problemas de la programación, y recomiendan combinar varios métodos en lenguajes de programación *multiparadigma*. Consulte *Multiparadigm Programming in Leda* de Timothy Budd (Addison-Wesley, 1995).

Un objeto tiene estado, comportamiento e identidad.

Esto significa que un objeto puede tener datos internos (lo que le proporciona el estado), métodos (para proporcionar un comportamiento) y que cada objeto puede ser diferenciado de forma unívoca de cualquier otro objeto; es decir, cada objeto tiene una dirección de memoria exclusiva.²

Todo objeto tiene una interfaz

Aristóteles fue probablemente el primero en estudiar cuidadosamente el concepto de *tipo*; hablaba de “la clase de peces y de la clase de pájaros”. La idea de que todos los objetos, aún siendo únicos, son también parte de una clase de objetos que tienen características y comportamientos comunes ya se empleó en el primer lenguaje orientado a objetos, el Simula-67, que ya usaba la palabra clave fundamental **class**, que permite introducir un nuevo tipo en un programa.

Simula, como su nombre implica, se creó para desarrollar simulaciones como la clásica del “problema del cajero de un banco”. En esta simulación, se tienen muchos cajeros, clientes, cuentas, transacciones y unidades monetarias, muchísimos “objetos”. Los objetos, que son idénticos excepto por su estado durante la ejecución de un programa, se agrupan en “clases de objetos”, que es de donde procede la palabra clave **class**. La creación de tipos de datos abstractos (clases) es un concepto fundamental en la programación orientada a objetos. Los tipos de datos abstractos funcionan casi exactamente como tipos predefinidos: pueden crearse variables de un tipo (llamadas *objetos* u *instancias* en la jerga de la POO) y manipular dichas variables (mediante el *envío de mensajes* o *solicitudes*, se envía un mensaje y el objeto sabe lo que tiene que hacer con él). Los miembros (elementos) de cada clase comparten algunos rasgos comunes. Cada cuenta tiene asociado un saldo, cada cajero puede aceptar un depósito, etc. Además, cada miembro tiene su propio estado. Cada cuenta tiene un saldo diferente y cada cajero tiene un nombre. Por tanto, los cajeros, clientes, cuentas, transacciones, etc., pueden representarse mediante una entidad unívoca en el programa informático. Esta entidad es el objeto y cada objeto pertenece a una determinada clase que define sus características y comportamientos.

Por tanto, aunque en la programación orientada a objetos lo que realmente se hace es crear nuevos tipos de datos, en la práctica, todos los lenguajes de programación orientada a objetos utilizan la palabra clave “**class**”. Cuando vea la palabra “**type**” (tipo) piense en “**class**” (clase), y viceversa.³

Dado que una clase describe un conjunto de objetos que tienen características (elementos de datos) y comportamientos (funcionalidad) idénticos, una clase realmente es un tipo de datos porque, por ejemplo, un número en coma flotante también tiene un conjunto de características y comportamientos. La diferencia está en que el programador define un clase para adaptar un problema en lugar de forzar el uso de un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Se puede ampliar el lenguaje de programación añadiendo nuevos tipos de datos específicos que se adapten a sus necesidades. El sistema de programación admite las nuevas clases y proporciona a todas ellas las comprobaciones de tipo que proporciona a los tipos predefinidos.

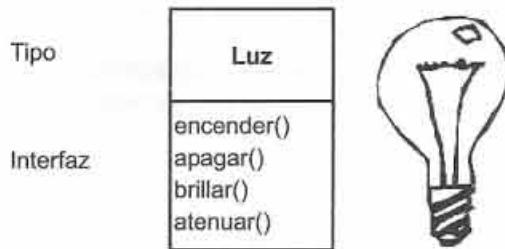
El enfoque orientado a objetos no está limitado a la creación de simulaciones. Se esté o no de acuerdo en que cualquier programa es una simulación del sistema que se está diseñando, el uso de las técnicas de la POO puede reducir fácilmente un gran conjunto de problemas a una sencilla solución.

Una vez que se ha definido una clase, se pueden crear tantos objetos de dicha clase como se desee y dichos objetos pueden manipularse como si fueran los elementos del problema que se está intentado resolver. Realmente, uno de los retos de la programación orientada a objetos es crear una correspondencia uno-a-uno entre los elementos del espacio del problema y los objetos del espacio de la solución.

Pero, ¿cómo se consigue que un objeto haga un trabajo útil para el programador? Debe haber una forma de hacer una solicitud al objeto para que haga algo, como por ejemplo, completar una transacción, dibujar algo en pantalla o encender un interruptor. Además, cada objeto sólo puede satisfacer ciertas solicitudes. Las solicitudes que se pueden hacer a un objeto se definen mediante su *interfaz* y es el tipo lo que determina la interfaz. Veamos un ejemplo con la representación de una bombilla:

² Realmente, esto es poco restrictivo, ya que pueden existir objetos en diferentes máquinas y espacios de direcciones, y también se pueden almacenar en disco. En estos casos, debe determinarse la identidad del objeto mediante alguna otra cosa que la dirección de memoria.

³ Algunas personas hacen una distinción, estableciendo que el tipo determina la interfaz mientras que la clase es una implementación concreta de dicha interfaz.



```
Luz lz = new Luz();
lz.encender();
```

La interfaz determina las solicitudes que se pueden hacer a un determinado objeto, por lo que debe existir un código en alguna parte que satisfaga dicha solicitud. Esto, junto con los datos ocultos, definen lo que denomina la *implementación*. Desde el punto de vista de la programación procedimental, esto no es complicado. Un tipo tiene un método asociado con cada posible solicitud; cuando se hace una determinada solicitud a un objeto, se llama a dicho método. Este proceso se resume diciendo que el programador “envía un mensaje” (hace una solicitud) a un objeto y el objeto sabe lo que tiene que hacer con ese mensaje (ejecuta el código).

En este ejemplo, el nombre del tipo/clase es **Luz**, el nombre de este objeto concreto **Luz** es **lz** y las solicitudes que se pueden hacer a un objeto **Luz** son encender, apagar, brillar o atenuar. Se ha creado un objeto **Luz** definiendo una “referencia” (**lz**) para dicho objeto e invocando **new** para hacer una solicitud a un nuevo objeto de dicho tipo. Para enviar un mensaje al objeto, se define el nombre del objeto y se relaciona con la solicitud del mensaje mediante un punto. Desde el punto de vista del usuario de una clase predefinida, esto es el no va más de la programación con objetos.

El diagrama anterior sigue el formato del lenguaje UML (*Unified Modeling Language*, lenguaje de modelado unificado). Cada clase se representa mediante un recuadro escribiendo el nombre del tipo en la parte superior, los *miembros de datos* en la zona intermedia y los *métodos* (las funciones de dicho objeto que reciben cualquier mensaje que el programador envíe a dicho objeto) en la parte inferior. A menudo, en estos diagramas sólo se muestran el nombre de la clase y los métodos públicos, no incluyéndose la zona intermedia, como en este caso. Si sólo se está interesado en el nombre de la clase, tampoco es necesario incluir la parte inferior.

Un objeto proporciona servicios

Cuando se está intentando desarrollar o comprender el diseño de un programa, una de las mejores formas de pensar en los objetos es como si fueran “proveedores de servicios”. El programa proporciona servicios al usuario y esto se conseguirá utilizando los servicios que ofrecen otros objetos. El objetivo es producir (o incluso mejor, localizar en las bibliotecas de código existentes) un conjunto de objetos que facilite los servicios idóneos para resolver el problema.

Una manera de empezar a hacer esto es preguntándose: “Si pudiera sacarlos de un sombrero mágico, ¿qué objetos resolvieran el problema de la forma más simple?”. Por ejemplo, suponga que quiere escribir un programa de contabilidad. Puede pensar en algunos objetos que contengan pantallas predefinidas para la introducción de los datos contables, otro conjunto de objetos que realicen los cálculos necesarios y un objeto que controle la impresión de los cheques y las facturas en toda clase de impresoras. Es posible que algunos de estos objetos ya existan, pero ¿cómo deben ser los que no existen? ¿Qué servicios deberían proporcionar *esos* objetos y qué objetos necesitarían para cumplir con sus obligaciones? Si se hace este planteamiento, llegará a un punto donde puede decir: “Este objeto es lo suficientemente sencillo como para escribirlo yo mismo” o “Estoy seguro de que este objeto ya tiene que existir”. Ésta es una forma razonable de descomponer un problema en un conjunto de objetos.

Pensar en un objeto como en un proveedor de servicios tiene una ventaja adicional: ayuda a mejorar la cohesión del objeto. Una *alta cohesión* es una cualidad fundamental del diseño software, lo que significa que los diferentes aspectos de un componente de software (tal como un objeto, aunque también podría aplicarse a un método o a una biblioteca de objetos) deben “ajustar bien entre sí”. Un problema que suelen tener los programadores cuando diseñan objetos es el de asignar demasiada funcionalidad al objeto. Por ejemplo, en el módulo para imprimir cheques, puede decidir que es necesario un objeto que sepa todo sobre cómo dar formato e imprimir. Probablemente, descubrirá que esto es demasiado para un solo objeto y que hay que emplear tres o más objetos. Un objeto puede ser un catálogo de todos los posibles diseños de cheque, al cual se le

puede consultar para obtener información sobre cómo imprimir un cheque. Otro objeto o conjunto de objetos puede ser una interfaz de impresión genérica que sepa todo sobre las diferentes clases de impresoras (pero nada sobre contabilidad; por ello, probablemente es un candidato para ser comprado en lugar de escribirlo uno mismo). Y un tercer objeto podría utilizar los servicios de los otros dos para llevar a cabo su tarea. Por tanto, cada objeto tiene un conjunto cohesivo de servicios que ofrecer. En un buen diseño orientado a objetos, cada objeto hace una cosa bien sin intentar hacer demasiadas cosas. Esto además de permitir descubrir objetos que pueden adquirirse (el objeto interfaz de impresora), también genera nuevos objetos que se reutilizarán en otros diseños.

Tratar los objetos como proveedores de servicios es una herramienta que simplifica mucho. No sólo es útil durante el proceso de diseño, sino también cuando alguien intenta comprender su propio código o reutilizar un objeto. Si se es capaz de ver el valor del objeto basándose en el servicio que proporciona, será mucho más fácil adaptarlo al diseño.

La implementación oculta

Resulta útil descomponer el campo de juego en *creadores de clases* (aquellos que crean nuevos tipos de datos) y en *programadores de clientes*⁴ (los consumidores de clases que emplean los tipos de datos en sus aplicaciones). El objetivo del programador cliente es recopilar una caja de herramientas completa de clases que usar para el desarrollo rápido de aplicaciones. El objetivo del creador de clases es construir una clase que exponga al programador cliente sólo lo que es necesario y mantenga todo lo demás oculto. ¿Por qué? Porque si está oculto, el programador cliente no puede acceder a ello, lo que significa que el creador de clases puede cambiar la parte oculta a voluntad sin preocuparse del impacto que la modificación pueda implicar. Normalmente, la parte oculta representa las vulnerabilidades internas de un objeto que un programador cliente poco cuidadoso o poco formado podría corromper fácilmente, por lo que ocultar la implementación reduce los errores en los programas.

En cualquier relación es importante tener límites que todas las partes implicadas tengan que respetar. Cuando se crea una biblioteca, se establece una relación con el programador de clientes, que también es un programador, pero que debe construir su aplicación utilizando su biblioteca, posiblemente con el fin de obtener una biblioteca más grande. Si todos los miembros de una clase están disponibles para cualquiera, entonces el programador de clientes puede hacer cualquier cosa con dicha clase y no hay forma de imponer reglas. Incluso cuando prefiera que el programador de clientes no manipule directamente algunos de los miembros de su clase, sin control de acceso no hay manera de impedirlo. Todo está a la vista del mundo.

Por tanto, la primera razón que justifica el control de acceso es mantener las manos de los programadores cliente apartadas de las partes que son necesarias para la operación interna de los tipos de datos, pero no de la parte correspondiente a la interfaz que los usuarios necesitan para resolver sus problemas concretos. Realmente, es un servicio para los programadores de clientes porque pueden ver fácilmente lo que es importante para ellos y lo que pueden ignorar.

La segunda razón del control de acceso es permitir al diseñador de bibliotecas cambiar el funcionamiento interno de la clase sin preocuparse de cómo afectará al programador de clientes. Por ejemplo, desea implementar una clase particular de una forma sencilla para facilitar el desarrollo y más tarde descubre que tiene que volver a escribirlo para que se ejecute más rápidamente. Si la interfaz y la implementación están claramente separadas y protegidas, podrá hacer esto fácilmente.

Java emplea tres palabras clave explícitamente para definir los límites en una clase: **public**, **private** y **protected**. Estos *modificadores de acceso* determinan quién puede usar las definiciones del modo siguiente: **public** indica que el elemento que le sigue está disponible para todo el mundo. Por otro lado, la palabra clave **private**, quiere decir que nadie puede acceder a dicho elemento excepto usted, el creador del tipo, dentro de los métodos de dicho tipo. **private** es un muro de ladillos entre usted y el programador de clientes. Si alguien intenta acceder a un miembro **private** obtendrá un error en tiempo de compilación. La palabra clave **protected** actúa como **private**, con la excepción de que una clase heredada tiene acceso a los miembros protegidos (**protected**), pero no a los privados (**private**). Veremos los temas sobre herencia enseguida.

Java también tiene un acceso “predeterminado”, que se emplea cuando no se aplica uno de los modificadores anteriores. Normalmente, esto se denomina *acceso de paquete*, ya que las clases pueden acceder a los miembros de otras clases que pertenecen al mismo *paquete* (componente de biblioteca), aunque fuera del paquete dichos miembros aparecen como privados (**private**).

⁴ Término acuñado por mi amigo Scott Meyers.

Reutilización de la implementación

Una vez que se ha creado y probado una clase, idealmente debería representar una unidad de código útil. Pero esta reutilización no siempre es tan fácil de conseguir como era de esperar; se necesita experiencia y perspicacia para generar un diseño de un objeto reutilizable. Pero, una vez que se dispone de tal diseño, parece implorar ser reutilizado. La reutilización de código es una de las grandes ventajas que proporcionan los lenguajes de programación orientada a objetos.

La forma más sencilla de reutilizar una clase consiste simplemente en emplear directamente un objeto de dicha clase, aunque también se puede colocar un objeto de dicha clase dentro de una clase nueva. Esto es lo que se denomina “crear un objeto miembro”. La nueva clase puede estar formada por cualquier número y tipo de otros objetos en cualquier combinación necesaria para conseguir la funcionalidad deseada en dicha nueva clase. Definir una nueva clase a partir de clases existentes se denomina *composición* (si la composición se realiza de forma dinámica, se llama *agregación*). A menudo se hace referencia a la composición como una relación “tiene un”, como en “un coche tiene un motor”.



Este diagrama UML indica la composición mediante un rombo relleno, que establece que hay un coche. Normalmente, yo utilizo una forma más sencilla: sólo una línea, sin el rombo, para indicar una asociación.⁵

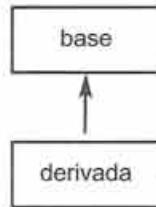
La composición conlleva una gran flexibilidad. Los objetos miembro de la nueva clase normalmente son privados, lo que les hace inaccesibles a los programadores de clientes que están usando la clase. Esto le permite cambiar dichos miembros sin disturbar al código cliente existente. Los objetos miembro también se pueden modificar en tiempo de ejecución, con el fin de cambiar dinámicamente el comportamiento del programa. La herencia, que se describe a continuación, no proporciona esta flexibilidad, ya que el compilador tiene que aplicar las restricciones en tiempo de compilación a las clases creadas por herencia.

Dado que la herencia es tan importante en la programación orientada a objetos, casi siempre se enfatiza mucho su uso, de manera que los programadores novatos pueden llegar a pensar que hay que emplearla en todas partes. Esto puede dar lugar a que se hagan diseños demasiado complejos y complicados. En lugar de esto, en primer lugar, cuando se van a crear nuevas clases debe considerarse la composición, ya que es más simple y flexible. Si aplica este método, sus diseños serán más inteligentes. Una vez que haya adquirido algo de experiencia, será razonablemente obvio cuándo se necesita emplear la herencia.

Herencia

Por sí misma, la idea de objeto es una buena herramienta. Permite unir datos y funcionalidad por *concepto*, lo que permite representar la idea del problema-espacio apropiada en lugar de forzar el uso de los idiomas de la máquina subyacente. Estos conceptos se expresan como unidades fundamentales en el lenguaje de programación utilizando la palabra clave *class*.

Sin embargo, es una pena abordar todo el problema para crear una clase y luego verse forzado a crear una clase nueva que podría tener una funcionalidad similar. Es mejor, si se puede, tomar la clase existente, clonarla y luego añadir o modificar lo que sea necesario al clon. Esto es lo que se logra con la *herencia*, con la excepción de que la clase original (llamada *clase base*, *superclase* o *clase padre*) se modifica, el clon “modificado” (denominado *clase derivada*, *clase heredada*, *subclase* o *clase hija*) también refleja los cambios.



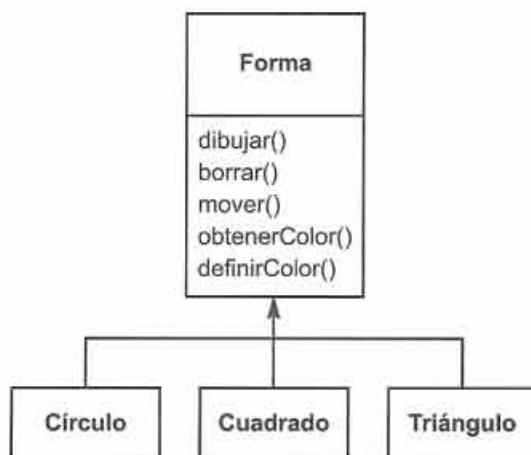
⁵ Normalmente, es suficiente grado de detalle para la mayoría de los diagramas y no es necesario especificar si se está usando una agregación o una composición.

La flecha de este diagrama UML apunta de la clase derivada a la clase base. Como veremos, puede haber más de una clase derivada.

Un tipo hace más que describir las restricciones definidas sobre un conjunto de objetos; también tiene una relación con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que el otro y también es posible que pueda manejar más mensajes (o manejarlos de forma diferente). La herencia expresa esta similitud entre tipos utilizando el concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos que los tipos derivados de él comparten. Es recomendable crear un tipo base para representar el núcleo de las ideas acerca de algunos de los objetos del sistema. A partir de ese tipo base, pueden deducirse otros tipos para expresar las diferentes formas de implementar ese núcleo.

Por ejemplo, una máquina para el reciclado de basura clasifica los desperdicios. El tipo base es “basura” y cada desperdicio tiene un peso, un valor, etc., y puede fragmentarse, mezclarse o descomponerse. A partir de esto, se derivan más tipos específicos de basura que pueden tener características adicionales (una botella tendrá un color) o comportamientos (el aluminio puede modelarse, el acero puede tener propiedades magnéticas). Además, algunos comportamientos pueden ser diferentes (el valor del papel depende de su tipo y condición). Utilizando la herencia, puede construir una jerarquía de tipos que exprese el problema que está intentando resolver en términos de sus tipos.

Un segundo ejemplo es el clásico ejemplo de la forma, quizás usado en los sistemas de diseño asistido por computadora o en la simulación de juegos. El tipo base es “forma” y cada forma tiene un tamaño, un color, una posición, etc. Cada forma puede dibujarse, borrarse, desplazarse, colorearse, etc. A partir de esto, se derivan (heredan) los tipos específicos de formas (círculo, cuadrado, triángulo, etc.), cada una con sus propias características adicionales y comportamientos. Por ejemplo, ciertas formas podrán voltearse. Algunos comportamientos pueden ser diferentes, como por ejemplo cuando se quiere calcular su área. La jerarquía de tipos engloba tanto las similitudes con las diferencias entre las formas.



Representar la solución en los mismos términos que el problema es muy útil, porque no se necesitan muchos modelos intermedios para pasar de una descripción del problema a una descripción de la solución. Con objetos, la jerarquía de tipos es el modelo principal, porque se puede pasar directamente de la descripción del sistema en el mundo real a la descripción del sistema mediante código. A pesar de esto, una de las dificultades que suelen tener los programadores con el diseño orientado a objetos es que es demasiado sencillo ir del principio hasta el final. Una mente formada para ver soluciones complejas puede, inicialmente, verse desconcertada por esta simplicidad.

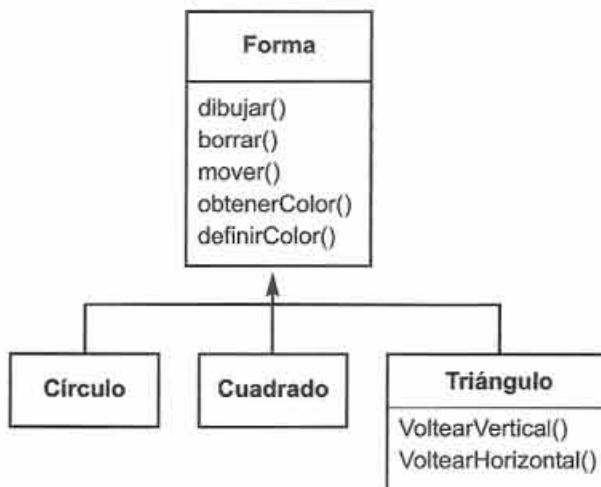
Cuando se hereda de un tipo existente, se crea un tipo nuevo. Este tipo nuevo no sólo contiene todos los miembros del tipo existente (aunque los privados están ocultos y son inaccesibles), sino lo que es más importante, duplica la interfaz de la clase base; es decir, todos los mensajes que se pueden enviar a los objetos de la clase base también se pueden enviar a los objetos de la clase derivada. Dado que conocemos el tipo de una clase por los mensajes que se le pueden enviar, esto quiere decir que la clase derivada *es del mismo tipo que la clase base*. En el ejemplo anterior, “un círculo es una forma”. Esta equivalencia de tipos a través de la herencia es uno de los caminos fundamentales para comprender el significado de la programación orientada a objetos.

Puesto que la clase base y la clase derivada tienen la misma interfaz, debe existir alguna implementación que vaya junto con dicha interfaz. Es decir, debe disponerse de algún código que se ejecute cuando un objeto recibe un mensaje concreto. Si

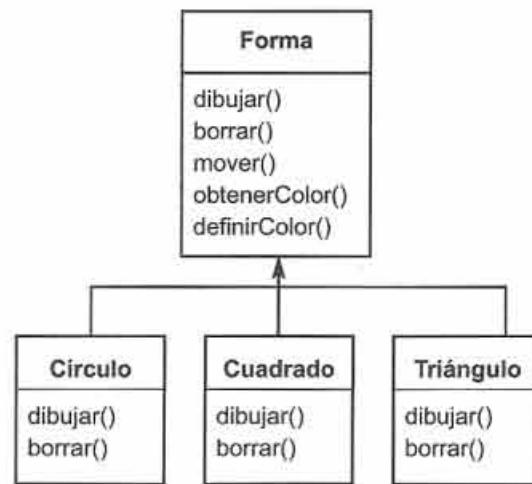
8 Piensa en Java

simplemente hereda una clase y no hace nada más, los métodos de la interfaz de la clase base pasan tal cual a la clase derivada, lo que significa que los objetos de la clase derivada no sólo tienen el mismo tipo sino que también tienen el mismo comportamiento, lo que no es especialmente interesante.

Hay dos formas de diferenciar la nueva clase derivada de la clase base original. La primera es bastante directa: simplemente, se añaden métodos nuevos a la clase derivada. Estos métodos nuevos no forman parte de la interfaz de la clase base, lo que significa que ésta simplemente no hacía todo lo que se necesitaba y se le han añadido más métodos. Este sencillo y primitivo uso de la herencia es, en ocasiones, la solución perfecta del problema que se tiene entre manos. Sin embargo, debe considerarse siempre la posibilidad de que la clase base pueda también necesitar esos métodos adicionales. Este proceso de descubrimiento e iteración en un diseño tiene lugar habitualmente en la programación orientada a objetos.



Aunque en ocasiones la herencia puede implicar (especialmente en Java, donde la palabra clave para herencia es **extends**) que se van a añadir métodos nuevos a la interfaz, no tiene que ser así necesariamente. La segunda y más importante forma de diferenciar la nueva clase es *cambiando* el comportamiento de un método existente de la clase base. Esto es lo que se denomina *sustitución* del método.



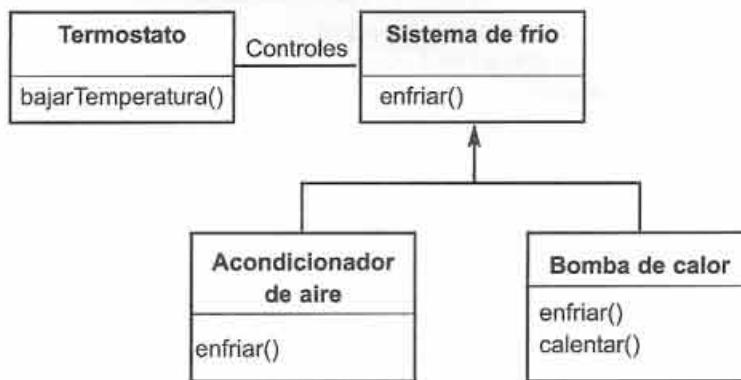
Para sustituir un método, basta con crear una nueva definición para el mismo en la clase derivada. Es decir, se usa el mismo método de interfaz, pero se quiere que haga algo diferente en el tipo nuevo.

Relaciones es-un y es-como-un

Es habitual que la herencia suscite un pequeño debate: ¿debe la herencia sustituir *sólo* los métodos de la clase base (y no añadir métodos nuevos que no existen en la clase base)? Esto significaría que la clase derivada es *exactamente* del mismo

tipo que la clase base, ya que tiene exactamente la misma interfaz. Como resultado, es posible sustituir de forma exacta un objeto de la clase derivada por uno de la clase base. Se podría pensar que esto es una *sustitución pura* y a menudo se denomina *principio de sustitución*. En cierto sentido, ésta es la forma ideal de tratar la herencia. A menudo, en este caso, la relación entre la clase base y las clases derivadas se dice que es una relación *es-un*, porque podemos decir, “un círculo *es una* forma”. Una manera de probar la herencia es determinando si se puede aplicar la relación *es-un* entre las clases y tiene sentido.

A veces es necesario añadir nuevos elementos de interfaz a un tipo derivado, ampliando la interfaz. El tipo nuevo puede todavía ser sustituido por el tipo base, pero la sustitución no es perfecta porque el tipo base no puede acceder a los métodos nuevos. Esto se describe como una relación *es-como-un*. El tipo nuevo tiene la interfaz del tipo antiguo pero también contiene otros métodos, por lo que realmente no se puede decir que sean exactos. Por ejemplo, considere un sistema de aire acondicionado. Suponga que su domicilio está equipado con todo el cableado para controlar el equipo, es decir, dispone de una interfaz que le permite controlar el aire frío. Imagine que el aparato de aire acondicionado se estropea y lo reemplaza por una bomba de calor, que puede generar tanto aire caliente como frío. La bomba de calor *es-como-un* aparato de aire acondicionado, pero tiene más funciones. Debido a que el sistema de control de su casa está diseñado sólo para controlar el aire frío, está restringido a la comunicación sólo con el sistema de frío del nuevo objeto. La interfaz del nuevo objeto se ha ampliado y el sistema existente sólo conoce la interfaz original.



Por supuesto, una vez que uno ve este diseño, está claro que la clase base “sistema de aire acondicionado” no es general y debería renombrarse como “sistema de control de temperatura” con el fin de poder incluir también el control del aire caliente, en esta situación, está claro que el principio de sustitución funcionará. Sin embargo, este diagrama es un ejemplo de lo que puede ocurrir en el diseño en el mundo real.

Cuando se ve claro que el principio de sustitución (la sustitución pura) es la única forma de poder hacer las cosas, debe aplicarse sin dudar. Sin embargo, habrá veces que no estará tan claro y será mejor añadir métodos nuevos a la interfaz de una clase derivada. La experiencia le proporcionará los conocimientos necesarios para saber qué método emplear en cada caso.

Objetos intercambiables con polimorfismo

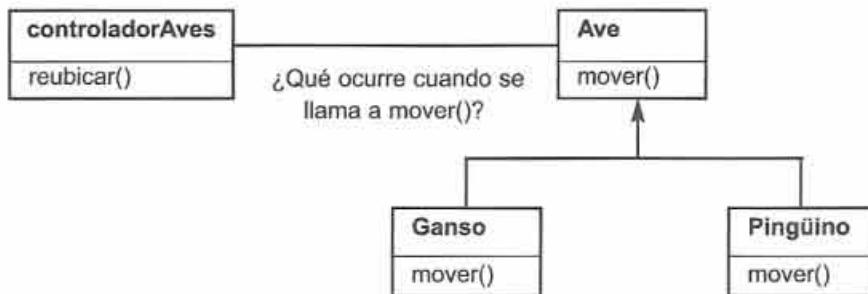
Cuando se trabaja con jerarquías de tipos, a menudo se desea tratar un objeto no como el tipo específico que es, sino como su tipo base. Esto permite escribir código que no dependa de tipos específicos. En el ejemplo de las formas, los métodos manipulan las formas genéricas, independientemente de que se trate de círculos, cuadrados, triángulos o cualquier otra forma que todavía no haya sido definida. Todas las formas pueden dibujarse, borrarse y moverse, por lo que estos métodos simplemente envían un mensaje a un objeto forma, sin preocuparse de cómo se enfrenta el objeto al mensaje.

Tal código no se ve afectado por la adición de tipos nuevos y esta adición de tipos nuevos es la forma más común de ampliar un programa orientado a objetos para manejar situaciones nuevas. Por ejemplo, puede derivar un subtipo nuevo de forma llamado pentágono sin modificar los métodos asociados sólo con las formas genéricas. Esta capacidad de ampliar fácilmente un diseño derivando nuevos subtipos es una de las principales formas de encapsular cambios. Esto mejora enormemente los diseños además de reducir el coste del mantenimiento del software.

Sin embargo, existe un problema cuando se intenta tratar los objetos de tipos derivados como sus tipos base genéricos (círculos como formas, bicicletas como automóviles, cormoranes como aves, etc.). Si un método dice a una forma que se dibuje, o a un automóvil genérico que se ponga en marcha o a un ave que se mueva, el compilador no puede saber en tiempo de

compilación de forma precisa qué parte del código tiene que ejecutar. Éste es el punto clave, cuando se envía el mensaje, el programador no desea saber qué parte del código se va a ejecutar; el método para dibujar se puede aplicar igualmente a un círculo, a un cuadrado o a un triángulo y los objetos ejecutarán el código apropiado dependiendo de su tipo específico.

Si no se sabe qué fragmento de código se ejecutará, entonces se añadirá un subtipo nuevo y el código que se ejecute puede ser diferente sin que sea necesario realizar cambios en el método que lo llama. Por tanto, el compilador no puede saber de forma precisa qué fragmento de código hay que ejecutar y ¿qué hace entonces? Por ejemplo, en el siguiente diagrama, el objeto **controladorAves** sólo funciona con los objetos genéricos **Ave** y no sabe exactamente de qué tipo son. Desde la perspectiva del objeto **controladorAves** esto es adecuado ya que no tiene que escribir código especial para determinar el tipo exacto de **Ave** con el que está trabajando ni el comportamiento de dicha **Ave**. Entonces, ¿cómo es que cuando se invoca al método **mover()** ignorando el tipo específico de **Ave**, se ejecuta el comportamiento correcto (un **Ganso** camina, vuela o nada y un **Pingüino** camina o nada)?



La respuesta es una de las principales novedades de la programación orientada a objetos: el compilador no puede hacer una llamada a función en el sentido tradicional. La llamada a función generada por un compilador no-POO hace lo que se denomina un *acoplamiento temprano*, término que es posible que no haya escuchado antes. Significa que el compilador genera una llamada a un nombre de función específico y el sistema de tiempo de ejecución resuelve esta llamada a la dirección absoluta del código que se va a ejecutar. En la POO, el programa no puede determinar la dirección del código hasta estar en tiempo de ejecución, por lo que se hace necesario algún otro esquema cuando se envía un mensaje a un objeto genérico.

Para resolver el problema, los lenguajes orientados a objetos utilizan el concepto de *acoplamiento tardío*. Cuando se envía un mensaje a un objeto, el código al que se está llamando no se determina hasta el tiempo de ejecución. El compilador no asegura que el método exista, realiza una comprobación de tipos con los argumentos y devuelve un valor, pero no sabe exactamente qué código tiene que ejecutar.

Para realizar el acoplamiento tardío, Java emplea un bit de código especial en lugar de una llamada absoluta. Este código calcula la dirección del cuerpo del método, utilizando la información almacenada en el objeto (este proceso se estudia en detalle en el Capítulo 8, *Polimorfismo*). Por tanto, cada objeto puede comportarse de forma diferente de acuerdo con los contenidos de dicho bit de código especial. Cuando se envía un mensaje a un objeto, realmente el objeto resuelve lo que tiene que hacer con dicho mensaje.

En algunos lenguajes debe establecerse explícitamente que un método tenga la flexibilidad de las propiedades del acoplamiento tardío (C++ utiliza la palabra clave **virtual** para ello). En estos lenguajes, de manera predeterminada, los métodos *no* se acoplan de forma dinámica. En Java, el acoplamiento dinámico es el comportamiento predeterminado y el programador no tiene que añadir ninguna palabra clave adicional para definir el polimorfismo.

Considere el ejemplo de las formas. La familia de clases (todas basadas en la misma interfaz uniforme) se ha mostrado en un diagrama anteriormente en el capítulo. Para demostrar el polimorfismo, queremos escribir un fragmento de código que ignore los detalles específicos del tipo y que sólo sea indicado para la clase base. Dicho código se *desacopla* de la información específica del tipo y por tanto es más sencillo de escribir y de comprender. Y, por ejemplo, si se añade un tipo nuevo como **Hexágono** a través de la herencia, el código que haya escrito funcionará tanto para el nuevo tipo de **Forma** como para los tipos existentes. Por tanto, el programa es *ampliable*.

Si escribe un método en Java (lo que pronto aprenderá a hacer) como el siguiente:

```

void hacerAlgo(Forma forma) {
    borrar.forma();
    // ...
    dibujar.forma();
}
  
```

Este método sirve para cualquier **Forma**, por lo que es independiente del tipo específico de objeto que se esté dibujando y borrando. Si alguna otra parte del programa utiliza el método `hacerAlgo()`:

```
Circulo circulo = new Circulo();
Triangulo triangulo = new Triangulo();
Linea linea = new Linea();
hacerAlgo (circulo);
hacerAlgo (triangulo);
hacerAlgo (linea);
```

Las llamadas a `hacerAlgo ()` funcionarán correctamente, independientemente del tipo exacto del objeto.

De hecho, éste es un buen truco. Considere la línea:

```
hacerAlgo (circulo);
```

Lo que ocurre aquí es que se está pasando un **Círculo** en un método que está esperando una **Forma**. Dado que un **Círculo** es una **Forma**, `hacerAlgo()` puede tratarlo como tal. Es decir, cualquier mensaje que `hacerAlgo()` pueda enviar a **Forma**, un **círculo** puede aceptarlo. Por tanto, actuar así es completamente seguro y lógico.

Llamamos a este proceso de tratar un tipo derivado como si fuera un tipo base *upcasting* (generalización). La palabra significa en inglés “proyección hacia arriba” y refleja la forma en que se dibujan habitualmente los diagramas de herencia, con el tipo base en la parte superior y las clases derivadas abriéndose en abanico hacia abajo, *upcasting* es, por tanto, efectuar una proyección sobre un tipo base, ascendiendo por el diagrama de herencia.



Un programa orientado a objetos siempre contiene alguna generalización, porque es la forma de desvincularse de tener que conocer el tipo exacto con que se trabaja. Veamos el código de `hacerAlgo()`:

```
forma.borrar();
// ...
forma.dibujar();
```

Observe que no se dice, “si eres un **Círculo**, hacer esto, si eres un **Cuadrado**, hacer esto, etc.”. Con este tipo de código lo que se hace es comprobar todos los tipos posibles de **Forma**, lo que resulta lioso y se necesitaría modificar cada vez que se añadiera una nueva clase de **Forma**. En este ejemplo, sólo se dice: “Eres una forma, te puedo `borrar()` y `dibujar()` teniendo en cuenta correctamente los detalles”.

Lo que más impresiona del código del método `hacerAlgo()` es que, de alguna manera se hace lo correcto. Llamar a `dibujar()` para **Círculo** da lugar a que se ejecute un código diferente que cuando se le llama para un **Cuadrado** o una **Linea**, pero cuando el mensaje `dibujar()` se envía a una **Forma** anónima, tiene lugar el comportamiento correcto basándose en el tipo real de la **Forma**. Esto es impresionante porque, como se ha dicho anteriormente, cuando el compilador Java está compilando el código de `hacerAlgo()`, no puede saber de forma exacta con qué tipos está tratando. Por ello, habitualmente se espera que llame a la versión de `borrar()` y `dibujar()` para la clase base **Forma** y no a la versión específica de **Círculo**, **Cuadrado** o **Linea**. Y sigue ocurriendo lo correcto gracias al polimorfismo. El compilador y el sistema de tiempo de ejecución controlan los detalles; todo lo que hay que saber es qué ocurre y, lo más importante, cómo diseñar haciendo uso de ello. Cuando se envía un mensaje a un objeto, el objeto hará lo correcto incluso cuando esté implicado el proceso de generalización.

La jerarquía de raíz única

Uno de los aspectos de la POO que tiene una importancia especial desde la introducción de C++ es si todas las clases en última instancia deberían ser heredadas de una única clase base. En Java (como en casi todos los demás lenguajes de POO

excepto C++) la respuesta es afirmativa. Y el nombre de esta clase base es simplemente **Object**. Resulta que las ventajas de una jerarquía de raíz única son enormes.

Todos los objetos de una jerarquía de raíz única tienen una interfaz en común, por lo que en última instancia son del mismo tipo fundamental. La alternativa (proporcionada por C++) es no saber que todo es del mismo tipo básico. Desde el punto de vista de la compatibilidad descendente, esto se ajusta al modelo de C mejor y puede pensarse que es menos restrictivo, pero cuando se quiere hacer programación orientada a objetos pura debe construirse una jerarquía propia con el fin de proporcionar la misma utilidad que se construye en otros lenguajes de programación orientada a objetos. Y en cualquier nueva biblioteca de clases que se adquiera, se empleará alguna otra interfaz incompatible. Requiere esfuerzo (y posiblemente herencia múltiple) hacer funcionar la nueva interfaz en un diseño propio. ¿Merece la pena entonces la “flexibilidad” adicional de C++? Si la necesita (si dispone ya de una gran cantidad de código en C), entonces es bastante valiosa. Si parte de cero, otras alternativas como Java a menudo resultan más productivas.

Puede garantizarse que todos los objetos de una jerarquía de raíz única tengan una determinada funcionalidad. Es posible realizar determinadas operaciones básicas sobre todos los objetos del sistema. Pueden crearse todos los objetos y el paso de argumentos se simplifica enormemente.

Una jerarquía de raíz única facilita mucho la implementación de un depurador de memoria, que es una de las mejoras fundamentales de Java sobre C++. Y dado que la información sobre el tipo de un objeto está garantizada en todos los objetos, nunca se encontrará con un objeto cuyo tipo no pueda determinarse. Esto es especialmente importante en las operaciones en el nivel del sistema, como por ejemplo el tratamiento de excepciones y para proporcionar un mayor grado de flexibilidad en la programación.

Contenedores

En general, no se sabe cuántos objetos se van a necesitar para resolver un determinado problema o cuánto tiempo va a llevar. Tampoco se sabe cómo se van a almacenar dichos objetos. ¿Cómo se puede saber cuánto espacio hay que crear si no se conoce dicha información hasta el momento de la ejecución?

La solución a la mayoría de los problemas en el diseño orientado a objetos parece algo poco serio, esta solución consiste en crear otro tipo de objeto. El nuevo tipo de objeto que resuelve este problema concreto almacena referencias a otros objetos. Por supuesto, se puede hacer lo mismo con una *matriz*, elemento que está disponible en la mayoría de los lenguajes. Pero este nuevo objeto, denominado contenedor (también se llama colección, pero la biblioteca de Java utiliza dicho término con un sentido diferente, por lo que en este libro emplearemos el término “contenedor”), se amplia por sí mismo cuando es necesario acomodar cualquier cosa que se quiera introducir en él. Por tanto, no necesitamos saber cuántos objetos pueden almacenarse en un contenedor. Basta con crear un objeto contenedor y dejarle a él que se ocupe de los detalles.

Afortunadamente, los buenos lenguajes de programación orientada a objetos incluyen un conjunto de contenedores como parte del paquete. En C++, ese conjunto forma parte de la biblioteca estándar C++ y a menudo se le denomina STL (*Standard Template Library*, biblioteca estándar de plantillas). Smalltalk tiene un conjunto muy completo de contenedores, mientras que Java tiene también numerosos contenedores en su biblioteca estándar. En algunas bibliotecas, se considera que uno o dos contenedores genéricos bastan y sobran para satisfacer todas las necesidades, mientras que en otras (por ejemplo, en Java) la biblioteca tiene diferentes tipos de contenedores para satisfacer necesidades distintas: varios tipos diferentes de clases **List** (para almacenar secuencias), **Maps** (también denominados matrices asociativas y que se emplean para asociar objetos con otros objetos), **Sets** (para almacenar un objeto de cada tipo) y otros componentes como colas, árboles, pilas, etc.

Desde el punto de vista del diseño, lo único que queremos es disponer de un contenedor que pueda manipularse para resolver nuestro problema. Si un mismo tipo de contenedor satisface todas las necesidades, no existe ninguna razón para disponer de varias clases de contenedor. Sin embargo, existen dos razones por las que sí es necesario poder disponer de diferentes contenedores. En primer lugar, cada tipo de contenedor proporciona su propio tipo de interfaz y su propio comportamiento externo. Una pila tiene una interfaz y un comportamiento distintos que una cola, que a su vez es distinto de un conjunto o una lista. Es posible que alguno de estos tipos de contenedor proporcione una solución más flexible a nuestro problema que los restantes tipos. En segundo lugar, contenedores diferentes tienen una eficiencia distinta a la hora de realizar determinadas operaciones. Por ejemplo, existen dos tipos básicos de contenedores de tipo List: **ArrayList** (lista matricial) y **LinkedList** (lista enlazada). Ambos son secuencias simples que pueden tener interfaces y comportamientos externos idénticos. Pero ciertas operaciones pueden llevar asociado un coste radicalmente distinto. La operación de acceder aleatoriamente a los elementos contenidos en un contenedor de tipo **ArrayList** es una operación de tiempo constante. Se tarda el mismo tiempo

independientemente de cuál sea el elemento que se haya seleccionado. Sin embargo, en un contenedor de tipo **LinkedList** resulta muy caro desplazarse a lo largo de la lista para seleccionar aleatoriamente un elemento, y se tarda más tiempo en localizar un elemento cuanto más atrás esté situado en la lista. Por otro lado, si se quiere insertar un elemento en mitad de la secuencia, es más barato hacerlo en un contenedor de tipo **LinkedList** que en otro de tipo **ArrayList**. Estas y otras operaciones pueden tener una eficiencia diferente dependiendo de la estructura subyacente de la secuencia. Podemos comenzar construyendo nuestro programa con un contenedor de tipo **LinkedList** y, a la hora de juzgar las prestaciones, cambiar a otro de tipo **ArrayList**. Debido a la abstracción obtenida mediante la interfaz List, podemos cambiar de un tipo de contenedor a otro con un impacto mínimo en el código.

Tipos parametrizados (genéricos)

Antes de Java SE5, los contenedores albergaban objetos del tipo universal de Java: **Object**. La jerarquía de raíz única indica que todo es de tipo **Object**, por lo que un contenedor que almacene objetos de tipo **Object** podrá almacenar cualquier cosa.⁶ Esto hacía que los contenedores fueran fáciles de reutilizar.

Para utilizar uno de estos contenedores, simplemente se añaden a él referencias a objetos y luego se las extrae. Sin embargo, puesto que el contenedor sólo permite almacenar objetos de tipo **Object**, al añadir una referencia a objeto al contenedor, esa referencia se transforma en una referencia a **Object** perdiendo así su carácter. Al extraerla, se obtiene una referencia a **Object** y no una referencia al tipo que se hubiera almacenado. En estas condiciones, ¿cómo podemos transformar esa referencia en algo que tenga el tipo específico de objeto que hubiéramos almacenado en el contenedor?

Lo que se hace es volver a utilizar el mecanismo de transformación de tipos (*cast*), pero esta vez no efectuamos una generalización, subiendo por la jerarquía de herencia, sino que efectuamos una especialización, descendiendo desde la jerarquía hasta alcanzar un tipo más específico. Este mecanismo de transformación de tipos se denomina especialización (*downcasting*). Con el mecanismo de generalización (*upcasting*), sabemos por ejemplo que un objeto **Círculo** es también de tipo **Forma**, por lo que resulta seguro realizar la transformación de tipos. Sin embargo, no todo objeto de tipo **Object** es necesariamente de tipo **Círculo** o **Forma** por lo que no resulta tan seguro realizar una especialización a menos que sepamos concretamente lo que estamos haciendo.

Sin embargo, esta operación no es del todo peligrosa, porque si efectuamos una conversión de tipos y transformamos el objeto a un tipo incorrecto, obtendremos un error de tipo de ejecución denominado *excepción* (lo que se describe más adelante). Sin embargo, cuando extraemos referencias a objetos de un contenedor, tenemos que disponer de alguna forma de recordar exactamente lo que son, con el fin de poder realizar la conversión de tipos apropiada.

El mecanismo de especialización y las comprobaciones en tiempo de ejecución requieren tiempo adicional para la ejecución del programa y un mayor esfuerzo por parte del programador. ¿No sería más lógico crear el contenedor de manera que éste supiera el tipo de los elementos que almacena, eliminando la necesidad de efectuar conversiones de tipos y evitando los errores asociados? La solución a este problema es el mecanismo de *tipos parametrizados*. Un tipo parametrizado es una clase que el compilador puede personalizar automáticamente para que funcione con cada tipo concreto. Por ejemplo, con un contenedor parametrizado, el compilador puede personalizar dicho contenedor para que sólo acepte y devuelva objetos **Forma**.

Uno de los cambios principales en Java SE5 es la adición de tipos parametrizados, que se denominan genéricos en Java. El uso de genéricos es fácilmente reconocible, ya que emplean corchetes angulares para encerrar alguna especificación de tipo, por ejemplo, puede crearse un contenedor de tipo **ArrayList** que almacene objetos de tipo **Forma** del siguiente modo:

```
ArrayList<Forma> formas = new ArrayList<Forma>();
```

También se han efectuado modificaciones en muchos de los componentes de las bibliotecas estándar para poder aprovechar el uso de genéricos. Como tendremos oportunidad de ver, los genéricos tienen una gran importancia en buena parte del código utilizado en este libro.

Creación y vida de los objetos

Una de las cuestiones críticas a la hora de trabajar con los objetos es la forma en que éstos se crean y se destruyen. Cada objeto consigue una serie de recursos, especialmente memoria, para poder simplemente existir. Cuando un objeto deja de

⁶ Los contenedores no permiten almacenar primitivas, pero la característica de *autoboxing* de Java SE5 hace que esta restricción tenga poca importancia. Hablaremos de esto en detalle más adelante en el libro.

ser necesario, es preciso eliminarlo, para que se liberen estos recursos y puedan emplearse en alguna otra cosa. En los casos más simples de programación, el problema de borrar los objetos no resulta demasiado complicado. Creamos el objeto, lo usamos mientras que es necesario y después lo destruimos. Sin embargo, no es difícil encontrarse situaciones bastante más complejas que ésta.

Suponga por ejemplo que estamos diseñando un sistema para gestionar el tráfico aéreo de un aeropuerto (el mismo modelo serviría para gestionar piezas en un almacén o para un sistema de alquiler de videos o para una tienda de venta de mascotas). A primera vista, el problema parece muy simple: creamos un contenedor para almacenar las aeronaves y luego creamos una nueva aeronave y la insertamos en el contenedor por cada una de las aeronaves que entran en la zona de control del tráfico aéreo. De cara al borrado, simplemente basta con eliminar el objeto aeronave apropiado en el momento en que el avión abandone la zona.

Pero es posible que tengamos algún otro sistema en el que queden registrados los datos acerca de los aviones; quizás se trate de datos que no requieran una atención tan inmediata como la de la función principal de control del tráfico aéreo. Puede que se trate de un registro de los planes de vuelo de todos los pequeños aeroplanos que salgan del aeropuerto. Entonces, podríamos definir un segundo contenedor para esos aeroplanos y, cada vez que se creara un objeto aeronave, se introduciría también en este segundo contenedor si se trata de un pequeño aeroplano. Entonces, algún proceso de segundo plano podría realizar operaciones sobre los objetos almacenados en este segundo contenedor en los momentos de inactividad.

Ahora el problema ya es más complicado: ¿cómo podemos saber cuándo hay que destruir los objetos? Aún cuando nosotros hayamos terminado de procesar un objeto, puede que alguna otra parte del sistema no lo haya hecho. Este mismo problema puede surgir en muchas otras situaciones, y puede llegar a resultar enormemente complejo de resolver en aquellos sistemas de programación (como C++) en los que es preciso borrar explícitamente un objeto cuando se ha terminado de utilizar.

¿Dónde se almacenan los datos correspondientes a un objeto y cómo se puede controlar el tiempo de vida del mismo? En C++, se adopta el enfoque de que el control de la eficiencia es el tema más importante, por lo que todas las decisiones quedan en manos del programador. Para conseguir la máxima velocidad de ejecución, las características de almacenamiento y del tiempo de vida del objeto pueden determinarse mientras se está escribiendo el programa, colocando los objetos en la pila (a estos objetos se los denomina en ocasiones variables automáticas o de ámbito) o en el área de almacenamiento estático. Esto hace que lo más prioritario sea la velocidad de asignación y liberación del almacenamiento, y este control puede resultar muy útil en muchas situaciones. Sin embargo, perdemos flexibilidad porque es preciso conocer la cantidad, el tiempo de vida y el tipo exacto de los objetos a la hora de escribir el programa. Si estamos tratando de resolver un problema más general, como por ejemplo, un programa de diseño asistido por computadora, un sistema de gestión de almacén o un sistema de control de tráfico aéreo, esta solución es demasiado restrictiva.

La segunda posibilidad consiste en crear los objetos dinámicamente en un área de memoria denominada círculo. Con este enfoque, no sabemos hasta el momento de la ejecución cuántos objetos van a ser necesarios, cuál va a ser su tiempo de vida ni cuál es su tipo exacto. Todas estas características se determinan en el momento en que se ejecuta el programa. Si hace falta un nuevo objeto, simplemente se crea en el círculo de memoria, en el preciso instante en que sea necesario. Puesto que el almacenamiento se gestiona dinámicamente en tiempo de ejecución, la cantidad de tiempo requerida para asignar el almacenamiento en el círculo de memoria puede ser bastante mayor que el tiempo necesario para crear un cierto espacio en la pila. La creación de espacio de almacenamiento en la pila requiere normalmente una única instrucción de ensamblador, para desplazar hacia abajo el puntero de la pila y otra instrucción para volver a desplazarlo hacia arriba. El tiempo necesario para crear un espacio de almacenamiento en el círculo de memoria depende del diseño del mecanismo de almacenamiento.

La solución dinámica se basa en la suposición, generalmente bastante lógica, de que los objetos suelen ser complicados, por lo que el tiempo adicional requerido para localizar el espacio de almacenamiento y luego liberarlo no tendrá demasiado impacto sobre el proceso de creación del objeto. Además, el mayor grado de flexibilidad que se obtiene resulta esencial para resolver los problemas de programación de carácter general.

Java utiliza exclusivamente un mecanismo dinámico de asignación de memoria⁷. Cada vez que se quiere crear un objeto, se utiliza el operador **new** para construir una instancia dinámica del objeto.

Sin embargo, existe otro problema, referido al tiempo de vida de un objeto. En aquellos lenguajes que permiten crear objetos en la pila, el compilador determina cuál es la duración del objeto y puede destruirlo automáticamente. Sin embargo, si creamos el objeto en el círculo de memoria, el compilador no sabe cuál es su tiempo de vida. En un lenguaje como C++,

⁷ Los tipos primitivos, de los que hablaremos en breve, representan un caso especial.

es preciso determinar mediante programa cuándo debe destruirse el objeto, lo que puede provocar pérdidas de memoria si no se realiza esta tarea correctamente (y este problema resulta bastante común en los programas C++). Java proporciona una característica denominada *depurador de memoria*, que descubre automáticamente cuándo un determinado objeto ya no está en uso, en cuyo caso lo destruye. Un depurador de memoria resulta mucho más cómodo que cualquier otra solución alternativa, porque reduce el número de problemas que el programador debe controlar, y reduce también la cantidad de código que hay que escribir. Además, lo que resulta más importante, el depurador de memoria proporciona un nivel mucho mayor de protección contra el insidioso problema de las fugas de memoria, que ha hecho que muchos proyectos en C++ fracasaran.

En Java, el depurador de memoria está diseñado para encargarse del problema de liberación de la memoria (aunque esto no incluye otros aspectos relativos al borrado de un objeto). El depurador de memoria “sabe” cuándo ya no se está usando un objeto, en cuyo caso libera automáticamente la memoria correspondiente a ese objeto. Esta característica, combinada con el hecho de que todos los objetos heredan de la clase raíz **Object**, y con el hecho de que sólo pueden crearse objetos de una manera (en el cúmulo de memoria), hace que el proceso de programación en Java sea mucho más simple que en C++, hay muchas menos decisiones que tomar y muchos menos problemas que resolver.

Tratamiento de excepciones: manejo de errores

Desde la aparición de los lenguajes de programación, el tratamiento de los errores ha constituido un problema peculiarmente difícil. Debido a que resulta muy complicado diseñar un buen esquema de tratamiento de errores, muchos lenguajes simplemente ignoran este problema, dejando que lo resuelvan los diseñadores de bibliotecas, que al final terminan desarrollando soluciones parciales que funcionan en muchas situaciones pero cuyas medidas pueden ser obviadas fácilmente; generalmente, basta con ignorarlas. Uno de los problemas principales de la mayoría de los esquemas de tratamiento de errores es que dependen de que el programador tenga cuidado a la hora de seguir un convenio preestablecido que no resulta obligatorio dentro del lenguaje. Si el programador no tiene cuidado (lo cual suele suceder cuando hay prisa por terminar un proyecto), puede olvidarse fácilmente de estos esquemas.

Los mecanismos de tratamiento de excepciones integran la gestión de errores directamente dentro del lenguaje de programación, en ocasiones, dentro incluso del sistema operativo. Una excepción no es más que un objeto “generado” en el lugar donde se ha producido el error y que puede ser “capturado” mediante una rutina apropiada de tratamiento de excepciones diseñada para gestionar dicho tipo particular de error. Es como si el tratamiento de excepciones fuera una ruta de ejecución paralela y diferente, que se toma cuando algo va mal. Y, como se utiliza una ruta de ejecución independiente, ésta no tiene porqué interferir con el código que se ejecuta normalmente. Esto hace que el código sea más simple de escribir, porque no es necesario comprobar constantemente la existencia de errores. Además, las excepciones generadas se diferencian de los típicos valores de error devueltos por los métodos o por los indicadores activados por los métodos para avisar que se ha producido una condición de error; tanto los valores como los indicadores de error podrían ser ignorados por el programador. Las excepciones no pueden ignorarse, por lo que se garantiza que en algún momento serán tratadas. Finalmente, las excepciones proporcionan un mecanismo para recuperarse de manera fiable de cualquier situación errónea. En lugar de limitarse a salir del programa, a menudo podemos corregir las cosas y restaurar la ejecución, lo que da como resultado programas mucho más robustos.

El tratamiento de excepciones de Java resulta muy sobresaliente entre los lenguajes de programación, porque en Java el tratamiento de excepciones estaba previsto desde el principio y estamos obligados a utilizarlo. Este esquema de tratamiento de excepciones es el único mecanismo aceptable en Java para informar de la existencia de errores. Si no se escribe el código de manera que trate adecuadamente las excepciones se obtiene un error en tiempo de compilación. Esta garantía de coherencia puede hacer que, en ocasiones, el tratamiento de errores sea mucho más sencillo.

Merece la pena resaltar que el tratamiento de excepciones no es una característica orientada a objetos, aunque en los lenguajes de programación orientada a objetos las excepciones se representan normalmente mediante un objeto. Los mecanismos de tratamiento de excepciones ya existían antes de que hicieran su aparición los lenguajes orientados a objetos.

Programación concurrente

Un concepto fundamental en el campo de la programación es la idea de poder gestionar más de una tarea al mismo tiempo. Muchos problemas de programación requieren que el programa detenga la tarea que estuviera realizando, resuelva algún

otro problema y luego vuelva al proceso principal. A lo largo del tiempo, se ha tratado de aplicar diversas soluciones a este problema. Inicialmente, los programadores que tenían un adecuado conocimiento de bajo nivel de la máquina sobre la que estaban programando escribían rutinas de servicio de interrupción, y la suspensión del proceso principal se llevaba a cabo mediante una interrupción hardware. Aunque este esquema funcionaba bien, resultaba complicado y no era portable, por lo que traducir un programa a un nuevo tipo de máquina resultaba bastante lento y muy caro.

En ocasiones, las interrupciones son necesarias para gestionar las tareas con requisitos críticos de tiempo, pero hay una amplia clase de problemas en la que tan sólo nos interesa dividir el problema en una serie de fragmentos que se ejecuten por separado (tareas), de modo que el programa completo pueda tener un mejor tiempo de respuesta. En un programa, estos fragmentos que se ejecutan por separado, se denominan *hebras* y el conjunto general se llama concurrencia. Un ejemplo bastante común de concurrencia es la interfaz de usuario. Utilizando distintas tareas, el usuario apretando un botón puede obtener una respuesta rápida, en lugar de tener que esperar a que el programa finalice con la tarea que está actualmente realizando.

Normalmente, las tareas son sólo una forma de asignar el tiempo disponible en un único procesador. Pero si el sistema operativo soporta múltiples procesadores, puede asignarse cada tarea a un procesador distinto, en cuyo caso las tareas pueden ejecutarse realmente en paralelo. Una de las ventajas de incluir los mecanismos de concurrencia en el nivel de lenguaje es que el programador no tiene que preocuparse de si hay varios procesadores o sólo uno; el programa se divide desde el punto de vista lógico en una serie de tareas, y si la máquina dispone de más de un procesador, el programa se ejecutará más rápido, sin necesidad de efectuar ningún ajuste especial.

Todo esto hace que la concurrencia parezca algo bastante sencillo, pero existe un problema: los recursos compartidos. Si se están ejecutando varias tareas que esperan poder acceder al mismo recurso, tendremos un problema de contienda entre las tareas. Por ejemplo, no puede haber dos procesadores enviando información a una misma impresora. Para resolver el problema, los recursos que puedan compartirse, como por ejemplo una impresora, deben bloquearse mientras estén siendo utilizados por una tarea. De manera que la forma de funcionar es la siguiente: una tarea bloquea un recurso, completa el trabajo que tuviera asignado y luego elimina el bloqueo para que alguna otra tarea pueda emplear el recurso.

Los mecanismos de concurrencia en Java están integrados dentro de lenguaje y Java SE5 ha mejorado significativamente el soporte de biblioteca para los mecanismos de concurrencia.

Java e Internet

Si Java no es, en definitiva, más que otro lenguaje informático de programación, podríamos preguntarnos por qué es tan importante y por qué se dice de él que representa una auténtica revolución dentro del campo de la programación. La respuesta no resulta obvia para aquéllos que provengan del campo de la programación tradicional. Aunque Java resulta muy útil para resolver problemas de programación en entornos autónomos, su importancia se debe a que permite resolver los problemas de programación que surgen en la World Wide Web.

¿Qué es la Web?

Al principio, la Web puede parecer algo misterioso, con todas esas palabras extrañas como "surfear," "presencia web" y "páginas de inicio". Resulta útil, para entender los conceptos, dar un paso atrás y tratar de comprender lo que la Web es realmente, pero para ello es necesario comprender primero lo que son los sistemas cliente/servidor, que constituyen otro campo de la informática lleno de conceptos bastante confusos.

Informática cliente/servidor

La idea principal en la que se basan los sistemas cliente/servidor es que podemos disponer de un repositorio centralizado de información (por ejemplo, algún tipo de datos dentro de una base de datos) que queramos distribuir bajo demanda a una serie de personas o de computadoras. Uno de los conceptos clave de las arquitecturas cliente/servidor es que el repositorio de información está centralizado, por lo que puede ser modificado sin que esas modificaciones se propaguen hasta los consumidores de la información. El repositorio de información, el software que distribuye la información y la máquina o máquinas donde esa información y ese software residen se denominan, en conjunto, "servidor". El software que reside en las máquinas consumidoras, que se comunica con el servidor, que extrae la información, que la procesa y que luego la muestra en la propia máquina consumidora se denomina cliente.

El concepto básico de informática cliente/servidor no es, por tanto, demasiado complicado. Los problemas surgen porque disponemos de un único servidor tratando de dar servicio a múltiples clientes al mismo tiempo. Generalmente, se utiliza algún tipo de sistema de gestión de bases de datos de modo que el diseñador “equilibra” la disposición de los datos entre distintas tablas, con el fin de optimizar el uso de los datos. Además, estos sistemas permiten a menudo que los clientes inserten nueva información dentro de un servidor. Esto quiere decir que es preciso garantizar que los nuevos datos de un cliente no sobreescriban los nuevos datos de otro cliente, al igual que hay que garantizar que no se pierdan datos en el proceso de añadirlos a la base de datos (este tipo de mecanismos se denomina procesamiento de transacciones). A medida que se realizan modificaciones en el software de cliente, es necesario diseñar el software, depurarlo e instalarlo en las máquinas cliente, lo que resulta ser más complicado y más caro de lo que en un principio cabría esperar. Resulta especialmente problemático soportar múltiples tipos de computadoras y de sistemas operativos. Finalmente, es necesario tener en cuenta también la cuestión crucial del rendimiento: puede que tengamos cientos de clientes enviando cientos de solicitudes al servidor en un momento dado, por lo que cualquier pequeño retraso puede llegar a ser verdaderamente crítico. Para minimizar la latencia, los programadores hacen un gran esfuerzo para tratar de descargar las tareas de procesamiento que en ocasiones se descargan en la máquina cliente, pero en otras ocasiones se descargan en otras máquinas situadas junto al servidor, utilizando un tipo especial de software denominado middleware, (el middleware se utiliza también para mejorar la facilidad de mantenimiento del sistema).

Esa idea tan simple de distribuir la información tiene tantos niveles de complejidad que el problema global puede parecer enigmáticamente insoluble. A pesar de lo cual, se trata de un problema crucial: la informática cliente/servidor representa aproximadamente la mitad de las actividades de programación en la actualidad. Este tipo de arquitectura es responsable de todo tipo de tareas, desde la introducción de pedidos y la realización de transacciones con tarjetas de crédito hasta la distribución de cualquier tipo de datos, como por ejemplo cotizaciones bursátiles, datos científicos, información de organismos gubernamentales. En el pasado, lo que hemos hecho es desarrollar soluciones individuales para problemas individuales, inventando una nueva solución en cada ocasión. Esas soluciones eran difíciles de diseñar y de utilizar, y el usuario se veía obligado a aprender una nueva interfaz en cada caso. De este modo, se llegó a un punto en que era necesario resolver el problema global de la informática cliente/servidor de una vez y para siempre.

La Web como un gigantesco servidor

La Web es, en la práctica, un sistema gigantesco de tipo cliente/servidor. En realidad, es todavía más complejo, ya que lo que tenemos es un conjunto de servidores y clientes que coexisten en una misma red de manera simultánea. El usuario no necesita ser consciente de ello, por lo que lo único que hace es conectarse con un servidor en cada momento e interactuar con él (aún cuando para llegar a ese servidor haya sido necesario ir saltando de servidor en servidor por todo el mundo hasta dar con el correcto).

Inicialmente, se trataba de un proceso muy simple de carácter unidireccional: el usuario envía una solicitud al servidor y éste le devolvía un archivo, que el software explorador de la máquina (es decir, el cliente) se encargaba de interpretar, efectuando todas las tareas de formateo en la propia máquina local. Pero al cabo de muy poco tiempo, los propietarios de servidores comenzaron a querer hacer cosas más complejas que simplemente suministrar páginas desde el servidor. Querían disponer de una capacidad completa cliente/servidor, de forma que el cliente pudiera, por ejemplo enviar información al servidor, realizar búsquedas en una base de datos instalada en el servidor, añadir nueva información al servidor o realizar un pedido (lo que requiere medidas especiales de seguridad). Éstos son los cambios que hemos vivido en los últimos años en el desarrollo de la Web.

Los exploradores web representaron un gran avance: permitieron implementar el concepto de que un mismo fragmento de información pudiera visualizarse en cualquier tipo de computadora sin necesidad de efectuar ninguna modificación. Sin embargo, los primeros exploradores eran bastante primitivos y se colapsaban rápidamente debido a las demandas que se les hacía. No resultaban peculiarmente interactivos y tendían a sobrecargar al servidor tanto como a la propia red Internet, porque cada vez que hacia falta hacer algo que requería programación, era necesario devolver la información al servidor para que éste la procesara. De esta forma, podía tardarse varios segundos o incluso minutos en averiguar simplemente que habíamos tecleado incorrectamente algo dentro de la solicitud. Como el explorador era simplemente un mecanismo de visualización no podía realizar ni siquiera la más simple de las tareas (por otro lado, resultaba bastante seguro ya que no podía ejecutar en la máquina local ningún programa que pudiera contener errores o virus).

Para resolver este problema, se han adoptado diferentes enfoques. Para empezar se han mejorado los estándares gráficos para poder disponer de mejores animaciones y vídeos dentro de los exploradores. El resto del problema sólo puede resol-

verse incorporando la capacidad de ejecutar programas en el extremo cliente, bajo control del explorador. Esto se denomina *programación del lado del cliente*.

Programación del lado del cliente

El diseño inicial de la Web, basado en una arquitectura servidor/explorador, permitía disponer de contenido interactivo, pero esa interactividad era completamente proporcionada por el servidor. El servidor generaba páginas estáticas para el explorador cliente, que simplemente se encargaba de interpretarlas y mostrarlas. El lenguaje básico HTML (*HyperText Markup Language*) contiene una serie de mecanismos simples para la introducción de datos: cuadros de introducción de texto, casillas de verificación, botones de opción, listas normales y listas desplegables, así como un botón que sólo podía programarse para borrar los datos del formulario o enviarlos al servidor. Ese proceso de envío se llevaba a cabo a través de la interfaz CGI (*Common Gateway Interface*) incluida en todos los servidores web. El texto incorporado en el envío le dice a la interfaz CGI lo que tiene que hacer. La acción más común, en este caso, consiste en ejecutar un programa ubicado en un servidor en un directorio normalmente llamado “cgi-bin” (si observa la barra de direcciones situada en la parte superior del explorador cuando pulse un botón en una página web, en ocasiones podrá ver las palabras “cgi-bin” como parte de la dirección). Estos programas del lado del servidor pueden escribirse en casi cualquier lenguaje. Perl es uno de los lenguajes más utilizados para este tipo de tareas, porque está diseñado específicamente para la manipulación de textos y es un lenguaje interpretado, por lo que se puede instalar en cualquier servidor independientemente de cuál sea su procesador o su sistema operativo. Sin embargo, otro lenguaje, Python (www.Python.org) se está abriendo camino rápidamente, debido a su mayor potencia y su mayor simplicidad.

Hay muchos sitios web potentes en la actualidad diseñados estrictamente con CGI, y lo cierto es que con CGI se puede hacer prácticamente de todo. Sin embargo, esos sitios web basados en programas CGI pueden llegar a ser rápidamente bastante complicados de mantener, y además pueden aparecer problemas en lo que se refiere al tiempo de respuesta. El tiempo de respuesta de un programa CGI depende de cuántos datos haya que enviar, de la carga del servidor y de la red Internet (además, el propio arranque de un programa CGI tiende a ser bastante lento). Los primeros diseñadores de la Web no previeron la rapidez con que el ancho de banda disponible iba a agotarse debido a los tipos de aplicaciones que la gente llegaría a desarrollar. Por ejemplo, es casi imposible diseñar de manera coherente una aplicación con gráficos dinámicos, porque es necesario crear un archivo GIF (*Graphics Interchange Format*) y enviarlo del servidor al cliente para cada versión de gráfico. Además, casi todos los usuarios hemos experimentado lo engorroso del proceso de validación de los datos dentro de un formulario enviado a través de la Web. El proceso es el siguiente: pulsamos el botón de envío de la página; se envían los datos al servidor, el servidor arranca un programa CGI que descubre un error, formatea una página HTML en la que nos informa del error y devuelve la página al cliente; a continuación, es necesario que el usuario retroceda una página y vuelva a intentarlo. Este enfoque no sólo resulta lento sino también poco elegante.

La solución consiste en usar un mecanismo de programación del lado del cliente. La mayoría de las computadoras de sobremesa que incluyen un explorador web son máquinas bastante potentes, capaces de realizar tareas muy complejas; con el enfoque original basado en HTML estático, esas potentes máquinas simplemente se limitan a esperar sin hacer nada, hasta que el servidor se digna a enviarles la siguiente página. La programación del lado del cliente permite asignar al explorador web todo el trabajo que pueda llevar a cabo, con lo que el resultado para los usuarios es una experiencia mucha más rápida y más interactiva a la hora de acceder a los sitios web.

El problema con las explicaciones acerca de la programación del lado del cliente es que no se diferencian mucho de las explicaciones relativas a la programación en general. Los parámetros son prácticamente idénticos, aunque la plataforma sea distinta: un explorador web es una especie de sistema operativo limitado. En último término, sigue siendo necesario diseñar programas, por lo que los problemas y soluciones que nos encontramos dentro del campo de la programación del lado del cliente son bastante tradicionales. En el resto de esta sección, vamos a repasar algunos de los principales problemas y técnicas que suelen encontrarse en el campo de la programación del lado del cliente.

Plug-ins

Uno de los avances más significativos en la programación del lado del cliente es el desarrollo de lo que se denomina *plug-in*. Se trata de un mecanismo mediante el que un programador puede añadir algún nuevo tipo de funcionalidad a un explorador descargando un fragmento de código que se inserta en el lugar apropiado dentro del explorador. Ese fragmento de código le dice al explorador: “A partir de ahora puedes realizar este nuevo tipo de actividad” (sólo es necesario descargar el

plug-in una vez). Podemos añadir nuevas formas de comportamiento, potentes y rápidas, a los exploradores mediante *plug-ins*, pero la escritura de un *plug-in* no resulta nada trivial, y por eso mismo no es conveniente acometer ese tipo de tarea como parte del proceso de construcción de un sitio web. El valor de un *plug-in* para la programación del lado del cliente es que permite a los programadores avanzados desarrollar extensiones y añadírselas a un explorador sin necesidad de pedir permiso al fabricante del explorador. De esta forma, los *plug-ins* proporcionan una especie de “puerta trasera” que permite la creación de nuevos lenguajes de programación del lado del cliente (aunque no todos los lenguajes se implementan como *plug-ins*).

Lenguajes de *script*

Los *plug-ins* dieron como resultado el desarrollo de lenguajes de *script* para los exploradores. Con un lenguaje de *script*, el código fuente del programa del lado del cliente se integra directamente dentro de la página HTML, y el *plug-in* que se encarga de interpretar ese lenguaje se activa de manera automática en el momento de visualizar la página HTML. Los lenguajes de *script* suelen ser razonablemente fáciles de comprender, y como están formados simplemente por texto que se incluye dentro de la propia página HTML, se cargan muy rápidamente como parte del acceso al servidor mediante el que se obtiene la página. La desventaja es que el código queda expuesto, ya que cualquiera puede verlo (y copiarlo). Generalmente, sin embargo, los programadores no llevan a cabo tareas extremadamente sofisticadas con los lenguajes de *script*, así que este problema no resulta particularmente grave.

Uno de los lenguajes de *script* que los exploradores web suelen soportar sin necesidad de un *plug-in* es JavaScript (el lenguaje JavaScript sólo se asemeja de forma bastante vaga a Java, por lo que hace falta un esfuerzo de aprendizaje adicional para llegar a dominarlo; recibió el nombre de JavaScript simplemente para aprovechar el impulso inicial de marketing de Java). Lamentablemente, cada explorador web implementaba originalmente JavaScript de forma distinta a los restantes exploradores web, e incluso en ocasiones, de forma diferente a otras versiones del mismo explorador. La estandarización de JavaScript mediante el diseño del lenguaje estándar ECMAScript ha resuelto parcialmente este problema, pero tuvo que transcurrir bastante tiempo hasta que los distintos exploradores adoptaron el estándar (el problema se complicó porque Microsoft trataba de conseguir sus propios objetivos presionando en favor de su lenguaje VBScript, que también se asemejaba vagamente a JavaScript). En general, es necesario llevar a cabo la programación de las páginas utilizando una especie de mínimo común denominador de JavaScript, si lo que queremos es que esas páginas puedan visualizarse en todos los tipos de exploradores. Por su parte, la solución de errores y la depuración en JavaScript son un auténtico lio. Como prueba de lo difícil que resulta diseñar un problema complejo con JavaScript, sólo muy recientemente alguien se ha atrevido a crear una aplicación compleja basada en él (Google, con GMail), y ese desarrollo requirió una dedicación y una experiencia realmente notables.

Lo que todo esto nos sugiere es que los lenguajes de *script* que se emplean en los exploradores web están diseñados, realmente, para resolver tipos específicos de problemas, principalmente el de la creación de interfaces gráficas de usuario (GUI) más ricas e interactivas. Sin embargo, un lenguaje de *script* puede resolver quizás un 80 por ciento de los problemas que podemos encontrar en la programación del lado del cliente. Es posible que los problemas que el lector quiera resolver estén incluidos dentro de ese 80 por ciento. Si esto es así, y teniendo en cuenta que los lenguajes de *script* permiten realizar los desarrollos de forma más fácil y rápida, probablemente sería conveniente ver si se puede resolver un tipo concreto de problema empleando un lenguaje de *script*, antes de considerar otras soluciones más complejas, como la programación en Java.

Java

Si un lenguaje de *script* puede resolver el 80 por ciento de los problemas de la programación del lado del cliente, ¿qué pasa con el otro 20 por ciento, con los “problemas realmente difíciles”? Java representa una solución bastante popular para este tipo de problemas. No sólo se trata de un potente lenguaje de programación diseñado para ser seguro, interplataforma e internacional, sino que continuamente está siendo ampliado para proporcionar nuevas características del lenguaje y nuevas bibliotecas que permiten gestionar de manera elegante una serie de problemas que resultan bastante difíciles de tratar en los lenguajes de programación tradicionales, como por ejemplo la concurrencia, el acceso a bases de datos, la programación en red y la informática distribuida. Java permite resolver los problemas de programación del lado del cliente utilizando *applets* y *Java Web Start*.

Un *applet* es un mini-programa que sólo puede ejecutarse sobre un explorador web. El *applet* se descarga automáticamente como parte de la página web (de la misma forma que se descarga automáticamente, por ejemplo, un gráfico). Cuando se activa el *applet*, ejecuta un programa. Este mecanismo de ejecución automática forma parte de la belleza de esta solución: nos proporciona una forma de distribuir automáticamente el software de cliente desde el servidor en el mismo momento en

que el usuario necesita ese software de cliente, y no antes. El usuario obtiene la última versión del software de cliente, libre de errores y sin necesidad de realizar complejas reinstalaciones. Debido a la forma en que se ha diseñado Java, el programador sólo tiene que crear un programa simple y ese programa funcionará automáticamente en todas las computadoras que dispongan de exploradores que incluyan un intérprete integrado de Java (lo que incluye la inmensa mayoría de máquinas). Puesto que Java es un lenguaje de programación completo podemos llevar a cabo la mayor cantidad de trabajo posible en el cliente, tanto antes como después de enviar solicitudes al servidor. Por ejemplo, no es necesario enviar una solicitud a través de Internet simplemente para descubrir que hemos escrito mal una fecha o algún otro parámetro; asimismo, la computadora cliente puede encargarse de manera rápida de la tarea de dibujar una serie de datos, en lugar de esperar a que el servidor genere el gráfico y devuelva una imagen al explorador. De este modo, no sólo aumentan de forma inmediata la velocidad y la capacidad de respuesta, sino que disminuyen también la carga de trabajo de los servidores y el tráfico de red, evitando así que todo Internet se ralentice.

Alternativas

Para ser honestos, los *applets* Java no han llegado a cumplir con las expectativas iniciales. Después del lanzamiento de Java, parecía que los *applets* era lo que más entusiasmaba a todo el mundo, porque iban a permitir finalmente realizar tareas serias de programación del lado del cliente, iban a mejorar la capacidad de respuesta de las aplicaciones basadas en Internet e iban a reducir el ancho de banda necesario. Las posibilidades que todo el mundo tenía en mente eran inmensas.

Sin embargo, hoy día nos podemos encontrar con unos *applets* realmente interesantes en la Web, pero la esperada migración masiva hacia los *applets* no llegó nunca a producirse. El principal problema era que la descarga de 10 MB necesaria para instalar el entorno de ejecución JRE (*Java Runtime Environment*) era demasiado para el usuario medio. El hecho de que Microsoft decidiera no incluir el entorno JRE dentro de Internet Explorer puede ser lo que acabó por determinar su aciago destino. Pero, sea como sea, lo cierto es que los *applets* Java no han llegado nunca a ser utilizados de forma masiva.

A pesar de todo, los *applets* y las aplicaciones *Java Web Start* siguen siendo adecuadas en algunas situaciones. En todos aquellos casos en que tengamos control sobre las máquinas de usuario, por ejemplo en una gran empresa, resulta razonable distribuir y actualizar las aplicaciones cliente utilizando este tipo de tecnologías, que nos pueden ahorrar una cantidad considerable de tiempo, esfuerzo y dinero, especialmente cuando es necesario realizar actualizaciones frecuentes.

En el Capítulo 22, *Interfaces gráficas de usuario*, analizaremos una buena tecnología bastante prometedora, Flex de Macromedia, que permite crear equivalentes a los *applets* basados en Flash. Como el reproductor Flash Player está disponible en más del 98 por ciento de todos los exploradores web (incluyendo Windows, Linux y Mac), puede considerarse como un estándar de facto. La instalación o actualización de Flash Player es asimismo rápida y fácil. El lenguaje ActionScript está basado en ECMAScript, por lo que resulta razonablemente familiar, pero Flex permite realizar las tareas de programación sin preocuparse acerca de las especificidades de los exploradores, por lo que resulta bastante más atractivo que JavaScript. Para la programación del lado del cliente se trata de una alternativa que merece la pena considerar.

.NET y C#

Durante un tiempo, el competidor principal de los *applets* de Java era ActiveX de Microsoft, aunque esta tecnología requería que en el cliente se estuviera ejecutando el sistema operativo Windows. Desde entonces, Microsoft ha desarrollado un competidor de Java: la plataforma .NET y el lenguaje de programación C#. La plataforma .NET es, aproximadamente, equivalente a la máquina virtual Java (JVM, *Java Virtual Machine*; es la plataforma software en la que se ejecutan los programas Java) y a las bibliotecas Java, mientras que C# tiene similitudes bastante evidentes con Java. Se trata, ciertamente, del mejor intento que Microsoft ha llevado a cabo en el área de los lenguajes de programación. Por supuesto, Microsoft partía con la considerable ventaja de conocer qué cosas habían funcionado de manera adecuada y qué cosas no funcionaban tan bien en Java, por lo que aprovechó esos conocimientos. Desde su concepción, es la primera vez que Java se ha encontrado con un verdadero competidor. Como resultado, los diseñadores de Java en Sun han analizado intensivamente C# y las razones por las que un programador podría sentirse tentado a adoptar ese lenguaje, y han respondido introduciendo significativas mejoras en Java, que han resultado en el lanzamiento de Java SE5.

Actualmente, la debilidad principal y el problema más importante en relación con .NET es si Microsoft permitirá portarlo completamente a otras plataformas. Ellos afirman que no hay ningún problema para esto, y el proyecto Mono (www.go-mono.com) dispone de una implementación parcial de .NET sobre Linux, pero hasta que la implementación sea completa y Microsoft decida no recortar ninguna parte de la misma, sigue siendo una apuesta arriesgada adoptar .NET como solución interplataforma.

Redes Internet e intranet

La Web es la solución más general para el problema de las arquitecturas cliente/servidor, por lo que tiene bastante sentido utilizar esta misma tecnología para resolver un cierto subconjunto de ese problema: el problema clásico de las arquitecturas cliente/servidor internas a una empresa. Con las técnicas tradicionales cliente/servidor, nos encontramos con el problema de la existencia de múltiples tipos de computadoras cliente, así como con la dificultad de instalar nuevo software de cliente; los exploradores web y la programación del lado del cliente permiten resolver fácilmente ambos problemas. Cuando se utiliza tecnología web para una red de información restringida a una empresa concreta, la arquitectura resultante se denomina intranet. Las intranets proporcionan un grado de seguridad mucho mayor que Internet, ya que podemos controlar físicamente el acceso a los equipos de la empresa. En términos de formación, una vez que los usuarios comprenden el concepto general de explorador les resulta mucho más fácil asumir las diferencias de aspecto entre las distintas páginas y *applets*, por lo que la curva de aprendizaje para los nuevos tipos de sistemas se reduce.

El problema de seguridad nos permite analizar una de las divisiones que parecen estarse formando de manera automática en el mundo de la programación del lado del cliente. Si nuestro programa se está ejecutando en Internet no sabemos en qué plataforma se ejecutará y además es necesario poner un cuidado adicional en no diseminar código que contenga errores. En estos casos, es necesario disponer de un lenguaje interplataforma y seguro, como por ejemplo, un lenguaje de *script* o Java.

Si nuestra aplicación se ejecuta en una intranet, es posible que el conjunto de restricciones sea distinto. No resulta extraño que todas las máquinas sean plataformas Intel/Windows. En una intranet, nosotros somos responsables de la calidad de nuestro propio código y podemos corregir los errores en el momento en que se descubran. Además, puede que ya dispongamos de una gran cantidad de código heredado que haya estado siendo utilizado en alguna arquitectura cliente/servidor más tradicional, en la que es necesario instalar físicamente los programas cliente cada vez que se lleva a cabo una actualización. El tiempo que se pierde a la hora de instalar actualizaciones es, precisamente, la principal razón para comenzar a utilizar exploradores, porque las actualizaciones son invisibles y automáticas (Java Web Start también constituye una solución a este problema). Si trabajamos en una intranet de este tipo, la solución más lógica consiste en seguir la ruta más corta que nos permita utilizar la base de código existente, en lugar de volver a escribir todos los programas en un nuevo lenguaje.

Al enfrentarse con este amplio conjunto de soluciones para los problemas de la programación del lado del cliente, el mejor plan de ataque consiste en realizar un análisis de coste-beneficio. Considere las restricciones que afectan a su problema y cuál sería la ruta más corta para encontrar una solución. Puesto que la programación del lado del cliente sigue siendo una programación en sentido tradicional, siempre resulta conveniente adoptar el enfoque de desarrollo más rápido en cada situación concreta. Ésta es la mejor manera de prepararse para los problemas que inevitablemente encontraremos a la hora de desarrollar los programas.

Programación del lado del servidor

En nuestro análisis, hemos ignorado hasta ahora la cuestión de la programación del lado del servidor, que es probablemente donde Java ha tenido su éxito más rotundo. ¿Qué sucede cuando enviamos una solicitud a un servidor? La mayor parte de las veces, la solicitud dice simplemente "Envíame este archivo". A continuación, el explorador interpreta el archivo de la forma apropiada, como página HTML, como imagen, como un *applet* de Java, como programa en lenguaje de *script*, etc.

Las solicitudes más complicadas dirigidas a los servidores suelen implicar una transacción de base de datos. Una situación bastante común consiste en enviar una solicitud para que se realice una búsqueda completa en una base de datos, encargándose a continuación el servidor de dar formato a los resultados como página HTML y enviar ésta al explorador (por supuesto, si el cliente dispone de un mayor grado de inteligencia, gracias a la utilización de Java o de un lenguaje de *script*, pueden enviarse los datos en bruto y formatearlos en el extremo cliente, lo que sería más rápido e impondría una menor carga de trabajo al servidor). Otro ejemplo: puede que queramos registrar nuestro nombre en una base de datos para unirnos a un grupo o realizar un pedido, lo que a su vez implica efectuar modificaciones en la base de datos. Estas solicitudes de base de datos deben procesarse mediante algún tipo de código situado en el lado del cliente; es a este tipo de programas a los que nos referimos a la hora de hablar de programación del lado del cliente. Tradicionalmente, la programación del lado del cliente se llevaba a cabo utilizando Perl, Python, C++, o algún otro lenguaje para crear programas CGI, pero con el tiempo se han desarrollado otros sistemas más sofisticados, entre los que se incluyen los servidores web basados en Java que permiten realizar todas las tareas de programación del lado del servidor en lenguaje Java, escribiendo lo que se denomina *servlets*. Los *servlets* y sus descendientes, las páginas JSP, son dos de las principales razones por las que las empresas que desarrollan sitios web están adoptando Java, especialmente porque dichas tecnologías eliminan los problemas derivados de tratar

con exploradores que dispongan de capacidades diferentes. Los temas de programación del lado del servidor se tratan en *Thinking in Enterprise Java* en el sitio web www.MindView.net.

A pesar de todo lo que hemos comentado acerca de Java y de Internet, Java es un lenguaje de programación de propósito general, que permite resolver los mismos tipos de problemas que podemos resolver con otros lenguajes. En este sentido, la ventaja de Java no radica sólo en su portabilidad, sino también en su programabilidad, su robustez, su amplia biblioteca estándar y las numerosas bibliotecas de otros fabricantes que ya están disponibles y que continúan siendo desarrolladas.

Resumen

Ya sabemos cuál es el aspecto básico de un programa procedimental: definiciones de datos y llamadas a funciones. Para comprender uno de esos programas es preciso analizarlo, examinando las llamadas a función y utilizando conceptos de bajo nivel con el fin de crear un modelo mental del programa. Ésta es la razón por la que necesitamos representaciones intermedias a la hora de diseñar programas procedimentales: en sí mismos, estos programas tienden a ser confusos, porque se utiliza una forma de expresarse que está más orientada hacia la computadora que hacia el programa que se trata de resolver.

Como la programación orientada a objetos añade numerosos conceptos nuevos, con respecto a los que podemos encontrar en un lenguaje procedural, la intuición nos dice que el programa Java resultante será más complicado que el programa procedural equivalente. Sin embargo, la realidad resulta gratamente sorprendente: un programa Java bien escrito es, generalmente, mucho más simple y mucho más fácil de comprender que un programa procedural. Lo que podemos ver al analizar el programa son las definiciones de los objetos que representan los conceptos de nuestro espacio de problema (en lugar de centrarse en la representación realizada dentro de la máquina), junto con mensajes que se envían a esos objetos para representar las actividades que tienen lugar en ese espacio de problema. Uno de los atractivos de la programación orientada a objetos, es que con un programa bien diseñado resulta fácil comprender el código sin más que leerlo. Asimismo, suele haber una cantidad de código bastante menor, porque buena parte de los problemas puede resolverse reutilizando código de las bibliotecas existentes.

La programación orientada a objetos y el lenguaje Java no resultan adecuados para todas las situaciones. Es importante evaluar cuáles son nuestras necesidades reales y determinar si Java permitirá satisfacerlas de forma óptima o si, por el contrario, es mejor emplear algún otro sistema de programación (incluyendo el que en la actualidad estemos usando). Si podemos estar seguros de que nuestras necesidades van a ser bastante especializadas en un futuro próximo, y si estamos sujetos a restricciones específicas que Java pueda no satisfacer, resulta recomendable investigar otras alternativas (en particular, mi recomendación sería echarle un vistazo a Python; véase www.Python.org). Si decide, a pesar de todo, utilizar el lenguaje Java, al menos comprenderá, después de efectuado ese análisis, cuáles serán las opciones existentes y por qué resultaba conveniente adoptar la decisión que finalmente haya tomado.

2

Todo es un objeto

“Si habláramos un lenguaje diferente, percibiríamos un mundo algo distinto”.

Ludwig Wittgenstein (1889-1951)

Aunque está basado en C++, Java es un lenguaje orientado a objetos más “puro”.

Tanto C++ como Java son lenguajes híbridos, pero en Java los diseñadores pensaron que esa hibridación no era tan importante como en C++. Un lenguaje híbrido permite utilizar múltiples estilos de programación; la razón por la que C++ es capaz de soportar la compatibilidad descendente con el lenguaje C. Puesto que C++ es un superconjunto del lenguaje C, incluye muchas de las características menos deseables de ese lenguaje, lo que hace que algunos aspectos del C++ sean demasiado complicados.

El lenguaje Java presupone que el programador sólo quiere realizar programación orientada a objetos. Esto quiere decir que, antes de empezar, es preciso cambiar nuestro esquema mental al del mundo de la orientación a objetos (a menos que ya hayamos efectuado esa transición). La ventaja que se obtiene gracias a este esfuerzo adicional es la capacidad de programar en un lenguaje que es más fácil de aprender y de utilizar que muchos otros lenguajes orientados a objetos. En este capítulo veremos los componentes básicos de un programa Java y comprobaremos que (casi) todo en Java es un objeto.

Los objetos se manipulan mediante referencias

Cada lenguaje de programación dispone de sus propios mecanismos para manipular los elementos almacenados en memoria. En ocasiones, el programador debe ser continuamente consciente del tipo de manipulación que se está efectuando. ¿Estamos tratando con el elemento directamente o con algún tipo de representación indirecta (un puntero en C o C++) , que haya que tratar con una sintaxis especial?

Todo esto se simplifica en Java. En Java, todo se trata como un objeto, utilizando una única sintaxis coherente. Aunque *tratamos* todo como un objeto, los identificadores que manipulamos son en realidad “referencias” a objetos.¹ Podríamos imaginarnos una TV (el objeto) y un mando a distancia (la referencia); mientras dispongamos de esta referencia tendremos una conexión con la televisión, pero cuando alguien nos dice “cambia de canal” o “baja el volumen”, lo que hacemos es manipular la referencia, que a su vez modifica el objeto. Si queremos movernos por la habitación y continuar controlando la TV, llevamos con nosotros el mando a distancia/referencia, no la televisión.

¹ Este punto puede suscitar enconados debates. Hay personas que sostienen que “claramente se trata de un puntero”, pero esto está presuponiendo una determinada implementación subyacente. Asimismo, las referencias en Java se parecen mucho más sintácticamente a las referencias C++ que a los punteros. En la primera edición de este libro decidí utilizar el término “descriptor” porque las referencias C++ y las referencias Java tienen diferencias notables. Yo mismo provenía del mundo del lenguaje C++ y no quería confundir a los programadores de C++, que suponía que constituirían la gran mayoría de personas interesadas en el lenguaje Java. En la segunda edición, decidí que “referencia” era el término más comúnmente utilizado, y que cualquiera que proveniera del mundo de C++ iba a enfrentarse a problemas mucho más graves que la terminología de las referencias, por lo que no tenía sentido usar una palabra distinta. Sin embargo, hay personas que están en desacuerdo incluso con el término “referencia”. En un determinado libro, pude leer que “resulta completamente equivocado decir que Java soporta el paso por referencia”, o que los identificadores de los objetos Java (de acuerdo con el autor del libro) son en realidad “referencias a objetos”. Por lo que (continúa el autor) todo se pasa *en la práctica* por valor. Según este autor, no se efectúa un paso por referencia, sino que se “pasa una referencia a objeto por valor”. Podríamos discutir acerca de la precisión de estas complicadas explicaciones, pero creo que el enfoque que he adoptado en este libro simplifica la compresión del concepto sin generar ningún tipo de problema (los puristas del lenguaje podrían sostener que estoy mintiendo, pero a eso respondería que lo que estoy haciendo es proporcionar una abstracción apropiada).

Asimismo, el mando a distancia puede existir de manera independiente, sin necesidad de que exista una televisión. En otras palabras, el hecho de que dispongamos de una referencia no implica necesariamente que haya un objeto conectado a la misma. De este modo, si queremos almacenar una palabra o una frase podemos crear una referencia de tipo **String**:

```
String s;
```

Pero con ello *sólo* habremos creado la referencia a un objeto. Si decidíramos enviar un mensaje a **s** en este punto, obtendríamos un error porque **s** no está asociado a nada (no hay televisión). Por tanto, una práctica más segura consiste en inicializar siempre las referencias en el momento de crearlas:

```
String s = "asdf";
```

Sin embargo, aquí estamos utilizando una característica especial de Java. Las cadenas de caracteres pueden inicializarse con un texto entre comillas. Normalmente, será necesario emplear un tipo más general de inicialización para los restantes objetos.

Es necesario crear todos los objetos

Al crear una referencia, lo que se desea es conectarla con un nuevo objeto. Para ello, en general, se emplea el operador **new**. La palabra clave **new** significa: “Crea un nuevo ejemplar de este tipo de objeto”. Por tanto, en el ejemplo anterior podríamos escribir:

```
String s = new String("asdf");
```

Esto no sólo dice: “Crea un nuevo objeto **String**”, sino que también proporciona información acerca de *cómo* crear el objeto suministrando una cadena de caracteres inicial.

Por supuesto, Java incluye una pléthora de tipos predefinidos, además de **String**. Lo más importante es que también podemos crear nuestros propios tipos. De hecho, la creación de nuevos tipos es la actividad fundamental en la programación Java, y eso es precisamente lo que aprenderemos a hacer en el resto del libro.

Los lugares de almacenamiento

Resulta útil tratar de visualizar la forma en que se dispone la información mientras se ejecuta el programa; en particular, es muy útil ver cómo está organizada la memoria. Disponemos de cinco lugares distintos en los que almacenar los datos:

- 1. Registros.** Se trata del tipo de almacenamiento más rápido, porque se encuentra en un lugar distinto al de los demás tipos de almacenamiento: dentro del procesador. Sin embargo, el número de registros está muy limitado, por lo que los registros se asignan a medida que son necesarios. No disponemos de un control directo sobre los mismos, ni tampoco podremos encontrar en los programas que los registros ni siquiera existan (C y C++, por el contrario, permiten sugerir al compilador que el almacenamiento se haga en un registro).
- 2. La pila.** Esta zona se encuentra en el área general de memoria de acceso aleatorio (RAM), pero el procesador proporciona un soporte directo para la pila, gracias al *puntero de pila*. El puntero de pila se desplaza hacia abajo para crear nueva memoria y hacia arriba para liberarla. Se trata de una forma extraordinariamente rápida y eficiente de asignar espacio de almacenamiento, sólo superada en rapidez por los registros. El sistema Java debe conocer, a la hora de crear el programa, el tiempo de vida exacto de todos los elementos que se almacenan en la pila. Esta restricción impone una serie de límites a la flexibilidad de los programas, por lo que aunque parte del almacenamiento dentro de Java se lleva a cabo en la pila (en concreto, las referencias a objetos), los propios objetos Java no se colocan nunca en la pila.
- 3. El círculo.** Es un área de memoria de propósito general (también situada dentro de la RAM) en la que se almacenan todos los objetos Java. El aspecto más atractivo del círculo de memoria, a diferencia de la pila, es que el compilador no necesita conocer de antemano durante cuánto tiempo se va a ocupar ese espacio de almacenamiento dentro del círculo. Por tanto, disponemos de un alto grado de flexibilidad a la hora de utilizar el espacio de almacenamiento que el círculo de memoria proporciona. Cada vez que hace falta un objeto, simplemente se escribe el código para crearlo utilizando **new**, y el espacio de almacenamiento correspondiente en el círculo se asigna en el momento de ejecutar el código. Por supuesto, esa flexibilidad tiene su precio: puede que sea necesario un tiempo más largo para asignar y liberar el espacio de almacenamiento del círculo de memoria, si lo comparamos con el tiempo necesario

para el almacenamiento en la pila (eso suponiendo que *pudiéramos* crear objetos en la pila en Java, al igual que se hace en C++).

4. **Almacenamiento constante.** Los valores constantes se suelen situar directamente dentro del código de programa, lo que resulta bastante seguro, ya que nunca varían. En ocasiones, las constantes se almacenan por separado, de forma que se pueden guardar opcionalmente en la memoria de sólo lectura (ROM, *read only memory*), especialmente en los sistemas integrados.²
5. **Almacenamiento fuera de la RAM.** Si los datos residen fuera del programa, podrán continuar existiendo mientras el programa no se esté ejecutando, sin que el programa tenga control sobre los mismos. Los dos ejemplos principales son los *objetos stream*, en los que los objetos se transforman en flujos de bytes, generalmente para enviarlos a otra máquina, los *objetos persistentes* en los que los objetos se almacenan en disco para que puedan conservar su estado incluso después de terminar el programa. El truco con estos tipos de almacenamiento consiste en transformar los objetos en algo que pueda existir en el otro medio de almacenamiento, y que, sin embargo, pueda recuperarse para transformarlo en un objeto normal basado en RAM cuando sea necesario. Java proporciona soporte para lo que se denomina *persistencia ligera* y otros mecanismos tales como JDBC e Hibernate proporcionan soporte más sofisticado para almacenar y extraer objetos utilizando bases de datos.

Caso especial: tipos primitivos

Hay un grupo de tipos que se emplean muy a menudo en programación y que requieren un tratamiento especial. Podemos considerarlos como tipos “primitivos”. La razón para ese tratamiento especial es que crear un objeto con **new**, una variable simple de pequeño tamaño, no resulta muy eficiente, porque **new** almacena los objetos en el círculo de memoria. Para estos tipos primitivos, Java utiliza la técnica empleada en Cy C++; es decir, en lugar de crear la variable con **new**, se crea una variable “automática” que *no es una referencia*. La variable almacena el valor directamente y se coloca en la pila, por lo que resulta mucho más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no cambian de una arquitectura de máquina a otra, a diferencia de lo que sucede en la mayoría de los lenguajes. Esta invariabilidad de los tamaños es una de las razones por la que los programas Java son más portables que los programas escritos en la mayoría de los demás lenguajes.

| Tipo primitivo | Tamaño | Mínimo | Máximo | Tipo envoltorio |
|----------------|---------|-----------|--------------------|------------------|
| boolean | — | — | — | Boolean |
| char | 16 bits | Unicode 0 | Unicode $2^{16}-1$ | Character |
| byte | 8 bits | -128 | +127 | Byte |
| short | 16 bits | -2^{15} | $+2^{15}-1$ | Short |
| int | 32 bits | -2^{31} | $+2^{31}-1$ | Integer |
| long | 64 bits | -2^{63} | $+2^{63}-1$ | Long |
| float | 32 bits | IEEE754 | IEEE754 | Float |
| double | 64 bits | IEEE754 | IEEE754 | Double |
| void | — | — | — | Void |

Todos los tipos numéricos tienen signo, por lo que Java no podrá encontrar ningún tipo sin signo.

El tamaño del tipo **boolean** no está especificado de manera explícita; tan sólo se define para que sea capaz de aceptar los valores literales **true** o **false**.

Las clases “envoltorio” para los tipos de datos primitivos permiten definir un objeto no primitivo en el círculo de memoria que represente a ese tipo primitivo. Por ejemplo:

² Un ejemplo sería el conjunto de cadenas de caracteres. Todas las cadenas literales y constantes que tengan un valor de tipo carácter se toman de forma automática y se asignan a un almacenamiento estático especial.

```
char c = 'x';
Character ch = new Character(c);
```

O también podría emplear:

```
Character ch = new Character('x');
```

La característica de Java SE5 denominada *autoboxing* permite realizar automáticamente la conversión de un tipo primitivo a un tipo envoltorio:

```
Character ch = 'x';
```

Y a la inversa:

```
char c = ch;
```

En un capítulo posterior veremos las razones que existen para utilizar envoltorios con los tipos primitivos.

Aritmética de alta precisión

Java incluye dos clases para realizar operaciones aritméticas de alta precisión: **BigInteger** y **BigDecimal**. Aunque estas clases caen aproximadamente en la misma categoría que las clases envoltorio, ninguna de las dos dispone del correspondiente tipo primitivo.

Ambas clases disponen de métodos que proporcionan operaciones análogas a las que se realizan con los tipos primitivos. Es decir, podemos hacer con un objeto **BigInteger** o **BigDecimal** cualquier cosa que podamos hacer con una variable **int** o **float**; simplemente deberemos utilizar llamadas a métodos en lugar de operadores. Asimismo, como son más complejas, las operaciones se ejecutarán más lentamente. En este caso, sacrificamos parte de la velocidad en aras de la precisión.

BigInteger soporta números enteros de precisión arbitraria. Esto quiere decir que podemos representar valores enteros de cualquier tamaño sin perder ninguna información en absoluto durante las operaciones.

BigDecimal se utiliza para números de coma fija y precisión arbitraria. Podemos utilizar estos números, por ejemplo, para realizar cálculos monetarios precisos.

Consulte la documentación del kit JDK para conocer más detalles acerca de los constructores y métodos que se pueden invocar para estas dos clases.

Matrices en Java

Casi todos los lenguajes de programación soportan algún tipo de matriz. La utilización de matrices en C y C++ es peligrosa, porque dichas matrices son sólo bloques de memoria. Si un programa accede a la matriz fuera de su correspondiente bloque de memoria o utiliza la memoria antes de la inicialización (lo cual son dos errores de programación comunes), los resultados serán impredecibles.

Uno de los principales objetivos de Java es la seguridad, por lo que en Java no se presentan muchos de los problemas que aquejan a los programadores en C y C++. Se garantiza que las matrices Java siempre se inicialicen y que no se pueda acceder a ellas fuera de su rango autorizado. Las comprobaciones de rango exigen pagar el precio de gastar una pequeña cantidad de memoria adicional para cada matriz, así como de verificar el índice en tiempo de ejecución, pero se supone que el incremento en productividad y la mejora de la seguridad compensan esas desventajas (y Java puede en ocasiones optimizar estas operaciones).

Cuando se crea una matriz de objetos, lo que se crea realmente es una matriz de referencias, y cada una de esas matrices se inicializa automáticamente con un valor especial que tiene su propia palabra clave: **null**. Cuando Java se encuentra un valor **null**, comprende que la referencia en cuestión no está apuntando a ningún objeto. Es necesario asignar un objeto a cada referencia antes de utilizarla, y si se intenta usar una referencia que siga teniendo el valor **null**, se informará del error en tiempo de ejecución. De este modo, en Java se evitan los errores comunes relacionados con las matrices.

También podemos crear una matriz de valores primitivos. De nuevo, el compilador se encarga de garantizar la inicialización, rellenando con ceros la memoria correspondiente a dicha matriz.

Hablaremos con detalle de las matrices en capítulos posteriores.

Nunca es necesario destruir un objeto

En la mayoría de los lenguajes de programación, el concepto de tiempo de vida de una variable representa una parte significativa del esfuerzo de programación. ¿Cuánto va a durar la variable? Si hay que destruirla, ¿cuándo debemos hacerlo? La confusión en lo que respecta al tiempo de vida de las variables puede generar una gran cantidad de errores de programación, y en esta sección vamos a ver que Java simplifica enormemente este problema al encargarse de realizar por nosotros todas las tareas de limpieza.

Ámbito

La mayoría de los lenguajes procedimentales incluyen el concepto de *ámbito*. El ámbito determina tanto la visibilidad como el tiempo de vida de los nombres definidos dentro del mismo. En C, C++ y Java, el ámbito está determinado por la colocación de las llaves {}, de modo que, por ejemplo:

```
{
    int x = 12;
    // x
    {
        int q = 96;
        // están disponibles tanto x como q
    }
    // sólo está disponible x
    // q está "fuera del ámbito"
}
```

Una variable definida dentro de un ámbito sólo está disponible hasta que ese ámbito termina.

Todo texto situado después de los caracteres `//` y hasta el final de la línea es un comentario.

El sangrado hace que el código Java sea más fácil de leer. Dado que Java es un lenguaje de formato libre, los espacios, tabuladores y retornos de carro adicionales no afectan al programa resultante.

No podemos hacer lo siguiente, a pesar de que sí es correcto en C y C++:

```
{
    int x = 12;
    {
        int x = 96; // Illegal
    }
}
```

El compilador nos indicaría que la variable `x` ya ha sido definida. Por tanto, no está permitida la posibilidad en C y C++ de “ocultar” una variable en un ámbito más grande, porque los diseñadores de Java pensaron que esta característica hacia los programas más confusos.

Ámbito de los objetos

Los objetos Java no tienen el mismo tiempo de vida que las primitivas. Cuando se crea un objeto Java usando `new`, ese objeto continúa existiendo una vez que se ha alcanzado el final del ámbito. Por tanto, si usamos:

```
{
    String s = new String("a string");
} // Fin del ámbito
```

la referencia `s` desaparece al final del ámbito. Sin embargo, el objeto `String` al que `s` estaba apuntando continuará ocupando memoria. En este fragmento de código, no hay forma de acceder al objeto después de alcanzar el final del ámbito, porque la única referencia a ese objeto está ahora fuera de ámbito. En capítulos posteriores veremos cómo pasar y duplicar una referencia a un objeto durante la ejecución de un programa.

Como los objetos que creamos con `new` continuarán existiendo mientras queramos, hay toda una serie de problemas típicos de programación en C++ que en Java se desvanecen. En C++ no sólo hay que asegurarse de que los objetos permanezcan mientras sean necesarios, sino que también es preciso destruirlos una vez que se ha terminado de usarlos.

Esto suscita una cuestión interesante. Si los objetos continúan existiendo en Java perpetuamente, ¿qué es lo que evita llenar la memoria y hacer que el programa se detenga? Éste es exactamente el tipo de problema que podía ocurrir en C++, y para resolverlo Java recurre a una especie de varita mágica. Java dispone de lo que se denomina *depurador de memoria*, que examina todos los objetos que hayan sido creados con `new` y determina cuáles no tienen ya ninguna referencia que les apunte. A continuación, Java libera la memoria correspondiente a esos objetos, de forma que pueda ser utilizada para crear otros objetos nuevos. Esto significa que nunca es necesario preocuparse de reclamar explícitamente la memoria. Basta con crear los objetos y, cuando dejen de ser necesarios, se borrarán automáticamente. Esto elimina un cierto tipo de problema de programación: las denominadas “fugas de memoria” que se producen cuando, en otros lenguajes, un programador se olvida de liberar la memoria.

Creación de nuevos tipos de datos: `class`

Si todo es un objeto, ¿qué es lo que determina cómo se comporta y qué aspecto tiene una clase concreta de objeto? Dicho de otro modo, ¿qué es lo que establece el *tipo* de un objeto? Cabría esperar que existiera una palabra clave denominada “type” (tipo), y de hecho tendría bastante sentido. Históricamente, sin embargo, la mayoría de los lenguajes orientados a objetos han utilizado la palabra clave `class` para decir “voy a definir cuál es el aspecto de un nuevo tipo de objeto”. La palabra clave `class` (que es tan común que la escribiremos en negrita normalmente a lo largo del libro) va seguida del nombre del nuevo tipo que queremos definir. Por ejemplo:

```
class ATypename { /* Aquí iría el cuerpo de la clase */ }
```

Este código permite definir un nuevo tipo, aunque en este caso el cuerpo de la clase sólo incluye un comentario (las barras inclinadas y los asteriscos junto con el texto forman el comentario, lo que veremos en detalle más adelante en este capítulo), así que no es mucho lo que podemos hacer con esta clase que acabamos de definir. Sin embargo, sí que podemos crear un objeto de este tipo utilizando `new`:

```
ATypename a = new ATypename();
```

Si bien es verdad que no podemos hacer que ese objeto lleve a cabo ninguna tarea útil (es decir, no le podemos enviar ningún mensaje interesante) hasta que definamos algunos métodos para ese objeto.

Campos y métodos

Cuando se define una clase (y todo lo que se hace en Java es definir clases, crear objetos de esa clase y enviar mensajes a dichos objetos), podemos incluir dos tipos de elementos dentro de la clase: *campos* (algunas veces denominados *miembros de datos*) y *métodos* (en ocasiones denominados *funciones miembro*). Un campo es un objeto de cualquier tipo con el que podemos comunicarnos a través de su referencia o bien un tipo primitivo. Si es una referencia a un objeto, hay que inicializar esa referencia para conectarla con un objeto real (usando `new`, como hemos visto anteriormente).

Cada objeto mantiene su propio almacenamiento para sus campos; los campos normales no son compartidos entre los distintos objetos. Aquí tiene un ejemplo de una clase con algunos campos definidos:

```
class DataOnly {
    int i;
    double d;
    boolean b;
}
```

Esta clase no *hace* nada, salvo almacenar una serie de datos. Sin embargo, podemos crear un objeto de esta clase de la forma siguiente:

```
DataOnly data = new DataOnly();
```

Podemos asignar valores a los campos, pero primero es preciso saber cómo referirnos a un miembro de un objeto. Para referirnos a él, es necesario indicar el nombre de la referencia al objeto, seguido de un punto y seguido del nombre del miembro concreto dentro del objeto:

```
objectReference.member
```

Por ejemplo:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

También es posible que el objeto contenga otros objetos, que a su vez contengan los datos que queremos modificar. En este caso, basta con utilizar los puntos correspondientes, como por ejemplo:

```
myPlane.leftTank.capacity = 100;
```

La clase **DataOnly** no puede hacer nada más que almacenar datos, ya que no dispone de ningún método. Para comprender cómo funcionan los métodos es preciso entender primero los conceptos de *argumentos* y *valores de retorno*, que vamos a describir en breve.

Valores predeterminados de los miembros de tipo primitivo

Cuando hay un tipo de datos primitivo como miembro de una clase, Java garantiza que se le asignará un valor predeterminado en caso de que no se inicialice:

| Tipo primitivo | Valor predeterminado |
|----------------|----------------------|
| boolean | false |
| char | '\u0000' (null) |
| byte | (byte)0 |
| short | (short)0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |

Los valores predeterminados son sólo los valores que Java garantiza cuando se emplea la variable *como miembro de una clase*. Esto garantiza que las variables miembro de tipo primitivo siempre sean inicializadas (algo que C++ no hace), reduciendo así una fuente de posibles errores. Sin embargo, este valor inicial puede que no sea correcto o ni siquiera legal para el programa que se esté escribiendo. Lo mejor es inicializar las variables siempre de forma explícita.

Esta garantía de inicialización no se aplica a las *variables locales*, aquellas que no son campos de clase. Por tanto, si dentro de la definición de un método tuviéramos:

```
int x;
```

Entonces **x** adoptaría algún valor arbitrario (como en C y C++), no siendo automáticamente inicializada con el valor cero. El programador es el responsable de asignar un valor apropiado antes de usar **x**. Si nos olvidamos, Java representa de todos modos una mejora con respecto a C++, ya que se obtiene un error en tiempo de compilación que nos informa de que la variable puede no haber sido inicializada (muchos compiladores de C++ nos advierten acerca de la existencia de variables no inicializadas, pero en Java esta falta de inicialización constituye un error).

Métodos, argumentos y valores de retorno

En muchos lenguajes (como C y C++), el término *función* se usa para describir una subrutina con nombre. El término más comúnmente utilizado en Java es el de *método*, queriendo hacer referencia a una forma de "llevar algo a cabo". Si lo desea, puede continuar pensando en términos de funciones; se trata sólo de una diferencia sintáctica, pero en este libro adoptaremos el uso común del término "método" dentro del mundo de Java.

Los métodos de Java determinan los mensajes que un objeto puede recibir. Las partes fundamentales de un método son el nombre, los argumentos, el tipo de retorno y el cuerpo. Ésta sería la forma básica de un método:

```
TipoRetorno NombreMetodo( /* Lista de argumentos */ ) {
    /* Cuerpo del método */
}
```

El tipo de retorno describe el valor devuelto por el método después de la invocación. La lista de argumentos proporciona la lista de los tipos y nombres de la información que hayamos pasado al método. El nombre del método y la lista de argumentos (que forman lo que se denomina *signatura* del método) identifican de forma unívoca al método.

Los métodos pueden crearse en Java únicamente como parte de una clase. Los métodos sólo se pueden invocar para un objeto³, y dicho objeto debe ser capaz de ejecutar esa invocación del método. Si tratamos de invocar un método correcto para un objeto obtendremos un mensaje de error en tiempo de compilación. Para invocar un método para un objeto, hay que nombrar el objeto seguido de un punto, seguido del nombre del método y de su lista de argumentos, como por ejemplo:

```
NombreObjeto.NombreMetodo(arg1, arg2, arg3);
```

Por ejemplo, suponga que tenemos un método `f()` que no tiene ningún argumento y que devuelve una valor de tipo `int`. Entonces, si tuviéramos un objeto denominado `a` para el que pudiera invocarse `f()` podríamos escribir:

```
int x = a.f();
```

El tipo del valor de retorno debe ser compatible con el tipo de `x`.

Este acto de invocar un método se denomina comúnmente *enviar un mensaje a un objeto*. En el ejemplo anterior, el mensaje es `f()` y el objeto es `a`. A menudo, cuando se quiere resumir lo que es la programación orientada a objetos se suele decir que consiste simplemente en “enviar mensajes a objetos”.

La lista de argumentos

La lista de argumentos del método especifica cuál es la información que se le pasa al método. Como puede suponerse, esta información (como todo lo demás en Java) adopta la forma de objetos. Por tanto, lo que hay que especificar en la lista de argumentos son los tipos de los objetos que hay pasar y el nombre que hay utilizar para cada uno. Como en cualquier otro caso dentro de Java en el que parece que estuviéramos gestionando objetos, lo que en realidad estaremos pasando son referencias⁴. Sin embargo, el tipo de la referencia debe ser correcto. Si el argumento es de tipo `String`, será preciso pasar un objeto `String`, porque de lo contrario el compilador nos daría un error.

Supongamos un cierto método que admite un objeto `String` como argumento. Para que la compilación se realice correctamente, la definición que habría que incluir dentro de la definición de la clase sería como la siguiente:

```
int storage(String s) {
    return s.length() * 2;
}
```

Este método nos dice cuántos bytes se requieren para almacenar la información contenida en un objeto `String` concreto (el tamaño de cada `char` en un objeto `String` es de 16 bits, o dos bytes, para soportar los caracteres Unicode). El argumento es de tipo `String` y se denomina `s`. Una vez que se pasa `s` al método, podemos tratarlo como cualquier otro objeto (podemos enviarle mensajes). Aquí, lo que se hace es invocar el método `length()`, que es uno de los métodos definidos para los objetos de tipo `String`; este método devuelve el número de caracteres que hay en una cadena.

También podemos ver en el ejemplo cómo se emplea la palabra clave `return`. Esta palabra clave se encarga de dos cosas: en primer lugar, quiere decir “sal del método, porque ya he terminado”, en segundo lugar, si el método ha generado un valor, ese valor se indica justo a continuación de una instrucción `return`, en este caso, el valor de retorno se genera evaluando la expresión `s.length() * 2`.

Podemos devolver un valor de cualquier tipo que deseemos, pero si no queremos devolver nada, tenemos que especificarlo, indicando que el método devuelve un valor de tipo `void`. He aquí algunos ejemplos:

³ Los métodos de tipo `static`, de los que pronto hablaremos, pueden invocarse para la clase, en lugar de para un objeto específico.

⁴ Con la excepción usual de los tipos de datos “especiales” que antes hemos mencionado: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`. En general, sin embargo, lo que se pasan son objetos; lo que realmente quiere decir que se pasan referencias a objetos.

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

Cuando el tipo de retorno es **void**, la palabra clave **return** se utiliza sólo para salir del método, por lo que resulta necesaria una vez que se alcanza el final del método. Podemos volver de un método en cualquier punto, pero si el tipo de retorno es distinto de **void**, entonces el compilador nos obligará (mediante los apropiados mensajes de error) a devolver un valor del tipo apropiado, independientemente del lugar en el que salgamos del método.

Llegados a este punto, podría parecer que un programa consiste, simplemente, en una serie de objetos con métodos que aceptan otros objetos como argumentos y envían mensajes a esos otros objetos. Realmente, esa descripción es bastante precisa, pero en el siguiente capítulo veremos cómo llevar a cabo el trabajo detallado de bajo nivel, tomando decisiones dentro de un método. Para los propósitos de este capítulo, el envío de mensajes nos resultará más que suficiente.

Construcción de un programa Java

Hay otras cuestiones que tenemos que entender antes de pasar a diseñar nuestro primer programa Java.

Visibilidad de los nombres

Uno de los problemas en cualquier lenguaje de programación es el de control de los nombres. Si utilizamos un nombre en un módulo de un programa y otro programador emplea el mismo nombre en otro módulo, ¿cómo podemos distinguir un nombre del otro y cómo podemos evitar la “colisión” de los dos nombres? En C, este problema es especialmente significativo, porque cada programa es, a menudo, un conjunto manejable de nombres. Las clases C++ (en las que se basan las clases Java) anidan las funciones dentro de clases para que no puedan colisionar con los nombres de función anidados dentro de otras clases. Sin embargo, C++ sigue permitiendo utilizar datos globales y funciones globales, así que las colisiones continúan siendo posibles. Para resolver este problema, C++ introdujo los denominados *espacios de nombres* utilizando palabras clave adicionales.

Java consiguió evitar todos estos problemas adoptando un enfoque completamente nuevo. Para generar un nombre no ambiguo para una biblioteca, los creadores de Java emplean los nombres de dominio de Internet en orden inverso, ya que se garantiza que los nombres de dominio son únicos. Puesto que mi nombre de dominio es **MindView.net**, una biblioteca de utilidades llamada foibles se denominaría, por ejemplo, **.net.mindview.utility.foibles**. Después del nombre de dominio invertido, los puntos tratan de representar subdirectorios.

En Java 1.0 y Java 1.1, las extensiones de dominio **com**, **edu**, **org**, **net**, etc., se escribían en mayúsculas por convenio, por lo que el nombre de la biblioteca sería **NET.mindview.utility.foibles**. Sin embargo, durante el desarrollo de Java 2 se descubrió que esto producía problemas, por lo que ahora los nombres completos de paquetes se escriben en minúsculas.

Este mecanismo implica que todos los archivos se encuentran, automáticamente, en sus propios espacios de nombres y que cada clase dentro de un archivo tiene un identificador único; el lenguaje se encarga de evitar automáticamente las colisiones de nombres.

Utilización de otros componentes

Cada vez que se quiera utilizar una clase predefinida en un programa, el compilador debería saber cómo localizarla. Por supuesto, puede que la clase ya exista en el mismo archivo de código fuente desde el que se la está invocando. En ese caso, basta con utilizar de manera directa la clase, aún cuando esa clase no esté definida hasta más adelante en el archivo (Java elimina los problemas denominados de “referencia anticipada”).

¿Qué sucede con las clases almacenadas en algún otro archivo? Cabría esperar que el compilador fuera lo suficientemente inteligente como para localizar la clase, pero hay un pequeño problema. Imagine que queremos emplear una clase con un nombre concreto, pero que existe más de una definición para dicha clase (probablemente, definiciones distintas). O peor, imagine que está escribiendo un programa y que a medida que lo escribe añade una nueva clase a la biblioteca que entra en conflicto con el nombre de otra clase ya existente.

Para resolver este problema, es preciso eliminar todas las ambigüedades potenciales. Esto se consigue informando al compilador Java de exactamente qué clases se desea emplear, utilizando para ello la palabra clave **import**. **import** le dice al compilador que cargue un paquete, que es una biblioteca de clases (en otros lenguajes, una biblioteca podría estar compuesta por funciones y datos, así como clases, pero recuerde que todo el código Java debe escribirse dentro de una clase).

La mayor parte de las veces utilizaremos componentes de las bibliotecas estándar Java incluidas con el compilador. Con estos componentes, no es necesario preocuparse sobre los largos nombres de dominio invertidos, basta con decir, por ejemplo:

```
import java.util.ArrayList;
```

para informar al compilador que queremos utilizar la clase **ArrayList** de Java. Sin embargo, **util** contiene diversas clases, y podemos usar varias de ellas sin declararlas todas ellas explícitamente. Podemos conseguir esto fácilmente utilizando ****** como comodín:

```
import java.util.*;
```

Resulta más común importar una colección de clases de esta forma que importar las clases individualmente.

La palabra clave static

Normalmente, cuando creamos una clase, lo que estamos haciendo es describir el aspecto de los objetos de esa clase y el modo en que éstos van a comportarse. En la práctica, no obtenemos ningún objeto hasta que creamos uno empleando **new**, que es el momento en que se asigna el almacenamiento y los métodos del objeto pasan a estar disponibles.

Existen dos situaciones en las que esta técnica no resulta suficiente. Una de ellas es cuando deseamos disponer de un único espacio de almacenamiento para un campo concreto, independientemente del número de objetos de esa clase que se cree, o incluso si no se crea ningún objeto de esa clase. La otra situación es cuando hace falta un método que no esté asociado con ningún objeto concreto de esa clase; en otras palabras, cuando necesitamos un método al que podamos invocar incluso aunque no se cree ningún objeto.

Podemos conseguir ambos efectos utilizando la palabra clave **static**. Cuando decimos que algo es **static**, quiere decir que ese campo o método concretos no están ligados a ninguna instancia concreta de objeto de esa clase. Por tanto, aún cuando nunca creáramos un objeto de esa clase, podríamos de todos modos invocar el método **static** o acceder a un campo **static**. Con los campos y métodos normales, que no tienen el atributo **static**, es preciso crear un objeto y usar ese objeto para acceder al campo o al método, ya que los campos y métodos que no son de tipo **static** deben conocer el objeto en particular con el que estén trabajando.⁵

Algunos lenguajes orientados a objetos utilizan los términos *datos de la clase y métodos de la clase*, para indicar que esos datos y métodos sólo existen para la clase como un todo, y no para ningún objeto concreto de esa clase. En ocasiones, en la literatura técnica relacionada con Java también se utilizan esos términos.

Para hacer que un campo o un método sea **static**, basta con colocar dicha palabra clave antes de la definición. Por ejemplo, el siguiente código generaría un campo **static** y le inicializaría:

```
class StaticTest {
    static int i = 47;
}
```

Ahora, aunque creáramos dos objetos **StaticTest**, existiría un único espacio de almacenamiento para **StaticTest.i**. Ambos objetos compartirían el mismo campo **i**. Considere el fragmento de código siguiente:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

En este punto, tanto **st1.i** como **st2.i** tienen el mismo valor, 47, ya que ambos hacen referencia a la misma posición de memoria.

⁵ Por supuesto, puesto que los métodos **static** no necesitan que se cree ningún objeto antes de utilizarlos, no pueden acceder *directamente* a ningún miembro o método que no sea **static**, por el procedimiento de invocar simplemente esos otros miembros sin hacer referencia a un objeto nominado (ya que los miembros y métodos que no son **static** deben estar asociados con un objeto concreto).

Hay dos formas de hacer referencia a una variable **static**. Como se indica en el ejemplo anterior, se la puede expresar a través de un objeto, escribiendo por ejemplo, **st2.i**. Podemos hacer referencia a ella directamente a través del nombre de la clase, lo que no puede hacerse con ningún nombre que no sea de tipo **static**.

```
StaticTest.i++;
```

El operador **++** incrementa en una unidad la variable. En este punto, tanto **st1.i** como **st2.i** tendrán el valor 48.

La forma preferida de hacer referencia a una variable **static** es utilizando el nombre de la clase. Esto no sólo permite enfatizar la naturaleza de tipo **static** de la variable, sino que en algunos casos proporciona al compilador mejores oportunidades para la optimización.

A los métodos **static** se les aplica una lógica similar. Podemos hacer referencia a un método **static** a través de un objeto, al igual que con cualquier otro método, o bien mediante la sintaxis adicional especial **NombreClase.método()**. Podemos definir un método **static** de manera similar:

```
class Incrementable {
    static void increment() { StaticTest.i++; }
}
```

Podemos ver que el método **increment()** de **Incrementable** incrementa el dato **i** de tipo **static** utilizando el operador **++**. Podemos invocar **increment()** de la forma típica a través de un objeto:

```
Incrementable sf = new Incrementable();
sf.increment();
```

O, dado que **increment()** es un método **static**, podemos invocarlo directamente a través de su clase:

```
Incrementable.increment();
```

Aunque **static**, cuando se aplica a un campo, modifica de manera evidente la forma en que se crean los datos (se crea un dato global para la clase, a diferencia de los campos que no son **static**, para los que se crea un campo para cada objeto), cuando se aplica esa palabra clave a un método no es tan dramático. Un uso importante de **static** para los métodos consiste en permitirnos invocar esos métodos sin necesidad de crear un objeto. Esta característica resulta esencial, como veremos, al definir el método **main()**, que es el punto de entrada para ejecutar una aplicación.

Nuestro primer programa Java

Finalmente, vamos a ver nuestro primer programa completo. Comenzaremos imprimiendo una cadena de caracteres y a continuación la fecha, empleando la clase **Date** de la biblioteca estándar de Java.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

Al principio de cada archivo de programa, es necesario incluir las instrucciones **import** necesarias para cargar las clases adicionales que se necesiten para el código incluido en ese archivo. Observe que hemos dicho "clases adicionales". La razón es que existe una cierta biblioteca de clases que se carga automáticamente en todo archivo Java: **java.lang**. Inicie el explorador web y consulte la documentación de Sun (si no ha descargado la documentación del kit JDK de <http://java.sun.com>, hágalo ahora).⁶ Observe que esta documentación no se suministra con el JDK; es necesario descargarla por separado). Si consulta la lista de paquetes, podrá ver todas las diferentes bibliotecas de clases incluidas con Java. Seleccione **java.lang**; esto hará que se muestre una lista de todas las clases que forman parte de esa biblioteca. Puesto que **java.lang** está implícitamente

⁶ El compilador Java y la documentación suministrados por Sun tienden a cambiar de manera frecuente, y el mejor lugar para obtenerlos es directamente en el sitio web de Sun. Descargándose esos archivos podrá disponer siempre de la versión más reciente.

en todo archivo de código Java, dichas clases estarán disponibles de manera automática. No existe ninguna clase **Date** en **java.lang**, lo que quiere decir que es preciso importar otra biblioteca para usar dicha clase. Si no sabe la biblioteca en la que se encuentra una clase concreta o si quiere ver todas las clases disponibles, puede seleccionar “Tree” en la documentación de Java. Con eso, podrá localizar todas las clases incluidas con Java. Entonces, utilice la función “Buscar” del explorador para encontrar **Date**. Cuando lo haga, podrá ver que aparece como **java.util.Date**, lo que nos permite deducir que se encuentra en la biblioteca **util** y que es necesario emplear la instrucción **import java.util.*** para usar **Date**.

Si vuelve atrás y selecciona **java.lang** y luego **System**, podrá ver que la clase **System** tiene varios campos; si selecciona **out**, descubrirá que se trata de un objeto **static PrintStream**. Puesto que es de tipo **static**, no es necesario crear ningún objeto empleando **new**. El objeto **out** está siempre disponible, por lo que podemos usarlo directamente. Lo que puede hacerse con este objeto **out** está determinado por su tipo: **PrintStream**. En la descripción se muestra **PrintStream** como un hipervínculo, por lo que al hacer clic sobre él podrá acceder a una lista de todos los métodos que se pueden invocar para **PrintStream**. Son bastantes métodos, y hablaremos de ellos más adelante en el libro. Por ahora, el único que nos interesa es **println()**, que en la práctica significa “imprime en la consola lo que te voy a pasar y termina con un carácter de avance de línea”. Así, en cualquier programa Java podemos escribir algo como esto:

```
System.out.println("Una cadena de caracteres");
```

cuando queramos mostrar información a través de la consola.

El nombre de la clase coincide con el nombre del archivo. Cuando estemos creando un programa independiente como éste, una de las clases del archivo deberá tener el mismo nombre que el propio archivo (el compilador se quejará si no lo hacemos así). Dicha clase debe contener un método denominado **main()** con la siguiente firma y tipo de retorno:

```
public static void main(String[] args) {
```

La palabra clave **public** quiere decir que el método está disponible para todo el mundo (lo que describiremos en detalle en el Capítulo 6, *Control de acceso*). El argumento de **main()** es una matriz de objetos **String**. En este programa, no se emplean los argumentos (**args**), pero el compilador Java insiste en que se incluya ese parámetro, ya que en él se almacenarán los argumentos de la línea de comandos.

La línea que imprime la fecha es bastante interesante:

```
System.out.println(new Date());
```

El argumento es un objeto **Date** que sólo se ha creado para enviar su valor (que se convierte automáticamente al tipo **String**) a **println()**. Una vez finalizada esta instrucción, dicho objeto **Date** resulta innecesario, y el depurador de memoria podrá encargarse de él en cualquier momento. Nosotros no necesitamos preocuparnos de borrar el objeto.

Al examinar la documentación del JDK en <http://java.sun.com>, podemos ver que **System** tiene muchos métodos que nos permiten producir muchos efectos interesantes (una de las características más potentes de Java es su amplio conjunto de bibliotecas estándar). Por ejemplo:

```
//: object>ShowProperties.java

public class ShowProperties {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
        System.out.println(System.getProperty("user.name"));
        System.out.println(
            System.getProperty("java.library.path"));
    }
} //:-
```

La primera linea de **main()** muestra todas las “propiedades” del sistema en que se está ejecutando el programa, por lo que nos proporciona la información del entorno. El método **list()** envía los resultados a su argumento, **System.out**. Más adelante en el libro veremos que los resultados pueden enviarse a cualquier parte, como por ejemplo, a un archivo. También podemos consultar una propiedad específica, por ejemplo, en este caso, el nombre de usuario y **java.library.path** (un poco más adelante explicaremos los comentarios bastante poco usuales situados al principio y al final de este fragmento de código).

Compilación y ejecución

Para compilar y ejecutar este programa, y cualquier otro programa de este libro, en primer lugar hay que tener un entorno de programación Java. Hay disponibles diversos entornos de programación Java de otros fabricantes, pero en el libro vamos a suponer que el lector está utilizando el kit de desarrollo JDK (Java Developer's Kit) de Sun, el cual es gratuito. Si está utilizando otro sistema de desarrollo⁷, tendrá que consultar la documentación de dicho sistema para ver cómo compilar y ejecutar los programas.

Acceda a Internet y vaya a <http://java.sun.com>. Allí podrá encontrar información y vínculos que le llevarán a través del proceso de descarga e instalación del JDK para su plataforma concreta.

Una vez instalado el JDK, y una vez modificada la información de ruta de la computadora para que ésta pueda encontrar `javac` y `java`, descargue y desempaque el código fuente correspondiente a este libro (puede encontrarlo en www.MindView.net). Esto creará un subdirectorio para cada capítulo del libro. Acceda al subdirectorio `objects` y escriba:

```
javac HelloDate.java
```

Este comando no debería producir ninguna respuesta. Si obtiene algún tipo de mensaje de error querrá decir que no ha instalado el JDK correctamente y tendrá que investigar cuál es el problema.

Por el contrario, si lo único que puede ver después de ejecutar el comando es una nueva línea de comandos, podrá escribir:

```
java HelloDate
```

y obtendrá el mensaje y la fecha como salida.

Puede utilizar este proceso para compilar y ejecutar cada uno de los programas de este libro. Sin embargo, podrá ver que el código fuente de este libro también dispone de un archivo denominado `build.xml` en cada capítulo, que contiene comandos "Ant" para construir automáticamente los archivos correspondientes a ese capítulo. Los archivos de generación y Ant (incluyendo las instrucciones para descargarlos) se describen más en detalle en el suplemento que podrá encontrar en <http://www.MindView.net/Books/BetterJava>, pero una vez que haya instalado Ant (desde <http://jakarta.apache.org/ant>) basta con que escriba '`ant`' en la línea de comandos para compilar y ejecutar los programas de cada capítulo. Si no ha instalado Ant todavía, puede escribir los comandos `javac` y `java` a mano.

Comentarios y documentación embebida

Existen dos tipos de comentarios en Java. El primero de ellos es el comentario de estilo C heredado de C++. Estos comentarios comienzan con `/*` y continúan, posiblemente a lo largo de varias líneas, hasta encontrar `*/`. Muchos programadores empiezan cada línea de un comando multilínea con un `*`, por lo que resulta bastante común ver comentarios como éste:

```
/* Este comentario
 * ocupa varias
 * líneas
 */
```

Recuerde, sin embargo, que en todo lo que hay entre `/*` y `*/` se ignora, por lo que no habría ninguna diferencia si dijéramos:

```
/* Este comentario ocupa
varias líneas */
```

El segundo tipo de comentario proviene de C++. Se trata del comentario monolínea que comienza con `//` y continúa hasta el final de la línea. Este tipo de comentario es bastante cómodo y se suele utilizar a menudo debido a su sencillez. No es necesario desplazarse de un sitio a otro del teclado para encontrar el carácter `/` y luego el carácter `*` (en su lugar, se pulsa la misma tecla dos veces), y tampoco es necesario cerrar el comentario. Así que resulta bastante habitual encontrar comentarios de este estilo:

```
// Éste es un comentario monolínea.
```

⁷ El compilador "jikes" de IBM es una alternativa bastante común, y es significativamente más rápido que Java C de Sun (aunque se están construyendo grupos de archivos utilizando Ant, la diferencia no es muy significativa). También hay diversos proyectos de código fuente abierto para crear compiladores Java, entornos de ejecución y bibliotecas.

Documentación mediante comentarios

Possiblemente el mayor problema de la tarea de documentar el código es el de mantener dicha documentación. Si la documentación y el código están separados, resulta bastante tedioso modificar la documentación cada vez que se cambia el código. La solución parece simple: integrar el código con la documentación. La forma más fácil de hacer esto es incluir todo en un mismo archivo. Sin embargo, para ello es necesario disponer de una sintaxis especial de comentarios que permita marcar la documentación, así como de una herramienta para extraer dichos comentarios y formatearlos de manera útil. Esto es precisamente lo que Java ha hecho.

La herramienta para extraer los comentarios se denomina *Javadoc*, y forma parte de la instalación del JDK. Utiliza parte de la tecnología del compilador Java para buscar los marcadores especiales de comentarios incluidos en el programa. No sólo extrae la información señalada por dichos marcadores, sino que también extrae el nombre de la clase o el nombre del método asociado al comentario. De esta forma, podemos generar una documentación de programa bastante digna con una cantidad mínima de trabajo.

La salida de Javadoc es un archivo HTML que se puede examinar con el explorador web. Javadoc permite crear y mantener un único archivo fuente y generar automáticamente una documentación bastante útil. Gracias a Javadoc, disponemos de un estándar bastante sencillo para la creación de documentación, por lo que podemos esperar, o incluso exigir, que todas las bibliotecas Java estén documentadas.

Además, podemos escribir nuestras propias rutinas de gestión Javadoc, denominadas *doclets*, si queremos realizar operaciones especiales con la información procesada por Javadoc (por ejemplo, para generar la salida en un formato distinto). Los *doclets* se explican en el suplemento que podrá encontrar en <http://MindView.net/Books/BetterJava>.

Lo que sigue es simplemente una introducción y una breve panorámica de los fundamentos de Javadoc. En la documentación del JDK podrá encontrar una descripción más exhaustiva. Cuando desempaque la documentación, vaya al subdirectorio "tooldocs" (o haga clic en el vínculo "tooldocs").

Sintaxis

Todos los comandos de Javadoc están incluidos en comentarios que tienen el marcador `/**`. Esos comentarios terminan con `*/` como es habitual. Existen dos formas principales de utilizar Javadoc: mediante HTML embebido o mediante "marcadores de documentación". Los *marcadores de documentación autónomos* son comandos que comienzan con '@' y se incluyen al principio de una línea de comentarios (sin embargo, un carácter '*' inicial será ignorado). Los *marcadores de documentación en línea* pueden incluirse en cualquier punto de un comentario Javadoc y también comienzan con '@', pero están encerrados entre llaves.

Existen tres "tipos" de documentación de comentarios, que se corresponden con el elemento al que el comentario precede: una clase, un campo o un método. En otras palabras, el comentario de clase aparece justo antes de la definición de la clase, el comentario de campo aparece justo antes de la definición de un campo y el comentario de un método aparece justo antes de la definición del mismo. He aquí un sencillo ejemplo:

```
//: object/Documentation1.java
/** Comentario de clase */
public class Documentation1 {
    /** Comentario de campo */
    public int i;
    /** Comentario de método */
    public void f() {}
} //:-
```

Observe que Javadoc sólo procesará la documentación de los comentarios para los miembros **public** y **protected**. Los comentarios para los miembros **private** y los miembros con acceso de paquete (véase el Capítulo 6, *Control de acceso*) se ignoran, no generándose ninguna salida para ellos (sin embargo, puede utilizar el indicador **private** para incluir también la documentación de los miembros **private**). Esto tiene bastante sentido, ya que sólo los miembros de tipo **public** y **protected** están disponibles fuera del archivo, por lo que esto se ajusta a la perspectiva del programador de clientes.

La salida del código anterior es un archivo HTML que tiene el mismo formato estándar que el resto de la documentación Java, por lo que los usuarios estarán acostumbrados a ese formato y podrán navegar fácilmente a través de las clases que

hayamos definido. Merece la pena introducir el ejemplo de código anterior, procesarlo mediante Javadoc y observar el archivo HTML resultante, para ver el tipo de salida que se genera.

HTML embebido

Javadoc pasa los comentarios HTML al documento HTML generado. Esto nos permite utilizar completamente las características HTML; sin embargo, el motivo principal de uso de ese lenguaje es dar formato al código, como por ejemplo en:

```
//: object/Documentation2.java
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
//:-
```

También podemos usar código HTML como en cualquier otro documento web, para dar formato al texto normal de las descripciones:

```
//: object/Documentation3.java
/**
 * Se puede <em>incluso</em> insertar una lista:
 * <ol>
 * <li> Elemento uno
 * <li> Elemento dos
 * <li> Elemento tres
 * </ol>
 */
//:-
```

Observe que, dentro del comentario de documentación, los asteriscos situados al principio de cada línea son ignorados por Javadoc, junto con los espacios iniciales. Javadoc reformatea todo para que se adapte al estilo estándar de la documentación. No utilice encabezados como `<h1>` o `<hr>` en el HTML embebido, porque Javadoc inserta sus propios encabezados y los que nosotros incluyamos interferirán con ellos.

Puede utilizarse HTML embebido en todos los tipos de comentarios de documentación: de clase, de campo y de método.

Algunos marcadores de ejemplo

He aquí algunos de los marcadores Javadoc disponibles para la documentación de código. Antes de tratar de utilizar Javadoc de forma seria, consulte la documentación de referencia de Javadoc dentro de la documentación del JDK, para ver las diferentes formas de uso de Javadoc.

`@see`

Este marcador permite hacer referencia a la documentación de otras clases. Javadoc generará el código HTML necesario, hipervinculando los marcadores `@see` a los otros fragmentos de documentación. Las formas posibles de uso de este marcador son:

```
@see nombreclase
@see nombreclase-completamente-cualificado
@see nombreclase-completamente-cualificado #nombre-método
```

Cada uno de estos marcadores añade una entrada “See Also” (Véase también) hipervinculada al archivo de documentación generado. Javadoc no comprueba los hipervínculos para ver si son válidos.

`{@link paquete.clase#miembro etiqueta }`

Muy similar a `@see`, excepto porque se puede utilizar en linea y emplea la *etiqueta* como texto del hipervínculo, en lugar de “See Also”.

{@docRoot}

Genera la ruta relativa al directorio raíz de la documentación. Resulta útil para introducir hipervínculos explícitos a páginas del árbol de documentación.

{@inheritDoc}

Este indicador hereda la documentación de la clase base más próxima de esta clase, insertándola en el comentario del documento actual.

@version

Tiene la forma:

```
@version información-versión
```

en el que **información-versión** es cualquier información significativa que queramos incluir. Cuando se añade el indicador **@version** en la línea de comandos Javadoc, la información de versión será mostrada de manera especial en la documentación HTML generada.

@author

Tiene la forma:

```
@author información-autor
```

donde **información-autor** es, normalmente, nuestro nombre, aunque también podríamos incluir nuestra dirección de correo electrónico o cualquier otra información apropiada. Cuando se incluye el indicador **@author** en la línea de comandos Javadoc, se inserta la información de autor de manera especial en la documentación HTML generada.

Podemos incluir múltiples marcadores de autor para incluir una lista de autores, pero es preciso poner esos marcadores de forma consecutiva. Toda la información de autor se agrupará en un único párrafo dentro del código HTML generado.

@since

Este marcador permite indicar la versión del código en la que se empezó a utilizar una característica concreta. En la documentación HTML de Java, se emplea este marcador para indicar qué versión del JDK se está utilizando.

@param

Se utiliza para la documentación de métodos y tiene la forma:

```
@param nombre-parámetro descripción
```

donde **nombre-parámetro** es el identificador dentro de la lista de parámetros del método y **descripción** es un texto que puede continuar en las líneas siguientes. La descripción se considera terminada cuando se encuentra un nuevo marcador de documentación. Puede haber múltiples marcadores de este tipo, normalmente uno por cada parámetro.

@return

Se utiliza para documentación de métodos y su formato es el siguiente:

```
@return descripción
```

donde **descripción** indica el significado del valor de retorno. Puede continuar en las líneas siguientes.

@throws

Las excepciones se estudian en el Capítulo 12, *Tratamiento de errores mediante excepciones*. Por resumir, se trata de objetos que pueden ser “generados” en un método si dicho método falla. Aunque sólo puede generarse un objeto excepción cada vez que invocamos un método, cada método concreto puede generar diferentes tipos de excepciones, todas las cuales habrá que describir, por lo que el formato del marcador de excepción es:

```
@throws nombre-clase-completamente-cualificado descripción
```

donde *nombre-clase-completamente-cualificado* proporciona un nombre no ambiguo de una clase de excepción definida en alguna otra parte y *descripción* (que puede ocupar las siguientes líneas) indica la razón por la que puede generarse este tipo concreto de excepción después de la llamada al método.

@deprecated

Se utiliza para indicar características que han quedado obsoletas debido a la introducción de alguna otra característica mejorada. Este marcador indicativo de la obsolescencia recomienda que no se utilice ya esa característica concreta, ya que es probable que en el futuro sea eliminada. Un método marcado como **@deprecated** hace que el compilador genere una advertencia si se usa. En Java SE5, el marcador Javadoc **@deprecated** ha quedado sustituido por la *anotación* **@Deprecated** (hablaremos de este tema en el Capítulo 20, *Anotaciones*).

Ejemplo de documentación

He aquí de nuevo nuestro primer programa Java, pero esta vez con los comentarios de documentación incluidos:

```
//: object/HelloDate.java
import java.util.*;

/** El primer programa de ejemplo del libro.
 * Muestra una cadena de caracteres y la fecha actual.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
 */
public class HelloDate {
    /** Punto de entrada a la clase y a la aplicación.
     * @param args matriz de argumentos de cadena
     * @throws exceptions No se generan excepciones
     */
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:36 MDT 2005
*///:-
```

La primera línea del archivo utiliza una técnica propia del autor del libro que consiste en incluir ‘//:’ como marcador especial en la línea de comentarios que contiene el nombre del archivo fuente. Dicha línea contiene la información de ruta del archivo (**object** indica este capítulo) seguida del nombre del archivo. La última línea también termina con un comentario, y éste (*///:-) indica el final del listado de código fuente, lo que permite actualizarlo automáticamente dentro del texto de este libro después de comprobarlo con un compilador y ejecutarlo.

El marcador **/* Output:** indica el comienzo de la salida que generará este archivo. Con esta forma concreta, se puede comprobar automáticamente para verificar su precisión. En este caso, el texto **(55% match)** indica al sistema de pruebas que la salida será bastante distinta en cada sucesiva ejecución del programa, por lo que sólo cabe esperar un 55 por ciento de correlación con la salida que aquí se muestre. La mayoría de los ejemplos del libro que generan una salida contendrán dicha salida comentada de esta forma, para que el lector pueda ver la salida y saber que lo que ha obtenido es correcto.

Estilo de codificación

El estilo descrito en el manual de convenios de código para Java, *Code Conventions for the Java Programming Language*⁸, consiste en poner en mayúscula la primera letra del nombre de una clase. Si el nombre de la clase está compuesto por varias

⁸ Para ahorrar espacio tanto en el libro como en las presentaciones de los seminarios no hemos podido seguir todas las directrices que se marcan en ese texto, pero el lector podrá ver que el estilo empleado en el libro se ajusta lo máximo posible al estándar recomendado en Java.

palabras, éstas se escriben juntas (es decir, no se emplean guiones bajos para separarlas) y se pone en mayúscula la primera letra de cada una de las palabras integrantes, como en:

```
class AllTheColorsOfTheRainbow { // ...
```

Para casi todos los demás elementos, como por ejemplo nombres de métodos, campos (variables miembro) y referencias a objetos, el estilo aceptado es igual que para las clases, *salvo* porque la primera letra del identificador se escribe en minúscula, por ejemplo:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

Recuerde que el usuario se verá obligado a escribir estos nombres tan largos, así que procure que su longitud no sea excesiva.

El código Java que podrá encontrar en las bibliotecas Sun también se ajusta a las directrices de ubicación de llaves de apertura y cierre que hemos utilizado en este libro.

Resumen

El objetivo de este capítulo es explicar los conceptos mínimos sobre Java necesarios para entender cómo se escribe un programa sencillo. Hemos expuesto una panorámica del lenguaje y algunas de las ideas básicas en que se fundamenta. Sin embargo, los ejemplos incluidos hasta ahora tenían todos ellos la forma “haga esto, luego lo otro y después lo de más allá”. En los dos capítulos siguientes vamos a presentar los operadores básicos utilizados en los programas Java, y luego mostraremos cómo controlar el flujo del programa.

Ejercicios

Normalmente, distribuiremos los ejercicios por todo el capítulo, pero en éste estábamos aprendiendo a escribir programas básicos, por lo que decidimos dejar los ejercicios para el final.

El número entre paréntesis incluido detrás del número de ejercicio es un indicador de la dificultad del ejercicio dentro de una escala de 1-10.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, a la venta en www.MindView.net.

Ejercicio 1: (2) Cree una clase que contenga una variable **int** y otra **char** que no estén inicializadas e imprima sus valores para verificar que Java se encarga de realizar una inicialización predeterminada.

Ejercicio 2: (1) A partir del ejemplo **HelloDate.java** de este capítulo, cree un programa “hello, world” que simplemente muestre dicha frase. Sólo necesitará un método dentro de la clase, (el método “main” que se ejecuta al arrancar el programa). Acuérdese de definir este método como **static** y de incluir la lista de argumentos, incluso aunque no la vaya a utilizar. Compile el programa con **javac** y ejecútelo con **java**. Si está utilizando otro entorno de desarrollo distinto de JDK, averigüe cómo compilar y ejecutar programas en dicho entorno.

Ejercicio 3: (1) Recopile los fragmentos de código relacionados con **ATypeName** y transfórmelos en un programa que se pueda compilar y ejecutar.

Ejercicio 4: (1) Transforme los fragmentos de código **DataOnly** en un programa que se pueda compilar y ejecutar.

Ejercicio 5: (1) Modifique el ejercicio anterior de modo que los valores de los datos en **DataOnly** se asignen e impriman en **main()**.

Ejercicio 6: (2) Escriba un programa que incluya e invoque el método **storage()** definido como fragmento de código en el capítulo.

- Ejercicio 7:** (1) Transforme los fragmentos de código **Incrementable** en un programa funcional.
- Ejercicio 8:** (3) Escriba un programa que demuestre que, independientemente de cuántos objetos se creen de una clase concreta, sólo hay una única instancia de un campo **static** concreto definido dentro de esa clase.
- Ejercicio 9:** (2) Escriba un programa que demuestre que el mecanismo automático de conversión de tipos funciona para todos los tipos primitivos y sus envoltorios.
- Ejercicio 10:** (2) Escriba un programa que imprima tres argumentos extraídos de la línea de comandos. Para hacer esto, necesitará acceder con el índice correspondiente a la matriz de objetos **String** extraída de la línea de comandos.
- Ejercicio 11:** (1) Transforme el ejemplo **AllTheColorsOfTheRainbow** en un programa que se pueda compilar y ejecutar.
- Ejercicio 12:** (2) Localice el código para la segunda versión de **HelloDate.java**, que es el ejemplo simple de documentación mediante comentarios. Ejecute **Javadoc** con ese archivo y compruebe los resultados con su explorador web.
- Ejercicio 13:** (1) Procese **Documentation1.java**, **Documentation2.java** y **Documentation3.java** con **Javadoc**. Verifique la documentación resultante con su explorador web.
- Ejercicio 14:** (1) Añada una lista HTML de elementos a la documentación del ejercicio anterior.
- Ejercicio 15:** (1) Tome el programa del Ejercicio 2 y añádale comentarios de documentación. Extraiga esos comentarios de documentación **Javadoc** para generar un archivo HTML y visualicelo con su explorador web.
- Ejercicio 16:** (1) En el Capítulo 5, *Inicialización y limpieza*, localice el ejemplo **Overloading.java** y añada documentación de tipo Javadoc. Extraiga los comentarios de documentación **Javadoc** para generar un archivo HTML y visualicelo con su explorador web.

Operadores

3

En el nivel inferior, los datos en Java se manipulan utilizando operadores.

Puesto que Java surgió a partir de C++, la mayoría de estos operadores les serán familiares a casi todos los programadores de C y C++. Java ha añadido también algunas mejoras y ha hecho algunas simplificaciones.

Si está familiarizado con la sintaxis de C o C++, puede pasar rápidamente por este capítulo y el siguiente, buscando aquellos aspectos en los que Java se diferencia de esos lenguajes. Por el contrario, si le asaltan dudas en estos dos capítulos, repase el seminario multimedia *Thinking in C*, que puede descargarlo gratuitamente en www.MindView.net. Contiene conferencias, presentaciones, ejercicios y soluciones específicamente diseñados para familiarizarse con los fundamentos necesarios para aprender Java.

Instrucciones simples de impresión

En el capítulo anterior, ya hemos presentado la instrucción de impresión en Java:

```
System.out.println("Un montón de texto que escribir");
```

Puede observar que esto no es sólo un montón de texto que escribir (lo que quiere decir muchos movimientos redundantes de los dedos), sino que también resulta bastante incómodo de leer. La mayoría de los lenguajes, tanto antes como después de Java, han adoptado una técnica mucho más sencilla para expresar esta instrucción de uso tan común.

En el Capítulo 6, *Control de acceso* se presenta el concepto de *importación estática* añadido a Java SE5, y se crea una pequeña biblioteca que permite simplificar la escritura de las instrucciones de impresión. Sin embargo, no es necesario comprender los detalles para comenzar a utilizar dicha biblioteca. Podemos reescribir el programa del último capítulo empleando esa nueva biblioteca:

```
//: operators>HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
    public static void main(String[] args) {
        print("Hello, it's: ");
        print(new Date());
    }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:05 MDT 2005
*///:-
```

Los resultados son mucho más limpios. Observe la inserción de la palabra clave **static** en la segunda instrucción **import**.

Para emplear esta biblioteca, es necesario descargar el paquete de código del libro desde www.MindView.net o desde alguno de los sitios espejo. Descomprima el árbol de código y añada el directorio raíz de dicho árbol de código a la variable de entorno CLASSPATH de su computadora (más adelante proporcionaremos una introducción completa a la variable de ruta CLASSPATH, pero es muy probable que el lector ya esté acostumbrado a lidiar con esa variable. De hecho, es una de las batallas más comunes que se presentan al intentar programar en Java).

Aunque la utilización de `net.mindview.util.Print` simplifica enormemente la mayor parte del código, su uso no está justificado en todas las ocasiones. Si sólo hay unas pocas instrucciones de impresión en un programa, es preferible no incluir la instrucción `import` y escribir el comando completo `System.out.println()`.

Ejercicio 1: (1) Escriba un programa que emplee tanto la forma “corta” como la normal de la instrucción de impresión.

Utilización de los operadores Java

Un operador toma uno o más argumentos y genera un nuevo valor. Los argumentos se incluyen de forma distinta a las llamadas normales a métodos, pero el efecto es el mismo. La suma, el operador más unario (+), la resta y el operador menos unario (-), la multiplicación (*), la división (/) y la asignación (=) funcionan de forma bastante similar a cualquier otro lenguaje de programación.

Todos los valores producen un valor a partir de sus operandos. Además, algunos operadores cambian el valor del operando, lo que se denomina *efecto colateral*. El uso más común de los operadores que modifican sus operandos consiste precisamente en crear ese efecto colateral, pero resulta conveniente pensar que el valor generado también está disponible para nuestro uso, al igual que sucede con los operadores que no tienen efectos colaterales.

Casi todos los operadores funcionan únicamente con primitivas. Las excepciones son ‘+=’, ‘-=’ y ‘!=’, que funcionan con todos los objetos (y son una fuente de confusión con los objetos). Además, la clase `String` soporta ‘+’ y ‘+=’.

Precedencia

La precedencia de los operadores define la forma en que se evalúa una expresión cuando hay presentes varios operadores. Java tiene una serie de reglas específicas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división se realizan antes que la suma y la resta. Los programadores se olvidan a menudo de las restantes reglas de precedencia, así que conviene utilizar paréntesis para que el orden de evaluación esté indicado de manera explícita. Por ejemplo, observe las instrucciones (1) y (2):

```
//: operators/Precedence.java

public class Precedence {
    public static void main(String[] args) {
        int x = 1, y = 2, z = 3;
        int a = x + y - 2/2 + z;           // (1)
        int b = x + (y - 2)/(2 + z);     // (2)
        System.out.println("a = " + a + " b = " + b);
    }
} /* Output:
a = 5 b = 1
*///:-
```

Estas instrucciones tienen aspecto similar, pero la salida nos muestra que sus significados son bastante distintos, debido a la utilización de paréntesis.

Observe que la instrucción `System.out.println()` incluye el operador ‘+’. En este contexto, ‘+’ significa “concatenación de cadenas de caracteres” y, en caso necesario, “conversión de cadenas de caracteres.” Cuando el compilador ve un objeto `String` seguido de ‘+’ seguido de un objeto no `String`, intenta convertirlo a un objeto `String`. Como puede ver en la salida, se ha efectuado correctamente la conversión de `Int` a `String` para `a` y `b`.

Asignación

La asignación se realiza con el operador `=`. Su significado es: “Toma el valor del lado derecho, a menudo denominado *rvalor*, y cópialo en el lado izquierdo (a menudo denominado *lvalor*)”. Un *rvalor* es cualquier constante, variable o expresión que genere un valor, pero un *lvalor* debe ser una variable determinada, designada mediante su nombre (es decir, debe existir un espacio físico para almacenar el valor). Por ejemplo, podemos asignar un valor constante a una variable:

```
a = 4;
```

pero no podemos asignar nada a un valor constante, ya que una constante no puede ser un lvalor (no podemos escribir `4 = a;`).

La asignación de primitivas es bastante sencilla. Puesto que la primitiva almacena el valor real y no una referencia a cualquier objeto, cuando se asignan primitivas se asigna el contenido de un lugar a otro. Por ejemplo, si escribimos `a = b` para primitivas, el contenido de `b` se copia en `a`. Si a continuación modificamos `a`, el valor de `b` no se verá afectado por esta modificación. Como programador es lo que cabría esperar en la mayoría de las situaciones.

Sin embargo, cuando asignamos objetos, la cosa cambia. Cuando manipulamos un objeto lo que manipulamos es la referencia, así que al asignar “de un objeto a otro”, lo que estamos haciendo en la práctica es copiar una referencia de un lugar a otro. Esto significa que si escribimos `c = d` para sendos objetos, lo que al final tendremos es que tanto `c` como `d` apuntan al objeto al que sólo `d` apuntaba originalmente. He aquí un ejemplo que ilustra este comportamiento:

```
//: operators/Assignment.java
// La asignación de objetos tiene su truco.
import static net.mindview.util.Print.*;

class Tank {
    int level;
}

public class Assignment {
    public static void main(String[] args) {
        Tank t1 = new Tank();
        Tank t2 = new Tank();
        t1.level = 9;
        t2.level = 47;
        print("1: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
        t1 = t2;
        print("2: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
        t1.level = 27;
        print("3: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
    }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:-
```

La clase `Tank` es simple, y se crean dos instancias de la misma (`t1` y `t2`) dentro de `main()`. Al campo `level` de cada objeto `Tank` se le asigna un valor distinto, luego se asigna `t2` a `t1`, y después se modifica `t1`. En muchos lenguajes de programación esperaríamos que `t1` y `t2` fueran independientes en todo momento, pero como hemos asignado una referencia, al cambiar el objeto `t1` se modifica también el objeto `t2`. Esto se debe a que tanto `t1` como `t2` contienen la misma referencia, que está apuntada en el mismo objeto (la referencia original contenida en `t1`, que apuntaba al objeto que tenía un valor de 9, fue sobreescrita durante la asignación y se ha perdido a todos los efectos; su objeto será eliminado por el depurador de memoria).

Este fenómeno a menudo se denomina *creación de alias*, y representa una de las características fundamentales del modo en que Java trabaja con los objetos. Pero, ¿qué sucede si no queremos que las dos referencias apunten al final a un mismo objeto? Podemos reescribir la asignación a otro nivel y utilizar:

```
t1.level = t2.level;
```

Esto hace que se mantengan independientes los dos objetos, en lugar de descartar uno de ellos y asociar `t1` y `t2` al mismo objeto. Más adelante veremos que manipular los campos dentro de los objetos resulta bastante confuso y va en contra de los principios de un buen diseño orientado a objetos. Se trata de un tema que no es nada trivial, así que tenga siempre presente que las asignaciones pueden causar sorpresas cuando se manejan objetos.

Ejercicio 2: (1) Cree una clase que contenga un valor `float` y utilicela para ilustrar el fenómeno de la creación de alias.

Creación de alias en las llamadas a métodos

El fenómeno de la creación de alias también puede manifestarse cuando se pasa un objeto a un método:

```
//: operators/PassObject.java
// El paso de objetos a los métodos puede no ser
// lo que cabría esperar.
import static net.mindview.util.Print.*;

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        print("1: x.c: " + x.c);
        f(x);
        print("2: x.c: " + x.c);
    }
} /* Output:
1: x.c: a
2: x.c: z
*/:-
```

En muchos lenguajes de programación, el método `f()` haría una copia de su argumento `Letter` dentro del ámbito del método, pero aquí, una vez más, lo que se está pasando es una referencia, por lo que la línea:

```
y.c = 'z';
```

lo que está haciendo es cambiar el objeto que está fuera de `f()`.

El fenómeno de creación de alias y su solución es un tema complejo del que se trata en uno de los suplementos en línea disponibles para este libro. Sin embargo, conviene que lo tenga presente desde ahora, con el fin de detectar posibles errores.

Ejercicio 3: (1) Cree una clase que contenga un valor `float` y utilicela para ilustrar el fenómeno de la creación de alias durante las llamadas a métodos.

Operadores matemáticos

Los operadores matemáticos básicos son iguales a los que hay disponibles en la mayoría de los lenguajes de programación: suma (+), resta (-), división (/), multiplicación (*) y módulo (%), que genera el resto de una división entera). La división entera trunca en lugar de redondear el resultado.

Java también utiliza la notación abreviada de C/C++ que realiza una operación y una asignación al mismo tiempo. Este tipo de operación se denota mediante un operador seguido de un signo de igual, y es coherente con todos los operadores del lenguaje (allí donde tenga sentido). Por ejemplo, para sumar 4 a la variable `x` y asignar el resultado a `x`, utilice: `x += 4`.

Este ejemplo muestra el uso de los operadores matemáticos:

```
//: operators/MathOps.java
// Ilustra los operadores matemáticos.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
    public static void main(String[] args) {
```

```

// Crea un generador de números aleatorios con una cierta semilla:
Random rand = new Random(47);
int i, j, k;
// Elegir valor entre 1 y 100:
j = rand.nextInt(100) + 1;
print("j : " + j);
k = rand.nextInt(100) + 1;
print("k : " + k);
i = j + k;
print("j + k : " + i);
i = j - k;
print("j - k : " + i);
i = k / j;
print("k / j : " + i);
i = k * j;
print("k * j : " + i);
i = k % j;
print("k % j : " + i);
j *= k;
print("j *= k : " + j);
// Pruebas con números en coma flotante:
float u, v, w; // Se aplica también a los de doble precisión
v = rand.nextFloat();
print("v : " + v);
w = rand.nextFloat();
print("w : " + w);
u = v + w;
print("v + w : " + u);
u = v - w;
print("v - w : " + u);
u = v * w;
print("v * w : " + u);
u = v / w;
print("v / w : " + u);
// Lo siguiente también funciona para char,
// byte, short, int, long y double:
u += v;
print("u += v : " + u);
u -= v;
print("u -= v : " + u);
u *= v;
print("u *= v : " + u);
u /= v;
print("u /= v : " + u);
}
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j *= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962

```

```
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:-
```

Para generar números, el programa crea en primer lugar un objeto **Random**. Si se crea un objeto **Random** sin ningún argumento, Java usa la hora actual como semilla para el generador de números aleatorios, y esto generaría una salida diferente en cada ejecución del programa. Sin embargo, en los ejemplos del libro, es importante que la salida que se muestra al final de los ejemplos sea lo más coherente posible, para poder verificarla con herramientas externas. Proporcionando una *semilla* (un valor de inicialización para el generador de números aleatorios que siempre genera la misma secuencia para una determinada semilla) al crear el objeto **Random**, se generarán siempre los mismos números aleatorios en cada ejecución del programa, por lo que la salida se podrá verificar.¹ Para generar una salida más variada, pruebe a eliminar la semilla en los ejemplos del libro.

El programa genera varios números aleatorios de distintos tipos con el objeto **Random** simplemente invocando los métodos **nextInt()** y **nextFloat()** (también se pueden invocar **nextLong()** o **nextDouble()**). El argumento de **nextInt()** establece la cota superior para el número generado. La cota superior es cero, lo cual no resulta deseable debido a la posibilidad de una división por cero, por lo que sumamos uno al resultado.

Ejercicio 4: (2) Escriba un programa que calcule la velocidad utilizando una distancia constante y un tiempo constante.

Operadores unarios más y menos

El menos unario (-) y el más unario (+) son los mismos operadores que la suma y la resta binarias. El compilador deduce cuál es el uso que se le quiere dar al operador a partir de la forma en que está escrita la expresión. Por ejemplo, la instrucción:

```
x = -a;
```

tiene un significado obvio. El compilador también podría deducir el uso correcto en:

```
x = a * -b;
```

pero esto podría ser algo confuso para el lector, por lo que a veces resulta más claro escribir:

```
x = a * (-b);
```

El menos unario invierte el signo de los datos. El más unario proporciona una simetría con respecto al menos unario, aunque no tiene ningún efecto.

Autoincremento y autodecremento

Java, como C, dispone de una serie de abreviaturas. Esas abreviaturas pueden hacer que resulte mucho más fácil escribir el código; en cuanto a la lectura, pueden simplificarla o complicarla.

Dos de las abreviaturas más utilizadas son los operadores de incremento y decremento (a menudo denominados operadores de autoincremento y autodecremento). El operador de decremento es -- y significa "disminuir en una unidad". El operador de incremento es ++ y significa "aumentar en una unidad". Por ejemplo, si a es un valor **int**, la expresión ++a es equivalente a (a = a + 1). Los operadores de incremento y decremento no sólo modifican la variable, sino que también generan el valor de la misma como resultado.

Hay dos versiones de cada tipo de operador, a menudo denominadas *prefija* y *postfija*. *Pre-incremento* significa que el operador ++ aparece antes de la variable, mientras que *post-incremento* significa que el operador ++ aparece detrás de la variable. De forma similar, *pre-decremento* quiere decir que el operador -- aparece antes de la variable y *post-decremento* significa que el operador -- aparece detrás de la variable. Para el pre-incremento y el pre-decremento (es decir, ++a o --a), se realiza primero la operación y luego se genera el valor. Para el post-incremento y el post-decremento (es decir, a++ o --a), se genera primero el valor y luego se realiza la operación. Por ejemplo:

¹ El número 47 se utilizaba como "número mágico" en una universidad en la que estudié, y desde entonces lo utilicé.

```

//: operators/AutoInc.java
// Ilustra los operadores ++ y --.
import static net.mindview.util.Print.*;

public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        print("i : " + i);
        print("++i : " + ++i); // Pre-incremento
        print("i++ : " + i++); // Post-incremento
        print("i : " + i);
        print("--i : " + --i); // Pre-decremento
        print("i-- : " + i--); // Post-decremento
        print("i : " + i);
    }
} /* Output:
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
*///:-

```

Puede ver qué para la forma prefija, se obtiene el valor después de realizada la operación, mientras que para la forma postfija, se obtiene el valor antes de que la operación se realice. Estos son los únicos operadores, además de los de asignación, que tienen efectos colaterales. Modifican el operando en lugar de simplemente utilizar su valor.

El operador de incremento es, precisamente, una de las explicaciones del por qué del nombre C++, que quiere decir “un paso más allá de C”. En una de las primeras presentaciones realizadas acerca de Java, Bill Joy (uno de los creadores de Java), dijo que “Java=C++---” (C más más menos menos), para sugerir que Java es C++ pero sin todas las complejidades innecesarias, por lo que resulta un lenguaje mucho más simple. A medida que vaya avanzando a lo largo del libro, podrá ver que muchas partes son más simples, aunque en algunos otros aspectos Java no resulta mucho más sencillo que C++.

Operadores relacionales

Los operadores relacionales generan un resultado de tipo **boolean**. Evalúan la relación existente entre los valores de los operandos. Una expresión relacional produce el valor **true** si la relación es cierta y **false** si no es cierta. Los operadores relacionales son: menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), equivalente (==) y no equivalente (!=). La equivalencia y la no equivalencia funcionan con todas las primitivas, pero las otras comparaciones no funcionan con el tipo **boolean**. Puesto que los valores **boolean** sólo pueden ser **true** o **false**, las relaciones “mayor que” y “menor que” no tienen sentido.

Comprobación de la equivalencia de objetos

Los operadores relacionales == y != también funcionan con todos los objetos, pero su significado suele confundir a los que comienzan a programar en Java. He aquí un ejemplo:

```

//: operators/Equivalence.java

public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
}

```

```
 } /* Output:
false
true
*/:-
```

La instrucción `System.out.println(n1 == n2)` imprimirá el resultado de la comparación booleana que contiene. Parece que la salida debería ser “true” y luego “false”, dado que ambos objetos `Integer` son iguales, aunque el *contenido* de los objetos son los mismos, las referencias no son iguales. Los operadores `==` y `!=` comparan referencias a objetos, por lo que la salida realmente es “false” y luego “true”. Naturalmente, este comportamiento suele sorprender al principio a los programadores.

¿Qué pasa si queremos comparar si el contenido de los objetos es equivalente? Entonces debemos utilizar el método especial `equals()` disponible para todos los objetos (no para las primitivas, que funcionan adecuadamente con `==` y `!=`). He aquí la forma en que se emplea:

```
//: operators/EqualsMethod.java

public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
} /* Output:
true
*/:-
```

El resultado es ahora el que esperábamos. Aunque, en realidad, las cosas no son tan sencillas. Si creamos nuestra propia clase, como por ejemplo:

```
//: operators/EqualsMethod2.java
// equals() predeterminado no compara los contenidos.

class Value {
    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
} /* Output:
false
*/:-
```

los resultados vuelven a confundirnos. El resultado es `false`. Esto se debe a que el comportamiento predeterminado de `equals()` consiste en comparar referencias. Por tanto, a menos que sustituymos `equals()` en nuestra nueva clase, no obtendremos el comportamiento deseado. Lamentablemente, no vamos a aprender a sustituir unos métodos por otros hasta el capítulo dedicado a la *Reutilización de clases*, y no veremos cuál es la forma adecuada de definir `equals()` hasta el Capítulo 17, *Análisis detallado de los contenedores*, pero mientras tanto tener en cuenta el comportamiento de `equals()` nos puede ahorrar algunos quebraderos de cabeza.

La mayoría de las clases de biblioteca Java implementan `equals()` de modo que compare el contenido de los objetos, en lugar de sus referencias.

Ejercicio 5: (2) Cree una clase denominada `Dog` (perro) que contenga dos objetos `String`: `name` (nombre) y `says` (ladrido). En `main()`, cree dos objetos perro con los nombres “spot” (que ladre diciendo “Ruff!”) y “scruffy” (que ladre diciendo, “Wurf!”). Después, muestre sus nombres y el sonido que hacen al ladrar.

Ejercicio 6: (3) Continuando con el Ejercicio 5, cree una nueva referencia **Dog** y asignela al objeto de nombre "spot". Realice una comparación utilizando **==** y **equals()** para todas las referencias.

Operadores lógicos

Cada uno de los operadores lógicos AND (**&&**), OR (**||**) y NOT (**!**) produce un valor **boolean** igual a **true** o **false** basándose en la relación lógica de sus argumentos. Este ejemplo utiliza los operadores relacionales y lógicos:

```
//: operators/Bool.java
// Operadores relacionales y lógicos.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        print("i = " + i);
        print("j = " + j);
        print("i > j is " + (i > j));
        print("i < j is " + (i < j));
        print("i >= j is " + (i >= j));
        print("i <= j is " + (i <= j));
        print("i == j is " + (i == j));
        print("i != j is " + (i != j));
        // Tratar int como boolean no es legal en Java:
        // print("i && j is " + (i && j));
        // print("i || j is " + (i || j));
        // print("!i is " + !i);
        print("(i < 10) && (j < 10) is "
              + ((i < 10) && (j < 10)) );
        print("(i < 10) || (j < 10) is "
              + ((i < 10) || (j < 10)) );
    }
} /* Output:
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
*///:-
```

Sólo podemos aplicar AND, OR o NOT a valores de tipo **boolean**. No podemos emplear un valor que no sea **boolean** como si fuera un valor booleano en una expresión lógica como a diferencia de lo que sucede en C y C++. Puede ver en el ejemplo los intentos fallidos de realizar esto, desactivados mediante marcas de comentarios **'//'** (esta sintaxis de comentarios permite la eliminación automática de comentarios para facilitar las pruebas). Sin embargo, las expresiones subsiguientes generan valores de tipo **boolean** utilizando comparaciones relacionales y a continuación realizan operaciones lógicas con los resultados.

Observe que un valor **boolean** se convierte automáticamente a una forma de tipo texto apropiada cuando se les usa en lugares donde lo que se espera es un valor de tipo **String**.

Puede reemplazar la definición de valores `int` en el programa anterior por cualquier otro tipo de dato primitivo excepto `boolean`. Sin embargo, teniendo en cuenta que la comparación de números en coma flotante es muy estricta, un número que difiera de cualquier otro, aunque sea en un valor pequeñísimo seguirá siendo distinto. Asimismo, cualquier número situado por encima de cero, aunque sea pequeñísimo, seguirá siendo distinto de cero.

Ejercicio 7: (3) Escriba un programa que simule el proceso de lanzar una moneda al aire.

Cortocircuitos

Al tratar con operadores lógicos, nos encontramos con un fenómeno denominado “cortocircuito”. Esto quiere decir que la expresión se evaluará únicamente *hasta* que la veracidad o la falsedad de la expresión completa pueda ser determinada de forma no ambigua. Como resultado, puede ser que las últimas partes de una expresión lógica no lleguen a evaluarse. He aquí un ejemplo que ilustra este fenómeno de cortocircuito.

```
//: operators/ShortCircuit.java
// Ilustra el comportamiento de cortocircuito
// al evaluar los operadores lógicos.
import static net.mindview.util.Print.*;

public class ShortCircuit {
    static boolean test1(int val) {
        print("test1(" + val + ")");
        print("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        print("test2(" + val + ")");
        print("result: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        print("test3(" + val + ")");
        print("result: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
        boolean b = test1(0) && test2(2) && test3(2);
        print("expression is " + b);
    }
} /* Output:
test1(0)
result: true
test2(2)
result: false
expression is false
*///:-
```

Cada una de las comprobaciones realiza una comparación con el argumento y devuelve `true` o `false`. También imprime la información necesaria para que veamos qué está siendo invocada. Las pruebas se utilizan en la expresión:

```
test1(0) && test2(2) && test3(2)
```

Lo natural sería pensar que las tres pruebas llegan a ejecutarse, pero la salida muestra que no es así. La primera de las pruebas produce un resultado `true`, por lo que continúa con la evaluación de la expresión. Sin embargo, la segunda prueba produce un resultado `false`. Dado que esto quiere decir que la expresión completa debe ser `false`, ¿por qué continuar con la evaluación del resto de la expresión? Esta evaluación podría consumir una cantidad considerable de recursos. La razón de que se produzca este tipo de cortocircuito es, de hecho, que podemos mejorar la velocidad del programa si no es necesario evaluar todas las partes de una expresión lógica.

Literales

Normalmente, cuando insertamos un valor literal en un programa, el compilador sabe exactamente qué tipo asignarle. En ocasiones, sin embargo, puede que ese tipo sea ambiguo. Cuando esto sucede, hay que guiar al compilador añadiendo cierta información adicional en la forma de caracteres asociados con el valor literal. El código siguiente muestra estos caracteres:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
    public static void main(String[] args) {
        int i1 = 0x2f; // Hexadecimal (minúscula)
        print("i1: " + Integer.toBinaryString(i1));
        int i2 = 0X2F; // Hexadecimal (mayúscula)
        print("i2: " + Integer.toBinaryString(i2));
        int i3 = 0177; // Octal (cero inicial)
        print("i3: " + Integer.toBinaryString(i3));
        char c = 0xffff; // máximo valor hex para char
        print("c: " + Integer.toBinaryString(c));
        byte b = 0x7f; // máximo valor hex para byte
        print("b: " + Integer.toBinaryString(b));
        short s = 0x7fff; // máximo valor hex para short
        print("s: " + Integer.toBinaryString(s));
        long n1 = 200L; // sufijo long
        long n2 = 200l; // sufijo long (pero puede ser confuso)
        long n3 = 200;
        float f1 = 1;
        float f2 = 1F; // sufijo float
        float f3 = 1f; // sufijo float
        double d1 = 1d; // sufijo double
        double d2 = 1D; // sufijo double
        // (Hex y Octal también funcionan con long)
    }
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 1111111111111111
*//*:~
```

Un carácter situado al final de un valor literal permite establecer su tipo. La **L** mayúscula o minúscula significa **long** (sin embargo, utilizar una **l** minúscula es confuso, porque puede parecerse al número uno). Una **F** mayúscula o minúscula significa **float**. Una **D** mayúscula o minúscula significa **double**.

Los valores hexadecimales (base 16), que funcionan con todos los tipos de datos enteros, se denotan mediante el prefijo **0x** o **0X** seguido de **0-9** o **a-f** en mayúscula o minúscula. Si se intenta inicializar una variable con un valor mayor que el máximo que puede contener (independientemente de la forma numérica del valor), el compilador dará un mensaje de error. Observe, en el código anterior, los valores hexadecimales máximos permitidos para **char**, **byte** y **short**. Si nos excedemos de éstos, el compilador transformará automáticamente el valor a **int** y nos dirá que necesitamos una *proyección hacia abajo* para la asignación (definiremos las proyecciones posteriormente en el capítulo). De esta forma, sabremos que nos hemos pasado del límite permitido.

Los valores octales (base 8) se denotan incluyendo un cero inicial en el número y utilizando sólo los dígitos 0-7.

No existe ninguna representación literal para los números binarios en C, C++ o Java. Sin embargo, a la hora de trabajar con notación hexadecimal y octal, a veces resulta útil mostrar la forma binaria de los resultados. Podemos hacer esto fácilmen-

te con los métodos `static toBinaryString()` de las clases `Integer` y `Long`. Observe que, cuando se pasan tipos más pequeños a `Integer.toBinaryString()`, el tipo se convierte automáticamente a `int`.

Ejercicio 8: (2) Demuestre que las notaciones hexadecimal y octal funcionan con los valores `long`. Utilice `Long.toBinaryString()` para mostrar los resultados.

Notación exponencial

Los exponentes utilizan una notación que a mí personalmente me resulta extraña:

```
//: operators/Exponents.java
// "e" significa "10 elevado a".

public class Exponents {
    public static void main(String[] args) {
        // 'e' en mayúscula o minúscula funcionan igual:
        float expFloat = 1.39e-43f;
        expFloat = 1.39E-43f;
        System.out.println(expFloat);
        double expDouble = 47e47d; // 'd' es opcional
        double expDouble2 = 47e47; // automáticamente double
        System.out.println(expDouble);
    }
} /* Output:
1.39E-43
4.7E48
*///:-
```

En el campo de las ciencias y de la ingeniería, ‘e’ hace referencia a la base de los logaritmos naturales, que es aproximadamente 2,718 (en Java hay disponible un valor `double` más preciso, que es `Math.E`). Esto se usa en expresiones de exponentiación, como por ejemplo $1.39 \times e^{-43}$, que significa 1.39×2.718^{-43} . Sin embargo, cuando se inventó el lenguaje de programación FORTRAN, decidieron que `e` significaría “diez elevado a”, lo cual es una decisión extraña, ya que FORTRAN fue diseñado para campos de la ciencia y de la ingeniería, así que cabría esperar que sus diseñadores tendrían en cuenta lo confuso de introducir esa ambigüedad². En cualquier caso, esta costumbre fue también introducida en C, C++ y ahora en Java. Por tanto, si el lector está habituado a pensar en `e` como en la base de los logaritmos naturales, tendrá que hacer una traducción mental cuando vea una expresión como `1.39 e-43f` en Java; ya que quiere decir 1.39×10^{-43} .

Observe que no es necesario utilizar el carácter sufijo cuando el compilador puede deducir el tipo apropiado. Con:

```
long n3 = 200;
```

no existe ninguna ambigüedad, por lo que una `L` después del 200 sería superfluo. Sin embargo, con:

```
float f4 = 1e-43f; // 10 elevado a
```

el compilador normalmente considera los números exponenciales como de tipo `double`, por lo que sin la `f` final, nos daría un error en el que nos informaría de que hay que usar una proyección para convertir el valor `double` a `float`.

Ejercicio 9: (1) Visualice los números más grande y más pequeño que se pueden representar con la notación exponencial en el tipo `float` y en el tipo `double`.

² John Kirkham escribe: “Comencé a escribir programas informáticos en 1962 en FORTRAN II en un IBM 1620. Por aquel entonces y a lo largo de las décadas de 1960 y 1970, FORTRAN era un lenguaje donde todo se escribía en mayúsculas. Probablemente, la razón era que muchos de los dispositivos de entrada eran antiguas unidades de teletipo que utilizaban el código Baudot de cinco bits que no disponían de minúsculas. La “E” en la notación exponencial era también mayúscula y no se confundía nunca con la base de los logaritmos naturales “e” que siempre se escribe en minúscula. La “E” simplemente quería decir exponencial, que era la base para el sistema de numeración que se estaba utilizando, que normalmente era 10. En aquella época, los programadores también empleaban los números octales. Aunque nunca vi que nadie lo utilizara, si yo hubiera visto un número octal en notación exponencial, habría considerado que estaba en base 8. La primera vez que vi un exponencial utilizando una “e” fue a finales de la década de 1970 y también a mí me pareció confuso; el problema surgió cuando empezaron a utilizarse minúsculas en FORTAN, no al principio. De hecho, disponíamos de funciones que podían usarse cuando se quisiera emplear la base de los logaritmos naturales, pero todas esas funciones se escribían en mayúsculas.”

Operadores bit a bit

Los operadores bit a bit permiten manipular bits individuales en un tipo de datos entero primitivo. Para generar el resultado, los operadores bit a bit realizan operaciones de álgebra booleana con los bits correspondientes de los dos argumentos.

Los operadores bit a bit proceden de la orientación a bajo nivel del lenguaje C, en el que a menudo se manipula el hardware directamente y es preciso configurar los bits de los registros hardware. Java se diseñó originalmente para integrarlo en codificadores para televisión, por lo que esta orientación a bajo nivel seguía teniendo sentido. Sin embargo, lo más probable es que no utilicemos demasiado esos operadores bit a bit en nuestros programas.

El operador bit a bit AND (**&**) genera un uno en el bit de salida si ambos bits de entrada son iguales a uno; en caso contrario, genera un cero. El operador OR bit a bit (**|**) genera un uno en el bit de salida si alguno de los bits de entrada es un uno y genera cero sólo si ambos bits de entrada son cero. El operador bit a bit EXCLUSIVE OR o XOR (**^**) genera un uno en el bit de salida si uno de los dos bits de entrada es un uno pero no ambos. El operador bit a bit NOT (**~**, también denominado operador de *complemento a uno*) es un operador unario, que sólo admite un argumento (todos los demás operadores bit a bit son operadores binarios). El operador bit a bit NOT genera el opuesto al bit de entrada, es uno si el bit de entrada es cero y es cero si el bit de entrada es uno.

Los operadores bit a bit y los operadores lógicos utilizan los mismos caracteres, por lo que resulta útil recurrir a un truco mnemónico para recordar cuál es el significado correcto. Como los bits son “pequeños” sólo se utiliza un carácter en los operadores bit a bit.

Los operadores bit a bit pueden combinarse con el signo **=** para unir la operación y la asignación: **&=**, **|=** y **^=** son operadores legítimos (puesto que **~** es un operador unario, no se puede combinar con el signo **=**).

El tipo **boolean** se trata como un valor de un único bit, por lo que es algo distinto de los otros tipos primitivos. Se puede realizar una operación AND, OR o XOR bit a bit, pero no se puede realizar una operación NOT bit a bit (presumiblemente, para evitar la confusión con la operación lógica NOT). Para los valores booleanos, los operadores bit a bit tienen el mismo efecto que los operadores lógicos, salvo porque no se aplica la regla de cortocircuito. Asimismo, las operaciones bit a bit con valores booleanos incluyen un operador lógico XOR que no forma parte de la lista de operadores “lógicos”. No se pueden emplear valores booleanos en expresiones de desplazamiento, las cuales vamos a describir a continuación.

Ejercicio 10: (3) Escriba un programa con dos valores constantes, uno en el que haya unos y ceros binarios alternados, con un cero en el dígito menos significativo, y el segundo con un valor también alternado pero con un uno en el dígito menos significativo (consejo: lo más fácil es usar constantes hexadecimales para esto). Tome estos dos valores y combínelos de todas las formas posibles utilizando los operadores bit a bit, y visualice los resultados utilizando **Integer.toBinaryString()**.

Operadores de desplazamiento

Los operadores de desplazamiento también sirven para manipular bits. Sólo se les puede utilizar con tipos primitivos enteros. El operador de desplazamiento a la izquierda (**<<**) genera como resultado el operando situado a la izquierda del operador después de desplazarlo hacia la izquierda el número de bits especificado a la derecha del operador (insertando ceros en los bits de menor peso). El operador de desplazamiento a la derecha con signo (**>>**) genera como resultado el operando situado a la izquierda del operador después de desplazarlo hacia la derecha el número de bits especificado a la derecha del operador. El desplazamiento a la derecha con signo **>>** utiliza lo que se denomina *extensión de signo*: si el valor es positivo, se insertan ceros en los bits de mayor peso; si el valor es negativo, se insertan unos en los bits de mayor peso. Java ha añadido también un desplazamiento a la derecha sin signo **>>>**, que utiliza lo que denomina *extensión con ceros*: independientemente del signo, se insertan ceros en los bits de mayor peso. Este operador no existe ni en C ni C++.

Si se desplaza un valor de tipo **char**, **byte** o **short**, será convertido a **int** antes de que el desplazamiento tenga lugar y el resultado será de tipo **int**. Sólo se utilizarán los bits de menor peso del lado derecho; esto evita que se realicen desplazamientos con un número de posiciones superior al número de bits de un valor **int**. Si se está operando con un valor **long**, se obtendrá un resultado de tipo **long** y sólo se emplearán los seis bits de menor peso del lado derecho, para así no poder desplazar más posiciones que el número de bits de un valor **long**.

Los desplazamientos se pueden combinar con el signo igual (**<<=** o **>>=** o **>>>=**). El lvalor se sustituye por el lvalor desplazado de acuerdo con lo que el rvalor marque. Existe un problema, sin embargo, con el desplazamiento a la derecha sin

signo combinado con la asignación. Si se usa con valores de tipo **byte** o **short**, no se obtienen los resultados correctos. En lugar de ello, se transforman a **int** y luego se desplazan a la derecha, pero a continuación se truncan al volver a asignar los valores a sus variables, por lo que se obtiene **-1** en esos casos. El siguiente ejemplo ilustra esta situación:

En el último desplazamiento, el valor resultante no se asigna de nuevo a **b**, sino que se imprime directamente, obteniéndose el comportamiento correcto.

He aquí un ejemplo que ilustra todos los operadores para el manejo de bits:

```
//: operators/BitManipulation.java
// Uso de los operadores bit a bit.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
```

```

int maxpos = 2147483647;
printBinaryInt("maxpos", maxpos);
int maxneg = -2147483648;
printBinaryInt("maxneg", maxneg);
printBinaryInt("i", i);
printBinaryInt("~i", ~i);
printBinaryInt("-i", -i);
printBinaryInt("j", j);
printBinaryInt("i & j", i & j);
printBinaryInt("i | j", i | j);
printBinaryInt("i ^ j", i ^ j);
printBinaryInt("i << 5", i << 5);
printBinaryInt("i >> 5", i >> 5);
printBinaryInt("(~i) >> 5", (~i) >> 5);
printBinaryInt("i >>> 5", i >>> 5);
printBinaryInt("(~i) >>> 5", (~i) >>> 5);

long l = rand.nextLong();
long m = rand.nextLong();
printBinaryLong("-1L", -1L);
printBinaryLong("+1L", +1L);
long ll = 9223372036854775807L;
printBinaryLong("maxpos", ll);
long lln = -9223372036854775808L;
printBinaryLong("maxneg", lln);
printBinaryLong("l", l);
printBinaryLong("-l", -l);
printBinaryLong("-1", -1);
printBinaryLong("m", m);
printBinaryLong("l & m", l & m);
printBinaryLong("l | m", l | m);
printBinaryLong("l ^ m", l ^ m);
printBinaryLong("l << 5", l << 5);
printBinaryLong("l >> 5", l >> 5);
printBinaryLong("(~l) >> 5", (~l) >> 5);
printBinaryLong("l >>> 5", l >>> 5);
printBinaryLong("(~l) >>> 5", (~l) >>> 5);
}

static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binary:\n" +
        Integer.toBinaryString(i));
}
static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binary:\n" +
        Long.toBinaryString(l));
}
/* Output:
-l, int: -1, binary:
111111111111111111111111111111
+l, int: 1, binary:
1
maxpos, int: 2147483647, binary:
111111111111111111111111111111
maxneg, int: -2147483648, binary:
10000000000000000000000000000000000000
i, int: -1172028779, binary:
1011101000100100010001010010101
-i, int: 1172028778, binary:
10001011101101110111010110101010

```

```

-i, int: 1172028779, binary:
 1000101110110111011110101101011
j, int: 1717241110, binary:
 1100110010110110000010100010110
i & j, int: 570425364, binary:
 100010000000000000000000010100
i | j, int: -25213033, binary:
 1111110011111110100011110010111
i ^ j, int: -695638397, binary:
 11011100011111110100011110000011
i << 5, int: 1149784736, binary:
 1000100100010000101001010100000
i >> 5, int: -36625900, binary:
 1111101110100010010001000010100
(~i) >> 5, int: 36625899, binary:
 1000101110110111011101011
i >>> 5, int: 97591828, binary:
 101110100010010001000010100
(~i) >>> 5, int: 36625899, binary:
 10001011101101110111101011
...
*/://:-/

```

Los dos métodos del final, `printBinaryInt()` y `printBinaryLong()`, toman un valor `int` o `long`, respectivamente, y lo imprimen en formato binario junto con una cadena de caracteres descriptiva. Además de demostrar el efecto de todos los operadores bit a bit para valores `int` y `long`, este ejemplo también muestra los valores mínimo, máximo, +1 y -1 para `int` y `long` para que vea el aspecto que tienen. Observe que el bit más alto representa el signo: 0 significa positivo y 1 significa negativo. En el ejemplo se muestra la salida de la parte correspondiente a los valores `int`.

La representación binaria de los números se denomina *complemento a dos con signo*.

Ejercicio 11: (3) Comience con un número que tenga un uno binario en la posición más significativa (consejo: utilice una constante hexadecimal). Emplee el operador de desplazamiento a la derecha con signo, desplace el valor a través de todas sus posiciones binarias, mostrando cada vez el resultado con `Integer.toBinaryString()`.

Ejercicio 12: (3) Comience con un número cuyos dígitos binarios sean todos iguales a uno. A continuación desplácelo a la izquierda y utilice el operador de desplazamiento a la derecha sin signo para desplazarlo a través de todas sus posiciones binarias, visualizando los resultados con `Integer.toBinaryString()`.

Ejercicio 13: (1) Escriba un método que muestre valores `char` en formato binario. Ejecútelo utilizando varios caracteres diferentes.

Operador ternario if-else

El operador *ternario*, también llamado operador *condicional* resulta inusual porque tiene tres operandos. Realmente se trata de un operador, porque genera un valor a diferencia de la instrucción `if-else` ordinaria, que veremos en la siguiente sección del capítulo. La expresión tiene la forma:

```
exp-booleana ? valor0 : valor1
```

Si `exp-booleana` se evalúa como `true`, se evalúa `valor0` y el resultado será el valor generado por el operador. Si `exp-booleana` es `false`, se evalúa `valor1` y su resultado pasará a ser el valor generado por el operador.

Por supuesto, podría utilizarse una instrucción `if-else` ordinaria (que se describe más adelante), pero el operador ternario es mucho más compacto. Aunque C (donde se originó este operador) se enorgullece de ser un lenguaje compacto, y el operador ternario puede que se haya introducido en parte por razones de eficiencia, conviene tener cuidado a la hora de emplearlo de forma cotidiana, ya que el código resultante puede llegar a ser poco legible.

El operador condicional es diferente de `if-else` porque genera un valor. He aquí un ejemplo donde se comparan ambas estructuras:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
    static int ternary(int i) {
        return i < 10 ? i * 100 : i * 10;
    }
    static int standardIfElse(int i) {
        if(i < 10)
            return i * 100;
        else
            return i * 10;
    }
    public static void main(String[] args) {
        print(ternary(9));
        print(ternary(10));
        print(standardIfElse(9));
        print(standardIfElse(10));
    }
} /* Output:
900
100
900
100
*///:-
```

Puede ver que el código de **ternary()** es más compacto de lo que sería si no dispusiéramos del operador ternario; la versión sin operador ternario se encuentra en **standardIfElse()**. Sin embargo, **standardIfElse()** es más fácil de comprender y además exige escribir muchos caracteres más. Así que asegúrese de ponderar bien las razones a la hora de elegir el operador ternario; normalmente, puede convenir utilizarlo cuando se quiera configurar una variable con uno de dos valores posibles.

Operadores + y += para String

Existe un uso especial de un operador en Java: los operadores + y += pueden usarse para concatenar cadenas, como ya hemos visto. Parece un uso bastante natural de estos operadores, aún cuando no encaje demasiado bien con la forma tradicional en que dichos operadores se emplean.

Esta capacidad le pareció adecuada a los diseñadores de C++, por lo que se añadió a C++ un mecanismo de *sobrecarga de operadores* para que los programadores C++ pudieran añadir nuevos significados casi a cualquier operador. Lamentablemente, la sobrecarga de operadores combinada con alguna de las otras restricciones de C++, resulta una característica excesivamente complicada para que los programadores la incluyan en el diseño de sus clases. Aunque la sobrecarga de operadores habría sido mucho más fácil de implementar en Java de lo que lo era en C++ (como se ha demostrado en el lenguaje C#, que *si* que dispone de un sencillo mecanismo de sobrecarga de operadores), se consideraba que esta característica seguía siendo demasiado compleja, por lo que los programadores Java no pueden implementar sus propios operadores sobrecargados, a diferencia de los programadores de C++ y C#.

El uso de los operadores para valores **String** presenta ciertos comportamientos interesantes. Si una expresión comienza con un valor **String**, todos los operandos que siguen también tendrán que ser cadenas de caracteres (recuerde que el compilador transforma automáticamente a **String** toda secuencia de caracteres encerrada entre comillas dobles).

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
    public static void main(String[] args) {
        int x = 0, y = 1, z = 2;
        String s = "x, y, z ";
        print(s + x + y + z);
```

```

        print(x + " " + s); // Convierte x a String
        s += "(summed) = "; // Operador de concatenación
        print(s + (x + y + z));
        print(" " + x); // Abreviatura de Integer.toString()
    }
} /* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*///:-

```

Observe que la salida de la primera instrucción de impresión es ‘012’ en lugar de sólo ‘3’, que es lo que obtendría si se estuvieran sumando los valores enteros. Esto es porque el compilador Java convierte `x`, `y` y `z` a su representación `String` y concatena esas cadenas de caracteres, en lugar de efectuar primero la suma. La segunda instrucción de impresión convierte la variable inicial a `String`, por lo que la conversión a cadena no depende de qué es lo que haya primero. Por último, podemos ver el uso del operador `+=` para añadir una cadena de caracteres a `s`, y el uso de paréntesis para controlar el orden de evaluación de la expresión, de modo que los valores enteros se sumen realmente antes de la visualización.

Observe el último ejemplo de `main()`: en ocasiones, se encontrará en los programas un valor `String` vacío seguido de `+` y una primitiva, como forma de realizar la conversión sin necesidad de invocar el método explícito más engorroso, (`Integer.toString()`, en este caso).

Errores comunes a la hora de utilizar operadores

Uno de los errores que se pueden producir a la hora de emplear operadores es el de tratar de no incluir los paréntesis cuando no se está del todo seguro acerca de la forma que se evaluará una expresión. Esto, que vale para muchos lenguajes también es cierto para Java.

Un error extremadamente común en C y C++ sería el siguiente:

```

while(x = y) {
    // ...
}

```

El programador estaba intentando, claramente, comprobar la equivalencia (`==`) en lugar de hacer una asignación. En C y C++ el resultado de esta asignación será siempre `true` si `y` es distinto de cero, por lo que probablemente se produzca un bucle infinito. En Java, el resultado de esta expresión no es de tipo `boolean`, pero el compilador espera un valor `boolean` y no realizará ninguna conversión a partir de un valor `int`, por lo que dará un error en tiempo de compilación y detectará el problema antes de que ni siquiera intentemos ejecutar el programa. Por tanto, este error nunca puede producirse en Java (la única posibilidad de que no se tenga un error de tiempo de compilación, es cuando `x` e `y` son de tipo `boolean`, en cuyo caso `x = y` es una expresión legal, aunque en el ejemplo anterior probablemente su uso se deba a un error).

Un problema similar en C y C++ consiste en utilizar los operadores bit a bit AND y OR en lugar de las versiones lógicas. Los operadores bit a bit AND y OR utilizan uno de los caracteres (`&` o `|`) mientras que los operadores lógicos AND y OR utilizan dos (`&&` y `||`). Al igual que `con = y ==`, resulta fácil confundirse y escribir sólo uno de los caracteres en lugar de dos. En Java, el compilador vuelve a evitar este tipo de error, porque no permite emplear un determinado tipo de datos en un lugar donde no sea correcto hacerlo.

Operadores de proyección

La palabra *proyección* (*cast*) hace referencia a la conversión explícita de datos de un tipo a otro. Java cambiará automáticamente un tipo de datos a otro cada vez que sea apropiado. Por ejemplo, si se asigna un valor entero a una variable de coma flotante, el compilador convertirá automáticamente el valor `int` a `float`. El mecanismo de conversión nos permite realizar esta conversión de manera explícita, o incluso forzarla en situaciones donde normalmente no tendría lugar.

Para realizar una proyección, coloque el tipo de datos deseado entre paréntesis a la izquierda del valor que haya que convertir, como en el siguiente ejemplo:

```
//: operators/Casting.java

public class Casting {
    public static void main(String[] args) {
        int i = 200;
        long lng = (long)i;
        lng = i; // "Ensanchamiento," por lo que no se requiere conversión
        long lng2 = (long)200;
        lng2 = 200;
        // Una "conversión de estrechamiento":
        i = (int)lng2; // Proyección requerida
    }
} //:-
```

Como podemos ver, resulta posible aplicar una proyección de tipo tanto a los valores numéricos como a las variables. Observe que se pueden también introducir proyecciones superfluas, por ejemplo, el compilador promocionará automáticamente un valor **int** a **long** cuando sea necesario. Sin embargo, podemos utilizar esas proyecciones superfluas con el fin de resaltar la operación o de clarificar el código. En otras situaciones, puede que la proyección de tipo sea esencial para que el código llegue a compilarse.

En C y C++, las operaciones de proyección de tipos pueden provocar algunos dolores de cabeza. En Java, la proyección de tipos resulta siempre segura, con la excepción de que, cuando se realiza una de las denominadas *conversiones de estrechamiento* (es decir, cuando se pasa de un tipo de datos que puede albergar más información a otro que no permite albergar tanta), se corre el riesgo de perder información. En estos casos, el compilador nos obliga a emplear una proyección, como diciéndonos: “Esta conversión puede ser peligrosa, si quieres que lo haga de todos modos, haz que esa proyección sea explícita”. Con una *conversión de ensanchamiento*, no hace falta una proyección explícita, porque el nuevo tipo permitirá albergar con creces la información del tipo anterior, de modo que nunca se puede perder información.

Java permite proyectar cualquier tipo primitivo a cualquier otro, excepto en el caso de **boolean**, que no permite efectuar ningún tipo de proyección. Los tipos de clase tampoco permiten efectuar proyecciones: para convertir uno de estos tipos en otro, deben existir métodos especiales (posteriormente, veremos que los objetos pueden proyectarse dentro de una *familia* de tipos; un **Olmo** puede proyectarse sobre un **Árbol** y viceversa, pero no sobre un tipo externo como pueda ser **Roca**).

Truncamiento y redondeo

Cuando se realizan conversiones de estrechamiento, es necesario prestar atención a los problemas de truncamiento y redondeo. Por ejemplo, si efectuamos una proyección de un valor de coma flotante sobre un valor entero, ¿qué es lo que haría Java? Por ejemplo, si tenemos el valor 29,7 y lo proyectamos sobre un **int**, ¿el valor resultante será 30 o 29? La respuesta a esta pregunta puede verse en el siguiente ejemplo:

```
//: operators/CastingNumbers.java
// ¿Qué ocurre cuando se proyecta un valor float
// o double sobre un valor entero?
import static net.mindview.util.Print.*;

public class CastingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print("(int)above: " + (int)above);
        print("(int)below: " + (int)below);
        print("(int)fabove: " + (int)fabove);
        print("(int)fbelow: " + (int)fbelow);
    }
} /* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:-
```

Así que la respuesta es que al efectuar la proyección de **float** o **double** a un valor entero, siempre se trunca el correspondiente número. Si quisieramos que el resultado se redondeara habría que utilizar los métodos **round()** de **java.lang.Math**:

```
//: operators/RoundingNumbers.java
// Redondeo de valores float y double.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print("Math.round(above): " + Math.round(above));
        print("Math.round(below): " + Math.round(below));
        print("Math.round(fabove): " + Math.round(fabove));
        print("Math.round(fbelow): " + Math.round(fbelow));
    }
} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:-
```

Puesto que **round()** es parte de **java.lang**, no hace falta ninguna instrucción adicional de importación para utilizarlo.

Promoción

Cuando comience a programar en Java, descubrirá que si hace operaciones matemáticas o bit a bit con tipos de datos primitivos más pequeños que **int** (es decir, **char**, **byte** o **short**), dichos valores serán promocionados a **int** antes de realizar las operaciones, y el valor resultante será de tipo **int**. Por tanto, si se quiere asignar el resultado de nuevo al tipo más pequeño, es necesario emplear una proyección (y, como estamos realizando una asignación a un tipo de menor tamaño, perderemos información). En general, el tipo de datos de mayor tamaño dentro de una expresión es el que determina el tamaño del resultado de esa expresión, si se multiplica un valor **float** por otro **double**, el resultado será **double**; si se suman un valor **int** y uno **long**, el resultado será **long**.

Java no tiene operador “sizeof”

En C y C++, el operador **sizeof()** nos dice el número de bytes asignado a un elemento de datos. La razón más importante para el uso de **sizeof()** en C y C++ es la portabilidad. Los diferentes tipos de datos pueden tener diferentes tamaños en distintas máquinas, por lo que el programador debe averiguar el tamaño de esos tipos a la hora de realizar operaciones que sean sensibles al tamaño. Por ejemplo, una computadora puede almacenar los enteros en 32 bits, mientras que otras podrían almacenarlos en 16 bits. Los programas podrían, así, almacenar valores de mayor tamaño en variables de tipo entero en la primera máquina. Como puede imaginarse, la portabilidad es un verdadero quebradero de cabeza para los programadores de C y C++.

Java no necesita un operador **sizeof()** para este propósito, porque todos los tipos de datos tienen el mismo tamaño en todas las máquinas. No es necesario que tengamos en cuenta la portabilidad en este nivel, ya que esa portabilidad forma parte del propio diseño del lenguaje.

Compendio de operadores

El siguiente ejemplo muestra qué tipos de datos primitivos pueden utilizarse con determinados operadores concretos. Básicamente, se trata del mismo ejemplo repetido una y otra vez pero empleando diferentes tipos de datos primitivos. El archivo se compilará sin errores porque las líneas que los incluyen están desactivadas mediante comentarios de tipo **!!**.

```
//: operators/AllOps.java
// Comprueba todos los operadores con todos los tipos de datos primitivos
```

```

// para mostrar cuáles son aceptables por el compilador Java.

public class AllOps {
    // Para aceptar los resultados de un test booleano:
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // Operadores aritméticos:
        //! x = x * y;
        //! x = x / y;
        //! x = x % y;
        //! x = x + y;
        //! x = x - y;
        //! x++;
        //! x--;
        //! x = +y;
        //! x = -y;
        // Relacionales y lógicos:
        //! f(x > y);
        //! f(x >= y);
        //! f(x < y);
        //! f(x <= y);
        f(x == y);
        f(x != y);
        f(!y);
        x = x && y;
        x = x || y;
        // Operadores bit a bit:
        //! x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Asignación compuesta:
        //! x += y;
        //! x -= y;
        //! x *= y;
        //! x /= y;
        //! x %= y;
        //! x <<= 1;
        //! x >>= 1;
        //! x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // Proyección:
        //! char c = (char)x;
        //! byte b = (byte)x;
        //! short s = (short)x;
        //! int i = (int)x;
        //! long l = (long)x;
        //! float f = (float)x;
        //! double d = (double)x;
    }
    void charTest(char x, char y) {
        // Operadores aritméticos:
        x = (char)(x * y);
        x = (char)(x / y);
    }
}

```

```

x = (char)(x % y);
x = (char)(x + y);
x = (char)(x - y);
x++;
x--;
x = (char)+y;
x = (char)-y;
// Relacionales y lógicos:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Operadores bit a bit:
x= (char)-y;
x = (char)(x & y);
x = (char)(x | y);
x = (char)(x ^ y);
x = (char)(x << 1);
x = (char)(x >> 1);
x = (char)(x >>> 1);
// Asignación compuesta:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= y;
x >= y;
x &= y;
x ^= y;
x |= y;
// Proyección:
//! boolean bl = (boolean)x;
byte b = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void byteTest(byte x, byte y) {
// Operadores aritméticos:
x = (byte)(x* y);
x = (byte)(x / y);
x = (byte)(x % y);
x = (byte)(x + y);
x = (byte)(x - y);
x++;
x--;
x = (byte)+ y;
x = (byte)- y;
// Relacionales y lógicos:
f(x > y);

```

```

f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Operadores bit a bit:
x = (byte)~y;
x = (byte)(x & y);
x = (byte)(x | y);
x = (byte)(x ^ y);
x = (byte)(x << 1);
x = (byte)(x >> 1);
x = (byte)(x >>> 1);
// Asignación compuesta:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= l;
x >= l;
x >>= l;
x &= y;
x ^= y;
x |= y;
// Proyección:
/// boolean bl = (boolean)x;
char c = (char)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void shortTest(short x, short y) {
    // Operadores aritméticos:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores bit a bit:
}

```

```

x = (short)-y;
x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
x = (short)(x >>> 1);
// Asignación compuesta:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= l;
x >= l;
x >>= l;
x &= y;
x ^= y;
x |= y;
// Proyección:
// boolean bl = (boolean)x;
char c = (char)x;
byte b = (byte)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void intTest(int x, int y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    // f(!x);
    // f(x && y);
    // f(x || y);
    // Operadores bit a bit:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Asignación compuesta:
    x += y;
}

```

```

x -= y;
x *= y;
x /= y;
x %= y;
x <= l;
x >= l;
x >>>= l;
x &= y;
x ^= y;
x |= y;
// Proyección:
//! boolean bl = (boolean)x;
char c = (char)x;
byte b = (byte)x;
short s = (short)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void longTest(long x, long y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores bit a bit:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << l;
    x = x >> l;
    x = x >>> l;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <= l;
    x >= l;
    x >>>= l;
    x &= y;
    x ^= y;
}

```

```

x |= y;
// Proyección:
//! boolean bl = (boolean)x;
char c = (char)x;
byte b = (byte)x;
short s = (short)x;
int i = (int)x;
float f = (float)x;
double d = (double)x;
}
void floatTest(float x, float y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores bit a bit:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <= 1;
    //! x >= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Proyección:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
}

```

```

}

void doubleTest(double x, double y) {
    // Operadores aritméticos:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relacionales y lógicos:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operadores bit a bit:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignación compuesta:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <= 1;
    //! x >= 1;
    //! x >>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Proyección:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
}
} //:-

```

Observe que **boolean** es bastante limitado. A una variable de este tipo se le pueden asignar los valores **true** y **false**, y se puede comprobar si el valor es verdadero o falso, pero no se pueden sumar valores booleanos ni realizar ningún otro tipo de operación con ellos.

En **char**, **byte** y **short**, puede ver el efecto de la promoción con los operadores aritméticos. Toda operación aritmética sobre cualquiera de estos tipos genera un resultado **int**, que después debe ser proyectado explícitamente al tipo original (una conversión de estrechamiento que puede perder información) para realizar la asignación a dicho tipo. Sin embargo, con los valo-

res **int** no es necesaria una proyección, porque todo es ya de tipo **int**. Sin embargo, no se crea que todas las operaciones son seguras. Si se multiplican dos valores **int** que sean lo suficientemente grandes, se producirá un desbordamiento en el resultado, como se ilustra en el siguiente ejemplo:

```
//: operators/Overflow.java
// ¡Sorpresa! Java permite los desbordamientos.

public class Overflow {
    public static void main(String[] args) {
        int big = Integer.MAX_VALUE;
        System.out.println("big = " + big);
        int bigger = big * 4;
        System.out.println("bigger = " + bigger);
    }
} /* Output:
big = 2147483647
bigger = -4
*///:-
```

No se obtiene ningún tipo de error o advertencia por parte del compilador, y tampoco se genera ninguna excepción en tiempo de ejecución. El lenguaje Java es muy bueno, aunque no hasta ese punto.

Las asignaciones compuestas *no* requieren proyecciones para **char**, **byte** o **short**, aún cuando estén realizando promociones que provocan los mismos resultados que las operaciones aritméticas directas. Esto resulta quizás algo sorprendente pero, por otro lado, la posibilidad de no incluir la proyección simplifica el código.

Como puede ver, con la excepción de **boolean**, podemos proyectar cualquier tipo primitivo sobre cualquier otro tipo primitivo. De nuevo, recalquemos que es preciso tener en cuenta los efectos de las conversiones de estrechamiento a la hora de realizar proyecciones sobre tipos de menor tamaño; en caso contrario, podríamos perder información inadvertidamente durante la proyección.

Ejercicio 14: (3) Escriba un método que tome dos argumentos de tipo **String** y utilice todas las comparaciones **boolean** para comparar las dos cadenas de caracteres e imprimir los resultados. Para las comparaciones **==** y **!=**, realice también la prueba con **equals()**. En **main()**, invoque el método que haya escrito, utilizando varios objetos **String** diferentes.

Resumen

Si tiene experiencia con algún lenguaje que emplee una sintaxis similar a la de C, podrá ver que los operadores de Java son tan similares que la curva aprendizaje es prácticamente nula. Si este capítulo le ha resultado difícil, asegúrese de echar un vistazo a la presentación multimedia *Thinking in C*, disponible en www.MindView.net.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, que está disponible para la venta en www.MindView.net.

Control de ejecución

4

Al igual que las criaturas sensibles, un programa debe manipular su mundo y tomar decisiones durante la ejecución. En Java, las decisiones se toman mediante las instrucciones de control de ejecución.

Java utiliza todas las instrucciones de control de ejecución de C, por lo que si ha programado antes con C o C++, la mayor parte de la información que vamos a ver en este capítulo le resultará familiar. La mayoría de los lenguajes de programación procedimental disponen de alguna clase de instrucciones de control, y suelen existir solapamientos entre los distintos lenguajes. En Java, las palabras clave incluyen **if-else**, **while**, **do-while**, **for**, **return**, **break** y una instrucción de selección denominada **switch**. Sin embargo, Java no soporta la despreciada instrucción **goto** (que a pesar de ello en ocasiones representa la forma más directa de resolver ciertos tipos de problemas). Se puede continuar realizando un salto de estilo **goto**, pero está mucho más restringido que un **goto** típico.

true y false

Todas las instrucciones condicionales utilizan la veracidad o falsedad de una expresión condicional para determinar la ruta de ejecución. Un ejemplo de expresión condicional sería **a == b**. Aquí, se utiliza el operador condicional **==** para ver si el valor de **a** es equivalente al valor de **b**. La expresión devuelve **true** o **false**. Podemos utilizar cualquiera de los operadores relacionales que hemos empleado en el capítulo anterior para escribir una instrucción condicional. Observe que Java no permite utilizar un número como **boolean**, a diferencia de lo que sucede en C y C++ (donde la veracidad se asocia con valores distintos cero y la falsedad con cero). Si quiere emplear un valor no **boolean** dentro de una prueba **boolean**, como por ejemplo **if(a)**, deberá primero convertir el valor al tipo **boolean** usando una expresión condicional, como por ejemplo **if(a != 0)**.

if-else

La instrucción **if-else** representa la forma más básica de controlar el flujo de un programa. La cláusula **else** es opcional, por lo que se puede ver **if** de dos formas distintas:

```
if(expresión-boleana)
    instrucción
o
if(expresión-boleana)
    instrucción
else
    instrucción
```

La *expresión-boleana* debe producir un resultado **boolean**. La *instrucción* puede ser una instrucción simple terminada en punto y coma o una instrucción compuesta, que es un grupo de instrucciones simples encerrado entre llaves. Allí donde empleemos la palabra *instrucción* querremos decir siempre que esa instrucción puede ser simple o compuesta.

Como ejemplo de **if-else**, he aquí un método **test()** que indica si un cierto valor está por encima, por debajo o es equivalente a un número objetivo:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
    static int result = 0;
    static void test(int testval, int target) {
        if(testval > target)
            result = +1;
        else if(testval < target)
            result = -1;
        else
            result = 0; // Coincidencia
    }
    public static void main(String[] args) {
        test(10, 5);
        print(result);
        test(5, 10);
        print(result);
        test(5, 5);
        print(result);
    }
} /* Output:
1
-1
0
*///:-
```

En la parte central de `test()`, también puede ver una instrucción “`else if`,” que no es una nueva palabra clave sino una instrucción `else` seguida de una nueva instrucción `if`.

Aunque Java, como sus antecesores C y C++, es un lenguaje de “formato libre” resulta habitual sangrar el cuerpo de las instrucciones de control de flujo, para que el lector pueda determinar fácilmente dónde comienzan y dónde terminan.

Iteración

Los bucles de ejecución se controlan mediante `while`, `do-while` y `for`, que a veces se clasifican como *instrucciones de iteración*. Una determinada instrucción se repite hasta que la *expresión-booleana* de control se evalúe como `false`. La forma de un bucle `while` es:

```
while(expresión-booleana)
    instrucción
```

La *expresión-booleana* se evalúa una vez al principio del bucle y se vuelve a evaluar antes de cada sucesiva iteración de la instrucción.

He aquí un ejemplo simple que genera números aleatorios hasta que se cumple una determinada condición.

```
//: control/WhileTest.java
// Ilustra el bucle while.

public class WhileTest {
    static boolean condition() {
        boolean result = Math.random() < 0.99;
        System.out.print(result + ", ");
        return result;
    }
    public static void main(String[] args) {
        while(condition())
            System.out.println("Inside 'while'");
            System.out.println("Exited 'while'");
    }
} /* (Ejecútelo para ver la salida) *///:-
```

El método **condition()** utiliza el método **random()** de tipo **static** de la biblioteca **Math**, que genera un valor **double** comprendido entre 0 y 1 (incluye 0, pero no 1.) El valor **result** proviene del operador de comparación **<**, que genera un resultado de tipo **boolean**. Si se imprime un valor **boolean**, automáticamente se obtiene la cadena de caracteres apropiada “true” o “false”. La expresión condicional para el bucle **while** dice: “repite las instrucciones del cuerpo mientras que **condition()** devuelva **true**”.

do-while

La forma de **do-while** es

```
do
    instrucción
    while(expresión-boleana);
```

La única diferencia entre **while** y **do-while** es que la instrucción del bucle **do-while** siempre se ejecuta al menos una vez, incluso aunque la expresión se evalúe como **false** la primera vez. En un bucle **while**, si la condición es **false** la primera vez, la instrucción nunca llega a ejecutarse. En la práctica, **do-while** es menos común que **while**.

for

El bucle **for** es quizás la forma de iteración más habitualmente utilizada. Este bucle realiza una inicialización antes de la primera iteración. Después realiza una prueba condicional y, al final de cada iteración, lleva a cabo alguna forma de “avance de paso”. La forma del bucle **for** es:

```
for(inicialización; expresión-boleana; paso)
    instrucción
```

Cualquiera de las expresiones *inicialización*, *expresión-boleana* o *paso* puede estar vacía. La expresión booleana se comprueba antes de cada iteración y, en cuanto se evalúe como **false**, la ejecución continúa en la línea que sigue a la instrucción **for**. Al final de cada bucle, se ejecuta el *paso*.

Los bucles **for** se suelen utilizar para tareas de “recuento”:

```
//: control>ListCharacters.java
// Ilustra los bucles "for" enumerando
// todas las letras minúsculas ASCII.

public class ListCharacters {
    public static void main(String[] args) {
        for(char c = 0; c < 128; c++)
            if(Character.isLowerCase(c))
                System.out.println("value: " + (int)c +
                    " character: " + c);
    }
} /* Output:
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 100 character: d
value: 101 character: e
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
...
*/

```

Observe que la variable **c** se define en el mismo lugar en el que se la utiliza, dentro de la expresión de control correspondiente al bucle **for**, en lugar de definirla al principio de **main()**. El ámbito de **c** es la instrucción controlada por **for**.

Este programa también emplea la clase "envoltorio" `java.lang.Character`, que no sólo envuelve el tipo primitivo `char` dentro de un objeto, sino que también proporciona otras utilidades. Aquí el método `static isLowerCase()` se usa para detectar si el carácter en cuestión es una letra minúscula.

Los lenguajes procedimentales tradicionales como C requieren que se definan todas las variables al comienzo de un bloque, de modo que cuando el compilador cree un bloque, pueda asignar espacio para esas variables. En Java y C++, se pueden distribuir las declaraciones de variables por todo el bloque, definiéndolas en el lugar que se las necesite. Esto permite un estilo más natural de codificación y hace que el código sea más fácil de entender.

- Ejercicio 1:** (1) Escriba un programa que imprima los valores comprendidos entre 1 y 100.
- Ejercicio 2:** (2) Escriba un programa que genere 25 valores `int` aleatorios. Para cada valor, utilice una instrucción `if-else` para clasificarlo como mayor que, menor que o igual a un segundo valor generado aleatoriamente.
- Ejercicio 3:** (1) Modifique el Ejercicio 2 para que el código quede rodeado por un bucle `while` "infinito". De este modo, el programa se ejecutará hasta que lo interrumpa desde el teclado (normalmente, pulsando Control-C).
- Ejercicio 4:** (3) Escriba un programa que utilice dos bucles `for` anidados y el operador de módulo (%) para detectar e imprimir números primos (números enteros que no son divisibles por ningún número excepto por sí mismos y por 1).
- Ejercicio 5:** (4) Repita el Ejercicio 10 del capítulo anterior, utilizando el operador ternario y una comprobación de tipo bit a bit para mostrar los unos y ceros en lugar de `Integer.toBinaryString()`.

El operador coma

Anteriormente en el capítulo, hemos dicho que el *operador coma* (no el *separador coma* que se emplea para separar definiciones y argumentos de métodos) sólo tiene un uso en Java: en la expresión de control de un bucle `for`. Tanto en la parte correspondiente a la inicialización como en la parte correspondiente al paso de la expresión de control, podemos incluir una serie de instrucciones separadas por comas, y dichas instrucciones se evaluarán secuencialmente.

Con el operador coma, podemos definir múltiples variables dentro de una instrucción `for`, pero todas ellas deben ser del mismo tipo:

```
//: control/CommaOperator.java

public class CommaOperator {
    public static void main(String[] args) {
        for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
            System.out.println("i = " + i + " j = " + j);
        }
    } /* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
*///:-
```

La definición `int` de la instrucción `for` cubre tanto a `i` como a `j`. La parte de inicialización puede tener cualquier número de definiciones de *un mismo tipo*. La capacidad de definir variables en una expresión de control está limitada a los bucles `for`. No puede emplearse esta técnica en ninguna otra de las restantes instrucciones de selección o iteración.

Puede ver que, tanto en la parte de inicialización como en la de paso, las instrucciones se evalúan en orden secuencial.

Sintaxis `foreach`

Java SE5 introduce una sintaxis `for` nueva, más sucinta, para utilizarla con matrices y contenedores (hablaremos más en detalle sobre este tipo de objetos en los Capítulos 16, *Matrices*, y 17, *Análisis detallado de los contenedores*). Esta sintaxis se denomina *sintaxis foreach* (para todos), y quiere decir que no es necesario crear una variable `int` para efectuar un recuento a través de una secuencia de elementos: el bucle `for` se encarga de generar cada elemento automáticamente.

Por ejemplo, suponga que tiene una matriz de valores **float** y que quiere seleccionar cada uno de los elementos de la matriz:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
    public static void main(String[] args) {
        Random rand = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
            f[i] = rand.nextFloat();
        for(float x : f)
            System.out.println(x);
    }
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:-
```

La matriz se rellena utilizando el antiguo bucle **for**, porque debe accederse a ella mediante un índice. Puede ver la sintaxis *foreach* en la línea:

```
for(float x : f) {
```

Esto define una variable **x** de tipo **float** y asigna secuencialmente cada elemento de **f** a **x**.

Cualquier método que devuelve una matriz es buen candidato para emplearlo con la sintaxis *foreach*. Por ejemplo, la clase **String** tiene un método **toCharArray()** que devuelve una matriz de **char**, por lo que podemos iterar fácilmente a través de los caracteres de una matriz:

```
//: control/ForEachString.java

public class ForEachString {
    public static void main(String[] args) {
        for(char c : "An African Swallow".toCharArray())
            System.out.print(c + " ");
    }
} /* Output:
A n   A f r i c a n   S w a l l o w
*///:-
```

Como podremos ver en el Capítulo 11, *Almacenamiento de objetos*, la sintaxis *foreach* también funciona con cualquier objeto que sea de tipo **Iterable**.

Muchas instrucciones **for** requieren ir paso a paso a través de una secuencia de valores enteros como ésta:

```
for(int i = 0; i < 100; i++)
```

Para este tipo de bloques, la sintaxis *foreach* no funcionará a menos que queramos crear primero una matriz de valores **int**. Para simplificar esta tarea, he creado un método denominado **range()** en **net.mindview.util.Range** que genera automáticamente la matriz apropiada. La intención es que **range()** se utilice como importación de tipo **static**:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;
```

```

public class ForEachInt {
    public static void main(String[] args) {
        for(int i : range(10)) // 0..9
            printnb(i + " ");
        print();
        for(int i : range(5, 10)) // 5..9
            printnb(i + " ");
        print();
        for(int i : range(5, 20, 3)) // 5..20 step 3
            printnb(i + " ");
        print();
    }
} /* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*///:-
```

El método **range()** está *sobrecargado*, lo que quiere decir que puede utilizarse el mismo método con diferentes listas de argumentos (en breve hablaremos del mecanismo de sobrecarga). La primera forma sobrecargada de **range()** empieza en cero y genera valores hasta el extremo superior del rango, sin incluir éste. La segunda forma comienza en el primer valor y va hasta un valor menos que el segundo, y la tercera forma incluye un valor de paso, de modo que los incrementos se realizan según ese valor. **range()** es una versión muy simple de lo que se denomina *generador*, que es un concepto del que hablaremos posteriormente en el libro.

Observe que aunque **range()** permite el uso de la sintaxis *foreach* en más lugares, mejorando así la legibilidad del código, es algo menos eficiente, por lo que se está utilizando el programa con el fin de conseguir la máxima velocidad, conviene que utilice un *perfilador*, que es una herramienta que mide el rendimiento del código.

Podrá observar también el uso de **printnb()** además de **print()**. El método **printnb()** no genera un carácter de nueva línea, por lo que permite escribir una línea en sucesivos fragmentos.

La sintaxis *foreach* no sólo ahorra tiempo a la hora de escribir el código. Lo más importante es que facilita la lectura y comunica perfectamente *qué* es lo que estamos tratando de hacer (obtener cada elemento de la matriz) en lugar de proporcionar los detalles acerca de *cómo* lo estamos haciendo (“Estoy creando este índice para poder usarlo en la selección de cada uno de los elementos de la matriz”). Utilizaremos la sintaxis *foreach* siempre que sea posible a lo largo del libro.

return

Diversas palabras clave representan lo que se llama un *salto incondicional*, lo que simplemente quiere decir que el salto en el flujo de ejecución se produce sin realizar previamente comprobación alguna. Dichas palabras clave incluyen **return**, **break**, **continue** y una forma de saltar a una instrucción etiquetada de forma similar a la instrucción **goto** de otros lenguajes.

La palabra clave **return** tiene dos objetivos: especifica qué valor devolverá un método (si no tiene un valor de retorno de tipo **void**) y hace que la ejecución salga del método actual devolviendo ese valor. Podemos reescribir el método **test()** precedente para aprovechar esta característica:

```

//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
    static int test(int testval, int target) {
        if(testval > target)
            return +1;
        else if(testval < target)
            return -1;
        else
            return 0; // Coincidencia
    }
}
```

```

public static void main(String[] args) {
    print(test(10, 5));
    print(test(5, 10));
    print(test(5, 5));
}
} /* Output:
1
-1
0
*///:-

```

No hay necesidad de la cláusula **else**, porque el método no continuará después de ejecutar una instrucción **return**.

Si no incluye una instrucción **return** en un método que devuelve un valor **void**, habrá una instrucción **return** implícita al final de ese método, así que no siempre es necesario incluir dicha instrucción. Sin embargo, si el método indica que va a devolver cualquier otro valor distinto de **void**, hay que garantizar que todas las rutas de ejecución del código devuelvan un valor.

Ejercicio 6: (2) Modifique los dos métodos **test()** de los dos programas anteriores para que admitan dos argumentos adicionales, **begin** y **end**, y para que se compruebe **testval** para ver si se encuentra dentro del rango comprendido entre **begin** y **end** (ambos incluidos).

break y continue

También se puede controlar el flujo del bucle dentro del cuerpo de cualquier instrucción de iteración utilizando **break** y **continue**. **break** provoca la salida del bucle sin ejecutar el resto de las instrucciones. La instrucción **continue** detiene la ejecución de la iteración actual y vuelve al principio del bucle para comenzar con la siguiente iteración.

Este programa muestra ejemplos de **break** y **continue** dentro de bucles **for** y **while**:

```

//: control/BreakAndContinue.java
// Ilustra las palabras clave break y continue.
import static net.mindview.util.Range.*;

public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Fuera del bucle
            if(i % 9 != 0) continue; // Siguiente iteración
            System.out.print(i + " ");
        }
        System.out.println();
        // Uso de foreach:
        for(int i : range(100)) {
            if(i == 74) break; // Fuera del bucle
            if(i % 9 != 0) continue; // Siguiente iteración
            System.out.print(i + " ");
        }
        System.out.println();
        int i = 0;
        // Un "bucle infinito":
        while(true) {
            i++;
            int j = i * 27;
            if(j == 1269) break; // Fuera del bucle for
            if(i % 10 != 0) continue; // Principio del bucle
            System.out.print(i + " ");
        }
    } /* Output:
1
-1
0
*///:-

```

```

0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:-

```

En el bucle **for**, el valor de **i** nunca llega a 100, porque la instrucción **break** hace que el bucle termine cuando **i** vale 74. Normalmente, utilizaremos una instrucción **break** como ésta sólo si no sabemos cuándo va a cumplirse la condición de terminación. La instrucción **continue** hace que la ejecución vuelva al principio del bucle de iteración (incrementando por tanto **i**) siempre que **i** no sea divisible por 9. Cuando lo sea, se imprimirá el valor.

El segundo bucle **for** muestra el uso de la sintaxis **foreach** y como ésta produce los mismos resultados.

Finalmente, podemos ver un bucle **while** “infinito” que se estaría ejecutando, en teoría, por siempre. Sin embargo, dentro del bucle hay una instrucción **break** que hará que salgamos del bucle. Además, podemos ver que la instrucción **continue** devuelve el control al principio del bucle sin ejecutar nada de lo que hay después de dicha instrucción **continue** (por tanto, la impresión sólo se produce en el segundo bucle cuando el valor de **i** es divisible por 10). En la salida, podemos ver que se imprime el valor 0, porque $0 \% 9$ da como resultado 0.

Una segunda forma del bucle infinito es **for(;;)**. El compilador trata tanto **while(true)** como **for(;;)** de la misma forma, por lo que podemos utilizar una de las dos formas según prefiramos.

Ejercicio 7: (1) Modifique el Ejercicio 1 para que el programa termine usando la palabra clave **break** con el valor 99. Intente utilizar **return** en su lugar.

La despreciada instrucción “goto”

La palabra clave **goto** ha estado presente en muchos lenguajes de programación desde el principio de la Informática. De hecho, **goto** representó la génesis de las técnicas de control de programa en los lenguajes ensambladores: “Si se cumple la condición A, salta aquí; en caso contrario, salta allí”. Si leemos el código ensamblador generado por casi todos los compiladores, podremos ver que el control de programa contiene muchos saltos (el compilador Java produce su propio “código ensamblador”, pero este código es ejecutado por la máquina virtual Java en lugar de ejecutarse directamente sobre un procesador hardware).

Una instrucción **goto** es un salto en el nivel de código fuente, y eso es lo que hizo que adquiriera una mala reputación. Si un programa va a saltar de un punto a otro, ¿no existe alguna forma de reorganizar el código para que el flujo de control no tenga que dar saltos? La instrucción **goto** llegó a ser verdaderamente puesta en cuestión con la publicación del famoso artículo “*Goto considered harmful*” de Edsger Dijkstra, y desde entonces la caza del **goto** se ha convertido en un deporte muy popular, forzando a los defensores de esa instrucción a ocultarse cuidadosamente.

Como suele suceder en casos como éste, la verdad está en el punto medio. El problema no está en el uso de **goto**, sino en su abuso, en determinadas situaciones especiales **goto** representa, de hecho, la mejor forma de estructurar el flujo.

Aunque **goto** es una palabra reservada en Java, no se utiliza en el lenguaje. Java no dispone de ninguna instrucción **goto**. Sin embargo, sí que dispone de algo que se asemeja a un salto, y que está integrado dentro de las palabras clave **break** y **continue**. No es un salto, sino más bien una forma salir de una instrucción de iteración. La razón por la que a menudo se asocia este mecanismo con las discusiones relativas a la instrucción **goto** es porque utiliza la misma técnica: una etiqueta.

Una etiqueta es un identificador seguido de un carácter de dos puntos, como se muestra aquí:

```
label1:
```

El único lugar en el que una etiqueta resulta útil en Java es justo antes de una instrucción de iteración. Y queremos decir exactamente *justo antes*: no resulta conveniente poner ninguna instrucción entre la etiqueta y la iteración. Y la única razón para colocar una etiqueta en una iteración es si vamos a anidar otra iteración o una instrucción **switch** (de la que hablaremos enseguida) dentro de ella. Esto se debe a que las palabras clave **break** y **continue** normalmente sólo interrumpirán el bucle actual, pero cuando se las usa como una etiqueta interrumpen todos los bucles hasta el lugar donde la etiqueta se haya definido:

```

label1:
iteración-externa {
    iteración-interna {

```

```

    //...
break; // (1)
//...
continue; // (2)
//...
continue label1; // (3)
//...
break label1; // (4)
}
}

```

En (1), la instrucción **break** hace que se salga de la iteración interna y que acabemos en la iteración externa. En (2), la instrucción **continue** hace que volvamos al principio de la iteración interna. Pero en (3), la instrucción **continue label1** hace que se salga de la iteración interna y de la iteración externa, hasta situarse en **label1**. Entonces, continúa de hecho con la iteración, pero comenzando en la iteración externa. En (4), la instrucción **break label1** también hace que nos salgamos de las dos iteraciones hasta situarnos en **label1**, pero sin volver a entrar en la iteración. De hecho, ambas iteraciones habrán finalizado.

He aquí un ejemplo de utilización de bucles **for**:

```

//: control/LabeledFor.java
// Bucle for con "break etiquetado" y "continue etiquetado".
import static net.mindview.util.Print.*;

public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer: // Aquí no se pueden incluir instrucciones
        for(; true ;) { // bucle infinito
            inner: // Aquí no se pueden incluir instrucciones
            for(; i < 10; i++) {
                print("i = " + i);
                if(i == 2) {
                    print("continue");
                    continue;
                }
                if(i == 3) {
                    print("break");
                    i++; // En caso contrario, i nunca
                         // se incrementa.
                    break;
                }
                if(i == 7) {
                    print("continue outer");
                    i++; // En caso contrario, i nunca
                         // se incrementa.
                    continue outer;
                }
                if(i == 8) {
                    print("break outer");
                    break outer;
                }
                for(int k = 0; k < 5; k++) {
                    if(k == 3) {
                        print("continue inner");
                        continue inner;
                    }
                }
            }
        }
    }
}

```

```

    // Aquí no se puede ejecutar break o continue para saltar a etiquetas
}
/* Output:
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*/

```

Observe que **break** hace que salgamos del bucle **for**, y que la expresión de incremento no se ejecuta hasta el final de la pasada a través del bucle **for**. Puesto que **break** se salta la expresión incremento, el incremento se realiza directamente en el caso de **i == 3**. La instrucción **continue outer** en el caso de **i == 7** también lleva al principio del bucle y también se salta el incremento, por lo que en este caso tenemos también que realizar el incremento directamente.

Si no fuera por la instrucción **break outer** no habría forma de salir del bucle externo desde dentro de un bucle interno, ya que **break** por sí misma sólo permite salir del bucle más interno (lo mismo cabría decir de **continue**).

Por supuesto, en aquellos casos en que salir de un bucle implique salir también del método, basta con ejecutar **return**.

He aquí una demostración de las instrucciones **break** y **continue** etiquetadas con bucles **while**:

```

//: control/LabeledWhile.java
// Bucles while con "break etiquetado" y "continue etiquetado".
import static net.mindview.util.Print.*;

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            print("Outer while loop");
            while(true) {
                i++;
                print("i = " + i);
                if(i == 1) {
                    print("continue");
                    continue;
                }
                if(i == 3) {
                    print("continue outer");
                    continue outer;
                }
                if(i == 5) {
                    print("break");
                    break;
                }
                if(i == 7) {

```

```

        print("break outer");
        break outer;
    }
}
} /* Output:
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
*///:-
```

Las mismas reglas siguen siendo ciertas para **while**:

1. Una instrucción **continue** normal hace que saltemos a la parte superior del bucle más interno y continuemos allí la ejecución.
2. Una instrucción **continue** etiquetada hace que saltemos hasta la etiqueta y que volvamos a ejecutar el bucle situado justo después de esa etiqueta.
3. Una instrucción **break** hace que finalice el bucle.
4. Una instrucción **break** etiquetada hace que finalicen todos los bucles hasta el que tiene la etiqueta, incluyendo este último.

Es importante recordar que la *única* razón para utilizar etiquetas en Java es si tenemos bucles anidados y queremos ejecutar una instrucción **break** o **continue** a través de más de un nivel.

En el artículo “*Goto considered harmful*” de Dijkstra, la objeción específica que él hacia era contra la utilización de etiquetas, no de la instrucción **goto**. Su observación era que el número de errores parecía incrementarse a medida que lo hacía el número de etiquetas dentro de un programa, y que las etiquetas en las instrucciones **goto** hacen que los programas sean más difíciles de analizar. Observe que las etiquetas de Java no presentan este problema, ya que están restringidas en cuanto a su ubicación y pueden utilizarse para transferir el control de forma arbitraria. También merece la pena observar que éste es uno de esos casos en los que se hace más útil una determinada característica del lenguaje reduciendo la potencia de la correspondiente instrucción.

switch

La palabra clave **switch** a veces se denomina *instrucción de selección*. La instrucción **switch** permite seleccionar entre distintos fragmentos de código basándose en el valor de una expresión entera. Su forma general es:

```

switch(selector-entero) {
    case valor-entero1 : instrucción; break;
    case valor-entero2 : instrucción; break;
    case valor-entero3 : instrucción; break;
    case valor-entero4 : instrucción; break;
    case valor-entero5 : instrucción; break;
    // ...
    default: instrucción;
}
```

Selector-entero es una expresión que genera un valor entero. La instrucción **switch** compara el resultado de *selector-entero* con cada *valor-entero*. Si encuentra una coincidencia, ejecuta la correspondiente *instrucción* (una sola instrucción o múltiples instrucciones; no hace falta usar llaves). Si no hay ninguna coincidencia, se ejecuta la *instrucción de default*.

Observará en la definición anterior que cada **case** finaliza con una instrucción **break**, lo que hace que la ejecución salte al final del cuerpo de la instrucción **switch**. Ésta es la forma convencional de construir una instrucción **switch**, pero la instrucción **break** es opcional. Si no se incluye, se ejecutará el código de las instrucciones **case** situadas a continuación hasta que se encuentre una instrucción **break**. Aunque normalmente este comportamiento no es el deseado, puede resultar útil en ocasiones para los programadores expertos. Observe que la última instrucción, situada después de la cláusula **default**, no tiene una instrucción **break** porque la ejecución continúa justo en el lugar donde **break** haría que continuara. Podemos incluir, sin que ello represente un problema, una instrucción **break** al final de la cláusula **default** si consideramos que resulta importante por razones de estilo.

La instrucción **switch** es una forma limpia de implementar selecciones multivía (es decir, selecciones donde hay que elegir entre diversas rutas de ejecución), pero requiere de un selector que se evalúe para dar un valor entero, como **int** o **char**. Si se desea emplear, por ejemplo, una cadena de caracteres o un número en coma flotante como selector, no funcionará en una instrucción **switch**. Para los tipos no enteros, es preciso emplear una serie de instrucciones **if**. Al final del siguiente capítulo, veremos que la nueva característica **enum** de Java SE5 ayuda a suavizar esta restricción, ya que los valores **enum** están diseñados para funcionar adecuadamente con la instrucción **switch**.

He aquí un ejemplo en el que se crean letras de manera aleatoria y se determina si son vocales o consonantes:

```
//: control/VowelsAndConsonants.java
// Ilustra la instrucción switch.
import java.util.*;
import static net.mindview.util.Print.*;

public class VowelsAndConsonants {
    public static void main(String[] args) {
        Random rand = new Random(47);
        for(int i = 0; i < 100; i++) {
            int c = rand.nextInt(26) + 'a';
            println((char)c + ", " + c + ": ");
            switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u': print("vowel");
                            break;
                case 'y':
                case 'w': print("Sometimes a vowel");
                            break;
                default: print("consonant");
            }
        }
    }
} /* Output:
y, 121: Sometimes a vowel
n, 110: consonant
z, 122: consonant
b, 98: consonant
r, 114: consonant
n, 110: consonant
y, 121: Sometimes a vowel
g, 103: consonant
c, 99: consonant
f, 102: consonant
o, 111: vowel
```

```
w, 119: Sometimes a vowel
z, 122: consonant
...
*///:-
```

Puesto que `Random.nextInt(26)` genera un valor comprendido entre 0 y 26, basta con sumar '`a`' para generar las letras minúsculas. Los caracteres encerrados entre comillas simples en las instrucciones `case` también generan valores enteros que se emplean para comparación.

Observe cómo las instrucciones `case` pueden "apilarse" unas encima de otras para proporcionar múltiples coincidencias para un determinado fragmento de código. Tenga también en cuenta que resulta esencial colocar la instrucción `break` al final de una cláusula `case` concreta; en caso contrario, el control no efectuaría el salto requerido y continuaría simplemente procesando el caso siguiente:

En la instrucción:

```
int c = rand.nextInt(26) + 'a';
```

`Random.nextInt()` genera un valor `int` aleatorio comprendido entre 0 y 25, al que se le suma el valor '`a`'. Esto quiere decir que '`a`' se convierte automáticamente a `int` para realizar la suma.

Para imprimir `c` como carácter, es necesario proyectarlo sobre el tipo `char`; en caso contrario, generaría una salida de tipo entero.

Ejercicio 8: (2) Cree una instrucción `switch` que imprima un mensaje para cada `case`, y coloque el `switch` dentro de un bucle `for` en el que se pruebe cada uno de los valores de `case`. Incluya una instrucción `break` después de cada `case` y compruebe los resultados; a continuación, elimine las instrucciones `break` y vea lo que sucede.

Ejercicio 9: (4) Una *secuencia de Fibonacci* es la secuencia de números 1, 1, 2, 3, 5, 8, 13, 21, 34, etc., donde cada número (a partir del tercero) es la suma de los dos anteriores. Cree un método que tome un entero como argumento y muestre esa cantidad de números de Fibonacci comenzando por el principio de la secuencia; por ejemplo, si ejecuta `java Fibonacci 5` (donde `Fibonacci` es el nombre de la clase) la salida sería: 1, 1, 2, 3, 5.

Ejercicio 10: (5) Un *número vampiro* tiene un número par de dígitos y se forma multiplicando una pareja de números que contengan la mitad del número de dígitos del resultado. Los dígitos se toman del número original en cualquier orden. No se permiten utilizar parejas de ceros finales. Entre los ejemplos tendríamos:

$1260 = 21 * 60$

$1827 = 21 * 87$

$2187 = 27 * 81$

Escriba un programa que determine todos los números vampiro de 4 dígitos (problema sugerido por Dan Forhan).

Resumen

Este capítulo concluye el estudio de las características fundamentales que podemos encontrar en la mayoría de los lenguajes de programación: cálculo, precedencia de operadores, proyección de tipos y mecanismos de selección e iteración. Ahora estamos listos para dar los siguientes pasos, que nos acercarán al mundo de la programación orientada a objetos. El siguiente capítulo cubrirá las importantes cuestiones de la inicialización y limpieza de objetos, a lo que seguirá, en el siguiente capítulo, el concepto esencial de ocultación de la implementación.

Pueden encontrarse las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Inicialización y limpieza

5

A medida que se abre paso la revolución informática, la programación “no segura” se ha convertido en uno de los mayores culpables del alto coste que tiene el desarrollo de programas.

Dos de las cuestiones relativas a la seguridad son la *inicialización* y la *limpieza*. Muchos errores en C se deben a que el programador se olvida de inicializar una variable. Esto resulta especialmente habitual con las bibliotecas, cuando los usuarios no saben cómo inicializar un componente en la biblioteca, e incluso ni siquiera son conscientes de que deban hacerlo. La limpieza también constituye un problema especial, porque resulta fácil olvidarse de un elemento una vez que se ha terminado de utilizar, ya que en ese momento deja de preocuparnos. Al no borrarlo, los recursos utilizados por ese elemento quedan retenidos y resulta fácil que los recursos se agoten (especialmente la memoria).

C++ introdujo el concepto de *constructor*, un método especial que se invoca automáticamente cada vez que se crea un objeto. Java también adoptó el concepto de constructor y además dispone de un depurador de memoria que se encarga de liberar automáticamente los recursos de memoria cuando ya no se los está utilizando. En este capítulo, se examinan las cuestiones relativas a la inicialización y la limpieza, así como el soporte que Java proporciona para ambas tareas.

Inicialización garantizada con el constructor

Podemos imaginar fácilmente que sería sencillo crear un método denominado **initialize()** para todas las clases que escribiríamos. El nombre es una indicación de que es necesario invocar el método antes de utilizar el objeto. Lamentablemente, esto indica que el usuario debe recordar que hay que invocar ese método. En Java, el diseñador de una clase puede garantizar la inicialización de todos los objetos proporcionando un constructor. Si una clase tiene un constructor, Java invoca automáticamente ese constructor cuando se crea un objeto, antes incluso de que los usuarios puedan llegar a utilizarlo. De este modo, la inicialización queda garantizada.

La siguiente cuestión es cómo debemos nombrar a este método, y existen dos problemas a este respecto. El primero es que cualquier nombre que usemos podría colisionar con otro nombre que quisiéramos emplear como miembro de la clase. El segundo problema es que debido a que el compilador es responsable de invocar el constructor, debe siempre conocer qué método invocar. La solución en C++ parece la más fácil y lógica, de modo que también se usa en Java: el nombre del constructor coincide con el nombre de la clase. De este modo, resulta fácil invocar ese método automáticamente durante la inicialización.

He aquí una clase simple con un constructor:

```
//: initialization/SimpleConstructor.java
// Ilustración de un constructor simple.

class Rock {
    Rock() { // Éste es el constructor
        System.out.print("Rock ");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
```

```

        new Rock();
    }
} /* Output:
Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
*///:-
```

Ahora, cuando se crea un objeto:

```
new Rock();
```

se asigna el correspondiente espacio de almacenamiento y se invoca el constructor. De este modo, se garantiza que el objeto está apropiadamente inicializado antes de poder utilizarlo.

Observe que el estilo de codificación consistente en poner la primera letra de todos los métodos en minúscula no se aplica a los constructores, ya que el nombre del constructor debe coincidir *exactamente* con el nombre de la clase.

Un constructor que no tome ningún argumento se denomina *constructor predeterminado*. Normalmente, la documentación de Java utiliza el término constructor *sin argumentos*, pero el término “constructor predeterminado” se ha estado utilizando durante muchos años antes de que Java apareciera, por lo que prefiero utilizar este último término. De todos modos, como cualquier otro método, el constructor puede también tener argumentos que nos permiten especificar *cómo* hay que crear el objeto. Podemos modificar fácilmente el ejemplo anterior para que el constructor tome un argumento:

```

//: initialization/SimpleConstructor2.java
// Los constructores pueden tener argumentos.

class Rock2 {
    Rock2(int i) {
        System.out.print("Rock " + i + " ");
    }
}

public class SimpleConstructor2 {
    public static void main(String[] args) {
        for(int i = 0; i < 8; i++)
            new Rock2(i);
    }
} /* Output:
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*///:-
```

Los argumentos del constructor proporcionan una forma de pasar parámetros para la inicialización de un objeto. Por ejemplo, si la clase **Tree** (árbol) tiene un constructor que toma como argumento un único número entero que indica la altura del árbol, podremos crear un objeto **Tree** como sigue:

```
Tree t = new Tree(12); // árbol de 12 metros
```

Si **Tree(int)** es el único constructor del que disponemos, el compilador no nos permitirá crear un objeto **Tree** de ninguna otra forma.

Los constructores eliminan una amplia clase de problemas y hacen que el código sea más fácil de leer. Por ejemplo, en el fragmento de código anterior, no vemos ninguna llamada explícita a ningún método **initialize()** que esté conceptualmente separado del acto de creación del objeto. En Java, la creación y la inicialización son conceptos unificados: no es posible tener la una sin la otra.

El constructor es un tipo de método poco usual, porque no tiene valor de retorno. Existe una clara diferencia entre esta circunstancia y los métodos que devuelven un valor de retorno **void**, en el sentido de que estos últimos métodos no devuelven nada, pero seguimos teniendo la opción de hacer que devuelvan algo. Los constructores no devuelven nada nunca, y no tenemos la opción de que se comporten de otro modo (la expresión **new** devuelve una referencia al objeto recién creado, pero el constructor mismo no tiene un valor de retorno). Si hubiera valor de retorno y pudieramos seleccionar cuál es, el compilador necesitaría saber qué hacer con ese valor de retorno.

Ejercicio 1: (1) Cree una clase que contenga una referencia de tipo **String** no inicializada. Demuestre que esta referencia la inicializa Java con el valor **null**.

Ejercicio 2: (2) Cree una clase con un campo **String** que se inicialice en el punto donde se defina, y otro campo que sea inicializado por el constructor. ¿Cuál es la diferencia entre las dos técnicas?

Sobrecarga de métodos

Una de las características más importantes en cualquier lenguaje de programación es el uso de nombres. Cuando se crea un objeto, se proporciona un nombre a un área de almacenamiento. Un método, por su parte, es un nombre que sirve para designar una acción. Utilizamos nombres para referirnos a todos los objetos y métodos. Una serie de nombres bien elegida creará un sistema que resultará más fácil de entender y modificar por otras personas. En cierto modo, este problema se parece al acto de escribir literatura: el objetivo es comunicarse con los lectores.

Todos los problemas surgen a la hora de aplicar el concepto de matiz del lenguaje humano a los lenguajes de programación. A menudo, una misma palabra tiene diferentes significados: es lo que se denomina palabras *polisémicas*, aunque en el campo de la programación diríamos que están *sobrecargadas*. Lo que hacemos normalmente es decir "Lava la camisa", "Lava el coche" y "Lava al perro": sería absurdo vernos forzados a decir "camisaLava la camisa", "cocheLava el coche" y "perroLava el perro" simplemente para que el oyente no se vea forzado a distinguir cuál es la acción que tiene que realizar. La mayoría de los lenguajes humanos son redundantes, de modo que podemos seguir determinando el significado aun cuando nos perdamos algunas de las palabras. No necesitamos identificadores univocos: podemos deducir el significado a partir del contexto.

La mayoría de los lenguajes de programación (y C en particular) exigen que dispongamos de un identificador unívoco para cada método (a menudo denominados *funciones* en dichos lenguajes). Así que *no se puede* tener una función denominada **print()** para imprimir enteros y otra denominada igualmente **print()** para imprimir números en coma flotante, cada una de las funciones necesitará un nombre distintivo.

En Java (y en C++), hay otro factor que obliga a sobrecargar los nombres de los métodos: el constructor. Puesto que el nombre del constructor está predeterminado por el nombre de la clase, sólo puede haber un nombre de constructor. Pero entonces, ¿qué sucede si queremos crear un objeto utilizando varias formas distintas? Por ejemplo, suponga que construimos una clase cuyos objetos pueden inicializarse de la forma normal o leyendo la información de un archivo. Harán falta dos constructores, el constructor predeterminado y otro que tome un objeto **String** como argumento, a través del cual suministremos el nombre del archivo que hay que utilizar para inicializar el objeto. Ambos métodos serán constructores, así que tendrán el mismo nombre: el nombre de la clase. Por tanto, la *sobrecarga* de métodos resulta esencial para poder utilizar el mismo nombre de método con diferentes tipos de argumentos. Y, aunque la sobrecarga de métodos es obligatoria para los constructores, también resulta útil de manera general y puede ser empleada con cualquier otro método.

He aquí un ejemplo que muestra tanto constructores sobrecargados como métodos normales sobrecargados:

```
//: initialization/Overloading.java
// Ilustración del mecanismo de sobrecarga
// tanto de constructores como de métodos normales.
import static net.mindview.util.Print.*;

class Tree {
    int height;
    Tree() {
        print("Planting a seedling");
        height = 0;
    }
    Tree(int initialHeight) {
        height = initialHeight;
        print("Creating new Tree that is " +
            height + " feet tall");
    }
    void info() {
        print("Tree is " + height + " feet tall");
    }
    void info(String s) {
        print(s + ": Tree is " + height + " feet tall");
    }
}
```

```

    }

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Constructor sobrecargado:
        new Tree();
    }
} /* Output:
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling
*///:-
```

Con estas definiciones, podemos crear un objeto **Tree** tanto a partir de una semilla, sin utilizar ningún argumento, como en forma de planta criada en vivero, en cuyo caso tendremos que indicar la altura que tiene. Para soportar este comportamiento, hay un constructor predeterminado y otro que toma como argumento la altura del árbol.

También podemos invocar el método **info()** de varias formas distintas. Por ejemplo, si queremos imprimir un mensaje adicional, podemos emplear **info(String)**, mientras que utilizariamos **info()** cuando no tengamos nada más que decir. Sería bastante extraño proporcionar dos nombres separados a cosas que se corresponden, obviamente, con un mismo concepto. Afortunadamente, la sobrecarga de métodos nos permite utilizar el mismo método para ambos.

Cómo se distingue entre métodos sobrecargados

Si los métodos tienen el mismo nombre, ¿cómo puede saber Java a qué método nos estamos refiriendo? Existe una regla muy simple: cada método sobrecargado debe tener una lista distintiva de tipos de argumentos.

Si pensamos en esta regla durante un momento, vemos que tiene bastantes sentido. ¿De qué otro modo podría un programador indicar la diferencia entre dos métodos que tienen el mismo nombre, si no es utilizando las diferencias entre los tipos de sus argumentos?

Incluso las diferencias en la ordenación de los argumentos son suficientes para distinguir dos métodos entre sí, aunque normalmente no conviene emplear esta técnica, dado que produce código difícil de mantener:

```

//: initialization/OverloadingOrder.java
// Sobre carga basada en el orden de los argumentos.
import static net.mindview.util.Print.*;

public class OverloadingOrder {
    static void f(String s, int i) {
        print("String: " + s + ", int: " + i);
```

```

}
static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
}
public static void main(String[] args) {
    f("String first", 11);
    f(99, "Int first");
}
/* Output:
String: String first, int: 11
int: 99, String: Int first
*///:-

```

Los dos métodos `f()` tienen argumentos idénticos, pero el orden es distinto y eso es lo que los hace diferentes.

Sobrecarga con primitivas

Una primitiva puede ser automáticamente convertida desde un tipo de menor tamaño a otro de mayor tamaño, y esto puede inducir a confusión cuando combinamos este mecanismo con el de sobrecarga. El siguiente ejemplo ilustra lo que sucede cuando se pasa una primitiva a un método sobrecargado:

```

//: initialization/PrimitiveOverloading.java
// Promoción de primitivas y sobrecarga.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
    void f1(char x) { printnb("f1(char) "); }
    void f1(byte x) { printnb("f1(byte) "); }
    void f1(short x) { printnb("f1(short) "); }
    void f1(int x) { printnb("f1(int) "); }
    void f1(long x) { printnb("f1(long) "); }
    void f1(float x) { printnb("f1(float) "); }
    void f1(double x) { printnb("f1(double) "); }

    void f2(byte x) { printnb("f2(byte) "); }
    void f2(short x) { printnb("f2(short) "); }
    void f2(int x) { printnb("f2(int) "); }
    void f2(long x) { printnb("f2(long) "); }
    void f2(float x) { printnb("f2(float) "); }
    void f2(double x) { printnb("f2(double) "); }

    void f3(short x) { printnb("f3(short) "); }
    void f3(int x) { printnb("f3(int) "); }
    void f3(long x) { printnb("f3(long) "); }
    void f3(float x) { printnb("f3(float) "); }
    void f3(double x) { printnb("f3(double) "); }

    void f4(int x) { printnb("f4(int) "); }
    void f4(long x) { printnb("f4(long) "); }
    void f4(float x) { printnb("f4(float) "); }
    void f4(double x) { printnb("f4(double) "); }

    void f5(long x) { printnb("f5(long) "); }
    void f5(float x) { printnb("f5(float) "); }
    void f5(double x) { printnb("f5(double) "); }

    void f6(float x) { printnb("f6(float) "); }
    void f6(double x) { printnb("f6(double) "); }

    void f7(double x) { printnb("f7(double) "); }
}

```

```

void testConstVal() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
}
void testChar() {
    char x = 'x';
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testByte() {
    byte x = 0;
    printnb("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testShort() {
    short x = 0;
    printnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testInt() {
    int x = 0;
    printnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testLong() {
    long x = 0;
    printnb("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testFloat() {
    float x = 0;
    printnb("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testDouble() {
    double x = 0;
    printnb("double: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
}
/* Output:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double) f7(double)
*///:-
```

Puede ver que el valor constante 5 se trata como **int**, por lo que si hay disponible un método sobrecargado que toma un objeto **int**, se utilizará dicho método. En todos los demás casos, si lo que tenemos es un tipo de datos más pequeño que el argumento del método, dicho tipo de datos será promocionado. **char** produce un efecto ligeramente diferente; ya que, si no se encuentra una correspondencia exacta con **char**, se le promociona a **int**.

¿Qué sucede si nuestro argumento es *mayor* que el argumento esperado por el método sobrecargado? Una modificación del programa anterior nos da la respuesta:

```
//: initialization/Demotion.java
// Reducción de primitivas y sobrecarga.
import static net.mindview.util.Print.*;

public class Demotion {
    void f1(char x) { print("f1(char)"); }
    void f1(byte x) { print("f1(byte)"); }
    void f1(short x) { print("f1(short)"); }
    void f1(int x) { print("f1(int)"); }
    void f1(long x) { print("f1(long)"); }
    void f1(float x) { print("f1(float)"); }
    void f1(double x) { print("f1(double)"); }

    void f2(char x) { print("f2(char)"); }
    void f2(byte x) { print("f2(byte)"); }
    void f2(short x) { print("f2(short)"); }
    void f2(int x) { print("f2(int)"); }
    void f2(long x) { print("f2(long)"); }
    void f2(float x) { print("f2(float)"); }

    void f3(char x) { print("f3(char)"); }
    void f3(byte x) { print("f3(byte)"); }
    void f3(short x) { print("f3(short)"); }
    void f3(int x) { print("f3(int)"); }
    void f3(long x) { print("f3(long)"); }

    void f4(char x) { print("f4(char)"); }
    void f4(byte x) { print("f4(byte)"); }
    void f4(short x) { print("f4(short)"); }
    void f4(int x) { print("f4(int)"); }

    void f5(char x) { print("f5(char)"); }
    void f5(byte x) { print("f5(byte)"); }
    void f5(short x) { print("f5(short)"); }

    void f6(char x) { print("f6(char)"); }
    void f6(byte x) { print("f6(byte)"); }

    void f7(char x) { print("f7(char)"); }

    void testDouble() {
        double x = 0;
        print("double argument:");
        f1(x);f2((float)x);f3((long)x);f4((int)x);
        f5((short)x);f6((byte)x);f7((char)x);
    }
    public static void main(String[] args) {
        Demotion p = new Demotion();
        p.testDouble();
    }
} /* Output:
double argument:
```

```
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*///:-
```

Aquí, los métodos admiten valores primitivos más pequeños. Si el argumento es de mayor anchura, entonces será necesario efectuar una conversión de estrechamiento mediante una proyección. Si no se hace esto, el compilador generará un mensaje de error.

Sobrecarga de los valores de retorno

Resulta bastante habitual hacerse la pregunta: “¿Por qué sólo tener en cuenta los nombres de clase y las listas de argumentos de los métodos? ¿Por qué no distinguir entre los métodos basándonos en sus valores de retorno?” Por ejemplo, los siguientes dos métodos, que tienen el mismo nombre y los mismos argumentos, pueden distinguirse fácilmente:

```
void f() {}
int f() { return 1; }
```

Esto podría funcionar siempre y cuando el compilador pudiera determinar inequivocamente el significado a partir del contexto, como por ejemplo en `int x = f()`. Sin embargo, el lenguaje nos permite invocar un método e ignorar el valor de retorno, que es una técnica que a menudo se denomina *invocar un método por su efecto colateral*, ya que no nos preocupa el valor de retorno, sino que queremos que tengan lugar los restantes efectos de la invocación al método. Luego entonces, si invocamos el método de esta forma:

```
E();
```

¿Cómo podría Java determinar qué método `f()` habría que invocar? ¿Y cómo podría determinarlo alguien que estuviera leyendo el código? Debido a este tipo de problemas, no podemos utilizar los tipos de los valores de retorno para distinguir los métodos sobrecargados.

Constructores predeterminados

Como se ha mencionado anteriormente, un constructor predeterminado (también denominado “constructor sin argumentos”) es aquel que no tiene argumentos y que se utiliza para crear un “objeto predeterminado”. Si creamos una clase que no tenga constructores, el compilador creará automáticamente un constructor predeterminado. Por ejemplo:

```
//: initialization/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird b = new Bird(); // Default!
    }
} *///:-
```

La expresión

```
new Bird()
```

crea un nuevo objeto y llama al constructor predeterminado, incluso aunque no se haya definido uno de manera explícita. Sin ese constructor predeterminado no dispondríamos de ningún método al que invocar para construir el objeto. Sin embargo, si definimos algún constructor (con o sin argumentos), el compilador *no sintetizará* ningún constructor por nosotros:

```
//: initialization/NoSynthesis.java

class Bird2 {
```

```

    Bird2(int i) {}
    Bird2(double d) {}
}

public class NoSynthesis {
    public static void main(String[] args) {
        //! Bird2 b = new Bird2(); // No hay predeterminado
        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
} //:=

```

Si escribimos:

```
new Bird2()
```

el compilador nos indicará que no puede localizar ningún constructor que se corresponda con la instrucción que hemos escrito. Cuando no definimos explícitamente ningún constructor, es como si el compilador dijera: "Es necesario utilizar *algún* constructor, así que déjame definir uno por ti". Pero, si escribimos al menos constructor, el compilador dice: "Has escrito un constructor, así que tu sabrás lo que estás haciendo; si no has incluido uno predeterminado es porque no quieres hacerlo".

Ejercicio 3: (1) Cree una clase con un constructor predeterminado (uno que no tome ningún argumento) que imprima un mensaje. Cree un objeto de esa clase.

Ejercicio 4: (1) Añada un constructor sobrecargado al ejercicio anterior que admita un argumento de tipo **String** e imprima la correspondiente cadena de caracteres junto con el mensaje.

Ejercicio 5: (2) Cree una clase denominada **Dog** con un método sobrecargado **bark()** (método "ladrar"). Este método debe estar sobrecargado basándose en diversos tipos de datos primitivos y debe imprimir diferentes tipos de ladridos, gruñidos, etc., dependiendo de la versión sobrecargada que se invoque. Escriba un método **main()** que invoque todas las distintas versiones.

Ejercicio 6: (1) Modifique el ejercicio anterior de modo que dos de los métodos sobrecargados tengan dos argumentos (de dos tipos distintos), pero en orden inverso uno respecto del otro. Verifique que estas definiciones funcionan.

Ejercicio 7: (1) Cree una clase sin ningún constructor y luego cree un objeto de esa clase en **main()** para verificar que se sintetiza automáticamente el constructor predeterminado.

La palabra clave **this**

Si tenemos dos objetos del mismo tipo llamados **a** y **b**, podemos preguntarnos cómo es posible invocar un método **peel()** para ambos objetos (*nota*: la palabra inglesa *peel* significa "pelar una fruta", que en este ejemplo de programación es una banana):

```

// initialization/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
    public static void main(String[] args) {
        Banana a = new Banana(),
               b = new Banana();
        a.peel(1);
        b.peel(2);
    }
} //:=

```

Si sólo hay un único método denominado **peel()**, ¿cómo puede ese método saber si está siendo llamado para el objeto **a** o para el objeto **b**?

Para poder escribir el código en una sintaxis cómoda orientada a objetos, en la que podamos “enviar un mensaje a un objeto”, el compilador se encarga de realizar un cierto trabajo entre bastidores por nosotros. Existe un primer argumento secreto pasado al método `peel()`, y ese argumento es la referencia al objeto que se está manipulando. De este modo, las dos llamadas a métodos se convierten en algo parecido a:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

Esto se realiza internamente y nosotros no podemos escribir estas expresiones y hacer que el compilador las acepte, pero este ejemplo nos basta para hacernos una idea de lo que sucede en la práctica.

Suponga que nos encontramos dentro de un método y queremos obtener la referencia al objeto actual. Puesto que el compilador pasa esa referencia *secretamente*, no existe ningún identificador para ella. Sin embargo, y para poder acceder a esa referencia, el lenguaje incluye una palabra clave específica: `this`. La palabra clave `this`, que sólo se puede emplear en métodos que no sean de tipo `static` devuelve la referencia al objeto para el cual ha sido invocado el método. Podemos tratar esta referencia del mismo modo que cualquier otra referencia a un objeto. Recuerde que, si está invocando un método de la clase desde dentro de otro método de esa misma clase, no es necesario utilizar `this`, sino que simplemente basta con invocar el método. La referencia `this` actual será utilizada automáticamente para el otro método. De este modo, podemos escribir:

```
//: initialization/Apricot.java
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
} /*:-*/
```

Dentro de `pit()`, *podríamos* decir `this.pick()` pero no hay ninguna necesidad de hacerlo.¹ El compilador se encarga de hacerlo automáticamente por nosotros. La palabra clave `this` sólo se usa en aquellos casos especiales en los que es necesario utilizar explícitamente la referencia al objeto actual. Por ejemplo, a menudo se usa en instrucciones `return` cuando se quiere devolver la referencia al objeto actual:

```
//: initialization/Leaf.java
// Uso simple de la palabra clave "this".

public class Leaf {
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
} /* Output:
i = 3
*/*:-*/
```

Puesto que `increment()` devuelve la referencia al objeto actual a través de la palabra clave `this`, pueden realizarse fácilmente múltiples operaciones con un mismo objeto.

La palabra clave `this` también resulta útil para pasar el objeto actual a otro método:

¹ Algunas personas escriben obsesivamente `this` delante de cada llamada a método o referencia a un campo argumentando que eso hace que el código sea “más claro y más explícito”. Mi consejo es que no lo haga. Existe una razón por la que utilizamos los lenguajes de alto nivel, y esa razón es que estos lenguajes se encargan de hacer buena parte del trabajo por nosotros. Si incluimos la palabra clave `this` cuando no es necesario, las personas que lean el código se sentirán confundidas, ya que los demás programas que hayan leído en cualquier parte *no utilizan* las palabra clave `this` de manera continua. Los programadores esperan que `this` sólo se use allí donde sea necesario. El seguir un estilo de codificación coherente y simple permite ahorrar tiempo y dinero.

```

//: initialization/PassingThis.java

class Person {
    public void eat(Apple apple) {
        Apple peeled = apple.getPealed();
        System.out.println("Yummy");
    }
}

class Peeler {
    static Apple peel(Apple apple) {
        // ... pelear
        return apple; // Peiada
    }
}

class Apple {
    Apple getPealed() { return Peeler.peel(this); }
}

public class PassingThis {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
} /* Output:
Yummy
*/

```

El objeto **Apple** (manzana) necesita invocar **Peeler.peel()**, que es un método de utilidad externo que lleva a cabo una operación que, por alguna razón, necesita ser externa a **Apple** (quizá ese método externo pueda aplicarse a muchas clases distintas y no queremos repetir el código). Para que el objeto pueda pasarse a sí mismo al método externo, es necesario emplear **this**.

Ejercicio 8: (1) Cree una clase con dos métodos. Dentro del primer método invoque al segundo método dos veces: la primera vez sin utilizar **this** y la segunda utilizando dicha palabra clave. Realice este ejemplo simplemente para ver cómo funciona el mecanismo, no debe utilizar esta forma de invocar a los métodos en la práctica.

Invocación de constructores desde otros constructores

Cuando se escriben varios constructores para una clase, existen ocasiones en las que conviene invocar a un constructor desde dentro de otro para no tener que duplicar el código. Podemos efectuar este tipo de llamadas utilizando la palabra clave **this**.

Normalmente, cuando escribimos **this**, es en el sentido de "este objeto" o el "objeto actual", y esa palabra clave genera, por sí misma, la referencia al objeto actual. Dentro de un constructor, la palabra clave **this** toma un significado distinto cuando se la proporciona una lista de argumentos: realiza una llamada explícita al constructor que se corresponda con esa lista de argumentos. De este modo, disponemos de una forma sencilla de invocar a otros constructores:

```

//: initialization/Flower.java
// Llamada a constructores con "this"
import static net.mindview.util.Print.*;

public class Flower {
    int petalCount = 0;
    String s = "initial value";
    Flower(int petals) {
        petalCount = petals;
        print("Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
}

```

```

Flower(String ss) {
    print("Constructor w/ String arg only, s = " + ss);
    s = ss;
}
Flower(String s, int petals) {
    this(petals);
//!    this(s); // ¡No podemos realizar dos invocaciones!
    this.s = s; // Otro uso de "this"
    print("String & int args");
}
Flower() {
    this("hi", 47);
    print("default constructor (no args)");
}
void printPetalCount() {
//! this(11); // ¡No dentro de un no-constructo!
    print("petalCount = " + petalCount + " s = "+ s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
}
/* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
*///:-
```

El constructor **Flower(String s, int petals)** muestra que, aunque podemos invocar un constructor utilizando **this**, no podemos invocar dos. Además, la llamada al constructor debe ser lo primero que hagamos, porque de lo contrario obtendremos un mensaje de error de compilación.

Este ejemplo también muestra otro modo de utilización de **this**. Puesto que el nombre del argumento **s** y el nombre del miembro de datos **s** son iguales, existe una ambigüedad. Podemos resolverla utilizando **this.s**, para dejar claro que estamos refiriéndonos al miembro de datos. Esta forma de utilización resulta muy habitual en el código Java y se emplea en numerosos lugares del libro.

En **printPetalCount()** podemos ver que el compilador no nos permite invocar un constructor desde dentro de cualquier método que no sea un constructor.

Ejercicio 9: (1) Cree una clase con dos constructores (sobrecargados). Utilizando **this**, invoque el segundo constructor desde dentro del primero.

El significado de **static**

Teniendo en mente el significado de la palabra clave **this**, podemos comprender mejor qué es lo que implica definir un método como **static**. Significa que no existirá ningún objeto **this** para ese método concreto. No se pueden invocar métodos no **static** desde dentro de los métodos **static**² (aunque la inversa sí es posible), y se puede invocar un método **static** para la propia clase, sin especificar ningún objeto. De hecho, esa es la principal aplicación de los métodos **static**: es como si estuviéramos creando el equivalente de un método global. Sin embargo, los métodos globales no están permitidos en Java, y el incluir el método **static** dentro de una clase permite a los objetos de esa clase acceder a métodos **static** y a campos de tipo **static**.

Algunas personas argumentan que los métodos estáticos no son orientados a objetos, ya que tienen la semántica de un método global. Un método estático no envía un mensaje a un objeto, ya que no existe referencia **this**. Probablemente se trate de

² El único caso en que esto puede hacerse es cuando se pasa al método **static** una referencia a un objeto (el método **static** también podría crear su propio objeto). Entonces, a través de la referencia (que ahora será, en la práctica, **this**), se pueden invocar métodos no **static** y acceder a campos no **static**. Pero, normalmente si queremos hacer algo como esto, lo mejor es que escribamos un método no **static** normal y corriente.

un argumento correcto, y si se encuentra alguna vez utilizando una gran cantidad de métodos estáticos probablemente convenga que vuelva a meditar sobre la estrategia que está empleando. Sin embargo, los métodos y valores estáticos resultan bastante prácticos, y hay ocasiones en las que de verdad son necesarios, por lo que la cuestión de si se trata de verdadera programación orientada a objetos es mejor dejársela a los teóricos.

Limpieza: finalización y depuración de memoria

Los programadores son conscientes de la importancia de la inicialización, pero a menudo se olvidan de que también la limpieza es importante. Después de todo, ¿quién necesita limpiar un valor `int`? Pero, con las bibliotecas, limitarse a olvidarse de un objeto después de haber acabado de utilizarlo no siempre resulta seguro. Por supuesto, Java dispone del depurador de memoria para reclamar la memoria ocupada por aquellos objetos que ya no estén siendo utilizados. Pero pensemos en lo que sucede en algunos casos poco usuales: suponga que el objeto que ha definido asigna memoria "especial" sin utilizar `new`. El depurador de memoria sólo sabe cómo liberar la memoria asignada con `new`, por lo que no sabrá cómo liberar la memoria "especial" del objeto. Para estos casos, Java proporciona un método denominado `finalize()` que se puede definir para la clase. He aquí cómo se supone que funciona ese método: cuando el depurador de memoria esté listo para liberar el almacenamiento utilizado por el objeto, invocará primero `finalize()` y sólo reclamará la memoria del objeto en la siguiente pasada del depurador de memoria. Por tanto, si decidimos utilizar `finalize()`, tendremos la posibilidad de realizar tareas de limpieza importantes en el momento en que se produzca la depuración de memoria.

Esta es una posible fuente de error de programación, porque algunos programadores, especialmente los que provienen del campo del C++, pueden confundirse inicialmente y considerar que `finalize()` es el destructor de C++, que es una función que se invoca siempre cuando se destruye un objeto. Es importante entender la distinción que existe entre C++ y Java a este respecto, porque en C++, los objetos siempre se destruyen (en un programa libre de errores), mientras que en Java, el depurador de memoria no siempre procesa los objetos. Dicho de otro modo:

1. Puede que el depurador de memoria no procese los objetos.
2. La depuración de memoria no es equivalente a la destrucción del objeto.

Si se recuerdan estos dos principios, se podrán evitar muchos problemas. Lo que quieren decir es que, si existe alguna actividad que deba de ser realizada antes de que un objeto deje de ser necesario, deberemos realizar dicha actividad nosotros mismos. Java no dispone de métodos destructores ni de ningún otro concepto similar, por lo que es necesario crear un método normal para llevar a cabo esta tarea de limpieza. Por ejemplo, suponga que, en el proceso de creación de un objeto, ese objeto se dibuja a sí mismo en la pantalla. Si no borramos explícitamente su imagen de la pantalla, puede que esa imagen nunca llegue a borrarse. Si incluimos una cierta funcionalidad de borrado dentro de `finalize()`, entonces si un objeto se ve sometido al proceso de depuración de memoria y se invoca `finalize()` (y recordemos que no existe ninguna garantía de que esto suceda), entonces se eliminará primero la imagen de la pantalla; pero si no incluimos explícitamente esa funcionalidad de borrado, esa imagen permanecerá.

Podemos encontrarnos con la situación de que nunca llegue a liberarse el espacio de almacenamiento de un objeto, debido a que el programa nunca se acerca a un punto en el que exista un riesgo de quedarse sin espacio de almacenamiento. Si el programa se completa y el depurador de memoria no llega a entrar en acción para eliminar el espacio de almacenamiento asignado a los objetos, dicho espacio será devuelto al sistema operativo en masa en el momento de salir del programa. Esta característica resulta bastante conveniente, porque el proceso de depuración de memoria implica un cierto gasto adicional de recursos de procesamiento, y si no se lleva a cabo, el gasto no se produce.

¿Para qué se utiliza `finalize()`?

Pero entonces, si no debe utilizarse `finalize()` como método de limpieza de propósito general, ¿para qué sirve este método?

Un tercer punto que hay que recordar es:

3. La depuración de memoria sólo se preocupa de la memoria.

Es decir, la única razón para la existencia del depurador de memoria es recuperar aquella memoria que el programa ya no está utilizando. Por tanto, cualquier actividad asociada con la depuración de memoria, y en especial el método `finalize()`, debe también encargarse únicamente de la memoria y de su desasignación.

Significa esto que, si el objeto contiene otros objetos, **finalize()** debe liberar explícitamente esos otros objetos? En realidad no: el depurador de memoria se encarga de liberar toda la memoria de objetos, independientemente de cómo estos hayan sido creados. En resumen, **finalize()** sólo es necesario en aquellos casos especiales en los que el objeto pueda asignar espacio de almacenamiento utilizando alguna técnica distinta de la propia creación de objetos. Algun lector especialmente atento podría argumentar: pero, si todo Java es un objeto, ¿cómo puede llegar a producirse esta situación?

Parece que **finalize()** se ha incluido en el lenguaje debido a la posibilidad de que el programador realice alguna actividad de estilo C, asignando memoria mediante algún mecanismo distinto del normalmente empleado en Java. Esto puede suceder, principalmente, a través de los *métodos nativos*, que son una forma de invocar desde Java código escrito en un lenguaje distinto de Java (los métodos nativos se estudian en el Apéndice B de la segunda edición electrónica de este libro, disponible en www.MindView.net). C y C++ son los únicos lenguajes actualmente soportados por los métodos nativos, pero como desde ellos se pueden invocar subprogramas en otros lenguajes, en la práctica podremos invocar cualquier cosa que queramos. Dentro del código no Java, podría invocarse la familia de funciones **malloc()** de C para asignar espacio de almacenamiento, y a menos que invoquemos **free()**, dicho espacio de almacenamiento no será liberado, provocando una fuga de memoria. Por supuesto, **free()** es una función de C y C++, por lo que sería necesario invocarla mediante un método nativo dentro de **finalize()**.

Después de estas explicaciones, el lector probablemente estará pensando que no va a tener que utilizar **finalize()** de forma demasiado frecuente.³ En efecto, es así: dicho método no es el lugar apropiado para realizar las tareas normales de limpieza. Pero, entonces, ¿dónde lleva a cabo esas tareas normales de limpieza?

Es necesario efectuar las tareas de limpieza

Para limpiar un objeto, el usuario de ese objeto debe invocar un método de limpieza en el lugar donde deseé que ésta se realice. Esto parece bastante sencillo, pero choca un poco con el concepto C++ de destructor. En C++, todos los objetos se destruyen; o, mejor dicho, todos los objetos *deberían* destruirse. Si el objeto C++ se crea como local (es decir, en la pila, lo cual no resulta posible en Java), la destrucción se produce en la llave de cierre del ámbito en que el objeto haya sido creado. Si el objeto se creó utilizando **new** (como en Java), el destructor se invoca cuando el programador llama al operador C++ **delete** (que no existe en Java). Si el programador de C++ olvida invocar **delete**, nunca se llamará al destructor y se producirá en la práctica una fuga de memoria (además de que las otras partes del objeto nunca llegarán a limpiarse). Este tipo de error puede ser muy difícil de localizar y es una de las principales razones para pasar de C++ a Java.

Por contraste, Java no permite crear objetos locales, sino que siempre es necesario utilizar **new**. Pero en Java, no existe ningún operador “**delete**” para liberar el objeto, porque el depurador de memoria se encarga de liberar el espacio de almacenamiento por nosotros. Por tanto, desde un punto de vista simplista, podríamos decir que debido a la depuración de memoria Java no dispone de destructores. Sin embargo, a medida que avancemos en el libro veremos que la presencia de un depurador de memoria no elimina ni la necesidad ni la utilidad de los destructores (y recuerde que no se debe invocar **finalize()** directamente, por lo que dicho método no constituye una solución). Si queremos que se realice algún tipo de limpieza distinta de la propia liberación del espacio de almacenamiento, sigue siendo necesario invocar explícitamente un método apropiado en Java, que será el equivalente del destructor de C++ pero sin la comodidad asociada a éste.

Recuerde que no están garantizadas ni la depuración de memoria ni la finalización. Si la máquina virtual Java (JVM) no está próxima a quedarse sin memoria, puede que no pierda tiempo recuperando espacio mediante el mecanismo de depuración de memoria.

La condición de terminación

En general, no podemos confiar en que **finalize()** sea invocado y es necesario crear métodos de “limpieza” separados e invocarlos explícitamente. Por tanto, parece que **finalize()** sólo resulta útil para oscuras tareas de limpieza de memoria que la mayoría de los programadores nunca van a tener que utilizar. Sin embargo, existe un caso interesante de uso de **finalize()** que no depende de que dicha función sea invocada todas las veces. Nos referimos a la verificación de la *condición de terminación*⁴ de un objeto.

³ Joshua Bloch es todavía más tajante en su sección “*Evite los finalizadores*”: “Los finalizadores son impredecibles, a menudo peligrosos y generalmente innecesarios”. *Effective JavaTM Programming Language Guide*, p. 20 (Addison-Wesley, 2001).

⁴ Un término acuñado por Bill Venners (www.Arkinia.com) en un seminario que impartimos conjuntamente.

En el momento en que ya no estemos interesados en un objeto (cuando esté listo para ser borrado) dicho objeto deberá encontrarse en un estado en el que su memoria debe ser liberada sin riesgo. Por ejemplo, si el objeto representa un archivo abierto, el programador deberá cerrar dicho archivo antes de que el objeto se vea sometido al proceso de depuración de memoria. Si alguna parte del objeto no se limpia apropiadamente, tendremos un error en el programa que será muy difícil de localizar. Podemos utilizar **finalize()** para descubrir esta condición, incluso aunque dicho método no sea siempre invocado. Si una de las finalizaciones nos permite detectar el error, habremos descubierto el problema, que es lo único que realmente nos importa.

He aquí un ejemplo simple de cómo podría emplearse dicho método:

```
//: initialization/TerminationCondition.java
// Uso de finalize() para detectar un objeto
// que no ha sido limpiado apropiadamente.

class Book {
    boolean checkedOut = false;
    Book(boolean checkOut) {
        checkedOut = checkOut;
    }
    void checkIn() {
        checkedOut = false;
    }
    protected void finalize() {
        if(checkedOut)
            System.out.println("Error: checked out");
        // Normalmente, también haremos esto:
        // super.finalize(); // Invocar la versión de la clase base
    }
}

public class TerminationCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Limpieza apropiada:
        novel.checkIn();
        // Falta la referencia, nos olvidamos de limpiar:
        new Book(true);
        // Forzar la depuración de memoria y la finalización:
        System.gc();
    }
} /* Output:
Error: checked out
*///:-
```

La condición de terminación es que se supone que todos los objetos **Book** (libro) deben ser devueltos (*check in*) antes de que los procese el depurador de memoria, pero en **main()** hay un error de programación por el que uno de los libros no es devuelto. Sin **finalize()** para verificar la condición de terminación, este error puede ser difícil de localizar.

Observe que se utiliza **System.gc()** para forzar la finalización. Pero, incluso aunque no usáramos ese método, resulta altamente probable que llegáramos a descubrir el objeto **Book** erróneo ejecutando repetidamente el programa (suponiendo que el programa asigne un espacio de almacenamiento suficiente como para provocar la ejecución del depurador de memoria).

Por regla general, debemos asumir que la versión de **finalize()** de la clase base también estará llevando a cabo alguna tarea importante, por lo que convendrá invocarla utilizando **super**, como puede verse en **Book.finalize()**. En este caso, hemos desactivado esa llamada mediante comentarios, porque requiere utilizar los mecanismos de tratamiento de excepciones de los que aún no hemos hablado en detalle.

Ejercicio 10: (2) Cree una clase con un método **finalize()** que imprima un mensaje. En **main()**, cree un objeto de esa clase. Explique el comportamiento del programa.

Ejercicio 11: (4) Modifique el ejercicio anterior de modo que siempre se invoque el método **finalize()**.

Ejercicio 12: (4) Cree una clase denominada **Tank** (tanque) que pueda ser llenado y vaciado, y cuya *condición de terminación* es que el objeto debe estar vacío en el momento de limpiarlo. Escriba un método **finalize()** que verifique esta condición de terminación. En **main()**, compruebe los posibles casos que pueden producirse al utilizar los objetos **Tank**.

Cómo funciona un depurador de memoria

Si su experiencia anterior es con lenguajes de programación en los que asignar espacio de almacenamiento a los objetos en el cùmulo de memoria resulta muy caro en términos de recursos de procesamiento, puede que piense que el mecanismo Java de asignar todo el espacio de almacenamiento (excepto para las primitivas) en el cùmulo de memoria es también caro. Sin embargo, resulta que el mecanismo de depuración de memoria puede contribuir significativamente a *acelerar* la velocidad de creación de los objetos. Esto puede parecer un poco extraño a primera vista (el que la liberación del espacio de almacenamiento afecte a la velocidad de asignación de dicho espacio) pero ésa es la forma en que funcionan algunas máquinas JVM, lo que implica que la asignación de espacio de almacenamiento en el cùmulo de memoria para los objetos Java puede ser casi tan rápida como crear espacio de almacenamiento *en la pila* en otros lenguajes.

Por ejemplo, podemos pensar en el cùmulo de memoria de C++ como si fuera una parcela de terreno en la que cada objeto ocupa su propio lote de espacio. Este terreno puede quedar abandonado y debe ser reutilizado. En algunas JVM el cùmulo de memoria Java es bastante distinto: se parece más a una cinta transportadora que se desplaza hacia adelante cada vez que se asigna un nuevo objeto. Esto quiere decir que la asignación de espacio de almacenamiento a los objetos es notablemente rápida: simplemente se desplaza hacia adelante el “*puntero del cùmulo de memoria*” para que apunte a un espacio vacío, por lo que equivale en la práctica a la asignación de espacio de almacenamiento en la pila en C++ (por supuesto, existe un cierto gasto adicional de recursos de procesamiento asociado a las tareas de administración del espacio, pero esos recursos son mínimos comparados con los necesarios para localizar espacio de almacenamiento).

Algun lector podría argumentar que el cùmulo de memoria no puede considerarse como una cinta transportadora, y que si lo consideramos de esa manera comenzarán a entrar en acción los mecanismos de paginación de memoria, desplazando información hacia y desde el disco, de modo que puede parecer que disponemos de más memoria de la que realmente existe. Los mecanismos de paginación afectan significativamente a la velocidad. Además, después de crear un número grande de objetos terminará por agotarse la memoria. El truco radica en que el depurador de memoria, mientras se encarga de liberar el espacio de almacenamiento que ya no es necesario, compacta también todos los objetos en el cùmulo de memoria, con lo que el efecto es que el “*puntero del cùmulo de memoria*” queda situado más cerca del comienzo de esa “cinta transportadora”, más alejado del punto en el que pueda producirse un fallo de página. El depurador de memoria se encarga de reordenar la información y hace posible utilizar ese modelo de cùmulo de memoria infinito de alta velocidad para asignar el espacio de almacenamiento.

Para entender el proceso de depuración de memoria en Java, resulta útil analizar cómo funcionan los esquemas de depuración de memoria en otros sistemas. Una técnica muy simple, pero muy lenta, de depuración de memoria es la que se denomina *recuento de referencias*. Esto quiere decir que cada objeto contiene un contador de referencias y que, cada vez que se asocia una referencia a ese objeto, ese contador de referencias se incrementa. De la misma forma, cada vez que una referencia se sale de ámbito o se la asigna el valor **null**, se reduce el contador de referencias. Este mecanismo de gestión del número de referencias representa un gasto adicional pequeño, pero constante, que tiene lugar mientras dura la ejecución del programa. El depurador de memoria recorre la lista completa de objetos, y donde encuentra uno cuyo contador de referencias sea cero, libera el espacio de almacenamiento correspondiente (sin embargo, los mecanismos basados en el recuento de referencias suelen liberar los objetos tan pronto como el contador pasa a valer cero). La desventaja es que, si hay una serie de objetos que se refieren circularmente entre sí, podemos encontrarnos con números de referencia distintos de cero a pesar de que los objetos ya no sean necesarios. La localización de esos grupos auto-referenciales exige al depurador de memoria realizar un trabajo adicional bastante significativo. Este mecanismo de recuento de referencias se suele utilizar de forma bastante habitual para explicar uno de los posibles mecanismos de depuración de memoria, pero no parece que se use en ninguna implementación de máquina JVM.

En otros esquemas más rápidos, la depuración de memoria no está basada en el recuento del número de referencia, en su lugar, se basa en el concepto de que cualquier objeto que no esté muerto debe, en último término, ser trazable hasta otra referencia que esté localizada en la pila o en almacenamiento estático. Esta cadena debe atravesar varios niveles de objetos. De este modo, si comenzamos en la pila y en el área de almacenamiento estático y vamos recorriendo todas las referencias, podremos localizar todos los objetos vivos. Para cada referencia que encontramos, debemos continuar con el proceso de

traza, entrando en el objeto al que apunta la referencia y siguiendo a continuación todas las referencias incluidas en ese objeto, entrando en los objetos a los que esas referencias apuntan, etc., hasta recorrer todo el árbol que se origina en la referencia situada en la pila o en almacenamiento estático. Cada objeto a través del cual pasemos seguirá estando vivo. Observe que no se presenta el problema de los grupos auto-referenciales: los objetos de esos grupos no serán recorridos durante este proceso de construcción del árbol, por lo que se puede deducir automáticamente que hay que depurarlos.

En la técnica que acabamos de describir, la JVM utiliza un esquema de depuración de memoria *adaptativo*, en el que lo que hace con los objetos vivos que localice dependerá de la variante del esquema que se esté utilizando actualmente. Una de esas variantes es *parar y copiar*, lo que quiere decir que (por razones que luego comentaremos) se detiene primero el programa (es decir, no se trata de un esquema de depuración de memoria que funcione en segundo plano). Entonces cada objeto vivo se copia desde un punto del cúmulo de memoria a otro, dejando detrás todos los objetos muertos. Además, a medida que se copien los objetos en la nueva zona del cúmulo de memoria, se los empaqueta de modo que ocupen un espacio de almacenamiento mínimo compactando así el área ocupada (y permitiendo que se asigne nuevo espacio de almacenamiento inmediatamente a continuación del área recién descrita, como antes hemos comentado).

Por supuesto, cuando se desplaza un objeto de un sitio a otro, es preciso modificar todas las referencias que apuntan al objeto. Las referencias que apunten al objeto desde el cúmulo de memoria o el área de almacenamiento estático pueden modificarse directamente, pero puede que haya otras referencias apuntando a este objeto que sean encontradas posteriormente, durante el proceso de construcción del árbol. Estas referencias se irán modificando a medida que sean encontradas (imagine, por ejemplo, que se utilizara una tabla para establecer la correspondencia entre las antiguas direcciones y las nuevas).

Hay dos problemas que hacen que estos denominados "depuradores copiadores" sean poco eficientes. El primero es la necesidad de disponer de dos áreas de cúmulo de memoria, para poder mover las secciones de memoria entre una y otra, lo que en la práctica significa que hace falta el doble de memoria de la necesaria. Algunas máquinas JVM resuelven este problema asignando el cúmulo de memoria de segmento en segmento, según sea necesario, y simplemente copiando de un segmento a otro.

El segundo problema es el propio proceso de copia. Una vez que el programa se estabilice, después de iniciada la ejecución, puede que no genere ningún objeto muerto o que genere muy pocos. A pesar de eso, el depurador copiador seguirá copiando toda la memoria de un sitio a otro, lo que constituye un desperdicio de recursos. Para evitar esto, algunas máquinas JVM detectan que no se están generando nuevos objetos muertos y comutan a un esquema distinto (ésta es la parte "adaptativa"). Este otro esquema se denomina *marcar y eliminar*, y es el que utilizaban de manera continua las anteriores versiones de la JVM de Sun. Para uso general, esta técnica de marcar y eliminar es demasiado lenta, pero resulta, sin embargo, muy rápida cuando sabemos de antemano que no se están generando objetos muertos.

La técnica de marcar y eliminar sigue la misma lógica de comenzar a partir de la pila y del almacenamiento estático y trazar todas las referencias para encontrar los objetos vivos. Sin embargo, cada vez que se encuentra un objeto vivo, éste se marca activando un indicador contenido en el mismo, pero sin aplicarle ningún mecanismos de depuración. Sólo cuando el proceso de marcado ha terminado se produce la limpieza. Durante esa fase, se libera la memoria asignada a los objetos muertos. Sin embargo, hay que observar que no se produce ningún proceso de copia, por lo que si el depurador de memoria decide compactar un cúmulo de memoria fragmentado, tendrá que hacerlo moviendo los objetos de un sitio a otro.

El concepto de "parar y copiar" hace referencia a la idea de que este tipo de depuración de memoria *no* se hace en segundo plano; en lugar de ello, se detiene el programa mientras tiene lugar la depuración de memoria. En la documentación técnica de Sun podrá encontrar muchas referencias al mecanismo de depuración de memoria donde se dice que se trata de un proceso de segundo plano de baja prioridad, pero la realidad es que la depuración de memoria no estaba implementada de esa forma en las primeras máquinas JVM de Sun. En lugar de ello, el depurador de memoria de Sun detenia el programa cuando detectaba que había poca memoria libre. La técnica de marcar y limpiar también requiere que se detenga el programa.

Como hemos mencionado anteriormente, en la máquina JVM descrita aquí, la memoria se asigna en bloques de gran tamaño. Si asignamos un objeto grande, éste obtendrá su propio bloque. La técnica de detención y copiado estricta requiere que se copien todos los objetos vivos desde el cúmulo de memoria de origen hasta un nuevo cúmulo de memoria antes de poder liberar el primero, lo que implica una gran cantidad de memoria. Utilizando bloques, el mecanismo de depuración de memoria puede normalmente copiar los objetos a los bloques muertos a medida que los va depurando. Cada bloque dispone de un *contador de generación* para ver si está vivo. En el caso normal, sólo se compactan los bloques creados desde la última pasada de depuración de memoria; para todos los demás bloques se incrementará el contador de generación si han sido referenciados desde algún sitio. Esto permite gestionar el caso normal en el que se dispone de un gran número de objetos temporales

de corta duración. Periódicamente, se hace una limpieza completa, en la que los objetos de gran tamaño seguirán sin ser copiados (simplemente se incrementará su contador de generación) y los bloques que contengan objetos pequeños se copiarán y compactarán. La máquina JVM monitoriza la eficiencia del depurador de memoria y, si este mecanismo se convierte en una pérdida de tiempo porque todos los objetos son de larga duración, comuta al mecanismo de marcado y limpieza. De forma similar, la JVM controla hasta qué punto es efectiva la técnica de marcado y limpieza, y si el cúmulo de memoria comienza a estar fragmentado, comuta al mecanismo de detención y copiado. Aquí es donde entra en acción la parte "adaptativa" del mecanismo, al que podríamos describir de manera rimbombante como : "mecanismo adaptativo generacional de detención-copiado y marcado-limpieza".

Existen varias posibles optimizaciones de la velocidad de una máquina JVM. Una especialmente importante afecta a la operación del cargador y es lo que se denomina compilador *just-in-time* (JIT). Un compilador JIT convierte parcial o totalmente un programa a código máquina nativo, de modo que dicho código no necesite ser interpretado por la JVM, con lo que se ejecutará mucho más rápido. Cuando debe cargarse una clase (normalmente, la primera vez que queremos crear un objeto en esa clase) se localiza el archivo .class y se carga en memoria el código intermedio correspondiente a dicha clase. En este punto, una posible técnica consiste en compilar simplemente todo el código, para generar un código máquina, pero esto tiene dos desventajas: necesita algo más de tiempo, lo cual (si tenemos en cuenta toda la vida del programa) puede representar una gran cantidad de recursos adicionales; e incrementa el tamaño del ejecutable (el código intermedio es bastante más compacto que el código JIT expandido), y esto puede provocar la aparición del fenómeno de paginación, lo que ralentiza enormemente los programas. Otra técnica alternativa es la *evaluación lenta*, que consiste en que el código intermedio no se compila para generar código máquina hasta el momento necesario. De este modo, puede que nunca se llegue a compilar el código que nunca llegue a ejecutarse. Las tecnologías HotSpot de Java en los kits de desarrollo JDK recientes adoptan una técnica similar, optimizando de manera incremental un fragmento de código cada vez que se ejecuta, por lo que cuanto más veces se ejecute, más rápido lo hará.

Inicialización de miembros

Java adopta medidas especiales para garantizar que las variables se inicialicen adecuadamente antes de usarlas. En el caso de las variables locales de un método, esta garantía se presenta en la forma de errores en tiempo de compilación. Por tanto, si escribimos:

```
void f() {
    int i;
    i++; // Error -- i no inicializada
}
```

obtendremos un mensaje de error que dice que *i* puede no haber sido no inicializada. Por supuesto, el compilador podría haber dado a *i* un valor predeterminado, pero el hecho de que exista una variable local no inicializada es un error del programador, y el valor predeterminado estaría encubriendo ese error. Forzando al programador a proporcionar un valor de inicialización, es más probable que se detecte el error.

Sin embargo, si hay un campo de tipo primitivo en una clase, las cosas son algo distintas. Como hemos visto en el Capítulo 2, *Todo es un objeto*, se garantiza que cada campo primitivo de una clase contendrá un valor inicial. He aquí un programa que permite verificar este hecho y mostrar los valores:

```
//: initialization/InitialValues.java
// Muestra los valores iniciales predeterminados.
import static net.mindview.util.Print.*;

public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;
```

```

void printInitialValues() {
    print("Data type      Initial value");
    print("boolean        " + t);
    print("char           [" + c + "]");
    print("byte           " + b);
    print("short          " + s);
    print("int            " + i);
    print("long           " + l);
    print("float          " + f);
    print("double         " + d);
    print("reference      " + reference);
}
public static void main(String[] args) {
    InitialValues iv = new InitialValues();
    iv.printInitialValues();
    /* También podríamos escribir:
    new InitialValues().printInitialValues();
    */
}
} /* Output:
Data type      Initial value
boolean        false
char           []
byte           0
short          0
int            0
long           0
float          0.0
double         0.0
reference      null
*///:-
```

Puede ver que, aún cuando no se han especificado los valores, los campos se inicializan automáticamente (el valor correspondiente a **char** es cero, lo que se imprime como un espacio). De modo que, al menos, no existe ningún riesgo de llegar a trabajar con variables no inicializadas.

Cuando definimos una referencia a objeto dentro de una clase sin inicializarse como nuevo objeto, dicha referencia se inicializa con el valor especial **null**.

Especificación de la inicialización

¿Qué sucede si queremos dar un valor inicial a una variable? Una forma directa de hacerlo consiste, simplemente, en asignar el valor en el punto en que definamos la variable dentro de la clase (observe que no se puede hacer esto en C++, aunque los programadores novatos de C++ siempre lo intentan). En el siguiente fragmento de código, se modifican las definiciones de los campos de la clase **InitialValues** para proporcionar los correspondientes valores iniciales:

```

//: initialization/InitialValues2.java
// Definición de valores iniciales explícitos.

public class InitialValues2 {
    boolean bool = true;
    char ch = 'x';
    byte b = 47;
    short s = 0xff;
    int i = 999;
    long lng = 1;
    float f = 3.14f;
    double d = 3.14159;
}
//:-
```

También podemos inicializar objetos no primitivos de la misma forma. Si **Depth** es una clase, podemos crear una variable e inicializarla como sigue:

```
//: initialization/Measurement.java
class Depth {}

public class Measurement {
    Depth d = new Depth();
    // ...
} //
```

Si no hubiéramos dado a **d** un valor inicial y tratáramos de usarlo de todos modos, obtendríamos un error de tiempo de ejecución denominado *excepción* (hablaremos de este tema en el Capítulo 12, *Tratamiento de errores mediante excepciones*).

Podemos incluso llamar a un método para proporcionar un valor de inicialización:

```
//: initialization/MethodInit.java
public class MethodInit {
    int i = f();
    int f() { return 11; }
} //
```

Por supuesto, este método puede tener argumentos, pero dichos argumentos no pueden ser otros miembros de la clase que todavía no hayan sido inicializados. Por tanto, podemos hacer esto:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
    int i = f();
    int j = g(i);
    int f() { return 11; }
    int g(int n) { return n * 10; }
} //
```

Pero no esto:

```
//: initialization/MethodInit3.java
public class MethodInit3 {
    //! int j = g(i); // Referencia anticipada ilegal
    int i = f();
    int f() { return 11; }
    int g(int n) { return n * 10; }
} //
```

Este es uno de los ejemplos en los que el compilador *se queja*, como es lógico, acerca de las referencias anticipadas, ya que este caso tiene que ver con el orden de inicialización más que con la forma en que se compila el programa.

Esta técnica de inicialización resulta bastante simple y directa. Tiene la limitación de que *todos* los objetos de tipo **InitialValues** contendrán el mismo valor de inicialización. En ocasiones, esto es, exactamente, lo que queremos, pero en otros casos hace falta más flexibilidad.

Inicialización mediante constructores

Podemos emplear el constructor para realizar la inicialización, y esto nos proporciona una mayor flexibilidad a la hora de programar, porque podemos invocar métodos y realizar acciones en tiempo de ejecución para determinar los valores iniciales. Sin embargo, es preciso tener presente una cosa: esto no excluye la inicialización automática que tiene lugar antes de entrar en el constructor. Así que, si escribimos por ejemplo:

```
//: initialization/Counter.java
public class Counter {
    int i;
    Counter() { i = 7; }
```

```
// ...
} //:-
```

entonces `i` se inicializará primero con el valor 0, y luego con el valor 7. Esto es cierto para todos los tipos primitivos y también para las referencias a objetos, incluyendo aquellos a los que se inicialice de manera explícita en el punto en el que se los defina. Por esta razón, el compilador no trata de obligarnos a inicializar los elementos dentro del constructor en ningún sitio concreto o antes de utilizarlos: la inicialización ya está garantizada.

Orden de inicialización

Dentro de una clase, el orden de inicialización se determina mediante el orden en que se definen las variables en la clase. Las definiciones de variables pueden estar dispersas a través de y entre las definiciones de métodos, pero las variables se inicializan antes de que se pueda invocar cualquier método, incluso el constructor. Por ejemplo:

```
//: initialization/OrderOfInitialization.java
// Ilustra el orden de inicialización.
import static net.mindview.util.Print.*;

// Cuando se invoca el constructor para crear un
// objeto Window, aparecerá el mensaje:
class Window {
    Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
    Window w1 = new Window(1); // Antes del constructor
    House() {
        // Mostrar que estamos en el constructor:
        print("House()");
        w3 = new Window(33); // Reinicializar w3
    }
    Window w2 = new Window(2); // Despues del constructor
    void f() { print("f()"); }
    Window w3 = new Window(3); // Al final
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f(); // Muestra que la construcción ha finalizado
    }
} /* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
```

En `House`, las definiciones de los objetos `Window` han sido dispersadas intencionadamente, para demostrar que todos ellos se inicializan antes de entrar en el constructor o de que suceda cualquier otra cosa. Además, `w3` se reinicializa dentro del constructor.

Examinando la salida, podemos ver que la referencia a `w3` se inicializa dos veces. Una vez antes y otra durante la llamada al constructor (el primer objeto será eliminado, por lo que podrá ser procesado por el depurador de memoria más adelante). Puede que esto no le parezca eficiente a primera vista, pero garantiza una inicialización adecuada: ¿qué sucedería si se definiera un constructor sobrecargado que *no* inicializara `w3` y no hubiera una inicialización “predeterminada” para `w3` en su definición?

Inicialización de datos estáticos

Sólo existe una única área de almacenamiento para un dato de tipo **static**, independientemente del número de objetos que se creen. No se puede aplicar la palabra clave **static** a las variables locales, así que sólo se aplica a los campos. Si un campo es una primitiva de tipo **static** y no se inicializa, obtendrá el valor inicial estándar correspondiente a su tipo. Si se trata de una referencia a un objeto, el valor predeterminado de inicialización será **null**.

Si desea colocar la inicialización en el punto de la definición, será similar al caso de las variables no estáticas.

Para ver *cuándo* se inicializa el almacenamiento de tipo **static**, ha aquí un ejemplo:

```
//: initialization/StaticInitialization.java
// Especificación de valores iniciales en una definición de clase.
import static net.mindview.util.Print.*;

class Bowl {
    Bowl(int marker) {
        print("Bowl(" + marker + ")");
    }
    void f1(int marker) {
        print("f1(" + marker + ")");
    }
}

class Table {
    static Bowl bowl1 = new Bowl(1);
    Table() {
        print("Table()");
        bowl1.f1(1);
    }
    void f2(int marker) {
        print("f2(" + marker + ")");
    }
    static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
    Bowl bowl3 = new Bowl(3);
    static Bowl bowl4 = new Bowl(4);
    Cupboard() {
        print("Cupboard()");
        bowl4.f1(2);
    }
    void f3(int marker) {
        print("f3(" + marker + ")");
    }
    static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        print("Creating new Cupboard() in main");
        new Cupboard();
        print("Creating new Cupboard() in main");
        new Cupboard();
        table.f2(1);
        cupboard.f3(1);
    }
    static Table table = new Table();
    static Cupboard cupboard = new Cupboard();
}
```

```

} /* Output:
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
*///:-
```

Bowl permite visualizar la creación de una clase, mientras que **Table** y **Cupboard** tienen miembros de tipo **static** de **Bowl** dispersos por sus correspondientes definiciones de clase. Observe que **Cupboard** crea un objeto **Bowl bowl3** no estático antes de las definiciones de tipo **static**.

Examinando la salida, podemos ver que la inicialización de **static** sólo tiene lugar en caso necesario. Si no se crea un objeto **Table** y nunca se hace referencia a **Table.bowl1** o **Table.bowl2**, los objetos **Bowl** estáticos **bowl1** y **bowl2** nunca se crearán. Sólo se inicializarán cuando se cree el *primer* objeto **Table** (cuando tenga lugar el primer acceso **static**). Después de eso, los objetos **static** no se reinicializan.

El orden de inicialización es el siguiente: primero se inicializan los objetos estáticos, si es que no han sido ya inicializados con una previa creación de objeto, y luego se inicializan los objetos no estáticos. Podemos ver que esto es así examinando la salida del programa. Para examinar **main()** (un método **static**), debe cargarse la clase **StaticInitialization**, después de lo cual se inicializan sus campos estáticos **table** y **cupboard**, lo que hace que *esas* clases se carguen y, como ambas contienen objetos **Bowl** estáticos, eso hace que se cargue la clase **Bowl**. Por tanto, todas las clases de este programa concreto se cargan antes de que dé comienzo **main()**. Éste no es el caso usual, porque en los programas típicos no tendremos todo vinculado entre sí a través de valores estáticos, como sucede en este ejemplo.

Para resumir el proceso de creación de un objeto, considere una clase **Dog**:

1. Aunque no utilice explícitamente la palabra clave **static**, el constructor es, en la práctica, un método **static**. Por tanto, la primera vez que se crea un objeto de tipo **Dog**, o la primera vez que se accede a un método estático o a un campo estático de la clase **Dog**, el intérprete de Java debe localizar **Dog.class**, para lo cual analiza la ruta de clases que en ese momento haya definido (*classpath*).
2. A medida que se carga **Dog.class** (creando un objeto **Class**, acerca del cual hablaremos posteriormente) se ejecutan todos sus inicializadores de tipo **static**. De este modo, la inicialización de tipo **static** sólo tiene lugar una vez, cuando se carga por primera vez el objeto **Class**.
3. Cuando se crea un nuevo objeto con **new Dog()**, el proceso de construcción del objeto **Dog** asigna primero el suficiente espacio de almacenamiento para el objeto **Dog** en el círculo de memoria.
4. Este espacio de almacenamiento se rellena con ceros, lo que asigna automáticamente sus valores predeterminados a todas las primitivas del objeto **Dog** (cero a los números y el equivalente para **boolean** y **char**); asimismo, este proceso hace que las referencias queden con el valor **null**.
5. Se ejecutan las inicializaciones especificadas en el lugar en el que se definan los campos.
6. Se ejecutan los constructores. Como podremos ver en el Capítulo 7, *Reutilización de las clases*, esto puede implicar un gran número de actividades, especialmente cuando estén implicados los mecanismos de herencia.

Inicialización static explícita

Java permite agrupar otras inicializaciones estáticas dentro de una "cláusula" **static** especial (en ocasiones denominada *bloque estático*) en una clase. El aspecto de esta cláusula es el siguiente:

```
//: initialization/Spoon.java
public class Spoon {
    static int i;
    static {
        i = 47;
    }
} //:-
```

Parece ser un método, pero se trata sólo de la palabra clave **static** seguida de un bloque de código. Este código, al igual que otras inicializaciones estáticas sólo se ejecuta una vez: la primera vez que se crea un objeto de esa clase o la primera vez que se accede a un miembro de tipo **static** de esa clase (incluso aunque nunca se cree un objeto de dicha clase). Por ejemplo:

```
//: initialization/ExplicitStatic.java
// Inicialización static explícita con la cláusula "static".
import static net.mindview.util.Print.*;

class Cup {
    Cup(int marker) {
        print("Cup(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

class Cups {
    static Cup cup1;
    static Cup cup2;
    static {
        cup1 = new Cup(1);
        cup2 = new Cup(2);
    }
    Cups() {
        print("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        print("Inside main()");
        Cups.cup1.f(99); // (1)
    }
    // static Cups cups1 = new Cups(); // (2)
    // static Cups cups2 = new Cups(); // (2)
} /* Output:
Inside main()
Cup(1)
Cup(2)
f(99)
*:-
```

Los inicializadores **static** para **Cups** se ejecutan cuando tiene lugar el acceso del objeto estático **cup1** en la línea marcada con (1), o si se desactiva mediante un comentario la línea (1) y se quitan los comentarios que desactivan las líneas marcadas (2). Si se desactivan mediante comentarios tanto (1) como (2), la inicialización **static** de **Cups** nunca tiene lugar, como

puede verse a la salida. Asimismo, da igual si se eliminan las marcas de comentario que están desactivando a una y otra de las líneas marcadas (2) o si se eliminan las marcas de ambas líneas; la inicialización estática tiene lugar una sola vez.

Ejercicio 13: (1) Verifique las afirmaciones contenidas en el párrafo anterior.

Ejercicio 14: (1) Cree una clase con un campo estático **String** que sea inicializado en el punto de definición, y otro campo que se inicialice mediante el bloque **static**. Añada un método **static** que imprima ambos campos y demuestre que ambos se inicializan antes de usarlos.

Inicialización de instancias no estáticas

Java proporciona una sintaxis similar, denominada *inicialización de instancia*, para inicializar las variables estáticas de cada objeto. He aquí un ejemplo:

```
//: initialization/Mugs.java
// "Inicialización de instancia" en Java.
import static net.mindview.util.Print.*;

class Mug {
    Mug(int marker) {
        print("Mug(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    {
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        print("mug1 & mug2 initialized");
    }
    Mugs() {
        print("Mugs()");
    }
    Mugs(int i) {
        print("Mugs(int)");
    }
    public static void main(String[] args) {
        print("Inside main()");
        new Mugs();
        print("new Mugs() completed");
        new Mugs(1);
        print("new Mugs(1) completed");
    }
} /* Output:
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
*///:-
```

Podemos ver que la cláusula de inicialización de instancia:

```
{
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
}
```

parece exactamente como la cláusula de inicialización estática, salvo porque falta la palabra clave **static**. Esta sintaxis es necesaria para soportar la inicialización de *clases internas anónimas* (véase el Capítulo 10, *Clases internas*), pero también nos permite garantizar que ciertas operaciones tendrán lugar independientemente de qué constructor explícito se invoque. Examinando la salida, podemos ver que la cláusula de inicialización de instancia se ejecuta antes de los dos constructores.

Ejercicio 15: (1) Cree una clase con un campo **String** que se inicialice mediante una cláusula de inicialización de instancia.

Inicialización de matrices

Una matriz es, simplemente, una secuencia de objetos o primitivas que son todos del mismo tipo y que se empaquetan juntos, utilizando un único nombre identificador. Las matrices se definen y usan mediante el *operador de indexación* `[]`. Para definir una referencia de una matriz, basta con incluir unos corchetes vacíos detrás del nombre del tipo:

```
int[] a1;
```

También puede colocar los corchetes después del identificador para obtener exactamente el mismo resultado:

```
int a1[];
```

Esto concuerda con las expectativas de los programadores de C y C++. Sin embargo, el primero de los dos estilos es una sintaxis más adecuada, ya que comunica mejor que el tipo que estamos definiendo es una “matriz de variables de tipo **int**”. Este estilo es el que emplearemos en el libro.

El compilador no permite especificar el tamaño de la matriz. Esto nos retrotrae al problema de las “referencias” anteriormente comentado. Todo lo que tenemos en este punto es una referencia a una matriz (habiéndole asignado el suficiente espacio de almacenamiento para esa referencia), sin que se haya asignado ningún espacio para el propio objeto matriz. Para crear espacio de almacenamiento para la matriz, es necesario escribir una expresión de inicialización. Para las matrices, la inicialización puede hacerse en cualquier lugar del código, pero también podemos utilizar una clase especial de expresión de inicialización que sólo puede emplearse en el punto donde se cree la matriz. Esta inicialización especial es un conjunto de valores encerrados entre llaves. En este caso, el compilador se ocupa de la asignación de espacio (el equivalente de utilizar **new**). Por ejemplo:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Pero entonces, ¿por qué íbamos a definir una referencia a una matriz sin definir la propia matriz?

```
int[] a2;
```

Bueno, la razón para definir una referencia sin definir la matriz asociada es que en Java es posible asignar una matriz a otra, por lo que podríamos escribir:

```
a2 = a1;
```

Lo que estamos haciendo con esto es, en realidad, copiar una referencia, como se ilustra a continuación:

```
//: initialization/ArraysOfPrimitives.java
import static net.mindview.util.Print.*;
public class ArraysOfPrimitives {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
```

```

        a2[i] = a2[i] + 1;
        for(int i = 0; i < a1.length; i++)
            print("a1[" + i + "] = " + a1[i]);
    }
} /* Output:
a1[0] = 2
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*///:-

```

Como puede ver, a **a1** se le da un valor de inicialización, pero a **a2** no; a **a2** se le asigna posteriormente un valor que en este caso es la referencia a otra matriz. Puesto que **a2** y **a1** apuntan ambas a la misma matriz, los cambios que se realicen a través de **a2** podrán verse en **a1**.

Todas las matrices tienen un miembro intrínseco (independientemente de si son matrices de objetos o matrices de primitivas) que puede consultarse (aunque no modificarse) para determinar cuántos miembros hay en la matriz. Este miembro es **length**. Puesto que las matrices en Java, al igual que en C y C++, comienzan a contar a partir del elemento cero, el elemento máximo que se puede indexar es **length - 1**. Si nos salimos de los límites, C y C++ lo aceptarán en silencio y permitirán que hagamos lo que queramos en la memoria, lo cual es el origen de muchos errores graves. Sin embargo, Java nos protege de tales problemas provocando un error de tiempo de ejecución (una *excepción*) si nos salimos de los límites.⁵

¿Qué sucede si no sabemos cuántos elementos vamos a necesitar en la matriz en el momento de escribir el programa? Simplemente, bastará con utilizar **new** para crear los elementos de la matriz. Aquí, **new** funciona incluso aunque se esté creando una matriz de primitivas (sin embargo, **new** no permite crear una primitiva simple que no forme parte de una matriz):

```

//: initialization/ArrayNew.java
// Creación de matrices con new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
    public static void main(String[] args) {
        int[] a;
        Random rand = new Random(47);
        a = new int[rand.nextInt(20)];
        print("length of a = " + a.length);
        print(Arrays.toString(a));
    }
} /* Output:
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*///:-

```

El tamaño de la matriz se selecciona aleatoriamente utilizando el método **Random.nextInt()**, que genera un valor entre cero y el que se le pase como argumento. Debido a la aleatoriedad, está claro que la creación de la matriz tiene lugar en tiempo de ejecución. Además, como la salida del programa muestra que los elementos de matriz de tipos primitivos se inicializan automáticamente con valores “vacíos” (para las variables numéricas y **char**, se inicializan con cero mientras que para variables **boolean** se inicializan con **false**).

El método **Arrays.toString()**, que forma parte de la biblioteca estándar **java.util**, genera una versión imprimible de una matriz unidimensional.

⁵ Por supuesto, comprobar cada acceso de una matriz cuesta tiempo y código, y no hay manera de desactivar esas comprobaciones, lo que quiere decir que los accesos a matrices pueden ser una fuente de inefficiencia en los programas, si se producen en alguna sección crítica. En aras de la seguridad en Internet y de la productividad de los programadores, los diseñadores en Java pensaron que resultaba conveniente pagar este precio para evitar los errores asociados con las matrices. Aunque el programador pueda sentirse tentado de escribir código para tratar de hacer que los accesos a las matrices sean más eficientes, esto es una pérdida de tiempo, porque las optimizaciones automáticas en tiempo de compilación y en tiempo de ejecución se encargan de acelerar los accesos a las matrices.

Por supuesto, en este caso la matriz también podía haber sido definida e inicializada en la misma instrucción:

```
int[] a = new int[rand.nextInt(20)];
```

Ésta es la forma preferible de hacerlo, siempre que se pueda.

Si se crea una matriz que no es de tipo primitivo, lo que se crea es una matriz de referencias. Considere el tipo envoltorio **Integer**, que es una clase y no una primitiva:

```
//: initialization/ArrayClassObj.java
// Creación de una matriz de objetos no primitivos.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] a = new Integer[rand.nextInt(20)];
        print("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            a[i] = rand.nextInt(500); // Conversión automática
        print(Arrays.toString(a));
    }
} /* Output: (Sample)
length of a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278, 498, 361, 20]
*///:-
```

Aquí, incluso después de invocar **new** para crear la matriz:

```
Integer[] a = new Integer[rand.nextInt(20)];
```

es sólo una matriz de referencias y la inicialización no se completa hasta que se inicialice la propia referencia creando un nuevo objeto **Integer** (mediante el mecanismo de conversión automática, en este caso):

```
a[i] = rand.nextInt(500);
```

Sin embargo, si nos olvidamos de crear un objeto, obtendremos una excepción en tiempo de ejecución cuando tratemos de utilizar esa posición vacía de la matriz.

También es posible inicializar matrices de objetos mediante una lista encerrada entre llaves. He aquí dos formas de hacerlo:

```
//: initialization/ArrayInit.java
// Inicialización de la matriz.
import java.util.*;

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            3, // Conversión automática
        };
        Integer[] b = new Integer[]{
            new Integer(1),
            new Integer(2),
            3, // Conversión automática
        };
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(b));
    }
} /* Output:
[1, 2, 3]
[1, 2, 3]
*///:-
```

En ambos casos, la coma final de la lista de inicializadores es opcional (esta característica permite un mantenimiento más fácil de las listas de gran tamaño).

Aunque la primera forma es útil, es más limitada, porque sólo puede emplearse en el punto donde se define la matriz. Podemos utilizar las formas segunda y tercera en cualquier lugar, incluso dentro de una llamada a un método. Por ejemplo, podríamos crear una matriz de objetos **String** para pasársela a otro método **main()**, con el fin de proporcionar argumentos de línea de comandos alternativos a ese método **main()**:

```
//: initialization/DynamicArray.java
// Inicialización de la matriz.

public class DynamicArray {
    public static void main(String[] args) {
        Other.main(new String[]{"fiddle", "de", "dum"});
    }
}

class Other {
    public static void main(String[] args) {
        for(String s : args)
            System.out.print(s + " ");
    }
} /* Output:
fiddle de dum
*///:-
```

La matriz creada para el argumento de **Other.main()** se crea en el punto correspondiente a la llamada al método, así que podemos incluso proporcionar argumentos alternativos en el momento de la llamada.

Ejercicio 16: (1) Cree una matriz de objetos **String** y asigne un objeto **String** a cada elemento. Imprima la matriz utilizando un bucle **for**.

Ejercicio 17: (2) Cree una clase con un constructor que tome un argumento **String**. Durante la construcción, imprima el argumento. Cree una matriz de referencias a objetos de esta clase, pero sin crear ningún objeto para asignarlo a la matriz. Cuando ejecute el programa, observe si se imprimen los mensajes de inicialización correspondientes a las llamadas al constructor.

Ejercicio 18: (1) Complete el ejercicio anterior creando objetos que asociar a la matriz de referencias.

Listas variables de argumentos

La segunda forma proporciona una sintaxis cómoda para crear e invocar métodos que pueden producir un efecto similar a las *listas variables de argumentos* de C (conocidas con el nombre de “*varargs*” en C). Esto puede incluir un número desconocido de argumentos que a su vez pueden ser de tipos desconocidos. Puesto que todas las clases se heredan en última instancia de la clase raíz común **Object** (tema del que hablaremos más adelante en el libro), podemos crear un método que admite una matriz de **Object** e invocarlo del siguiente modo:

```
//: initialization/VarArgs.java
// Uso de la sintaxis de matriz para crear listas variables de argumentos.

class A {}

public class VarArgs {
    static void printArray(Object[] args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        printArray(new Object[]{
            "fiddle", "de", "dum"
        });
    }
}
```

```

        new Integer(47), new Float(3.14), new Double(11.11)
    });
printArray(new Object[] {"one", "two", "three" });
printArray(new Object[]{new A(), new A(), new A()});
}
} /* Output: (Sample)
47 3.14 11.11
one two three
A@1a46e30 A@3e25a5 A@19821f
*///:-
```

Podemos ver que **print()** admite una matriz de tipo **Object**, y recorre la matriz utilizando la sintaxis *foreach* imprimiendo cada objeto. Las clases de la biblioteca estándar de Java generan una salida más comprensible, pero los objetos de las clases que hemos creado aquí imprimen el nombre de la clase, seguido de un signo de '@' y de una serie de dígitos hexadecimales. Por tanto, el comportamiento predeterminado (si no se define un método **toString()** para la clase, como veremos posteriormente en el libro) consiste en imprimir el nombre de la clase y la dirección del objeto.

Es posible que se encuentre con código anterior a Java SE5 escrito como el anterior para generar listas variables de argumentos. Sin embargo, en Java SE5, esta característica largo tiempo demandada ha sido finalmente añadida, por lo que ahora podemos emplear puntos suspensivos para definir una lista variable de argumentos, como puede ver en **printArray()**:

```

//: initialization/NewVarArgs.java
// Uso de la sintaxis de matrices para crear listas variables de argumentos.

public class NewVarArgs {
    static void printArray(Object... args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        // Admite elementos individuales:
        printArray(new Integer(47), new Float(3.14),
            new Double(11.11));
        printArray(47, 3.14F, 11.11);
        printArray("one", "two", "three");
        printArray(new A(), new A(), new A());
        // O una matriz:
        printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
        printArray(); // Se admite una lista vacía
    }
} /* Output: (75% match)
47 3.14 11.11
47 3.14 11.11
one two three
A@1bab50a A@c3c749 A@150bd4d
1 2 3 4
*///:-
```

Con *varargs*, ya no es necesario escribir explícitamente la sintaxis de la matriz: el compilador se encargará de completarla automáticamente cuando especifiquemos *varargs*. Seguimos obteniendo una matriz, lo cual es la razón de que **print()** siga pudiendo utilizar la sintaxis *foreach* para iterar a través de la matriz. Sin embargo, se trata de algo más que una simple conversión automática entre una lista de elementos y una matriz. Observe la penúltima línea del programa, en la que una matriz de elementos **Integer** (creados con la característica de conversión automática) se proyecta sobre una matriz **Object** (para evitar que el compilador genere una advertencia) y se pasa a **printArray()**. Obviamente, el compilador determina que esto es ya una matriz, por lo que no realiza ninguna conversión con ella. De modo que, si tenemos un grupo de elementos, podemos pasarlo como una lista, y si ya tenemos una matriz, se aceptará esa matriz como lista variable de argumentos.

La última línea del programa muestra que es posible pasar cero argumentos a una lista *vararg*. Esto resulta útil cuando existen argumentos finales opcionales:

```
//: initialization/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
    static void f(int required, String... trailing) {
        System.out.print("required: " + required + " ");
        for(String s : trailing)
            System.out.print(s + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(1, "one");
        f(2, "two", "three");
        f(0);
    }
} /* Output:
required: 1 one
required: 2 two three
required: 0
*///:-
```

Esto muestra también cómo se pueden utilizar *varargs* con un tipo especificado distinto de **Object**. Aquí, todos los *varargs* deben ser objetos **String**. Se puede utilizar cualquier tipo de argumentos en las listas *varargs*, incluyendo tipos primitivos. El siguiente ejemplo también muestra que la lista *vararg* se transforma en una matriz y que, si no hay nada en la lista, se tratará como una matriz de tamaño cero.

```
//: initialization/VarargType.java

public class VarargType {
    static void f(Character... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length);
    }
    static void g(int... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length);
    }
    public static void main(String[] args) {
        f('a');
        f();
        g(1);
        g();
        System.out.println("int[]: " + new int[0].getClass());
    }
} /* Output:
class [Ljava.lang.Character; length 1
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int[]: class [I
*///:-
```

El método *getClass()* es parte de **Object**, y lo analizaremos en detalle en el Capítulo 14, *Información de tipos*. Devuelve la clase de un objeto, y cuando se imprime esa clase, se ve una representación del tipo de la clase en forma de cadena de caracteres codificada. El carácter inicial '[' indica que se trata de una matriz del tipo situado a continuación. La 'I' indica una primitiva **int**; para comprobarlo, hemos creado una matriz de **int** en la última línea y hemos impreso su tipo. Esto permite comprobar que la utilización de *varargs* no depende de la característica de conversión automática, sino que utiliza en la práctica los tipos primitivos.

Sin embargo, las listas *vararg* funcionan perfectamente con la característica de conversión automática. Por ejemplo:

```
//: initialization/AutoboxingVarargs.java

public class AutoboxingVarargs {
    public static void f(Integer... args) {
        for(Integer i : args)
            System.out.print(i + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(new Integer(1), new Integer(2));
        f(4, 5, 6, 7, 8, 9);
        f(10, new Integer(11), 12);
    }
} /* Output:
1 2
4 5 6 7 8 9
10 11 12
*///:-
```

Observe que se pueden mezclar los tipos en una misma lista de argumentos, y que la característica de conversión automática promociona selectivamente los argumentos **int** a **Integer**.

Las listas *vararg* complican el proceso de sobrecarga, aunque éste parezca suficientemente seguro a primera vista:

```
//: initialization/OverloadingVarargs.java

public class OverloadingVarargs {
    static void f(Character... args) {
        System.out.print("first");
        for(Character c : args)
            System.out.print(" " + c);
        System.out.println();
    }
    static void f(Integer... args) {
        System.out.print("second");
        for(Integer i : args)
            System.out.print(" " + i);
        System.out.println();
    }
    static void f(Long... args) {
        System.out.println("third");
    }
    public static void main(String[] args) {
        f('a', 'b', 'c');
        f(1);
        f(2, 1);
        f(0);
        f(0L);
        //! f(); // No se compilará -- ambiguo
    }
} /* Output:
first a b c
second 1
second 2 1
second 0
third
*///:-
```

En cada caso, el compilador está utilizando la característica de conversión automática para determinar qué método sobre cargado hay que utilizar, e invocará el método que se ajuste de la forma más específica.

Pero cuando se invoca `f()` sin argumentos, el compilador no tiene forma de saber qué método debe llamar. Aunque este error es comprensible, probablemente sorprenda al programador de programas cliente.

Podemos tratar de resolver el problema añadiendo un argumento no-*vararg* a uno de los métodos:

```
//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (Won't compile)

public class OverloadingVarargs2 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(Character... args) {
        System.out.print("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} /*:-*/
```

El marcador de comentario **{CompileTimeError}** excluye este archivo del proceso de construcción Ant del libro. Si lo compila a mano podrá ver el mensaje de error:

reference to f is ambiguous, both method f(float,java.lang.Character...) in OverloadingVarargs2 and method f(java.lang.Character...) in OverloadingVarargs2 match

Si proporciona a *ambos* métodos un argumento no-*vararg*, funcionará perfectamente:

```
//: initialization/OverloadingVarargs3.java

public class OverloadingVarargs3 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(char c, Character... args) {
        System.out.println("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} /* Output:
first
second
*:-*/
```

Generalmente, sólo debe utilizarse una lista variable de argumentos en una única versión de un método sobrecargado. O bien, considere el no utilizar la lista variable de argumentos en absoluto.

Ejercicio 19: (2) Escriba un método que admita una matriz *vararg* de tipo **String**. Verifique que puede pasar una lista separada por comas de objetos **String** o una matriz **String[]** a este método.

Ejercicio 20: (1) Cree un método **main()** que utilice *varargs* en lugar de la sintaxis **main()** normal. Imprima todos los elementos de la matriz **args** resultante. Pruebe el método con diversos conjuntos de argumentos de línea de comandos.

Tipos enumerados

Una adición aparentemente poco importante en Java SE5 es la palabra clave **enum**, que nos facilita mucho las cosas cuando necesitamos agrupar y utilizar un conjunto de *tipos enumerados*. En el pasado, nos veíamos forzados a crear un conjunto

to de valores enteros constantes, pero estos conjuntos de valores no suelen casar muy bien con los conjuntos que se necesitan definir y son, por tanto, más arriesgados y difíciles de utilizar. Los tipos enumerados representan una necesidad tan común que C, C++ y diversos otros lenguajes siempre los han tenido. Antes de Java SE5, los programadores de Java estaban obligados a conocer muchos detalles y a tener mucho cuidado si querían emular apropiadamente el efecto de **enum**. Ahora, Java dispone también de **enum**, y lo ha implementado de una manera mucho más completa que la que podemos encontrar en C/C++. He aquí un ejemplo simple:

```
//: initialization/Spiciness.java
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} /*:-
```

Esto crea un tipo enumerado denominado **Spiciness** con cinco valores nominados. Puesto que las instancias de los tipos enumerados son constantes, se suelen escribir en mayúsculas por convenio (si hay múltiples palabras en un nombre, se separan mediante guiones bajos).

Para utilizar un tipo **enum**, creamos una referencia de ese tipo y la asignamos una instancia:

```
//: initialization/SimpleEnumUse.java
public class SimpleEnumUse {
    public static void main(String[] args) {
        Spiciness howHot = Spiciness.MEDIUM;
        System.out.println(howHot);
    }
} /* Output:
MEDIUM
*:-
```

El compilador añade automáticamente una serie de características útiles cuando creamos un tipo **enum**. Por ejemplo, crea un método **toString()** para que podamos visualizar fácilmente el nombre de una instancia **enum**, y ésa es precisamente la forma en que la instrucción de impresión anterior nos ha permitido generar la salida del programa. El compilador también crea un método **ordinal()** para indicar el orden de declaración de una constante **enum** concreta, y un método **static values()** que genera una matriz de valores con las constantes **enum** en el orden en que fueron declaradas:

```
//: initialization/EnumOrder.java
public class EnumOrder {
    public static void main(String[] args) {
        for(Spiciness s : Spiciness.values())
            System.out.println(s + ", ordinal " + s.ordinal());
    }
} /* Output:
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
*:-
```

Aunque los tipos enumerados **enum** parecen ser un nuevo tipo de datos, esta palabra clave sólo provoca que el compilador realice una serie de actividades mientras genera una clase para el tipo **enum**, por lo que un **enum** puede tratarse en muchos sentidos como si fuera una clase de cualquier otro tipo. De hecho, los tipos **enum** son clases y tienen sus propios métodos.

Una característica especialmente atractiva es la forma en que pueden usarse los tipos **enum** dentro de las instrucciones **switch**:

```
//: initialization/Burrito.java
public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree; }
    public void describe() {
        System.out.print("This burrito is ");
```

```

switch(degree) {
    case NOT:    System.out.println("not spicy at all.");
                  break;
    case MILD:
    case MEDIUM: System.out.println("a little hot.");
                   break;
    case HOT:
    case FLAMING:
    default:     System.out.println("maybe too hot.");
}
}

public static void main(String[] args) {
    Burrito
    plain = new Burrito(Spiciness.NOT),
    greenChile = new Burrito(Spiciness.MEDIUM),
    jalapeno = new Burrito(Spiciness.HOT);
    plain.describe();
    greenChile.describe();
    jalapeno.describe();
}
} /* Output:
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
*///:-

```

Puesto que una instrucción **switch** se emplea para seleccionar dentro de un conjunto limitado de posibilidades, se complementa perfectamente con un tipo **enum**. Observe cómo los nombres **enum** indican de una manera mucho más clara qué es lo que pretende hacer el programa.

En general, podemos utilizar un tipo **enum** como si fuera otra forma de crear un tipo de datos, y limitarnos luego a utilizar los resultados. En realidad, eso es lo importante, que no es necesario prestar demasiada atención a su uso, porque resulta bastante simple. Antes de la introducción de **enum** en Java SE5, era necesario realizar un gran esfuerzo para construir un tipo enumerado equivalente que se pudiera emplear de forma segura.

Este breve análisis es suficiente para poder comprender y utilizar los tipos enumerados básicos, pero examinaremos estos tipos enumerados más profundamente en el Capítulo 19, *Tipos enumerados*.

Ejercicio 21: (1) Cree un tipo **enum** con los seis tipos de billetes de euro de menor valor. Recorra en bucle los valores utilizando **values()** e imprima cada valor y su orden correspondiente con **ordinal()**.

Ejercicio 22: (2) Escriba una instrucción **switch** para el tipo **enum** del ejercicio anterior. En cada **case**, imprima una descripción de ese billete concreto.

Resumen

Este aparentemente elaborado mecanismo de inicialización, el constructor, nos indica la importancia crítica que las tareas de inicialización tienen dentro del lenguaje. Cuando Bjarne Stroustrup, el inventor de C++, estaba diseñando ese lenguaje, una de las primeras cosas en las que se fijó al analizar la productividad en C fue que la inicialización inadecuada de las variables es responsable de una parte significativa de los problemas de programación. Este tipo de errores son difíciles de localizar, y lo mismo cabría decir de las tareas de limpieza inapropiadas. Puesto que los constructores nos permiten garantizar una inicialización y limpieza adecuadas (el compilador no permitirá crear un objeto sin las apropiadas llamadas a un constructor), la seguridad y el control están garantizados.

En C++, la destrucción también es muy importante, porque los objetos creados con **new** deben ser destruidos explicitamente. En Java, el depurador de memoria libera automáticamente la memoria de los objetos que no son necesarios, por lo que el método de limpieza equivalente en Java no es necesario en muchas ocasiones (pero cuando lo es, es preciso implementarlo explícitamente). En aquellos casos donde no se necesite un comportamiento similar al de los destructores, el mecanismo de depuración de memoria de Java simplifica enormemente la programación y mejora también en gran medida la

seguridad de la gestión de memoria. Algunos depuradores de memoria pueden incluso limpiar otros recursos, como los recursos gráficos y los descriptores de archivos. Sin embargo, el depurador de memoria hace que se incremente el coste de ejecución, resultando difícil evaluar adecuadamente ese coste, debido a la lentitud que históricamente han tenido los intérpretes de Java. Aunque a lo largo del tiempo se ha mejorado significativamente la velocidad de Java, un problema de la velocidad ha supuesto un obstáculo a la hora de adoptar este lenguaje en ciertos tipos de problemas de programación.

Debido a que está garantizado que todos los objetos se construyan, los constructores son más complejos de lo que aquí hemos mencionado. En particular, cuando se crean nuevas clases utilizando los mecanismos de *composición* o de *herencia*, también se mantiene la garantía de construcción, siendo necesaria una cierta sintaxis adicional para soportar este mecanismo. Hablaremos de la composición, de la herencia y del efecto que ambos mecanismos tienen en los constructores en próximos capítulos.

Puedes encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, que está disponible para la venta en www.MindView.net.

Control de acceso

6

El control de acceso (u ocultación de la implementación) trata acerca de “que no salgan las cosas a la primera”.

Todos los buenos escritores, incluyendo aquellos que escriben software, saben que un cierto trabajo no está terminado hasta después de haber sido reescrito, a menudo muchas veces. Si dejamos un fragmento de código encima de la mesa durante un tiempo y luego volvemos a él, lo más probable es que veamos una forma mucho mejor de escribirlo. Ésta es una de las principales motivaciones para el trabajo de *rediseño*, que consiste en reescribir código que ya funciona con el fin de hacerlo más legible, comprensible y, por tanto, mantenable.¹

Sin embargo, existe una cierta tensión en este deseo de modificar y mejorar el código. A menudo, existen consumidores (*programadores de cliente*) que dependen de que ciertos aspectos de nuestro código continúen siendo iguales. Por tanto, nosotros queremos modificar el código, pero ellos quieren que siga siendo igual. Es por eso que una de las principales consideraciones en el diseño orientado a objetos es la de “separar las cosas que cambian de las cosas que permanecen”.

Esto es particularmente importante para las bibliotecas. Los consumidores de una biblioteca deben poder confiar en el elemento que están utilizando, y saber que no necesitarán reescribir el código si se publica una nueva versión de la biblioteca. Por otro lado, el creador de la biblioteca debe tener la libertad de realizar modificaciones y mejoras, con la confianza de que el código del cliente no se verá afectado por esos cambios.

Estos objetivos pueden conseguirse adoptando el convenio adecuado. Por ejemplo, el programador de la biblioteca debe aceptar no eliminar los métodos existentes a la hora de modificar una clase de la biblioteca, ya que eso haría que dejara de funcionar el código del programador de clientes. Sin embargo, la situación inversa es un poco más compleja de resolver. En el caso de un campo, ¿cómo puede saber el creador de la biblioteca a qué campos han accedido los programadores de clientes? Lo mismo cabe decir de los métodos que sólo forman parte de la implementación de una clase y que no están para ser usados directamente por el programador de clientes. ¿Qué pasa si el creador de la biblioteca quiere deshacerse de una implementación anterior y sustituirla por una nueva? Si se modifica alguno de esos miembros, podría dejar de funcionar el código de algún programa cliente. Por tanto, el creador de la biblioteca tiene las manos atadas y no puede modificar nada.

Para resolver este problema, Java proporciona *especificadores de acceso* que permiten al creador de la biblioteca decir qué cosas están disponibles para el programa cliente y qué cosas no lo están. Los niveles de control de acceso, ordenados de mayor a menor acceso, son **public**, **protected**, acceso de paquete (que no tienen una palabra clave asociada) y **private**. Leyendo el párrafo anterior, podríamos pensar que, como diseñadores de bibliotecas, conviene mantener todas las cosas lo más “privadas” posible y exponer sólo aquellos métodos que queramos que el programa cliente utilice. Esto es cierto, aunque a menudo resulta antinatural para aquellas personas acostumbradas a programar en otros lenguajes (especialmente C) y que están acostumbradas a acceder a todo sin ninguna restricción. Cuando lleguen al final del capítulo, estas personas estarán convencidas de la utilidad de los controles de acceso en Java.

Sin embargo, el concepto de biblioteca de componentes y el control acerca de quién puede acceder a los componentes de esa biblioteca no es completo. Sigue quedando pendiente la cuestión de cómo empaquetar los componentes para formar

¹ Consulte *Refactoring: Improving the Design of Existing Code*, de Martin Fowler, et al. (Addison-Wesley, 1999). Ocasionalmente, algunas personas argumentarán en contra de las tareas de rediseño, sugiriendo que un código que ya funciona es perfectamente adecuado, por lo que resulta una pérdida de tiempo tratar de rediseñarlo. El problema con esta forma de pensar es que la parte del león en lo que se refiere al tiempo y al dinero consumidos por un proyecto no está en la escritura inicial del código, sino en su mantenimiento. Hacer el código más fácil de entender permite ahorrar una gran cantidad de dinero.

una unidad de biblioteca cohesionada. Este aspecto se controla mediante la palabra clave **package** en Java, y los especificadores de acceso se verán afectados por el hecho de que una clase se encuentra en el mismo paquete o en otro paquete distinto. Por tanto, para comenzar este capítulo, veamos primero cómo se incluyen componentes de biblioteca en los paquetes. Con eso, seremos capaces de entender completamente el significado de los especificadores de acceso.

package: la unidad de biblioteca

Un paquete contiene un grupo de clases, organizadas conjuntamente dentro de un mismo *espacio de nombres*.

Por ejemplo, existe una biblioteca de utilidad que forma parte de la distribución estándar de Java, organizada bajo el espacio de nombres **java.util**. Una de las clases de **java.util** se denomina **ArrayList**. Una forma de utilizar un objeto **ArrayList** consiste en especificar el nombre completo **java.util.ArrayList**.

```
//: access/FullQualification.java

public class FullQualification {
    public static void main(String[] args) {
        java.util.ArrayList list = new java.util.ArrayList();
    }
} //:-
```

Sin embargo, este procedimiento se vuelve rápidamente tedioso, por lo que suele ser más cómodo utilizar en su lugar la palabra clave **import**. Si queremos importar una única clase, podemos indicar esa clase en la instrucción **import**:

```
//: access/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
    public static void main(String[] args) {
        ArrayList list = new java.util.ArrayList();
    }
} //:-
```

Ahora podemos usar **ArrayList** sin ningún cualificador. Sin embargo, no tendremos a nuestra disposición ninguna de las otras clases de **java.util**. Para importar todas, basta con utilizar '*' tal como hemos visto en los ejemplos del libro.

```
import java.util.*;
```

La razón para efectuar estas importaciones es proporcionar un mecanismo para gestionar los espacios de nombres. Los nombres de todos los miembros de las clases están aislados de las clases restantes. Un método **f()** de la clase **A** no coincidirá con un método **f()** que tenga la misma firma en la clase **B**. ¿Pero qué sucede con los nombres de las clases? Suponga que creamos una clase **Stack** en una máquina que ya disponga de otra clase **Stack** escrita por alguna otra persona. Esta posibilidad de colisión de los nombres es la que hace que sea tan importante disponer de un control completo de los espacios de nombres en Java, para poder crear una combinación de identificadores unívoca para cada clase.

La mayoría de los ejemplos que hemos visto hasta ahora en el libro se almacenaban en un único archivo y habían sido diseñados para uso local, por lo que no nos hemos preocupado de los nombres de paquete. Lo cierto es que estos ejemplos sí estaban incluidos en paquetes: el *paquete predeterminado* o “innominado”. Ciertamente, ésta es una opción viable y trataremos de utilizarla siempre que sea posible en el resto del libro, en aras de la simplicidad. Sin embargo, si lo que pretendemos es crear bibliotecas o programas que puedan cooperar con otros programas Java que estén en la misma máquina, deberemos tener en cuenta que hay que evitar las posibles colisiones entre nombres de clases.

Cuando se crea un archivo de código fuente para Java, normalmente se le denomina *unidad de compilación* (y también, en ocasiones, *unidad de traducción*). Cada unidad de compilación debe tener un nombre que termine en **.java**, y dentro de la unidad de compilación puede haber una clase **public** que debe tener el mismo nombre del archivo (incluyendo el uso de mayúsculas y minúsculas, pero excluyendo la extensión **.java** del nombre del archivo). Sólo puede haber *una* clase **public** en cada unidad de compilación; en caso contrario, el compilador se quejará. Si existen clases adicionales en esa unidad de compilación, estarán ocultas para el mundo exterior al paquete, porque *no son public*, y simplemente se tratará de clases “soporte” para la clase **public** principal.

Organización del código

Cuando se compila un archivo **.java**, se obtiene un archivo de salida para cada clase del archivo **.java**. Cada archivo de salida tiene el nombre de una de las clases del archivo **.java**, pero con la extensión **.class**. De este modo, podemos llegar a obtener un gran número de archivos **.class** a partir de un número pequeño de archivos **.java**. Si el lector ha programado anteriormente en algún lenguaje compilado, estará acostumbrado al hecho de que el compilador genere algún formato intermedio (normalmente un archivo "obj") que luego se empaqueta con otros del mismo tipo utilizando un montador (para crear un archivo ejecutable) o un gestor de biblioteca (para crear una biblioteca). Ésta no es la forma de funcionar de Java. Un programa funcional es un conjunto de archivos **.class**, que se puede empaquetar y comprimir en un archivo JAR (Java ARchive), utilizando el archivador **jar** de Java. El intérprete de Java es responsable de localizar, cargar e interpretar² estos archivos.

Una biblioteca es un grupo de estos archivos de clase. Cada archivo fuente suele tener una clase **public** y un número arbitrario de clases no públicas, por lo que no sólo existe un componente **public** para cada archivo fuente. Si queremos especificar que todos estos componentes (cada uno con sus propios archivos **.java** y **.class** separados) deben agruparse, podemos utilizar la palabra clave **package**.

Si usamos una instrucción **package**, debe aparecer como la primera linea no de comentario en el archivo. Cuando escribimos:

```
package access;
```

estamos indicando que esta unidad de compilación es parte de una biblioteca denominada **access**. Dicho de otro modo, estamos especificando que el nombre de clase pública situado dentro de esta unidad de compilación debe integrarse bajo el "paraguas" correspondiente al nombre **access**, de modo que cualquiera que quiera usar ese nombre deberá especificarlo por completo o utilizar la palabra clave **import** en combinación con **access**, utilizando las opciones que ya hemos mencionado anteriormente (observe que el convenio que se emplea para los nombres de paquetes Java consiste en emplear letras minúsculas, incluso para las palabras intermedias).

Por ejemplo, suponga que el nombre de un archivo es **MyClass.java**. Esto quiere decir que sólo puede haber una clase **public** en dicho archivo y que el nombre de esa clase debe ser **MyClass** (respetando el uso de mayúsculas y minúsculas):

```
//: access/mypackage/MyClass.java
package access.mypackage;

public class MyClass {
    // ...
} //
```

Ahora, si alguien quiere utilizar **MyClass** o cualquiera otra de las clases públicas de **access**, deberá emplear la palabra clave **import** para que estén disponibles esos nombres definidos en el paquete **access**. La alternativa consiste en especificar el nombre completamente cualificado:

```
//: access/QualifiedMyClass.java

public class QualifiedMyClass {
    public static void main(String[] args) {
        access.mypackage.MyClass m =
            new access.mypackage.MyClass();
    }
} //
```

La palabra clave **import** permite que este ejemplo tenga un aspecto mucho más simple:

```
//: access/ImportedMyClass.java
import access.mypackage.*;
```

² No hay ninguna característica de Java que nos obligue a utilizar un intérprete. Existen compiladores Java de código nativo que generan un único archivo ejecutable.

```

public class Imported MyClass {
    public static void main(String[] args) {
        MyClass m = new MyClass();
    }
} //:-

```

Merece la pena tener presente que lo que las palabras clave **package** e **import** nos permiten hacer, como diseñadores de bibliotecas, es dividir el espacio de nombres global único, para que los nombres no colisionen, independientemente de cuántas personas se conecten a Internet y comiencen a escribir clases en Java.

Creación de nombres de paquete únicos

El lector se habrá percatado de que, dado que un paquete nunca estará realmente "empaquetado" en un solo archivo, podrá estar compuesto por muchos archivos **.class**, por lo que el sistema de archivos puede llegar a estar un tanto abarrotado. Para evitar el desorden, una medida lógica que podemos tomar sería colocar todos los archivos **.class** correspondientes a un paquete concreto dentro de un mismo directorio; es decir, aprovechar la estructura de archivos jerárquica del sistema operativo. Ésta es una de las formas mediante las que Java trata de evitar el problema de la excesiva acumulación de archivos; veremos esto de otra forma cuando más adelante presentemos la utilidad **jar**.

Recopilar los archivos de un paquete dentro de un mismo subdirectorío resuelve también otros dos problemas: la creación de nombres de paquete únicos y la localización de aquellas clases que puedan estar perdidas en algún lugar de la estructura de directorios. Esto se consigue codificando la ruta correspondiente a la ubicación del archivo **.class** dentro del nombre del paquete. Por convenio, la primera parte del nombre del paquete es el nombre de dominio Internet invertido del creador de la clase. Dado que está garantizado que los nombres de dominio Internet sean únicos, si seguimos este convenio nuestro nombre de paquete será único y nunca se producirá una colisión de nombres (es decir, salvo que perdamos el derecho a utilizar el nombre de dominio y la persona que lo comience a utilizar se dedique a escribir código Java con los mismos nombres de ruta que usted utilizó). Por supuesto, si no disponemos de nuestro propio nombre de dominio, deberemos concebir una combinación que resulte lo suficientemente improbable (como por ejemplo la combinación de nuestro nombre y apellidos) para crear nombres de paquete únicos. Si ha decidido comenzar a publicar código Java, merece la pena que haga un pequeño esfuerzo para obtener un nombre de dominio.

La segunda parte de esta solución consiste en establecer la correspondencia entre los nombres de paquete y los directorios de la máquina, de modo que cuando el programa Java se ejecute y necesite cargar el archivo **.class**, pueda localizar el directorio en el que ese archivo **.class** resida.

El intérprete Java actúa de la forma siguiente. Primero, localiza la variable de entorno **CLASSPATH**³ (que se fija a través del sistema operativo y en ocasiones es definida por el programa de instalación que instala Java con una herramienta basada en Java en la máquina). **CLASSPATH** contiene uno o más directorios que se utilizan como raíces para buscar los archivos **.class**. Comenzando por esa raíz, el intérprete toma el nombre de paquete y sustituye cada punto por una barra inclinada para generar un nombre de ruta a partir de la raíz **CLASSPATH** (por lo que el paquete **package foo.bar.baz** se convertiría en **foo\bar\baz** o **foo/bar/baz** o, posiblemente, en alguna otra cosa, dependiendo del sistema operativo). Esto se concatena a continuación con las diversas entradas que se encuentren en la variable **CLASSPATH**. Será en ese subdirectorío donde el intérprete busque el archivo **.class** que tenga un nombre que se corresponda con la clase que se esté intentando crear (también busca en algunos directorios estándar relativos al lugar en el que reside el intérprete Java).

Para comprender esto, considere por ejemplo mi nombre de dominio, que es **MindView.net**. Invertiendo éste y pasándolo a minúsculas, **net.mindview** establece mi nombre global único para mis clases (antiguamente, las extensiones **com**, **edu**, **org**, etc., estaban en mayúsculas en los paquetes Java, pero esto se modificó en Java 2 para que todo el nombre del paquete estuviera en minúsculas). Puedo subdividir este espacio de nombres todavía más creando, por ejemplo, una biblioteca denominada **simple**, por lo que tendré un nombre de paquete que será:

```
package net.mindview.simple;
```

Ahora, este nombre de paquete puede utilizarse como espacio de nombres paraguas para los siguientes dos archivos:

³ Cuando nos refiramos a la variable de entorno, utilizaremos letras mayúsculas (**CLASSPATH**).

```
//: net/mindview/simple/Vector.java
// Creación de un paquete.
package net.mindview.simple;

public class Vector {
    public Vector() {
        System.out.println("net.mindview.simple.Vector");
    }
} /*:-
```

Como hemos mencionado antes, la instrucción **package** debe ser la primera línea de no comentario dentro del código del archivo. El segundo archivo tiene un aspecto parecido:

```
//: net/mindview/simple/List.java
// Creación de un paquete.
package net.mindview.simple;

public class List {
    public List() {
        System.out.println("net.mindview.simple.List");
    }
} /*:-
```

Ambos archivos se ubicarán en el siguiente subdirectorio de mi sistema:

```
C:\DOC\JavaT\net\mindview\simple
```

Observe que la primera línea de comentario en cada archivo del libro indica la ubicación del directorio donde se encuentra ese archivo dentro del árbol del código fuente; esto se usa para la herramienta automática de extracción de código que he empleado con el libro.

Si examinamos esta ruta, podemos ver el nombre del paquete **net.mindview.simple**, pero ¿qué pasa con la primera parte de la ruta? De esa parte se encarga la variable de entorno **CLASSPATH**, que en mi máquina es:

```
CLASSPATH=. ;D:\JAVA\LIB;C:\DOC\JavaT
```

Podemos ver que la variable de entorno **CLASSPATH** puede contener una serie de rutas de búsqueda alternativas.

Sin embargo, existe una variación cuando se usan archivos JAR. Es necesario poner el nombre real del archivo JAR en la variable de ruta, y no simplemente la ruta donde está ubicado. Así, para un archivo JAR denominado **grape.jar**, la variable de ruta incluiría:

```
CLASSPATH=. ;D:\JAVA\LIB;C:\flavors\grape.jar
```

Una vez que la variable de ruta de búsqueda se ha configurado apropiadamente, el siguiente archivo puede ubicarse en cualquier directorio:

```
//: access/LibTest.java
// Utiliza la biblioteca.
import net.mindview.simple.*;

public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
    }
} /* Output:
net.mindview.simple.Vector
net.mindview.simple.List
*/*:-
```

Cuando el compilador se encuentra con la instrucción **import** correspondiente a la biblioteca **simple**, comienza a explorar todos los directorios especificados por **CLASSPATH**, en busca del subdirectorio **net/mindview/simple**, y luego busca los archivos compilados con los nombres apropiados (**Vector.class** para **Vector** y **List.class** para **List**). Observe que tanto las dos clases como los métodos deseados de **Vector** y **List** tienen que ser de tipo **public**.

La configuración de CLASSPATH resultaba tan enigmática para los usuarios de Java inexpertos (al menos lo era para mí cuando comencé con el lenguaje) que Sun ha hecho que en el kit JDK de las versiones más recientes de Java se comporte de forma algo más inteligente. Se encontrará, cuando lo instale, que aunque no configure la variable CLASSPATH, podrá compilar y ejecutar programas Java básicos. Sin embargo, para compilar y ejecutar el paquete de código fuente de este libro (disponible en www.MindView.net), necesitará añadir a la variable CLASSPATH el directorio base del árbol de código.

Ejercicio 1: (1) Cree una clase dentro de un paquete. Cree una instancia de esa clase fuera de dicho paquete.

Colisiones

¿Qué sucede si se importan dos bibliotecas mediante '*' y ambas incluyen los mismos nombres? Por ejemplo, suponga que un programa hace esto:

```
import net.mindview.simple.*;
import java.util.*;
```

Puesto que **java.util.*** también contiene una clase **Vector**, esto provocaría una potencial colisión. Sin embargo, mientras que no lleguemos a escribir el código que provoque en efecto la colisión, no pasa nada. Resulta bastante conveniente que esto sea así, ya que de otro modo nos veríamos forzados a escribir un montón de cosas para evitar colisiones que realmente nunca iban a suceder.

La colisión *sí* que se producirá si ahora intentamos construir un **Vector**:

```
Vector v = new Vector();
```

¿A qué clase **Vector** se refiere esta línea? El compilador no puede saberlo, como tampoco puede saberlo el lector. Así que el compilador generará un error y nos obligará a ser más explícitos. Si queremos utilizar el **Vector** Java estándar, por ejemplo, deberemos escribir:

```
java.util.Vector v = new java.util.Vector();
```

Puesto que esto (junto con la variable CLASSPATH) especifica completamente la ubicación de la clase **Vector** deseada, no existirá en realidad ninguna necesidad de emplear la instrucción **import java.util.***, a menos que vayamos a utilizar alguna otra clase definida en **java.util**.

Alternativamente, podemos utilizar la instrucción de importación de una única clase para prevenir las colisiones, siempre y cuando no empleemos los dos nombres que entran en colisión dentro de un mismo programa (en cuyo caso, no tendremos más remedio que especificar completamente los nombres).

Ejercicio 2: (1) Tome los fragmentos de código de esta sección y transfórmelos en un programa para verificar que se producen las colisiones que hemos mencionado.

Una biblioteca personalizada de herramientas

Armados con este conocimiento, ahora podemos crear nuestras propias bibliotecas de herramientas, para reducir o eliminar la escritura de código duplicado. Considere, por ejemplo, el alias que hemos estado utilizando para **System.out.println()**, con el fin de reducir la cantidad de información tecleada. Esto puede ser parte de una clase denominada **Print**, de modo que dispondriamos de una instrucción estática de impresión bastante más legible:

```
//: net/mindview/util/Print.java
// Métodos de impresión que pueden usarse sin
// cualificadores, empleando importaciones estáticas de Java SES:
package net.mindview.util;
import java.io.*;

public class Print {
    // Imprimir con una nueva línea:
    public static void print(Object obj) {
        System.out.println(obj);
    }
    // Imprimir una nueva línea sola:
}
```

```

public static void print() {
    System.out.println();
}
// Imprimir sin salto de línea:
public static void printnb(Object obj) {
    System.out.print(obj);
}
// Si nuevo printf() de Java SE5 (de C):
public static PrintStream
printf(String format, Object... args) {
    return System.out.printf(format, args);
}
*/:-
```

Podemos utilizar estas abreviaturas de impresión para imprimir cualquier cosa, bien con la inserción de una nueva línea (`print()`) o sin una nueva línea (`printnb()`).

Como habrá adivinado, este archivo deberá estar ubicado en un directorio que comience en una de las ubicaciones definidas en CLASSPATH y que luego continúe con `net/mindview`. Después de compilar, los métodos `static print()` y `printnb()` pueden emplearse en cualquier lugar del sistema utilizando una instrucción `import static`:

```

//: access/PrintTest.java
// Usa los métodos estáticos de impresión de Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
    public static void main(String[] args) {
        print("Available from now on!");
        print(100);
        print(100L);
        print(3.14159);
    }
} /* Output:
Available from now on!
100
100
3.14159
*/:-
```

Un segundo componente de esta biblioteca pueden ser los métodos `range()`, que hemos presentado en el Capítulo 4, *Control de la ejecución*, y que permiten el uso de la sintaxis `foreach` para secuencias simples de enteros:

```

//: net/mindview/util/Range.java
// Métodos de creación de matrices que se pueden usar sin
// cualificadores, con importaciones estáticas Java SE5;
package net.mindview.util;

public class Range {
    // Generar una secuencia [0..n]
    public static int[] range(int n) {
        int[] result = new int[n];
        for(int i = 0; i < n; i++)
            result[i] = i;
        return result;
    }
    // Generar una secuencia [start..end]
    public static int[] range(int start, int end) {
        int sz = end - start;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + i;
```

```

        return result;
    }
    // Generar una secuencia [start..end] con incremento igual a step
    public static int[] range(int start, int end, int step) {
        int sz = (end - start)/step;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + (i * step);
        return result;
    }
} //:-

```

A partir de ahora, cuando desarrolle cualquier nueva utilidad que le parezca interesante la podrá añadir a su propia biblioteca. A lo largo del libro podrá ver que cómo añadiremos más componentes a la biblioteca **net.mindview.util**.

Utilización de importaciones para modificar el comportamiento

Una característica que se echa en falta en Java es la *compilación condicional* que existe en C y que permite cambiar una variable indicadora y obtener un comportamiento diferente sin variar ninguna otra parte del código. La razón por la que dicha característica no se ha incorporado a Java es, probablemente, porque la mayor parte de las veces se utiliza en C para resolver los problemas interplataforma: dependiendo de la plataforma de destino se compilan diferentes partes del código. Puesto que Java está pensado para ser automáticamente un lenguaje interplataforma no debería ser necesaria.

Sin embargo, existen otras aplicaciones interesantes de la compilación condicional. Un uso bastante común es durante la depuración del código. Las características de depuración se activan durante el desarrollo y se desactivan en el momento de lanzar el producto. Podemos conseguir el mismo efecto modificando el paquete que se importe dentro de nuestro programa, con el fin de conmutar entre el código utilizado en la versión de depuración y el empleado en la versión de producción. Esta misma técnica puede utilizarse para cualquier código de tipo condicional.

Ejercicio 3: (2) Cree dos paquetes: **debug** y **debugoff**, que contengan una clase idéntica con un método **debug()**. La primera versión debe mostrar su argumento **String** en la consola, mientras que la segunda no debe hacer nada. Utilice una línea **static import** para importar la clase en un programa de prueba y demuestre el efecto de la compilación condicional.

Un consejo sobre los nombres de paquete

Merece la pena recordar que cada vez que creamos un paquete, estamos especificando implicitamente una estructura de directorio en el momento de dar al paquete un nombre. El paquete *debe estar* en el directorio indicado por su nombre, que deberá ser un directorio alcanzable a partir de la ruta indicada en CLASSPATH. Experimentar con la palabra clave **package** puede ser algo frustrante al principio, porque a menos que respetemos la regla que establece la correspondencia entre nombres de paquete y rutas de directorio, obtendremos un montón de misteriosos mensajes en tiempo de ejecución que nos dicen que el sistema no puede encontrar una clase concreta, incluso aunque esa clase esté ahí en el mismo directorio. Si obtiene un mensaje como éste, desactive mediante un comentario la instrucción **package** y compruebe si el programa funciona, si lo hace, ya sabe dónde está el problema.

Observe que el código compilado se coloca a menudo en un directorio distinto de aquel en el que reside el código fuente, pero la ruta al código compilado deberá seguir siendo localizable por la JVM utilizando la variable CLASSPATH.

Especificadores de acceso Java

Los especificadores de acceso Java **public**, **protected** y **private** se colocan delante de cada definición de cada miembro de la clase, ya sea éste un campo o un método. Cada especificador de acceso sólo controla el acceso para esa definición concreta.

Si no proporciona un especificador de acceso, querrá decir que ese miembro tiene "acceso de paquete". Por tanto, de una forma u otra, todo tiene asociado algún tipo de control de acceso. En las secciones siguientes, vamos a analizar los diversos tipos de acceso.

Acceso de paquete

En los ejemplos de los capítulos anteriores no hemos utilizado especificadores de acceso. El acceso predeterminado no tiene asociada ninguna palabra clave, pero comúnmente se hace referencia a él como *acceso de paquete* (y también, en ocasiones, "acceso amigable"). Este tipo de acceso significa que todas las demás clases del paquete actual tendrán acceso a ese miembro, pero para las clases situadas fuera del paquete ese miembro aparecerá como **private**. Puesto que cada unidad de compilación (cada archivo) sólo puede pertenecer a un mismo paquete, todas las clases dentro de una misma unidad de compilación estarán automáticamente disponibles para las otras mediante el acceso de paquete.

El acceso de paquete nos permite agrupar en un mismo paquete una serie de clases relacionadas para que puedan interactuar fácilmente entre sí. Cuando se colocan las clases juntas en un paquete, garantizando así el acceso mutuo a sus miembros definidos con acceso de paquete, estamos en cierto modo garantizando que el código de ese paquete sea "propiedad" nuestra. Resulta bastante lógico que sólo el código que sea de nuestra propiedad disponga de acceso de paquete al resto del código que nos pertenezca. En cierto modo, podríamos decir que el acceso de paquete hace que tenga sentido el agrupar las clases dentro de un paquete. En muchos lenguajes, la forma en que se hagan las definiciones en los archivos puede ser arbitraria, pero en Java nos vemos impelidos a organizarlas de una forma lógica. Además, podemos aprovechar la definición del paquete para excluir aquellas clases que no deban tener acceso a las clases que se definen en el paquete actual.

Cada clase se encarga de controlar qué código tiene acceso a sus miembros. El código de los restantes paquetes no puede presentarse sin más y esperar que le muestren los miembros **protected**, los miembros con acceso de paquete y los miembros **private** de una determinada clase. La única forma de conceder acceso a un miembro consiste en:

1. Hacer dicho miembro **public**. Entonces, todo el mundo podrá acceder a él.
2. Hacer que ese miembro tenga acceso de paquete, por el procedimiento de no incluir ningún especificador de acceso, y colocar las otras clases que deban acceder a él dentro del mismo paquete. Entonces, las restantes clases del paquete podrán acceder a ese miembro.
3. Como veremos en el Capítulo 7, *Reutilización de clases*, cuando se introduce la herencia, una clase heredada puede acceder tanto a los miembros **protected** como a los miembros **public** (pero no a los miembros **private**). Esa clase podrá acceder a los miembros con acceso de paquete sólo si las dos clases se encuentran en el mismo paquete. Pero, por el momento, vamos a olvidarnos de los temas de herencia y del especificador de acceso **protected**.
4. Proporcionar métodos "de acceso/mutadores" (también denominados métodos "get/set") que permitan leer y cambiar el valor. Éste es el enfoque más civilizado en términos de programación orientada a objetos, y resulta fundamental en JavaBeans, como podrá ver en el Capítulo 22, *Interfaces gráficas de usuario*.

public: acceso de interfaz

Cuando se utiliza la palabra clave **public**, ésta quiere decir que la declaración de miembros situada inmediatamente a continuación suya está disponible para todo el mundo, y en particular para el programa cliente que utilice la biblioteca. Suponga que definimos un paquete **dessert** que contiene la siguiente unidad de compilación:

```
//: access/dessert/Cookie.java
// Crea una biblioteca.
package access.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void bite() { System.out.println("bite"); }
} //:-
```

Recuerde que el archivo de clase producido por **Cookie.java** debe residir en un subdirectorio denominado **dessert**, dentro de un directorio **access** (que hace referencia al Capítulo 6, *Control de acceso* de este libro) que a su vez deberá estar bajo uno de los directorios CLASSPATH. No cometa el error de pensar que Java siempre examinará el directorio actual como uno de los puntos de partida de su búsqueda. Si no ha incluido un '.' como una de las rutas dentro de CLASSPATH, Java no examinará ese directorio.

Si ahora creamos un programa que usa **Cookie**:

```
//: access/Dinner.java
// Usa la biblioteca.
import access.dessert.*;

public class Dinner {
    public static void main(String[] args) {
        Cookie x = new Cookie();
        //! x.bite(); // No puede acceder
    }
} /* Output:
Cookie constructor
*///:-
```

podemos crear un objeto **Cookie**, dado que su constructor es **public** y que la clase también es **public** (más adelante, profundizaremos más en el concepto de clase **public**). Sin embargo, el miembro **bite()** es inaccesible desde de **Dinner.java** ya que **bite()** sólo proporciona acceso dentro del paquete **dessert**, así que el compilador nos impedirá utilizarlo.

El paquete predeterminado

Puede que le sorprenda descubrir que el siguiente código sí que puede compilarse, a pesar de que parece que no cumple con las reglas:

```
//: access/Cake.java
// Accede a una clase en una unidad de compilación separada.

class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
} /* Output:
Pie.f()
*///:-
```

En un segundo archivo del mismo directorio tenemos:

```
//: access/Pie.java
// La otra clase.

class Pie {
    void f() { System.out.println("Pie.f()"); }
} //:-
```

Inicialmente, cabría pensar que estos dos archivos no tienen nada que ver entre si, a pesar de lo cual **Cake** es capaz de crear un objeto **Pie** y de invocar su método **f()** (observe que debe tener `.` en su variable CLASSPATH para que los archivos se compilen). Lo que parecería lógico es que **Pie** y **f()** tengan acceso de paquete y no estén, por tanto, disponibles para **Cake**. Es verdad que *tienen* acceso de paquete, esa parte de la suposición es correcta. Pero la razón por la que están disponibles en **Cake.java** es porque se encuentran en el mismo directorio y no tienen ningún nombre explícito de paquete. Java trata los archivos de este tipo como si fueran implicitamente parte del “paquete predeterminado” de ese directorio, y por tanto proporciona acceso de paquete a todos los restantes archivos situados en ese directorio.

private: ¡no lo toque!

La palabra clave **private** significa que nadie puede acceder a ese miembro salvo la propia clase que lo contiene, utilizando para el acceso los propios métodos de la clase. El resto de las clases del mismo paquete no puede acceder a los miembros privados, así que el efecto resultante es como si estuviéramos protegiendo a la clase contra nosotros mismos. Por otro lado, resulta bastante común que un paquete sea creado por varias personas que colaboran entre si, por lo que **private** permite modificar libremente ese miembro sin preocuparse de si afectará a otras clases del mismo paquete.

El acceso de paquete predeterminado proporciona a menudo un nivel adecuado de ocultación; recuerde que un miembro con acceso de paquete resulta inaccesible para todos los programas cliente que utilicen esa clase. Esto resulta bastante conveniente, ya que el acceso predeterminado es el que normalmente se utiliza (y el que se obtiene si nos olvidamos de añadir especificadores de control de acceso). Por tanto, lo que normalmente haremos será pensar qué miembros queremos definir explícitamente como públicos para que los utilicen los programas cliente; como resultado, uno tendería a pensar que la palabra clave **private** no se utiliza muy a menudo, ya que se pueden realizar los diseños sin ella. Sin embargo, la realidad es que el uso coherente de **private** tiene una gran importancia, especialmente en el caso de la programación multihilo (como veremos en el Capítulo 21, *Concurrencia*).

He aquí un ejemplo del uso de **private**:

```
//: access/IceCream.java
// Ilustra la palabra clave "private".

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        // Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} ///:-
```

Este ejemplo nos permite ver un caso en el que **private** resulta muy útil: queremos tener control sobre el modo en que se crea un objeto y evitar que nadie pueda acceder directamente a un constructor concreto (o a todos ellos). En el ejemplo anterior, no podemos crear un objeto **Sundae** a través de su constructor; en lugar de ello, tenemos que invocar el método **makeASundae()** para que se encargue de hacerlo por nosotros.⁴

Cualquier método del que estemos seguros de que sólo actúa como método “auxiliar” de la clase puede ser definido como privado para garantizar que no lo utilicemos accidentalmente en ningún otro lugar del paquete, lo que nos impediría modificar o eliminar el método. Definir un método como privado nos garantiza que podamos modificarlo libremente en el futuro.

Lo mismo sucede para los campos privados definidos dentro de una clase. A menos que tengamos que exponer la implementación subyacente (lo cual es bastante menos habitual de lo que podría pensarse), conviene definir todos los campos como privados. Sin embargo, el hecho de que una referencia a un objeto sea de tipo **private** en una clase no quiere decir que algún otro objeto no pueda tener una referencia de tipo **public** al mismo objeto (consulte los suplementos en línea del libro para conocer más detalles acerca de los problemas de los alias).

protected: acceso de herencia

Para poder comprender el especificador de acceso **protected**, es necesario que demos un salto hacia adelante. En primer lugar, debemos tener presente que no es necesario comprender esta sección para poder continuar leyendo el libro hasta llegar al capítulo dedicado a la herencia (el Capítulo 7, *Reutilización de clases*). Pero para ser exhaustivos, hemos incluido aquí una breve descripción y un ejemplo de uso de **protected**.

La palabra clave **protected** trata con un concepto denominado *herencia*, que toma una clase existente (a la que denominaremos *clase base*) y añade nuevos miembros a esa clase sin tocar la clase existente. También se puede modificar el comportamiento de los miembros existentes en la clase. Para heredar de una clase, tenemos que especificar que nuestra nueva clase amplía (**extends**) una clase existente, como en la siguiente línea:

```
class Foo extends Bar {
```

⁴ Existe otro efecto en este caso: puesto que el único constructor definido es el predeterminado y éste se ha definido como **private**, se impedirá que nadie herede esta clase (lo cual es un tema del que hablaremos más adelante).

El resto de la definición de la clase tiene el aspecto habitual.

Si creamos un nuevo paquete y heredamos de una clase situada en otro paquete, los únicos miembros a los que tendremos acceso son los miembros públicos del paquete original (por supuesto, si la herencia se realiza dentro del *mismo* paquete, se podrán manipular todos los miembros que tengan acceso de paquete). En ocasiones, el creador de la clase base puede tomar un miembro concreto y garantizar el acceso a las clases derivadas, pero no al mundo en general. Eso es precisamente lo que hace la palabra clave **protected**. Esta palabra clave también proporciona acceso de paquete, es decir, las restantes clases del mismo paquete podrán acceder a los elementos protegidos.

Si volvemos al archivo **Cookie.java**, la siguiente clase *no puede* invocar el miembro **bite()** que tiene acceso de paquete:

```
//: access/ChocolateChip.java
// No se puede usar un miembro con acceso de paquete desde otro paquete.
import access.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println("ChocolateChip constructor");
    }
    public void chomp() {
        //: bite(); // No se puede acceder a bite
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        x.chomp();
    }
} /* Output:
Cookie constructor
ChocolateChip constructor
*///:-
```

Uno de los aspectos interesantes de la herencia es que, si existe un método **bite()** en la clase **Cookie**, también existirá en todas las clases que hereden de **Cookie**. Pero como **bite()** tiene acceso de paquete y está situado en un paquete distinto, no estará disponible para nosotros en el nuevo paquete. Por supuesto, podríamos hacer que ese método fuera público, pero entonces todo el mundo tendría acceso y puede que no sea eso lo que queramos. Si modificamos la clase **Cookie** de la forma siguiente:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void bite() {
        System.out.println("bite");
    }
} ///:-
```

ahora **bite()** estará disponible para toda aquella clase que herede de **Cookie**:

```
//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
    public ChocolateChip2() {
        System.out.println("ChocolateChip2 constructor");
    }
    public void chomp() { bite(); } // Método protegido
```

```

public static void main(String[] args) {
    ChocolateChip2 x = new ChocolateChip2();
    x.chomp();
}
/* Output:
Cookie constructor
ChocolateChip2 constructor
bite
*///:-

```

Observe que, aunque `bite()` también tiene acceso de paquete, *no es* de tipo **public**.

Ejercicio 4: (2) Demuestre que los métodos protegidos (**protected**) tienen acceso de paquete pero no son públicos.

Ejercicio 5: (2) Cree una clase con campos y métodos de tipo **public**, **private**, **protected** y con acceso de paquete. Cree un objeto de esa clase y vea los tipos de mensajes de compilación que se obtienen cuando se intenta acceder a todos los miembros de la clase. Tenga en cuenta que las clases que se encuentran en el mismo directorio forman parte del paquete “predeterminado”.

Ejercicio 6: (1) Cree una clase con datos protegidos. Cree una segunda clase en el mismo archivo con un método que manipule los datos protegidos de la primera clase.

Interfaz e implementación

El mecanismo de control de acceso se denomina a menudo *ocultación de la implementación*. El envolver los datos y los métodos dentro de la clase, en combinación con el mecanismo de ocultación de la implementación se denomina a menudo *encapsulación*.⁵ El resultado es un tipo de datos con una serie de características y comportamientos.

El mecanismo de control de acceso levanta una serie de fronteras dentro de un tipo de datos por dos razones importantes. La primera es establecer qué es lo que los programas cliente pueden usar o no. Podemos entonces diseñar como queramos los mecanismos internos dentro de la clase, sin preocuparnos de que los programas cliente utilicen accidentalmente esos mecanismos internos como parte de la interfaz que deberían estar empleando.

Esto nos lleva directamente a la segunda de las razones, que consiste en separar la interfaz de la implementación. Si esa clase se utiliza dentro de un conjunto de programas, pero los programas cliente tan sólo pueden enviar mensajes a la interfaz pública, tendremos libertad para modificar cualquier cosa que no sea pública (es decir, los miembros con acceso de paquete, protegidos o privados) sin miedo de que el código cliente deje de funcionar.

Para que las cosas sean más claras, resulta conveniente a la hora de crear las clases, situar los miembros públicos al principio, seguidos de los miembros protegidos, los miembros con acceso de paquete y los miembros privados. La ventaja es que el usuario de la clase puede comenzar a leer desde el principio y ver en primer lugar lo que para él es lo más importante (los miembros de tipo **public**, que son aquellos a los que podrá accederse desde fuera del archivo), y dejar de leer en cuanto encuentre los miembros no públicos, que forman parte de la implementación interna.

```

//: access/OrganizedByAccess.java

public class OrganizedByAccess {
    public void pub1() { /* ... */ }
    public void pub2() { /* ... */ }
    public void pub3() { /* ... */ }
    private void priv1() { /* ... */ }
    private void priv2() { /* ... */ }
    private void priv3() { /* ... */ }
    private int i;
    /**
    */
} //://:-

```

⁵ Sin embargo, mucha gente utiliza la palabra encapsulación para referirse en exclusiva a la ocultación de la implementación.

Esto sólo facilita parcialmente la lectura, porque la interfaz y la implementación siguen estando mezcladas. En otras palabras, el lector seguirá pudiendo ver el código fuente (la implementación), porque está incluido en la clase. Además, el sistema de documentación basado en comentarios soportado por Javadoc no concede demasiada importancia a la legibilidad del código por parte de los programadores de clientes. El mostrar la interfaz al consumidor de una clase es, en realidad, trabajo del explorador de clases, que es una herramienta que se encarga de examinar todas las clases disponibles y mostrarnos lo que se puede hacer con ellas (es decir, qué miembros están disponibles) en una forma apropiada. En Java, visualizar la documentación del JDK con un explorador web nos proporciona el mismo resultado que si utilizáramos un explorador de clases.

Acceso de clase

En Java, los especificadores de acceso también pueden emplearse para determinar qué clases *de una biblioteca* estarán disponibles para los usuarios de esa biblioteca. Si queremos que una cierta clase esté disponible para un programa cliente, tendremos que utilizar la palabra clave **public** en la definición de la propia clase. Con esto se controla si los programas cliente pueden siquiera crear un objeto de esa clase.

Para controlar el acceso a una clase, el especificador debe situarse delante de la palabra clave **class**. Podemos escribir:

```
public class Widget {
```

Ahora, si el nombre de la biblioteca es **access**, cualquier programa cliente podrá acceder a **Widget** mediante la instrucción

```
import access.Widget;
```

O

```
import access.*;
```

Sin embargo, existen una serie de restricciones adicionales:

1. Sólo puede haber una clase **public** por cada unidad de compilación (archivo). La idea es que cada unidad de compilación tiene una única interfaz pública representada por esa clase pública. Además de esa clase, puede tener tantas clases de soporte con acceso de paquete como deseemos. Si tenemos más de una clase pública dentro de una unidad de compilación, el compilador generará un mensaje de error.
2. El nombre de la clase **public** debe corresponderse exactamente con el nombre del archivo que contiene dicha unidad de compilación, incluyendo el uso de mayúsculas y minúsculas. De modo que para **Widget**, el nombre del archivo deberá ser **Widget.java** y no **widet.java** ni **WIDGET.java**. De nuevo, obtendremos un error en tiempo de compilación si los nombres no concuerdan.
3. Resulta posible, aunque no muy normal, tener una unidad de compilación sin ninguna clase **public**. En este caso, podemos dar al archivo el nombre que queramos (aunque si lo denominamos de forma arbitraria confundiremos a las personas que tengan que leer y mantener el código).

¿Qué sucede si tenemos una clase dentro de **access** que sólo estamos empleando para llevar a cabo las tareas realizadas por **Widget** o alguna otra clase de tipo **public** de **access**? Puede que no queramos molestarnos en crear la documentación para el programador de clientes y que pensemos que algo más adelante quizás queramos modificar las cosas completamente, eliminando esa clase y sustituyéndola por otra. Para poder disponer de esta flexibilidad, necesitamos garantizar que ningún programa cliente dependa de nuestros detalles concretos de implementación que están ocultos dentro de **access**. Para conseguir esto, basta con no incluir la palabra clave **public** en la clase, en cuyo caso tendrá acceso de paquete (dicha clase sólo podrá ser usada dentro de ese paquete).

Ejercicio 7: (1) Cree una biblioteca usando los fragmentos de código con los que hemos descrito **access** y **Widget**. Cree un objeto **Widget** dentro de una clase que no forme parte del paquete **access**.

Cuando creamos una clase con acceso de paquete, sigue siendo conveniente definir los campos de esa clase como **private** (siempre deben hacerse los campos lo más privados posible), pero generalmente resulta razonable dar a los métodos el mismo tipo de acceso que a la clase (acceso de paquete). Puesto que una clase con acceso de paquete sólo se utiliza normalmente dentro del paquete, sólo hará falta definir como públicos los métodos de esa clase si nos vemos obligados a ello; además, en esos casos, el compilador ya se encargará de informarnos.

Observe que una clase no puede ser **private** (ya que eso haría que fuera inaccesible para todo el mundo salvo para la propia clase) ni **protected**.⁶ Así que sólo tenemos dos opciones para especificar el acceso a una clase: acceso de paquete o **public**. Si no queremos que nadie tenga acceso a la clase, podemos definir todos los constructores como privados, impidiendo que nadie cree un objeto de dicha clase salvo nosotros, que podremos hacerlo dentro de un miembro de tipo **static** de esa clase. He aquí un ejemplo:

```
//: access/Lunch.java
// Ilustra los especificadores de acceso a clases. Define una
// clase como privada con constructores privados:

class Soup1 {
    private Soup1() {}
    // (1) Permite la creación a través de un método estático:
    public static Soup1 makeSoup() {
        return new Soup1();
    }
}

class Soup2 {
    private Soup2() {}
    // (2) Crea un objeto estático y devuelve una referencia
    // cuando se solicita.(El patrón "Singleton"):
    private static Soup2 ps1 = new Soup2();
    public static Soup2 access() {
        return ps1;
    }
    public void f() {}
}

// Sólo se permite una clase pública por archivo:
public class Lunch {
    void testPrivate() {
        // ¡No se puede hacer! Constructor privado:
        //! Soup1 soup = new Soup1();
    }
    void testStatic() {
        Soup1 soup = Soup1.makeSoup();
    }
    void testSingleton() {
        Soup2.access().f();
    }
} ///:-
```

Hasta ahora, la mayoría de los métodos devolvían **void** o un tipo primitivo, por lo que la definición:

```
public static Soup1 makeSoup() {
    return new Soup1();
}
```

puede parecer algo confusa a primera vista. La palabra **Soup1** antes del nombre del método (**makeSoup**) dice lo que el método devuelve. Hasta ahora en el libro, esta palabra normalmente era **void**, lo que significa que no devuelve nada. Pero también podemos devolver una referencia a un objeto, que es lo que estamos haciendo aquí. Este método devuelve una referencia a un objeto de la clase **Soup1**.

Las clases **Soup1** y **Soup2** muestran cómo impedir la creación directa de objetos de una clase definiendo todos los constructores como privados. Recuerde que, si no creamos explícitamente al menos un constructor, se creará automáticamente el constructor predeterminado (un constructor sin argumentos). Escribiendo el constructor predeterminado, garantizamos

⁶ En realidad, una *clase interna* puede ser privada o protegida, pero se trata de un caso especial. Hablaremos de ello en el Capítulo 10, *Clases internas*.

que no sea escrito automáticamente. Definiéndolo como privado nadie podrá crear un objeto de esa clase. Pero entonces, ¿cómo podrá alguien usar esta clase? En el ejemplo anterior se muestran dos opciones. En **Soup1**, se crea un método **static** que crea un nuevo objeto **Soup1** y devuelve una referencia al mismo. Esto puede ser útil si queremos realizar algunas operaciones adicionales con el objeto **Soup1** antes de devolverlo, o si queremos llevar la cuenta de cuántos objetos **Soup1** se han creado (por ejemplo, para restringir el número total de objetos).

Soup2 utiliza lo que se denomina un *patrón de diseño*, de lo que se habla en *Thinking in Patterns (with Java)* en www.MindView.net. Este patrón concreto se denomina *Solitario (singleton)*, porque sólo permite crear un único objeto. El objeto de la clase **Soup2** se crea como un miembro **static private** de **Soup2**, por lo que existe un objeto y sólo uno, y no se puede acceder a él salvo a través del método público **access()**.

Como hemos mencionado anteriormente, si no utilizamos un especificador de acceso para una clase, ésta tendrá acceso de paquete. Esto significa que cualquier otra clase del paquete podrá crear un objeto de esa clase, pero no se podrán crear objetos de esa clase desde fuera del paquete (recuerde que todos los archivos de un mismo directorio que no tengan declaraciones **package** explícitas forman parte, implicitamente, del paquete predeterminado de dicho directorio). Sin embargo, si un miembro estático de esa clase es de tipo **public**, el programa cliente podrá seguir accediendo a ese miembro estático, aún cuando no podrá crear un objeto de esa clase.

Ejercicio 8: (4) Siguiendo la forma del ejemplo **Lunch.java**, cree una clase denominada **ConnectionManager** que gestione una matriz fija de objetos **Connection**. El programa cliente no debe poder crear explicitamente objetos **Connection**, sino que sólo debe poder obtenerlos a través de un método estático de **ConnectionManager**. Cuando **ConnectionManager** se quede sin objetos, devolverá una referencia **null**. Pruebe las clases con un programa **main()**.

Ejercicio 9: (2) Cree el siguiente archivo en el directorio **access/local** (dentro de su ruta CLASSPATH):

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
    public PackagedClass() {
        System.out.println("Creating a packaged class");
    }
}
```

A continuación cree el siguiente archivo en un directorio distinto de **access/local**:

```
// access/foreign/Foreign.java
package access.foreign;
import access.local.*;

public class Foreign {
    public static void main(String[] args) {
        PackagedClass pc = new PackagedClass();
    }
}
```

Explique por qué el compilador crea un error. ¿Se resolvería el error si la clase **Foreign** fuera parte del paquete **access.local**?

Resumen

En cualquier relación, resulta importante definir una serie de fronteras que sean respetadas por todos los participantes. Cuando se crea una biblioteca, se establece una relación con el usuario de esa biblioteca (el programador de clientes), que es un programador como nosotros, aunque lo que hace es utilizar la biblioteca para construir una aplicación u otra biblioteca de mayor tamaño.

Sin una serie de reglas, los programas cliente podrían hacer lo que quisieran con todos los miembros de una clase, aún cuando nosotros prefiriéramos que no manipularan algunos de los miembros. Todo estaría expuesto a ojos de todo el mundo.

En este capítulo hemos visto cómo se construyen bibliotecas de clases: en primer lugar, la forma en que se puede empaquetar un grupo de clases en una biblioteca y, en segundo lugar, la forma en que las clases pueden controlar el acceso a sus miembros.

Las estimaciones indican que los proyectos de programación en C empiezan a fallar en algún punto entre las 50.000 y 100.000 líneas de código, porque C tiene un único espacio de nombres y esos nombres empiezan a colisionar, obligando a realizar un esfuerzo adicional de gestión. En Java, la palabra clave **package**, el esquema de denominación de paquetes y la palabra clave **import** nos proporcionan un control completo sobre los nombres, por lo que puede evitarse fácilmente el problema de la colisión de nombres.

Existen dos razones para controlar el acceso a los miembros. La primera consiste en evitar que los usuarios puedan husmear en aquellas partes del código que no deben tocar. Esas partes son necesarias para la operación interna de la clase, pero no forman parte de la interfaz que el programa cliente necesita. Por tanto, definir los métodos y los campos como privados constituye un servicio para los programadores de clientes, porque así éstos pueden ver fácilmente qué cosas son importantes para ellos y qué cosas pueden ignorar. Simplifica su comprensión de la clase.

La segunda razón, todavía más importante, para definir mecanismos de control de acceso consiste en permitir al diseñador de bibliotecas modificar el funcionamiento interno de una clase sin preocuparse de si ello puede afectar a los programas cliente. Por ejemplo, podemos diseñar al principio una clase de una cierta manera y luego descubrir que una cierta reestructuración del código podría aumentar enormemente la velocidad. Si la interfaz y la implementación están claramente protegidas, podemos realizar las modificaciones sin forzar a los programadores de clientes a reescribir su código. Los mecanismos de control de acceso garantizan que ningún programa cliente dependa de la implementación subyacente de una clase.

Cuando disponemos de la capacidad de modificar la implementación subyacente, no sólo tenemos la libertad de mejorar nuestro diseño, sino también la libertad de cometer errores. Independientemente del cuidado que pongamos en la planificación y el diseño, los errores son inevitables. Saber que resulta relativamente seguro cometer esos errores significa que tendremos menos miedo a experimentar, que aprenderemos más rápidamente y que finalizaremos nuestro proyecto con mucha más antelación.

La interfaz pública de una clase es lo que el usuario puede ver, así que constituye la parte de la clase que más importancia tiene que definamos "correctamente" durante la fase de análisis del diseño. Pero incluso aquí existen posibilidades de modificación. Si no definimos correctamente la interfaz a la primera podemos *añadir* más métodos siempre y cuando no eliminemos ningún método que los programas cliente ya hayan utilizado en su código.

Observe que el mecanismo de control de acceso se centra en una relación (y en un tipo de comunicación) entre un creador de bibliotecas y los clientes externos de esas bibliotecas. Existen muchas situaciones en las que esta relación no está presente. Por ejemplo, imagine que todo el código que escribimos es para nosotros mismos o que estamos trabajando en estrecha relación con un pequeño grupo de personas y que todo lo que se escriba se va a incluir en un mismo paquete. En esas situaciones, el tipo de comunicación es diferente, y la rígida adhesión a una serie de reglas de acceso puede que no sea la solución óptima. En esos casos, el acceso predeterminado (de paquete) puede que sea perfectamente adecuado.

Las soluciones a los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Reutilización de clases

7

Una de las características más atractivas de Java es la posibilidad de reutilizar el código. Pero, para ser verdaderamente revolucionario es necesario ser capaz de hacer mucho más que simplemente copiar el código y modificarlo.

Ésta es la técnica que se ha estado utilizando en lenguajes procedimentales como C, y no ha funcionado muy bien. Como todo lo demás en Java, la solución que este lenguaje proporciona gira alrededor del concepto de clase. Reutilizando el código creamos nuevas clases, pero en lugar de crearlo partiendo de cero, usamos las clases existentes que alguien haya construido y depurado anteriormente.

El truco estriba en utilizar las clases sin modificar el código existente. En este capítulo vamos a ver dos formas de llevar esto a cabo. La primera de ellas es bastante simple. Nos limitamos a crear objetos de una clase ya existente dentro de una nueva clase. Esto se denomina *composición*, porque la nueva clase está compuesta de objetos de otras clases existentes. Con esto, simplemente estamos reutilizando la funcionalidad del código, no su forma.

La segunda técnica es más sutil. Se basa en crear una nueva clase como un *tipo de* una clase existente. Literalmente lo que hacemos es tomar la forma de la clase existente y añadirla código sin modificarla. Esta técnica se denomina *herencia*, y el compilador se encarga de realizar la mayor parte del trabajo. La herencia es una de las piedras angulares de la programación orientada a objetos y tiene una serie de implicaciones adicionales que analizaremos en el Capítulo 8, *Poliformismo*.

Resulta que buena parte de la sintaxis y el comportamiento son similares tanto en la composición como en la herencia (lo que tiene bastante sentido, ya que ambas son formas de construir nuevos tipos a partir de otros tipos existentes). En este capítulo, vamos a presentar ambos mecanismos de reutilización de código.

Sintaxis de la composición

Hemos utilizado el mecanismo de composición de forma bastante frecuente en el libro hasta este momento. Simplemente, basta con colocar referencias a objetos dentro de las clases. Por ejemplo, suponga que queremos construir un objeto que almacene varios objetos **String**, un par de primitivas y otro objeto de otra clase. Para los objetos no primitivos, lo que hacemos es colocar referencias dentro de la nueva clase, pero sin embargo las primitivas se definen directamente:

```
//: reusing/SprinklerSystem.java
// Composición para la reutilización de código.

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = "Constructed";
    }
    public String toString() { return s; }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
```

```

private WaterSource source = new WaterSource();
private int i;
private float f;
public String toString() {
    return
        "valve1 = " + valve1 + " " +
        "valve2 = " + valve2 + " " +
        "valve3 = " + valve3 + " " +
        "valve4 = " + valve4 + "\n" +
        "i = " + i + " " + "f = " + f + " " +
        "source = " + source;
}
public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
}
} /* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = Constructed
*///:-

```

Uno de los métodos definidos en ambas clases es especial: `toString()`. Todo objeto no primitivo tiene un método `toString()` que se invoca en aquellas situaciones especiales en las que el compilador quiere una cadena de caracteres `String` pero lo que tiene es un objeto. Así, en la expresión contenida en `SprinklerSystem.toString()`:

```
"source = " + source;
```

el compilador ve que estamos intentando añadir un objeto `String` ("source =") a un objeto `WaterSource`. Puesto que sólo podemos "añadir" un objeto `String` a otro objeto `String`, el compilador dice "voy a convertir `source` en un objeto `String` invocando `toString()`". Después de hacer esto, puede combinar las dos cadenas de caracteres y pasar el objeto `String` resultante a `System.out.println()` (o, de forma equivalente, a los métodos estáticos `print()` y `println()` que hemos definido en este libro). Siempre que queramos permitir este tipo de comportamiento dentro de una clase que creemos, nos bastará con escribir un método `toString()`.

Las primitivas que son campos de una clase se inicializan automáticamente con el valor cero, como hemos indicado en el Capítulo 2, *Todo es un objeto*. Pero las referencias a objetos se inicializan con el valor `null`, y si tratamos de invocar métodos para cualquiera de ellas obtendremos una excepción (un error en tiempo de ejecución). Afortunadamente, podemos imprimir una referencia `null` sin que se genere una excepción.

Tiene bastante sentido que el compilador no cree un objeto predeterminado para todas las referencias, porque eso obligaría a un gasto adicional de recursos completamente innecesarios en muchos casos. Si queremos inicializar las referencias, podemos hacerlo de las siguientes formas:

1. En el lugar en el que los objetos se definan. Esto significa que estarán siempre inicializados antes de que se invoque el constructor.
2. En el constructor correspondiente a esa clase.
3. Justo antes de donde se necesite utilizar el objeto. Esto se suele denominar *inicialización diferida*. Puede reducir el gasto adicional de procesamiento en aquellas situaciones en las que la creación de los objetos resulte muy cara y no sea necesario crear el objeto todas las veces.
4. Utilizando la técnica de *inicialización de instancia*.

El siguiente ejemplo ilustra las cuatro técnicas:

```

//: reusing/Bath.java
// Inicialización mediante constructor con composición.
import static net.mindview.util.Print.*;

class Soap {
    private String s;

```

```

Soap() {
    print("Soap()");
    s = "Constructed";
}
public String toString() { return s; }

}

public class Bath {
    private String // Inicialización en el punto de definición:
        s1 = "Happy",
        s2 = "Happy",
        s3, s4;
    private Soap castille;
    private int i;
    private float toy;
    public Bath() {
        print("Inside Bath()");
        s3 = "Joy";
        toy = 3.14f;
        castille = new Soap();
    }
    // Inicialización de instancia:
    { i = 47; }
    public String toString() {
        if(s4 == null) // Inicialización diferida:
            s4 = "Joy";
        return
            "s1 = " + s1 + "\n" +
            "s2 = " + s2 + "\n" +
            "s3 = " + s3 + "\n" +
            "s4 = " + s4 + "\n" +
            "i = " + i + "\n" +
            "toy = " + toy + "\n" +
            "castille = " + castille;
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        print(b);
    }
} /* Output:
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
*///:-

```

Observe que en el constructor **Bath**, se ejecuta una instrucción antes de que tenga lugar ninguna de las inicializaciones. Cuando no se realiza la inicialización en el punto de definición, sigue sin haber ninguna garantía de que se vaya a realizar la inicialización antes de enviar un mensaje a una referencia al objeto, salvo la inevitable excepción en tiempo de ejecución.

Cuando se invoca **toString()**, asigna un valor a **s4** de modo que todos los campos estarán apropiadamente inicializados en el momento de usarlos.

Ejercicio 1: (2) Cree una clase simple. Dentro de una segunda clase, defina una referencia a un objeto de la segunda clase. Utilice el mecanismo de inicialización diferida para instanciar este objeto.

Sintaxis de la herencia

La herencia es una parte esencial de Java (y todos los lenguajes orientados a objetos). En la práctica, siempre estamos utilizando la herencia cada vez que creamos una clase, porque a menos que heredemos explícitamente de alguna otra clase, estaremos heredando implícitamente de la clase raíz estándar **Object** de Java.

La sintaxis de composición es obvia, pero la herencia utiliza una sintaxis especial. Cuando heredamos, lo que hacemos es decir “esta nueva clase es similar a esa antigua clase”. Especificamos esto en el código situado antes de la llave de apertura del cuerpo de la clase, utilizando la palabra clave **extends** seguida del nombre de la *clase base*. Cuando hacemos esto, automáticamente obtenemos todos los campos y métodos de la clase base. He aquí una clase:

```
//: reusing/Detergent.java
// Sintaxis y propiedades de la herencia.
import static net.mindview.util.Print.*;

class Cleanser {
    private String s = "Cleanser";
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        print(x);
    }
}

public class Detergent extends Cleanser {
    // Cambio de un método:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Invocar versión de la clase base
    }
    // Añadir métodos a la interfaz:
    public void foam() { append(" foam()"); }
    // Probar la nueva clase:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        print(x);
        print("Testing base class:");
        Cleanser.main(args);
    }
} /* Output:
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Testing base class:
Cleanser dilute() apply() scrub()
*///:r-
```

Esto nos permite ilustrar una serie de características. En primer lugar, en el método **Cleanser.append()**, se concatenan cadenas de caracteres con **s** utilizando el operador **`+=`**, que es uno de los operadores, junto con **`+`**, que los diseñadores de Java “han sobrecargado” para que funcionen con cadenas de caracteres.

En segundo lugar, tanto **Cleanser** como **Detergent** contienen un método **main()**. Podemos crear un método **main()** para cada una de nuestras clases; esta técnica de colocar un método **main()** en cada clase permite probar fácilmente cada una de

ellas. Y no es necesario eliminar el método `main()` cuando hayamos terminado; podemos dejarlo para las pruebas posteriores.

Aún cuando tengamos muchas clases en un programa, sólo se invocará el método `main()` de la clase especificada en la línea de comandos. Por tanto, en este caso, cuando escribimos `java Detergent`, se invocará `Detergent.main()`. Pero también podemos escribir `java Cleanser` para invocar a `Cleanser.main()`, aún cuando `Cleanser` no sea una clase pública. Incluso aunque una clase tenga acceso de paquete, si el método `main()` es público, también se podrá acceder a él.

Aquí, podemos ver que `Detergent.main()` llama a `Cleanser.main()` explicitamente, pasándole los mismos argumentos de la línea de comandos (sin embargo, podríamos pasarle cualquier matriz de objetos `String`).

Es importante que todos los métodos de `Cleanser` sean públicos. Recuerde que, si no se indica ningún especificador de acceso, los miembros adoptarán de forma predeterminada el acceso de paquete, lo que sólo permite el acceso a los otros miembros del paquete. Por tanto, *dentro de este paquete*, cualquiera podría usar esos métodos si no hubiera ningún especificador de acceso. `Detergent`, por ejemplo, no tendría ningún problema. Sin embargo, si alguna clase de algún otro paquete fuera a heredar de `Cleanser`, sólo podría acceder a los miembros de tipo `public`. Así que, para permitir la herencia, como regla general deberemos definir todos los campos como `private` y todos los métodos como `public` (los miembros de tipo `protected` también permiten el acceso por parte de las clases derivadas; analizaremos este tema más adelante). Por supuesto, en algunos casos particulares será necesario hacer excepciones, pero esta directriz suele resultar bastante útil.

`Cleanser` dispone de una serie de métodos en su interfaz: `append()`, `dilute()`, `apply()`, `scrub()` y `toString()`. Puesto que `Detergent` *deriva de* `Cleanser` (gracias a la palabra clave `extends`), automáticamente obtendrá todos estos métodos como parte de su interfaz, aún cuando no los veamos explícitamente definidos en `Detergent`. Por tanto, podemos considerar la herencia como un modo de reutilizar la clase.

Como podemos ver en `scrub()`, es posible tomar un método que haya sido definido en la clase base y modificarlo. En este caso, puede también que queramos invocar el método de la clase base desde dentro de la nueva versión. Pero dentro de `scrub()`, no podemos simplemente invocar a `scrub()`, ya que eso produciría una llamada recursiva, que no es exactamente lo que queremos. Para resolver este problema, la palabra clave `super` de Java hace referencia a la “superclase” de la que ha heredado la clase actual. Por tanto, la expresión `super.scrub()` invoca a la versión de la clase base del método `scrub()`.

Cuando se hereda, no estamos limitados a utilizar los métodos de la clase base. También podemos añadir nuevos métodos a la clase derivada, de la misma forma que los añadiríamos a otra clase: definiéndolos. El método `foam()` es un ejemplo de esto.

En `Detergent.main()` podemos ver que, para un objeto `Detergent`, podemos invocar todos los métodos disponibles en `Cleanser` así como en `Detergent` (es decir, `foam()`).

Ejemplo 2: (2) Cree una nueva clase que herede de la clase `Detergent`. Sustituya el método `scrub()` y añada un nuevo método denominado `sterilize()`.

Inicialización de la clase base

Puesto que ahora tenemos dos clases (la clase base y la clase derivada) en lugar de una, puede resultar un poco confuso tratar de imaginar cuál es el objeto resultante generado por una clase derivada. Desde fuera, parece como si la nueva clase tuviera la misma interfaz que la clase base y, quizás algunos métodos y campos adicionales. Pero el mecanismo de herencia no se limita a copiar la interfaz de la clase base. Cuando se crea un objeto de la clase derivada, éste contiene en su interior un *subobjeto* de la clase base. Este subobjeto es idéntico a lo que tendríamos si hubiéramos creado directamente un objeto de la clase base. Lo que sucede, simplemente, es que, visto desde el exterior, el subobjeto de la clase base está envuelto por el objeto de la clase derivada.

Por supuesto, es esencial que el subobjeto de la clase base se inicialice correctamente, y sólo hay una forma de garantizar esto: realizar la inicialización en el constructor invocando al constructor de la clase base, que es quien tiene todos los conocimientos y todos los privilegios para llevar a cabo adecuadamente la inicialización de la clase base. Java inserta automáticamente llamadas al constructor de la clase base dentro del constructor de la clase derivada. El siguiente ejemplo muestra este mecanismo en acción con tres niveles de herencia:

```
//: reusing/Cartoon.java
// Llamadas a constructores durante la herencia.
```

```

import static net.mindview.util.Print.*;

class Art {
    Art() { print("Art constructor"); }
}

class Drawing extends Art {
    Drawing() { print("Drawing constructor"); }
}

public class Cartoon extends Drawing {
    public Cartoon() { print("Cartoon constructor"); }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} /* Output:
Art constructor
Drawing constructor
Cartoon constructor
*///:-
```

Como puede ver, la construcción tiene lugar desde la base hacia “afuera”, por lo que la clase base se inicializa antes de que los constructores de la clase derivada puedan acceder a ella. Incluso aunque no creáramos un constructor para **Cartoon()**, el compilador sintetizaría un constructor predeterminado que invocaría al constructor de la clase base.

Ejercicio 3: (2) Demuestre la afirmación anterior.

Ejercicio 4: (2) Demuestre que los constructores de la clase base (a) se invocan siempre y (b) se invocan antes que los constructores de la clase derivada.

Ejercicio 5: (1) Cree dos clases, **A** y **B**, con constructores predeterminados (listas de argumentos vacías) que impriman un mensaje informando de la construcción de cada objeto. Cree una nueva clase llamada **C** que herede de **A**, y cree un miembro de la clase **B** dentro de **C**. No cree un constructor para **C**. Cree un objeto de la clase **C** y observe los resultados.

Constructores con argumentos

El ejemplo anterior tiene constructores predeterminados; es decir, que no tienen argumentos. Resulta fácil para el compilador invocar estos constructores, porque no existe ninguna duda acerca de qué argumentos hay que pasar. Si no existe un constructor predeterminado en la clase base, o si se quiere invocar un constructor de la clase base que tenga argumentos, será necesario escribir explícitamente una llamada al constructor de la clase base utilizando la palabra clave **super** y la lista de argumentos apropiada:

```

//: reusing/Chess.java
// Herencia, constructores y argumentos.
import static net.mindview.util.Print.*;

class Game {
    Game(int i) {
        print("Game constructor");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        print("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
```

```

Chess() {
    super(11);
    print("Chess constructor");
}
public static void main(String[] args) {
    Chess x = new Chess();
}
} /* Output:
Game constructor
BoardGame constructor
Chess constructor
*///:i-

```

Si no se invoca el constructor de la clase base en **BoardGame()**, el compilador se quejará de que no puede localizar un constructor de la forma **Game()**. Además, la llamada al constructor de la clase base *debe* ser la primera cosa que se haga en el constructor de la clase derivada (el compilador se encargará de recordárselo si se olvida de ello).

Ejercicio 6: (1) Utilizando **Chess.java**, demuestre las afirmaciones del párrafo anterior.

Ejercicio 7: (1) Modifique el Ejercicio 5 de modo que A y B tengan constructores con argumentos en lugar de constructores predeterminados. Escriba un constructor para C que realice toda la inicialización dentro del constructor de C.

Ejercicio 8: (1) Cree una clase base que sólo tenga un constructor no predeterminado y una clase derivada que tenga un constructor predeterminado (sin argumentos) y otro no predeterminado. En los constructores de la clase derivada, invoque al constructor de la clase base.

Ejercicio 9: (2) Cree una clase denominada **Root** que contenga una instancia de cada una de las siguientes clases (que también deberá crear): **Component1**, **Component2** y **Component3**. Derive una clase **Stem** de **Root** que también contenga una instancia de cada “componente”. Todas las clases deben tener constructores predeterminados que impriman un mensaje relativo a la clase.

Ejercicio 10: (1) Modifique el ejercicio anterior de modo que cada clase sólo tenga constructores no predeterminados.

Delegación

Una tercera relación, que no está directamente soportada en Java, se denomina *delegación*. Se encuentra a caballo entre la herencia y la composición, por que lo que hacemos es incluir un objeto miembro en la clase que estemos construyendo (como en la composición), pero al mismo tiempo exponemos todos los métodos del objeto miembro en nuestra nueva clase (como en la herencia). Por ejemplo, una nave espacial (*spaceship*) necesita un módulo de control:

```

//: reusing/SpaceShipControls.java

public class SpaceShipControls {
    void up(int velocity) {}
    void down(int velocity) {}
    void left(int velocity) {}
    void right(int velocity) {}
    void forward(int velocity) {}
    void back(int velocity) {}
    void turboBoost() {}
} ///:i-

```

Una forma de construir la nave espacial consistiría en emplear el mecanismo de herencia:

```

//: reusing/SpaceShip.java

public class SpaceShip extends SpaceShipControls {
    private String name;
    public SpaceShip(String name) { this.name = name; }
    public String toString() { return name; }
}

```

```

public static void main(String[] args) {
    SpaceShip protector = new SpaceShip("NSEA Protector");
    protector.forward(100);
}
} //:-

```

Sin embargo, un objeto **SpaceShip** no es realmente "un tipo de" objeto **SpaceShipControls**, aún cuando podamos, por ejemplo, "decirle" al objeto **SpaceShip** que avance hacia adelante (`forward()`). Resulta más preciso decir que la nave espacial (el objeto **SpaceShip**) *contiene* un módulo de control (objeto **SpaceShipControls**), y que, al mismo tiempo, todos los métodos de **SpaceShipControls** deben quedar expuestos en el objeto **SpaceShip**. El mecanismo de delegación permite resolver este dilema:

```

//: reusing/SpaceShipDelegation.java

public class SpaceShipDelegation {
    private String name;
    private SpaceShipControls controls =
        new SpaceShipControls();
    public SpaceShipDelegation(String name) {
        this.name = name;
    }
    // Métodos delegados:
    public void back(int velocity) {
        controls.back(velocity);
    }
    public void down(int velocity) {
        controls.down(velocity);
    }
    public void forward(int velocity) {
        controls.forward(velocity);
    }
    public void left(int velocity) {
        controls.left(velocity);
    }
    public void right(int velocity) {
        controls.right(velocity);
    }
    public void turboBoost() {
        controls.turboBoost();
    }
    public void up(int velocity) {
        controls.up(velocity);
    }
    public static void main(String[] args) {
        SpaceShipDelegation protector =
            new SpaceShipDelegation("NSEA Protector");
        protector.forward(100);
    }
} //:-

```

Como puede ver los métodos se redirigen hacia el método **controls** subyacente, y la interfaz es, por tanto, la misma que con la herencia. Sin embargo, tenemos más control con la delegación, ya que podemos decidir proporcionar únicamente un subconjunto de los métodos contenidos en el objeto miembro.

Aunque el lenguaje Java no soporta directamente la delegación, las herramientas de desarrollo sí que suelen hacerlo. Por ejemplo, el ejemplo anterior se ha generado automáticamente utilizando el entorno integrado de desarrollo JetBrains Idea.

Ejercicio 11: (3) Modifique **Detergent.java** de modo que utilice el mecanismo de delegación.

Combinación de la composición y de la herencia

Resulta bastante común utilizar los mecanismos de composición y de herencia conjuntamente. El siguiente ejemplo muestra la creación de una clase más compleja utilizando tanto la herencia como la composición, junto con la necesaria inicialización mediante los constructores:

```
//: reusing/PlaceSetting.java
// Combinación de la composición y la herencia.
import static net.mindview.util.Print.*;

class Plate {
    Plate(int i) {
        print("Plate constructor");
    }
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        print("DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {
        print("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        print("Spoon constructor");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        print("Fork constructor");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        print("Knife constructor");
    }
}

// Una forma cultural de hacer algo:
class Custom {
    Custom(int i) {
        print("Custom constructor");
    }
}

public class PlaceSetting extends Custom {
    private Spoon sp;
```

```

private Fork frk;
private Knife kn;
private DinnerPlate pl;
public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    print("PlaceSetting constructor");
}
public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
}
/* Output:
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
*///:-
```

Aunque el compilador nos obliga a inicializar la clase base y requiere que lo hagamos al principio del constructor, no se asegura de que inicialicemos los objetos miembro, así que es preciso prestar atención a este detalle.

Resulta asombrosa la forma tan limpia en que las clases quedan separadas. No es necesario siquiera disponer del código fuente de los métodos para poder reutilizar ese código. Como mucho, nos basta con limitarnos a importar un paquete (esto es cierto tanto para la herencia como para la composición).

Cómo garantizar una limpieza apropiada

Java no tiene el concepto C++ de *destructor*, que es un método que se invoca automáticamente cuando se destruye un objeto. La razón es, probablemente, que en Java la práctica habitual es olvidarse de los objetos en lugar de destruirlos, permitiendo al depurador de memoria reclamar la memoria cuando sea necesario.

A menudo, esto resulta suficiente, pero hay veces en que la clase puede realizar determinadas actividades durante su tiempo de vida que obligan a realizar la limpieza. Como hemos mencionado en el Capítulo 5, *Inicialización y limpieza*, no podemos saber cuándo va a ser invocado el depurador de memoria, o ni siquiera si va a ser invocado. Por tanto, si queremos limpiar algo concreto relacionado con una clase, deberemos escribir explícitamente un método especial para hacerlo y asegurarnos de que el programador de clientes sepa que debe invocar dicho método. Además, como se describe en el Capítulo 12, *Tratamiento de errores con excepciones*, debemos protegernos frente a la aceleración de posibles excepciones incorporando dicha actividad de limpieza en una cláusula **finally**.

Considere un ejemplo de un sistema de diseño asistido por computadora que dibuje imágenes en la pantalla:

```

//: reusing/CADSystem.java
// Cómo garantizar una limpieza adecuada.
package reusing;
import static net.mindview.util.Print.*;

class Shape {
    Shape(int i) { print("Shape constructor"); }
    void dispose() { print("Shape dispose"); }
}
```

```

class Circle extends Shape {
    Circle(int i) {
        super(i);
        print("Drawing Circle");
    }
    void dispose() {
        print("Erasing Circle");
        super.dispose();
    }
}

class Triangle extends Shape {
    Triangle(int i) {
        super(i);
        print("Drawing Triangle");
    }
    void dispose() {
        print("Erasing Triangle");
        super.dispose();
    }
}

class Line extends Shape {
    private int start, end;
    Line(int start, int end) {
        super(start);
        this.start = start;
        this.end = end;
        print("Drawing Line: " + start + ", " + end);
    }
    void dispose() {
        print("Erasing Line: " + start + ", " + end);
        super.dispose();
    }
}

public class CADSystem extends Shape {
    private Circle c;
    private Triangle t;
    private Line[] lines = new Line[3];
    public CADSystem(int i) {
        super(i + 1);
        for(int j = 0; j < lines.length; j++)
            lines[j] = new Line(j, j*j);
        c = new Circle(1);
        t = new Triangle(1);
        print("Combined constructor");
    }
    public void dispose() {
        print("CADSystem.dispose()");
        // El orden de limpieza es el inverso
        // al orden de inicialización:
        t.dispose();
        c.dispose();
        for(int i = lines.length - 1; i >= 0; i--)
            lines[i].dispose();
        super.dispose();
    }
    public static void main(String[] args) {

```

```

CADSystem x = new CADSystem(47);
try {
    // Código y tratamiento de excepciones...
} finally {
    x.dispose();
}
} /* Output:
Shape constructor
Shape constructor
Drawing Line: 0, 0
Shape constructor
Drawing Line: 1, 1
Shape constructor
Drawing Line: 2, 4
Shape constructor
Drawing Circle
Shape constructor
Drawing Triangle
Combined constructor
CADSystem.dispose()
Erasing Triangle
Shape dispose
Erasing Circle
Shape dispose
Erasing Line: 2, 4
Shape dispose
Erasing Line: 1, 1
Shape dispose
Erasing Line: 0, 0
Shape dispose
Shape dispose
*///:-

```

Todo en este sistema es algún tipo de objeto **Shape** (que a su vez es un tipo de **Object**, puesto que hereda implicitamente de la clase raíz). Cada clase sustituye el método **dispose()** de **Shape**, además de invocar la versión de dicho método de la clase base utilizando **super**. Las clases **Shape** específicas, **Circle**, **Triangle** y **Line**, tienen constructores que “dibujan” las formas geométricas correspondientes, aunque cualquier método que se invoque durante la vida del objeto puede llegar a ser responsable de hacer algo que luego requiera una cierta tarea de limpieza. Cada clase tiene su propio método **dispose()** para restaurar todas esas cosas que no están relacionadas con la memoria y dejarlas en el estado en que estaban antes de que el objeto se creara.

En **main()**, hay dos palabras clave que no habíamos visto antes y que no van a explicarse en detalle hasta el Capítulo 12, *Tratamiento de errores mediante excepciones*: **try** y **finally**. La palabra clave **try** indica que el bloque situado a continuación suyo (delimitado por llaves) es una *región protegida*, lo que quiere decir que se la otorga un tratamiento especial. Uno de estos tratamientos especiales consiste en que el código de la cláusula **finally** situada a continuación de esta región protegida *siempre* se ejecuta, independientemente de cómo se salga de bloque **try** (con el tratamiento de excepciones, es posible salir de un bloque **try** de diversas formas distintas de la normal). Aquí, la cláusula **finally** dice: “Llama siempre a **dispose()** para **x**, independientemente de lo que suceda”.

En el método de limpieza, (**dispose()**, en este caso), también hay que prestar atención al orden de llamada de los métodos de limpieza de la clase base y de los objetos miembro, en caso de que un subobjeto dependa de otro. En general, hay que seguir la misma forma que imponen los compiladores de C++ para los destructores: primero hay que realizar toda la tarea de limpieza específica de la clase, en orden inverso a su creación (en general, esto requiere que los elementos de la clase base sigan siendo viables). A continuación, se invoca el método de limpieza de la clase base, como se ilustra en el ejemplo.

Hay muchos casos en los que el tema de la limpieza no constituye un problema, bastando con dejar que el depurador de memoria realice su tarea. Pero cuando hay que llevar a cabo una limpieza explícita se requieren grandes dosis de diligencia y atención, porque el depurador de memoria no sirve de gran ayuda en este aspecto. El depurador de memoria puede que no

llegue nunca a ser invocado y, en caso de que lo sea, podría reclamar los objetos en el orden que quisiera. No podemos confiar en la depuración de memoria para nada que no sea reclamar las zonas de memoria no utilizadas. Si queremos que las tareas de limpieza se lleven a cabo, es necesario definir nuestros propios métodos de limpieza y no emplear `finalize()`.

Ejercicio 12: (3) Añada una jerarquía adecuada de métodos `dispose()` a todas las clases del Ejercicio 9.

Ocultación de nombres

Si una clave base de Java tiene un nombre de método varias veces sobrecargado, redefinir dicho nombre de método en la clase derivada no ocultará ninguna de las versiones de la clase base (a diferencia de lo que sucede en C++). Por tanto, el mecanismo de sobrecarga funciona independientemente de si el método ha sido definido en este nivel o en una clase base:

```
//: reusing/Hide.java
// La sobrecarga de un nombre de método de la clase base en una
// clase derivada no oculta las versiones de la clase base.
import static net.mindview.util.Print.*;

class Homer {
    char doh(char c) {
        print("doh(char)");
        return 'd';
    }
    float doh(float f) {
        print("doh(float)");
        return 1.0f;
    }
}

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {
        print("doh(Milhouse)");
    }
}

public class Hide {
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
} /* Output:
doh(float)
doh(char)
doh(float)
doh(Milhouse)
*///:-
```

Podemos ver que todos los métodos sobrecargados de `Homer` están disponibles en `Bart`, aunque `Bart` introduce un nuevo método sobrecargado (si se hiciera esto en C++ se ocultarían los métodos de la clase base). Como veremos en el siguiente capítulo, lo más común es sobrecargar los métodos del mismo nombre, utilizando exactamente la misma firma y el mismo tipo de retorno de la clase de retorno. En caso contrario, el código podría resultar confuso (lo cual es la razón por la que C++ oculta todos los métodos de la clase base, para que no cometamos lo que muy probablemente se trata de un error).

Java SE5 ha añadido al lenguaje la anotación `@Override`, que no es una palabra clave pero puede usarse como si lo fuera. Cuando queremos sustituir un método, podemos añadir esta anotación y el compilador generará un mensaje de error si sobre-cargamos accidentalmente el método en lugar de sustituirlo:

```
//: reusing/Lisa.java
// {CompileTimeError} (Won't compile)

class Lisa extends Homer {
    @Override void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
} //:-
```

El marcador `{CompileTimeError}` excluye el archivo del proceso de construcción con Ant de este libro, pero si lo compila a mano podrá ver el mensaje de error:

```
method does not override a method from its superclass
```

La anotación `@Override` evitara, así, que sobrecarguemos accidentalmente un método cuando no es eso lo que queremos hacer.

Ejercicio 13: (2) Cree una clase con un método que esté sobrecargado tres veces. Defina una nueva clase que herede de la anterior y añada una nueva versión sobrecargada del método. Muestre que los cuatro métodos están disponibles en la clase derivada.

Cómo elegir entre la composición y la herencia

Tanto la composición como la herencia nos permiten incluir subobjetos dentro de una nueva clase (la composición lo hace de forma explícita, mientras que en el caso de la herencia esto se hace de forma implícita). Puede que se esté preguntando cuál es la diferencia entre ambos mecanismos y cuándo conviene elegir entre uno y otro.

Generalmente, la composición se usa cuando se desea incorporar la funcionalidad de la clase existente dentro de la nueva clase pero no su interfaz. En otras palabras, lo que hacemos es integrar un objeto para poderlo utilizar con el fin de poder implementar ciertas características en la nueva clase, pero el usuario de la nueva clase verá la interfaz que hayamos definido para la nueva clase en lugar de la interfaz del objeto incrustado. Para conseguir este efecto, lo que hacemos es integrar objetos `private` de clases existentes dentro de la nueva clase.

Algunas veces, tiene sentido permitir que el usuario de la clase acceda directamente a la composición de esa nueva clase, es decir, hacer que los objetos miembros sean públicos. Los objetos miembros utilizan la técnica de la ocultación de la implementación por sí mismos, así que no existe ningún riesgo. Cuando el usuario sabe que estamos ensamblando un conjunto de elementos, normalmente, puede comprender mejor la interfaz que hayamos definido. Un ejemplo sería un objeto `car` (`coche`):

```
//: reusing/Car.java
// Composición con objetos públicos.

class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

class Door {
    public Window window = new Window();
    public void open() {}
```

```

    public void close() {}
}

public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door
        left = new Door(),
        right = new Door(); // 2-door
    public Car() {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
} //:-

```

Puesto que en este caso la composición de un coche forma parte del análisis del problema (y no simplemente del diseño subyacente), hacer los miembros públicos ayuda al programador de clientes a entender cómo utilizar la clase, y disminuye también la complejidad del código que el creador de la clase tiene que desarrollar. Sin embargo, tenga en cuenta que éste es un caso especial y que, en general, los campos deberían ser privados.

Cuando recurrimos al mecanismos de herencia, lo que hacemos es tomar una clase existente y definir una versión especial de la misma. En general, esto quiere decir que estaremos tomando una clase de propósito general y especializándola para una necesidad concreta. Si reflexionamos un poco acerca de ello, podremos ver que no tendría ningún sentido componer un coche utilizando un objeto vehículo, ya que un coche no contiene vehículo, sino que *es* un vehículo. La relación *es-un* se expresa mediante la herencia, mientras que la relación *tiene-un* se expresa mediante la composición.

Ejercicio 14: (1) En **Car.java** añada un método **service()** a **Engine** e invoque este método desde **main()**.

protected

Ahora que ya hemos tomado contacto con la herencia, vemos que la palabra clave **protected** adquiere su pleno significado. En un mundo ideal, la palabra clave **private** resultaría suficiente, pero en los proyectos reales, hay veces en las que queremos ocultar algo a ojos del mundo pero permitir que accedan a ese algo los miembros de las clases derivadas.

La palabra clave **protected** es homenaje al pragmatismo, lo que dice es: "Este elemento es privado en lo que respecta al usuario de la clase, pero está disponible para cualquiera que herede de esta clase y para todo lo demás que se encuentre en el mismo paquete (**protected** también proporciona acceso de paquete)".

Aunque es posible crear campos de tipo **protected** (protegidos), lo mejor es definir los campos como privados; esto nos permitirá conservar siempre el derecho a modificar la implementación subyacente. Entonces, podemos permitir que los herederos de nuestra clase dispongan de un método controlado utilizando métodos **protected**:

```

//: reusing/Orc.java
// La palabra clave protected.
import static net.mindview.util.Print.*;

class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return "I'm a Villain and my name is " + name;
    }
}

```

```

public class Orc extends Villain {
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Disponible porque está protegido.
        this.orcNumber = orcNumber;
    }
    public String toString() {
        return "Orc " + orcNumber + ": " + super.toString();
    }
    public static void main(String[] args) {
        Orc orc = new Orc("Limburger", 12);
        print(orc);
        orc.change("Bob", 19);
        print(orc);
    }
} /* Output:
Orc 12: I'm a Villain and my name is Limburger
Orc 19: I'm a Villain and my name is Bob
*///:-
```

Puede ver que `change()` tiene acceso a `set()` porque es de tipo **protected**. Observe también la forma en que se ha definido el método `toString()` de `Orc` en términos de `toString()` de la clase base.

Ejercicio 15: (2) Cree una clase dentro de un paquete. Esa clase debe estar dentro de un paquete. Esa clase debe contener un método **protected**. Fuera del paquete, trate de invocar el método **protected** y explique los resultados. Ahora defina otra clase que herede de la anterior e invoque el método **protected** desde un método de la clase derivada.

Upcasting (generalización)

El aspecto más importante de la herencia no es que proporciona métodos para la nueva clase, sino la relación expresada entre la nueva clase y la clase base. Esta relación puede resumirse diciendo que “la nueva clase *es un tipo de* la clase existente”.

Esta descripción no es simplemente una forma elegante de explicar la herencia, sino que está soportada directamente por el lenguaje. Por ejemplo, considere una clase base denominada **Instrument** que represente instrumentos musicales y una clase derivada denominada **Wind** (instrumentos de viento). Puesto que la herencia garantiza que todos los métodos de la clase base estén disponibles también en la clase derivada, cualquier mensaje que envíemos a la clase base puede enviarse también a la clase derivada. Si la clase **Instrument** tiene un método `play()` (tocar el instrumento), también lo tendrán los instrumentos de la clase **Wind**. Esto significa que podemos decir con propiedad que un objeto **Wind** es también un objeto de tipo **Instrument**. El siguiente ejemplo ilustra cómo soporta el compilador esta idea.

```

//: reusing/Wind.java
// Herencia y generalización.

class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

// Los objetos instrumentos de viento
// porque tienen la misma interfaz;
```

```

public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Generalización
    }
} //:-.

```

Lo más interesante de este ejemplo es el método **tune()** (afinar), que acepta una referencia a un objeto **Instrument**. Sin embargo, en **Wind.main()** al método **tune()** se le entrega una referencia a un objeto **Wind**. Dado que Java es muy estricto en lo que respecta a las comprobaciones de tipos, parece extraño que un método que acepta un determinado tipo pueda aceptar también otro tipo distinto, hasta que nos demos cuenta de que un objeto **Wind** también es un objeto **Instrument** y de que no existe ningún método que **tune()** pudiera invocar para un objeto **Instrument** y que no se encuentre también en **Wind**. Dentro de **tune()**, el código funciona tanto para los objetos **Instrument** como para cualquier otra cosa derivada de **Instrument**, y el acto de convertir una referencia a un objeto **Wind** en otra referencia a **Instrument** se denomina *upcasting* (generalización).

¿Por qué generalizar?

El término está basado en la forma en que se vienen dibujando tradicionalmente los diagramas de herencia de clase. Con la raíz en la parte superior de la página y las clases derivadas distribuyéndose hacia abajo (por supuesto, podríamos dibujar los diagramas de cualquier otra manera que nos resultara útil). El diagrama de herencia para **Wind.java** será entonces:



Al realizar una proyección de un tipo derivado al tipo base, nos movemos *hacia arriba* en el diagrama de herencia, y esa es la razón de que en inglés se utilice el término *upcasting* (*up* = arriba, *cast* = proyección). El *upcasting* o generalización siempre resulta seguro, porque estamos pasando de un tipo más específico a otro más general. Es decir, la clase derivada es un superconjunto de la clase base. Puede que la clase derivada contenga más métodos que la clase base, pero debe contener *al menos* los métodos de la clase base. Lo único que puede ocurrir con la interfaz de la clase durante la generalización es que pierda métodos, no que los gane, y ésta es la razón por la que el compilador permite la generalización sin efectuar ningún tipo de proyección explícita y sin emplear ninguna notación especial.

También podemos realizar el inverso de la generalización, que se denomina *downcasting* (especialización), pero esto lleva asociado un cierto dilema que examinaremos más en detalle en el siguiente capítulo, y en el Capítulo 14, *Información de tipos*.

Nueva comparación entre la composición y la herencia

En la programación orientada a objetos, la forma más habitual de crear y utilizar código consiste en empaquetar los datos y métodos en una clase y usar los objetos de dicha clase. También utilizamos otras clases existentes para construir nuevas clases utilizando el mecanismo de composición. Menos frecuentemente, debemos utilizar el mecanismo de herencia. Por tanto, aunque al enseñar programación orientada a objetos se suele hacer un gran hincapié en el tema de la herencia, eso no quiere decir que se la deba usar en todo momento. Por el contrario, conviene emplearla con mesura, y sólo cuando esté claro que la herencia resulta útil. Una de las formas más claras de determinar si debe utilizarse composición o herencia consiste en preguntarse si va a ser necesario recurrir en algún momento al mecanismo de generalización de la nueva clase a la clase base. Si es necesario usar dicho mecanismo, entonces la herencia será necesaria, pero si ese mecanismo no hace falta conviene meditar si verdaderamente hay que emplear la herencia. En el Capítulo 8, *Polimorfismo* se proporciona una de las razones más importantes para utilizar la generalización, pero si se acuerda de preguntarse “*¿voy a necesitar generalizar en algún momento?*” tendrá una buena forma de optar entre la composición y la herencia.

Ejercicio 16: (2) Cree una clase denominada **Amphibian** (anfibio). A partir de ésta, defina una nueva clase denominada **Frog** (rana) que herede de la anterior. Incluya una serie de métodos apropiados en la clase base. En

`main()`, cree un objeto **Frog** y realice una generalización a **Amphibian**, demostrando que todos los métodos siguen funcionando.

Ejercicio 17: (1) Modifique el Ejercicio 16 para que el objeto **Frog** sustituya las definiciones de métodos de la clase base (proporcione las nuevas definiciones utilizando las mismas signaturas de métodos). Observe lo que sucede en `main()`.

La palabra clave final

La palabra clave de Java **final** tiene significados ligeramente diferentes dependiendo del contexto, pero en general quiere decir: “Este elemento no puede modificarse”. Puede haber dos razones para que no queramos permitir los cambios: diseño y eficiencia. Puesto que estas dos razones son muy diferentes entre sí, resulta bastante posible utilizar la palabra clave **final** de manera inadecuada.

En las siguientes secciones vamos a ver los tres lugares donde **final** puede utilizarse: para los datos, para los métodos y para las clases.

Datos final

Muchos lenguajes de programación disponen de alguna forma de comunicarle al compilador que un elemento de datos es “constante”. Las constantes son útiles por dos razones:

1. Puede tratarse de una *constante de tiempo de compilación* que nunca va a cambiar.
2. Puede tratarse de un valor inicializado en tiempo de ejecución que no queremos que cambie.

En el caso de una constante de tiempo de compilación, el compilador está autorizado a “compactar” el valor constante en todos aquellos cálculos que se le utilice; es decir, el cálculo puede realizarse en tiempo de compilación eliminando así ciertos cálculos en tiempo de ejecución. En Java, estos tipos de constantes deben ser primitivas y se expresan con la palabra clave **final**. En el momento de definir una de tales constantes, es preciso definir un valor.

Un campo que sea a la vez **static** y **final** sólo tendrá una zona de almacenamiento que no puede nunca ser modificada.

Cuando **final** se utiliza con referencias a objetos en lugar de con primitivas, el significado puede ser confuso. Con una primitiva, **final** hace que el *valor* sea constante, pero con una referencia a objeto lo que **final** hace constante es la *referencia*. Una vez inicializada la referencia a un objeto, nunca se la puede cambiar para que apunte a otro objeto. Sin embargo, el propio objeto sí que puede ser modificado; Java no proporciona ninguna manera para hacer que el objeto arbitrario sea constante (podemos, sin embargo, escribir nuestras clases de modo que tengan el efecto de que los objetos sean constantes). Esta restricción incluye a las matrices, que son también objetos.

He aquí un ejemplo donde se ilustra el uso de los campos **final**. Observe que, por convenio, los campos que son a la vez **static** y **final** (es decir, constantes de tiempo de compilación) se escriben en mayúsculas, utilizando guiones bajos para separar las palabras.

```
//: reusing/FinalData.java
// Efecto de final sobre los campos.
import java.util.*;
import static net.mindview.util.Print.*;

class Value {
    int i; // Acceso de paquete
    public Value(int i) { this.i = i; }
}

public class FinalData {
    private static Random rand = new Random(47);
    private String id;
    public FinalData(String id) { this.id = id; }
    // Pueden ser constantes de tiempo de compilación:
    private final int valueOne = 9;
```

```

private static final int VALUE_TWO = 99;
// Constante pública típica:
public static final int VALUE_THREE = 39;
// No pueden ser constantes de tiempo de compilación:
private final int i4 = rand.nextInt(20);
static final int INT_5 = rand.nextInt(20);
private Value v1 = new Value(11);
private final Value v2 = new Value(22);
private static final Value VAL_3 = new Value(33);
// Matrices:
private final int[] a = { 1, 2, 3, 4, 5, 6 };
public String toString() {
    return id + ":" + i4 + ", " + INT_5 + ", ";
}
public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    // fd1.valueOne++; // Error: no se puede modificar el valor
    fd1.v2.i++; // ;El objeto no es constante!
    fd1.v1 = new Value(9); // OK -- no es final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a[i]++; // ;El objeto no es constante!
    // fd1.v2 = new Value(0); // Error: no se puede
    // fd1.VAL_3 = new Value(1); // cambiar la referencia
    // fd1.a = new int[3];
    print(fd1);
    print("Creating new FinalData");
    FinalData fd2 = new FinalData("fd2");
    print(fd1);
    print(fd2);
}
} /* Output:
fd1: i4 = 15, INT_5 = 18
Creating new FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*///:-

```

Dado que **valueOne** y **VALUE_TWO** son primitivas **final** con valores definidos en tiempo de compilación, ambas pueden usarse como constantes de tiempo de compilación y no se diferencian en ningún aspecto importante. **VALUE_THREE** es la forma más típica en que podrá ver definidas dichas constantes: **public** que se pueden usar fuera del paquete, **static** para enfatizar que sólo hay una y **final** para decir que se trata de una constante. Observe que las primitivas **final static** con valores iniciales constantes (es decir, constantes en tiempo de compilación) se designan con letras mayúsculas por convenio, separando las palabras mediante guiones bajos (al igual que las constantes en C, que es el lenguaje en el que surgió este convenio).

El que algo sea **final** no implica necesariamente el que su valor se conozca en tiempo de compilación. El ejemplo ilustra está inicializando **i4** e **INT_5** en tiempo de ejecución, mediante números generados aleatoriamente. Esta parte del ejemplo también genera la diferencia entre hacer un valor **final** estático o no estático. Esta diferencia sólo se hace patente cuando los valores se inicializan en tiempo de ejecución, ya que el compilador trata de la misma manera los valores de tiempo de compilación (en muchas ocasiones, optimizando el código para eliminar esas constantes). La diferencia se muestra cuando se ejecuta el programa. Observe que los valores de **i4** para **fd1** y **fd2** son distintos, mientras que el valor para **INT_5** no cambia porque creamos el segundo objeto **FinalData**. Esto se debe a que es estático y se inicializa una sola vez durante la carga y no cada vez que se crea un nuevo objeto.

Las variables **v1** a **VAL_3** ilustran el significado de una referencia **final**. Como puede ver en **main()**, el hecho de que **v2** sea **final** no quiere decir que se pueda modificar su valor. Puesto que es una referencia, **final** significa que no se puede asociar **v2** con un nuevo objeto. Podemos ver que la afirmación también es cierta para las matrices, que son otro de tipo de referencia (no hay ninguna forma que yo conozca de hacer que las referencias a una matriz sean **final**). Hacer las referencias **final** parece menos útil que definir las primitivas como **final**.

Ejercicio 18: (2) Cree una clase con un campo static final y un campo final y demuestre la diferencia entre los dos.

Valores final en blanco

Java permite la creación de *valores finales en blanco*, que son campos que se declaran como **final** pero que no se les proporciona un valor de inicialización. En todos los casos, el valor **final** en blanco *debe ser* inicializado antes de utilizarlo, y el compilador se encargará de hacer que esto sea así. Sin embargo, los valores **final** en blanco proporcionan mucha más flexibilidad en el uso de la palabra clave **final** ya que, por ejemplo, un campo **final** dentro de una clase podrá con esto ser diferente para cada objeto y mantener aún así su carácter de inmutable. He aquí un ejemplo:

```
//: reusing/BlankFinal.java
// Campos final "en blanco".

class Poppet {
    private int i;
    Poppet(int ii) { i = ii; }
}

public class BlankFinal {
    private final int i = 0; // Valor final inicializado
    private final int j; // Valor final en blanco
    private final Poppet p; // Referencia final en blanco
    // Los valores final en blanco DEBEN inicializarse en el constructor:
    public BlankFinal() {
        j = 1; // Inicializar valor final en blanco
        p = new Poppet(1); // Inicializar referencia final en blanco
    }
    public BlankFinal(int x) {
        j = x; // Inicializar valor final en blanco
        p = new Poppet(x); // Inicializar referencia final en blanco
    }
    public static void main(String[] args) {
        new BlankFinal();
        new BlankFinal(47);
    }
} //:-
```

Estamos obligados a realizar asignaciones a los valores **final** utilizando una expresión en el punto de definición del campo o bien en cada constructor. De esa forma, se garantizará que el campo **final** esté siempre inicializado antes de utilizarlo.

Ejercicio 19: (2) Cree una clase con una referencia **final** en blanco a un objeto. Realice la inicialización de la referencia **final** en blanco dentro de todos los constructores. Demuestre que se garantiza que el valor **final** estará inicializado antes de utilizarlo, y que no se puede modificar una vez inicializado.

Argumentos final

Java permite definir argumentos **final** declarándolos como tales en la lista de argumentos. Esto significa que dentro del método no se podrá cambiar aquello a lo que apunte la referencia del argumento:

```
//: reusing/FinalArguments.java
// Uso de "final" con argumentos de métodos.

class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegal -- g es final
    }
}
```

```

void without(Gizmo g) {
    g = new Gizmo(); // OK -- g no es final
    g.spin();
}

// void f(final int i) { i++; } // No se puede cambiar
// Las primitivas final sólo pueden leerse:
int g(final int i) { return i + 1; }
public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
}
}

```

Los métodos `f()` y `g()` muestran lo que sucede cuando los argumentos primitivos son `final`: se puede leer el argumento, pero no modificarlo. Esta característica se utiliza principalmente para pasar datos a las clases internas anónimas, lo cual es un tema del que hablaremos en el Capítulo 10, *Clases Internas*.

Métodos final

Hay dos razones para utilizar métodos `final`. La primera es "bloquear" el método para impedir que cualquier clase que herede de ésta cambie su significado. Esto se hace por razones de diseño cuando queremos asegurarnos de que se retenga el comportamiento de un método durante la herencia y que ese método pueda ser sustituido.

La segunda razón por la que se ha sugerido en el pasado la utilización de los métodos `final` es la eficiencia. En las implementaciones anteriores de Java, si definíamos un método como `final`, permitíamos al compilador convertir todas las llamadas a ese método en llamadas *en línea*. Cuando el compilador veía una llamada a un método `final`, podía (a su discreción) saltarse el modo normal de insertar el código correspondiente al mecanismo de llamada al método (insertar los argumentos en la pila, saltar al código del método y ejecutarlo, saltar hacia atrás y eliminar de la pila los argumentos y tratar el valor de retorno), para sustituir en su lugar la llamada al método por una copia del propio código contenido en el cuerpo del método. Esto elimina el gasto adicional de recursos asociado a la llamada al método. Por supuesto, si el método es de gran tamaño, el código empezará a crecer enormemente y probablemente no detectemos ninguna mejora de velocidad por la utilización de métodos en línea, ya que la mejora será insignificante comparada con la cantidad de tiempo invertida dentro del método.

En las versiones más recientes de Java, la máquina virtual (en particular, la tecnología *hotspot*) puede detectar estas situaciones y eliminar el paso adicional de indirección, por lo que ya no es necesario (de hecho, se desaconseja por regla general) utilizar `final` para tratar de ayudar al optimizador. Con Java SE5/6, lo que debemos hacer es dejar que el compilador y la JVM se encarguen de las cuestiones de eficiencia, y sólo debemos definir un método como `final` si queremos impedir explícitamente la sustitución del método en las clases derivadas.¹

final y private

Los métodos privados de una clase son implicitamente de tipo `final` (finales). Puesto que no se puede acceder a un método privado, es imposible sustituirlo en una clase derivada. Podemos añadir el especificador `final` a un método `private`, pero no tendrá ningún efecto adicional.

Este tema puede causar algo de confusión, porque si se trata de sustituir un método `private` (que implicitamente es `final`), parece que el mecanismo funciona y el compilador no proporciona ningún mensaje de error.

```

//: reusing/FinalOverridingIllusion.java
// Tan sólo parece que podemos sustituir
// un método privado o un método privado final.
import static net.mindview.util.Print.*;

```

¹No pierda el tiempo tratando de optimizar prematuramente. Si el sistema funciona y es demasiado lento, resulta dudoso que lo pueda solventar con la palabra clave `final`. <http://MindView.net/Books/BetterJava> contiene información acerca de las técnicas de perfilado, que pueden servir de ayuda a la hora de acelerar los programas.

```

class WithFinals {
    // Idéntico al uso de "private" sola:
    private final void f() { print("WithFinals.f()"); }
    // También automáticamente "final":
    private void g() { print("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        print("OverridingPrivate.f()");
    }
    private void g() {
        print("OverridingPrivate.g()");
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        print("OverridingPrivate2.f()");
    }
    public void g() {
        print("OverridingPrivate2.g()");
    }
}

public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // Se puede generalizar:
        OverridingPrivate op = op2;
        // Pero no se pueden invocar los métodos:
        //! op.f();
        //! op.g();
        // Lo mismo aquí:
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
    }
} /* Output:
OverridingPrivate2.f()
OverridingPrivate2.g()
*///:-
```

La “sustitución de los métodos” sólo puede tener lugar si el método forma parte de la clase base. En otras palabras, es necesario poder generalizar un objeto a su tipo base y poder invocar al mismo método (este tema resultará más claro en el siguiente capítulo). Si un método es privado, no forma parte de la interfaz de la clase base. Se trata simplemente de un cierto código que está oculto dentro de la clase y que sucede que tiene ese nombre, pero si creamos un método **public**, **protected** o con acceso de paquete con el mismo nombre en la clase derivada, no habrá ninguna conexión con el método que resulte que tiene el mismo nombre en la clase base. Es decir, no habremos sustituido el método, sino que simplemente habremos creado otro nuevo. Puesto que un método **private** es inalcanzable y resulta invisible a efectos prácticos, a lo único que afecta es a la organización del código de la clase en la que haya sido definido.

Ejercicio 20: (1) Demuestre que la anotación **@Override** resuelve el problema descrito en esta sección.

Ejercicio 21: (1) Cree una clase con un método **final**. Cree otra clase que herede de la clase anterior y trate de sustituir ese método.

Clases final

Cuando decimos que toda una clase es de tipo **final** (precediendo su definición con la palabra clave **final**), lo que estamos diciendo es que no queremos heredar de esta clase ni permitir que nadie más lo haga. En otras palabras, por alguna razón, el diseño de nuestra clase es de tal naturaleza que nunca va a ser necesario efectuar ningún cambio, o bien no queremos que nadie defina clases derivadas por razones de seguridad.

```
//: reusing/Jurassic.java
// Definición de una clase completa como final.

class SmallBrain {}

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//: class Further extends Dinosaur {}
// error: no se puede heredar de la clase final 'Dinosaur'

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:-
```

Observe que los campos de una clase **final** pueden ser de tipo **final** o no, según deseemos. A los campos de tipo **final** se les aplican las mismas reglas independientemente de si la clase está definida como **final**. Sin embargo, como la clase impide la herencia, *todos los métodos* en una clase **final** son *implicitamente final*, ya que no existe ninguna forma de sustituirlos. Podemos añadir el especificador **final** a un método de una clase **final**, pero no tiene ningún efecto adicional.

Ejercicio 22: (1) Cree una clase **final** y trate de definir otra clase que herede de ella.

Una advertencia sobre final

En ocasiones, uno puede verse tentado a definir un método como **final** a la hora de definir una clase, pensando que nadie podría querer sustituir ese método. A veces, es cierto que las cosas pueden ser así.

Pero tenga cuidado con las suposiciones que realiza. En general, es difícil prever cómo se va a reutilizar una clase, especialmente si se trata de una clase de propósito general. Si define un método como **final**, puede que esté impidiendo reutilizar la clase a través del mecanismo de herencia en algún otro proyecto de programación, simplemente porque no había llegado a imaginar que esa clase pudiera llegar a emplearse de esa forma.

La biblioteca estándar de Java es un buen ejemplo de esto. En particular, la clase **Vector** de Java 1.0/1.1 se utilizaba de forma bastante común y todavía podía haber resultado más útil si, por consideraciones de eficiencia (que no pasaban de ser una ilusión), no se hubieran hecho todos los métodos **final**. Resulta fácil imaginar que alguien quiera heredar una clase tan fundamental y tan útil y sustituir sus métodos, pero los diseñadores decidieron por alguna razón que esto no era apropiado. Resulta bastante irónico que se tomara esa decisión por dos razones distintas. En primer lugar, **Stack** (pila) hereda de **Vector**, lo que quiere decir que **Stack** es un **Vector**, lo que no es realmente cierto desde un punto vista lógico. En cualquier caso, se trata de un ejemplo en el que los propios diseñadores de Java decidieron que una determinada clase heredara de **Vector**. Cuando crearon **Stack** de esta forma, debieron darse cuenta de que los métodos **final** eran bastante restrictivos.

En segundo lugar, muchos de los métodos más importantes de **Vector**, como **addElement()** y **elementAt()** están sincronizados (**synchronized**). Como veremos en el Capítulo 21, *Concurrencia*, esta característica restringe significativamente las

prestaciones, lo que probablemente anula cualquier ganancia proporcionada por **final**. Este ejemplo tiende a avalar la teoría de que los programadores suelen equivocarse siempre a la hora de decidir dónde hay que optimizar. Resulta un poco penoso que un diseño tan pobre terminara siendo incluido en la biblioteca estándar para que todo el mundo tuviera que sufrirlo (afortunadamente, la moderna biblioteca de contenedores Java sustituye **Vector** por **ArrayList**, que se comporta de una forma mucho más civilizada; lamentablemente, hoy día se sigue escribiendo código que utiliza la antigua biblioteca de contenedores).

Resulta también interesante observar que **Hashtable**, otra clase importante de la biblioteca estándar 1.0/1.1 de Java *no* tiene ningún método **final**. Como ya se ha mencionado, es patente que algunas de las clases fueran diseñadas por personas distintas (tendrá la ocasión de comprobar que los nombres de los métodos en **Hashtable** son mucho más breves comparados con los de **Vector**, lo que constituye otra prueba de esta afirmación). Esto es, precisamente, el tipo de cosas que *no resultan* obvias para los consumidores de una biblioteca de clases. Cuando las cosas no son coherentes, damos más trabajo del necesario a los usuarios, lo cual es otro argumento en favor de las revisiones de diseño y de los programas (observe que la biblioteca actual de contenedores Java sustituye **Hashtable** por **HashMap**).

Inicialización y carga de clases

En los lenguajes más tradicionales, los programas se cargan de una vez como parte del proceso de arranque. Este proceso va seguido del de inicialización y luego del programa. El proceso de inicialización en estos lenguajes debe controlarse cuidadosamente para que el orden de inicialización de los valores estáticos no cause problemas. Por ejemplo, C++ tiene problemas si uno de los valores estáticos espera que otro valor estático sea válido antes de que el segundo haya sido inicializado.

Java no tiene este problema porque adopta una técnica de carga completamente distinta. Ésta es una de las actividades que se facilitan enormemente porque en Java todo es un objeto. Recuerde que el código compilado de cada clase está almacenado en su propio archivo separado. Dicho archivo no se carga hasta que ese código sea necesario. En general, podemos decir que “el código de las clases se carga en el lugar que por primera vez se utiliza”. Usualmente, dicho lugar es cuando se construye el primer objeto de esa clase, aunque la carga también puede tener lugar cuando se acceda a un campo estático o a un método estático.²

El lugar del primer uso es también el lugar donde se produce la inicialización de los elementos estáticos. Todos los elementos estáticos y el bloque de código **static** se inicializarán en orden textual (es decir, en el orden en el que estén escritos en la definición de la clase), en el punto donde se produzca la carga. Por supuesto, los valores estáticos sólo se inicializan una vez.

Inicialización con herencia

Resulta útil analizar el proceso de inicialización completo, incluyendo la herencia, para hacerse una idea general. Considere el siguiente ejemplo:

```
//: reusing/Beetle.java
// El proceso completo de inicialización.
import static net.mindview.util.Print.*;

class Insect {
    private int i = 9;
    protected int j;
    Insect() {
        print("i = " + i + ", j = " + j);
        j = 39;
    }
    private static int x1 =
        printInit("static Insect.x1 initialized");
    static int printInit(String s) {
```

²El constructor es también un método estático, aún cuando la palabra clave **static** no sea explícita. Por tanto, para ser precisos, una clase se carga por primera vez cuando se accede a cualquiera de sus miembros estáticos.

```

        print(s);
        return 47;
    }

}

public class Beetle extends Insect {
    private int k = printInit("Beetle.k initialized");
    public Beetle() {
        print("k = " + k);
        print("j = " + j);
    }
    private static int x2 =
        printInit("static Beetle.x2 initialized");
    public static void main(String[] args) {
        print("Beetle constructor");
        Beetle b = new Beetle();
    }
} /* Output:
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
*///:-

```

Lo primero que sucede cuando se ejecuta Java con **Beetle** es que se trata de acceder a **Beetle.main()** (un método estático), por lo que el cargador localiza el código compilado correspondiente a la clase **Beetle** (en un archivo denominado **Beetle.class**). Durante el proceso de carga, el cargador observa que tiene una clase base (eso es lo que dice la palabra clave **extends**), por lo que procede a cargarlo. Esto sucederá independientemente de si se va a construir un objeto de dicha clase base (pruebe a desactivar con comentarios la creación del objeto como demostración).

Si la clase base tiene a su vez otra clase base, esa segunda clase base se cargaría, y así sucesivamente. A continuación, se realiza la inicialización **static** en la clase base raíz (en este caso, **Insect**), y luego en la siguiente clase derivada, etc. Esto es importante, porque la inicialización **static** de la clase derivada puede depender de que el miembro de la clase base haya sido inicializado adecuadamente.

En este punto, ya se habrán cargado todas las clases necesarias, por lo que se podrá crear el objeto. En primer lugar, se asignan los valores predeterminados a todas las primitivas de este objeto y se configuran las referencias a objetos con el valor **null** (esto sucede en una única pasada, escribiendo ceros binarios en la memoria del objeto). A continuación, se invoca el constructor de la clase base. En este caso la llamada es automática, pero también podemos especificar la llamada al constructor de la clase base (como primera operación dentro del constructor **Beetle()**) utilizando **super**. El constructor de la clase base pasa a través del mismo proceso y en el mismo orden que el constructor de la clase derivada. Después de que se complete el constructor de la clase base, las variables de instancia se inicializan en orden textual. Finalmente, se ejecuta el resto del cuerpo del constructor.

Ejercicio 23: (2) Demuestre que el proceso de carga de una clase sólo tiene lugar una vez. Demuestre que la carga puede ser provocada por la creación de la primera instancia de esa clase o por el acceso a un miembro estático de la misma.

Ejercicio 24: (2) En **Beetle.java**, defina una nueva clase que represente un tipo específico de la clase **Beetle** de la que debe heredar, siguiendo el mismo formato que las clases existentes. Trace y explique los resultados de salida.

Resumen

Tanto la herencia como la composición permiten crear nuevos tipos a partir de los tipos existentes. La composición reutiliza los tipos existentes como parte de la implementación subyacente del nuevo tipo, mientras que la herencia reutiliza la interfaz.

Con la herencia, la clase derivada tiene la interfaz de la clase base, así que puede ser *generalizada* hacia la base, lo cual resulta crítico para el polimorfismo, como veremos en el siguiente capítulo.

A pesar del gran énfasis que se pone en las cuestiones de herencia cuando hablamos de programación orientada a objetos, al comenzar un diseño suele ser preferible la composición (o preferiblemente la delegación) en una primera aproximación, y utilizar la herencia sólo cuando sea claramente necesario. La composición tiende a ser más flexible. Además, utilizando la herencia como artificio añadido a un tipo que se haya incluido como miembro de una clase, se puede modificar el tipo exacto (y por tanto el comportamiento) de esos objetos miembro en tiempo de ejecución. De este modo, se puede modificar el comportamiento del objeto compuesto en tiempo de ejecución.

A la hora de diseñar un sistema, nuestro objetivo consiste en localizar o crear un conjunto de clases en el que cada clase tenga un uso específico y no sea ni demasiado grande (que abarque tanta funcionalidad que sea difícil de reutilizar) ni incómodamente pequeña (de modo que no se pueda emplear por sí misma o sin añadirla funcionalidad). Cuando los diseños comienzan a complicarse demasiado, suele resultar útil añadir más objetos descomponiendo los existentes en otros más pequeños.

Cuando se proponga diseñar un sistema, es importante tener en cuenta que el desarrollo de los programas es un proceso incremental, al igual que el proceso de aprendizaje humano. El proceso de diseño necesita de la experimentación; podemos hacer las tareas de análisis que queramos pero es imposible que lleguemos a conocer todas las respuestas en el momento de arrancar el proyecto. Tendrá mucho más éxito (y podrá probar las cosas antes) si comienza a “hacer crecer” el proyecto como una criatura orgánica en constante evolución, en lugar de construir todo de una vez, como si fuera un rascacielos de cristal. La herencia y la composición son dos de las herramientas más fundamentales en la programación orientada a objetos a la hora de realizar tales experimentos en el curso de un proyecto.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Polimorfismo

8

“En ocasiones me he preguntado, ‘Sr. Babbage, si introduce en la máquina datos erróneos, ¿podrá suministrar las respuestas correctas? Debo confesar que no soy capaz de comprender qué tipo de confusión de ideas puede hacer que alguien planteé semejante pregunta’”.

Charles Babbage (1791-1871)

El polimorfismo es la tercera de las características esenciales de un lenguaje de programación orientado a objetos, después de la abstracción de datos y de la herencia.

Proporciona otra dimensión de separación entre la interfaz y la implementación, con el fin de desacoplar el *qué* con respecto al *cómo*. El polimorfismo permite mejorar la organización y la legibilidad de código, así como crear programas *ampliables* que puedan “hacerse crecer” no sólo durante el proceso original de desarrollo del proyecto, sino también cada vez que se desean adherir nuevas características.

El mecanismo de encapsulación crea nuevos tipos de datos combinando diversas características y comportamientos. La técnica de la ocultación de la implementación separa la interfaz de la implementación haciendo que los detalles sean de tipo privado. Este tipo de organización mecánica tiene bastante sentido para las personas que tengan experiencia con la programación procedimental. Pero el polimorfismo trata la cuestión del acoplamiento en términos de *tipos*. En el último capítulo, hemos visto que la herencia permite tratar un objeto como si fuera de su propio tipo o como si fuera del tipo base. Esta capacidad resulta crítica, porque permite tratar varios tipos (todos ellos derivados del mismo tipo base) como si fueran un único tipo, pudiéndose utilizar un mismo fragmento de código para procesar de la misma manera todos esos tipos diferentes. La llamada a un método polimórfico permite que cada tipo exprese su distinción con respecto a los otros tipos similares, siempre y cuando ambos deriven del mismo tipo base. Esta distinción se expresa mediante diferencias en el comportamiento de los métodos que se pueden invocar a través de la clase base.

En este capítulo, vamos a estudiar el tema del polimorfismo (también denominado *acoplamiento dinámico* o *acoplamiento tardío* o *acoplamiento en tiempo de ejecución*) comenzando por los conceptos más básicos y proporcionando ejemplos simples en los que nos fijaremos tan sólo en el comportamiento polimórfico de los programas.

Nuevas consideraciones sobre la generalización

En el último capítulo hemos visto cómo puede utilizarse un objeto como si fuera de su propio tipo o como si fuera un objeto del tipo base. El acto de tomar una referencia a un objeto y tratarla como si fuera una referencia a su tipo base se denomina *generalización* (*upcasting*) debido a la forma en que se dibujan los árboles de herencia, en los que la clase base se suele representar en la parte superior.

También vimos en el capítulo anterior cómo surgió el problema a este respecto, el cual se ilustra en el siguiente ejemplo sobre instrumentos musicales.

En primer lugar, puesto que en muchos de estos ejemplos los instrumentos hacen sonar notas (**Note**), vamos a crear una enumeración separada **Note**, dentro de un paquete:

```
//: polymorphism/music/Note.java
// Notas para tocar en los instrumentos musicales.
```

```

package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // Etc.
} //:-
```

Los tipos **enum** se han presentado en el Capítulo 5, *Inicialización y limpieza*.

Aquí **Wind** es un tipo de **Instrument**; por tanto, **Wind** hereda de **Instrument**:

```

//: polymorphism/music/Instrument.java
package polymorphism.music;
import static net.mindview.util.Print.*;

class Instrument {
    public void play(Note n) {
        print("Instrument.play()");
    }
}
//:-
```

```

//: polymorphism/music/Wind.java
package polymorphism.music;

// Los objetos Wind son instrumentos
// porque tienen la misma interfaz;
public class Wind extends Instrument {
    // Redefinición de un método de la interfaz;
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
}
//:-
```

```

//: polymorphism/music/Music.java
// Herencia y generalización;
package polymorphism.music;

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Generalización
    }
} /* Output:
Wind.play() MIDDLE_C
*:-
```

El método **Music.tune()** acepta una referencia a **Instrument**, pero también a cualquier cosa que se derive de **Instrument**. Podemos ver que esto sucede en **main()**, donde se pasa una referencia **Wind** a **tune()**, sin que sea necesario efectuar ninguna proyección. Esto resulta perfectamente lógico: la interfaz de **Instrument** debe existir en **Wind**, porque **Wind** hereda de **Instrument**. La generalización de **Wind** a **Instrument** puede “estrechar” dicha interfaz, pero en ningún caso esa interfaz podrá llegar a ser más pequeña que la interfaz completa de **Instrument**.

Por qué olvidar el tipo de un objeto

Music.java puede resultarle un poco extraño. ¿Por qué alguien debería *olvidar* intencionadamente el tipo de un objeto? Esto es lo que sucede cuando efectuamos una generalización, y parece que sería mucho más sencillo si **tune()** simplemente tomará una referencia a **Wind** como argumento. Esto plantea un punto esencial: si hiciéramos eso, necesitaríamos escribir un

nuevo método `tune()` para cada clase derivada de `Instrument` que incluyéramos en nuestro sistema. Suponga que siguiéramos esta forma de razonar y añadiéramos dos instrumentos `Stringed` (instrumentos de cuerda) y `Brass` (instrumentos de metal):

```
//: polymorphism/music/Music2.java
// Sobrecarga en lugar de generalización.
package polymorphism.music;
import static net.mindview.util.Print.*;

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // Sin generalización
        tune(violin);
        tune(frenchHorn);
    }
} /* Output:
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
*///:-
```

Esta solución funciona, pero presenta una desventaja importante: es necesario escribir métodos específicos del tipo para cada nueva clase derivada de `Instrument` que añadamos. Esto significa, en primer lugar, un mayor esfuerzo de programación, pero también quiere decir que si queremos añadir un nuevo método como `tune()` o un nuevo tipo de clase derivada de `Instrument`, el trabajo adicional necesario es considerable. Si a esto le añadimos el hecho de que el compilador no nos dará ningún mensaje de error si nos olvidamos de sobrecargar alguno de los métodos, todo el proceso de gestión de los tipos se vuelve immanejable.

¿No sería mucho más fácil, si pudiéramos, limitarnos a escribir un único método que tomara la clase base como argumento y no ninguna de las clases derivadas específicas? En otras palabras: ¿no sería mucho más adecuado si pudiéramos olvidarnos de que hay clases derivadas y escribir el código de manera que sólo se entendiera con la clase base?

Eso es exactamente lo que el polimorfismo nos permite hacer. Sin embargo, la mayoría de los programadores que proceden del campo de los lenguajes de programación procedimentales suelen tener problemas a la hora de entender cómo funciona el polimorfismo.

Ejercicio 1: (2) Cree una clase **Cycle**, con subclases **Unicycle**, **Bicycle** y **Tricycle**. Demuestre que se puede generalizar una instancia de cada tipo a **Cycle** mediante un método **ride()**.

El secreto

La dificultad con **Music.java** puede verse ejecutando el programa. La salida es **Wind.play()**. Se trata claramente de la salida deseada, pero no parece tener sentido que el programa funcione de esa forma. Examinemos el método **tune()**:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}
```

El método recibe una referencia a **Instrument**. De modo que ¿cómo puede el compilador saber que esta referencia a **Instrument** apunta a un objeto **Wind** en este caso y no a un objeto **Brass** o **Stringed**? El compilador no puede saberlo. Para comprender mejor esta cuestión, resulta útil que examinemos el tema del *acoplamiento*.

Acoplamiento de las llamadas a métodos

El hecho de conectar una llamada con el cuerpo del método se denomina *acoplamiento*. Cuando se realiza el acoplamiento antes de ejecutar el programa (es decir, cuando lo realizan el compilador y el montador, si es que existe uno), el proceso se llama *acoplamiento temprano (early binding)*. Puede que haya oido este término antes porque en los lenguajes procedimentales, como por ejemplo C, sólo existe un tipo de llamadas a métodos y ese tipo es precisamente, el acoplamiento temprano, así que no existe ninguna posibilidad de elegir.

La parte confusa del programa anterior es precisamente la que se refiere al acoplamiento temprano, porque el compilador no puede saber cuál es el método correcto que hay que llamar cuando sólo dispone de una referencia **Instrument**.

La solución es el *acoplamiento tardío (late binding)*, que quiere decir que el acoplamiento tiene lugar en tiempo de ejecución basándose en el tipo del objeto. El acoplamiento tardío también se denomina *acoplamiento dinámico* o *acoplamiento en tiempo de ejecución*. Cuando un lenguaje implementa el mecanismo de acoplamiento tardío, debe haber alguna manera de determinar el tipo del objeto en tiempo de ejecución, con el fin de llamar al método apropiado. En otras palabras, el compilador sigue sin saber cuál es el tipo del objeto, pero el mecanismo de invocación del método lo averigua y llama al cuerpo de método correcto. El mecanismo de acoplamiento tardío varía de un lenguaje a otro, pero podemos considerar que en todos los objetos debe incorporarse una cierta información sobre el tipo del objeto.

El mecanismo de acoplamiento de métodos en Java utiliza el acoplamiento tardío a menos que el método sea estático o de tipo **final** (los métodos **private** son implicitamente **final**). Esto quiere decir que, normalmente, no es necesario tomar ninguna decisión acerca de si debe producirse el acoplamiento tardío, ya que éste tendrá lugar automáticamente.

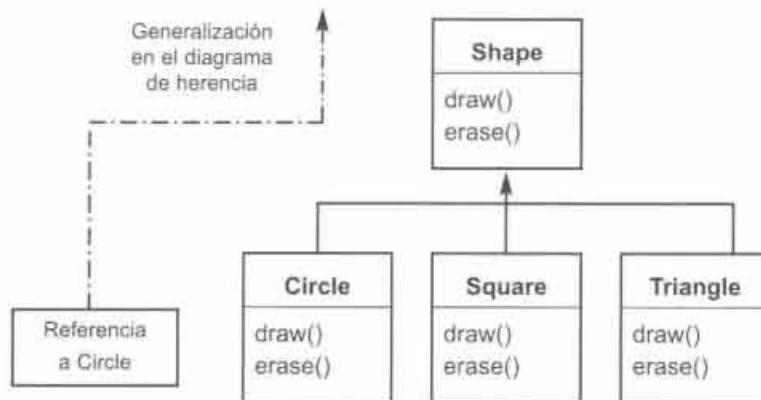
¿Para qué queríamos declarar un método como **final**? Como hemos indicado en el capítulo anterior, esto evita que nadie pueda sustituir dicho método en las clases derivadas. Además, y todavía más importante, esta palabra clave desactiva en la práctica el acoplamiento dinámico, o más bien le dice al compilador que el acoplamiento dinámico no es necesario. Esto permite que el compilador genere un código ligeramente más eficiente para las llamadas a métodos **final**. Sin embargo, en la mayoría de los casos, no será perceptible la ganancia de velocidad en el programa, por lo que lo mejor es utilizar **final** únicamente por decisión de diseño, y no como intento de mejorar las prestaciones.

Especificación del comportamiento correcto

Ahora que sabemos que todo el acoplamiento de métodos en Java tiene lugar polimórficamente a través del acoplamiento tardío, podemos escribir el código de forma que se comunique con la clase base, a sabiendas de que todos los casos donde estén involucradas las clases derivadas funcionarán correctamente con el mismo código. O, dicho de otro modo, “enviamos un mensaje a un objeto y dejamos que el objeto averigüe qué es lo que tiene que hacer”.

El ejemplo clásico en la programación orientada a objetos es el de las “formas”. Se suele utilizar comúnmente porque resulta fácil de visualizar, pero lamentablemente puede hacer que los programadores inexpertos piensen que la programación orientada a objetos sólo sirve para la programación gráfica, lo cual, por supuesto, no es cierto.

El ejemplo de las formas tiene una clase base denominada **Shape** (forma) y varios tipos derivados: **Circle** (circulo), **Square** (cuadrado), **Triangle** (triángulo), etc. La razón por la que este ejemplo es tan adecuado es porque es fácil decir “un circulo es un tipo de forma” y que el lector lo entienda. El diagrama de herencia muestra las relaciones:



La generalización puede tener lugar en una instrucción tan simple como la siguiente:

```
s = new Circle();
```

Aquí, se crea un objeto **Circle**, y la referencia resultante se asigna inmediatamente a un objeto **Shape**, (lo que podría parecer un error asignar un tipo a otro); sin embargo, es perfectamente correcto, porque un objeto **Circle** *es* una forma (**Shape**) debido a la herencia. Por tanto, el compilador aceptará la instrucción y no generará ningún mensaje de error.

Suponga que invoca uno de los métodos de la clase base (que han sido sustituidos en las clases derivadas):

```
s.draw();
```

De nuevo, cabría esperar que se invocara el método **draw()** de **Shape** porque, después de todo, esto es una referencia a **Shape**, así que ¿cómo podría el compilador hacer cualquier otra cosa? Sin embargo, se invoca el método apropiado **Circle.draw()** debido al acoplamiento tardío (polimorfismo).

El siguiente ejemplo presenta las formas de una manera ligeramente distinta. En primer lugar, vamos a crear una biblioteca reutilizable de tipos **Shape**:

```

//: polymorphism/shape/Shape.java
package polymorphism.shape;

public class Shape {
    public void draw() {}
    public void erase() {}
} //:~

//: polymorphism/shape/Circle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Circle extends Shape {
    public void draw() { print("Circle.draw()"); }
    public void erase() { print("Circle.erase()"); }
} //:~

//: polymorphism/shape/Square.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Square extends Shape {
    public void draw() { print("Square.draw()"); }
}
  
```

```

    public void erase() { print("Square.erase()"); }
} //:-~


//: polymorphism/shape/Triangle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Triangle extends Shape {
    public void draw() { print("Triangle.draw()"); }
    public void erase() { print("Triangle.erase()"); }
} //:-~


//: polymorphism/shape/RandomShapeGenerator.java
// Una "fábrica" que genera formas aleatoriamente.
package polymorphism.shape;
import java.util.*;

public class RandomShapeGenerator {
    private Random rand = new Random(47);
    public Shape next() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
} //:-~


//: polymorphism/Shapes.java
// Polimorfismo en Java.
import polymorphism.shape.*;

public class Shapes {
    private static RandomShapeGenerator gen =
        new RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Rellena la matriz con formas:
        for(int i = 0; i < s.length; i++)
            s[i] = gen.next();
        // Realiza llamadas a métodos polimórficos:
        for(Shape shp : s)
            shp.draw();
    }
} /* Output:
Triangle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Circle.draw()
*//:-~

```

La clase base **Shape** establece la interfaz común para cualquier otra clase que herede de **Shape**; en términos conceptuales, representa a todas las formas que puedan dibujarse y borrarse. Cada clase derivada sustituye estas definiciones con el fin de proporcionar un comportamiento distintivo para cada tipo específico de forma.

RandomShapeGenerator es una especie de “fábrica” que genera una referencia a un objeto **Shape** aleatoriamente seleccionado cada vez que se invoca su método **next()**. Observe que el *upcasting* se produce en las instrucciones **return**, cada una de las cuales toma una referencia a **Circle**, **Square** o **Triangle** y la devuelve desde **next()** con el tipo de retorno, **Shape**. Por tanto, cada vez que se invoca **next()**, nunca tenemos la oportunidad de ver de qué tipo específico se trata, ya que siempre obtenemos una referencia genérica a **Shape**.

main() contiene una matriz de referencias **Shape** que se rellena mediante llamadas a **RandomShapeGenerator.next()**. En este punto, sabemos que tenemos objetos **Shape**, pero no podemos ser más específicos (ni tampoco puede serlo el compilador). Sin embargo, cuando recorremos esta matriz e invocamos **draw()** para cada objeto, tiene lugar el comportamiento correspondiente a cada tipo específico, como por arte de magia, tal y como puede ver si analiza la salida que se obtiene al ejecutar el programa.

La razón de crear las formas aleatoriamente es que así puede percibirse mejor que el compilador no puede tener ningún conocimiento especial que le permite hacer las llamadas correctas en tiempo de compilación. Todas las llamadas a **draw()** deben tener lugar mediante el mecanismo de acoplamiento dinámico.

Ejercicio 2: (1) Añada la anotación **@Override** al ejemplo de procesamiento de formas.

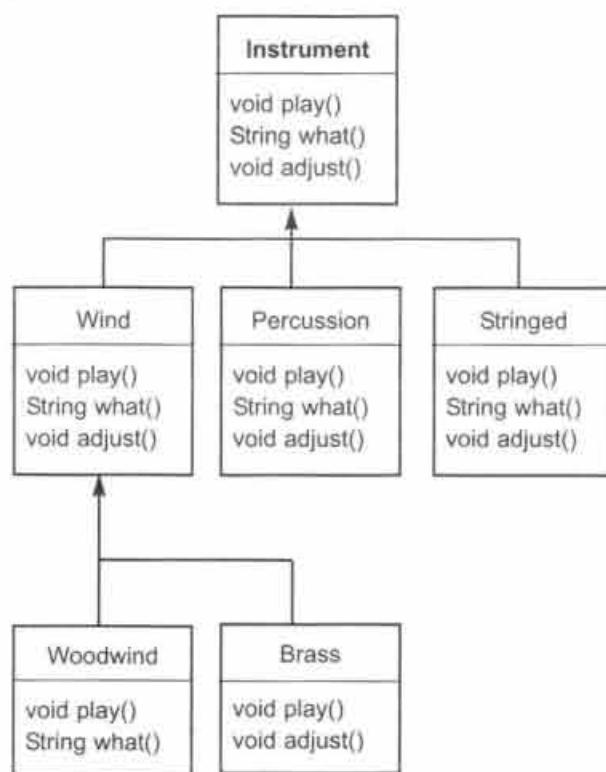
Ejercicio 3: (1) Añada un nuevo método a la clase base de **Shapes.java** que imprima un mensaje, pero sin sustituirlo en las clases derivadas. Explique lo que sucede. Ahora, sustitúyalo en una de las clases derivadas pero no en las otras y vea lo que sucede. Finalmente, sustitúyalo en todas las clases derivadas.

Ejercicio 4: (2) Añada un nuevo tipo de objeto **Shape** a **Shapes.java** y verifique en **main()** que el polimorfismo funciona para el nuevo tipo al igual que para los tipos anteriores.

Ejercicio 5: (1) Partiendo del Ejercicio 1, añada un método **wheels()** a **Cycle**, que devuelva el número de ruedas. Modifique **ride()** para invocar **wheels()** y verifique que funciona el polimorfismo.

Ampliabilidad

Volvamos ahora al ejemplo de los instrumentos musicales. Debido al polimorfismo, podemos añadir al sistema todos los nuevos tipos que deseemos sin modificar el método **tune()**. En un programa orientado a objetos bien diseñado, la mayoría de los métodos (o todos ellos) seguirán el método de **tune()** y sólo se comunicarán con la interfaz de la clase base. Ese tipo



de programas es *extensible (ampliable)* porque puede añadir nueva funcionalidad heredando nuevos tipos de datos a partir de la clase base común. Los métodos que manipulan la interfaz de la clase base no necesitarán ser modificados para poder utilizar las nuevas clases.

Considere lo que sucede si tomamos el ejemplo de los instrumentos y añadimos más métodos a la clase base y una serie de clases nuevas. Puede ver el diagrama correspondiente al final de la página anterior.

Todas estas nuevas clases funcionan correctamente con el método antiguo `tune()`, sin necesidad de modificarlo. Incluso si `tune()` se encontrara en un archivo separado y añadiéramos nuevos métodos a la interfaz de `Instrument`, `tune()` seguiría funcionando correctamente, sin necesidad de recompilarlo. He aquí la implementación del diagrama:

```
//: polymorphism/music3/Music3.java
// Un programa ampliable.
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
    void play(Note n) { print("Instrument.play() " + n); }
    String what() { return "Instrument"; }
    void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
    void play(Note n) { print("Wind.play() " + n); }
    String what() { return "Wind"; }
    void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
    void play(Note n) { print("Percussion.play() " + n); }
    String what() { return "Percussion"; }
    void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
    void play(Note n) { print("Stringed.play() " + n); }
    String what() { return "Stringed"; }
    void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
    void play(Note n) { print("Brass.play() " + n); }
    void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
    void play(Note n) { print("Woodwind.play() " + n); }
    String what() { return "Woodwind"; }
}

public class Music3 {
    // No importa el tipo, por lo que los nuevos
    // tipos añadidos al sistema funcionan bien:
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
```

```

        tune(i);
    }
    public static void main(String[] args) {
        // Upcasting durante la adición a la matriz:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:-

```

Los nuevos métodos son **what()**, que devuelve una referencia **String** con una descripción de la clase y **adjust()**, que proporciona alguna forma de ajustar cada instrumento.

En **main()**, cuando insertamos algo dentro de la matriz **orchestra**, se produce automáticamente una generalización a **Instrument**.

Podemos ver que el método **tune()** es completamente ignorante de todos los cambios de código que han tenido lugar alrededor suyo, a pesar de lo cual sigue funcionando perfectamente. Ésta es, exactamente, la funcionalidad que se supone que el polimorfismo debe proporcionar. Los cambios en el código no generan ningún problema en aquellas partes del programa que no deban verse afectadas. Dicho de otra forma, el polimorfismo es una técnica importante con la que el programador puede "separar" las cosas que cambian de las cosas que permanecen.

Ejercicio 6: (1) Modifique **Music3.java** de modo que **what()** se convierta en el método **toString()** del objeto raíz **Object**. Pruebe a imprimir los objetos **Instrument** utilizando **System.out.println()** (sin efectuar ninguna proyección de tipo).

Ejercicio 7: (2) Añada un nuevo tipo de objeto **Instrument** a **Music3.java** y verifique que el polimorfismo funciona para el nuevo tipo.

Ejercicio 8: (2) Modifique **Music3.java** para que genere aleatoriamente objetos **Instrument** de la misma forma que lo hace **Shapes.java**.

Ejercicio 9: (3) Cree una jerarquía de herencia **Rodent**: **Mouse**, **Gerbil**, **Hamster**, etc (roedor: ratón, jerbo, hamster, etc.). En la clase base proporcione los métodos que son comunes para todos los roedores, y sustituya estos métodos en las clases derivadas para obtener diferentes comportamientos dependiendo del tipo específico de roedor. Cree una matriz de objetos **Rodent**, rellénela con diferentes tipos específicos de roedores e invoque los métodos de la clase base para ver lo que sucede.

Ejercicio 10: (3) Cree una clase base con dos métodos. En el primer método, invoque el segundo método. Defina una clase que herede de la anterior y sustituya el segundo método. Cree un objeto de la clase derivada, realice una generalización (*upcasting*) al tipo base y llame al primer método. Explique lo que sucede.

Error: "sustitución" de métodos private

He aquí un ejemplo de error de un programa que se puede cometer de manera inadvertida:

```

//: polymorphism/PrivateOverride.java
// Intento de sustituir un método privado.
package polymorphism;
import static net.mindview.util.Print.*;

```

```

public class PrivateOverride {
    private void f() { print("private f()"); }
    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

class Derived extends PrivateOverride {
    public void f() { print("public f()"); }
} /* Output:
private f()
*///:-
```

Podría esperar, razonablemente, que la salida fuera “**public f()**”, pero los métodos privados son automáticamente de tipo **final**, y están también ocultos a ojos de la clase derivada. Por esta razón, el método **f()** de la clase derivada es, en este caso, un método completamente nuevo, ni siquiera está sobrecargado, ya que la versión de **f()** en la clase base no es visible en **Derived**.

El resultado de esto es que sólo los métodos no privados pueden ser sustituidos, así que hay que estar atento al intento incorrecto de sustituir métodos de tipo **private**, ya que esos intentos no generan ninguna advertencia del compilador, sino que el sistema no hará, seguramente, lo que se espera. Para evitar las confusiones, conviene utilizar en la clase derivada un nombre diferente al del método **private** de la clase base.

Error: campos y métodos static

Una vez familiarizados con el tema del polimorfismo, podemos tender a pensar que todo ocurre polimórficamente. Sin embargo, las únicas llamadas que pueden ser polimórficas son las llamadas a métodos normales. Por ejemplo, si accedemos a un campo directamente, ese acceso se resolverá en tiempo de compilación, como se ilustra en el siguiente ejemplo:¹

```

//: polymorphism/FieldAccess.java
// El acceso directo a un campo se determina en tiempo de compilación.

class Super {
    public int field = 0;
    public int getField() { return field; }
}

class Sub extends Super {
    public int field = 1;
    public int getField() { return field; }
    public int getSuperField() { return super.field; }
}

public class FieldAccess {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
            ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " +
            sub.field + ", sub.getField() = " +
            sub.getField() +
            ", sub.getSuperField() = " +
            sub.getSuperField());
```

¹ Gracias a Randy Nichols por plantear esta cuestión.

```

    }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:-
```

Cuando un objeto **Sub** se generaliza a una referencia **Super**, los accesos a los campos son resueltos por el compilador, por lo que no son polimórficos. En este ejemplo, hay asignado un espacio de almacenamiento distinto para **Super.field** y **Sub.field**. Por tanto, **Sub** contiene realmente dos campos denominados **field**: el suyo propio y el que obtiene a partir de **Super**. Sin embargo, cuando se hace referencia al campo **field** de **Super** no se genera de forma predeterminada una referencia a la versión almacenada en **Super**; para poder acceder al campo **field** de **Super** es necesario escribir explícitamente **super.field**.

Aunque esto último pueda parecer algo confuso, en la práctica no llega a plantearse casi nunca, por una razón: por regla general, se definen todos los campos como **private**, por lo que no se accede a ellos directamente, sino sólo como efecto secundario de la invocación a métodos. Además, probablemente nunca le demos el mismo nombre de la clase base a un campo de la clase derivada, ya que eso resultaría muy confuso.

Si un método es de tipo **static**, no se comporta de forma polimórfica:

```

//: polymorphism/StaticPolymorphism.java
// Los métodos estáticos no son polimórficos.

class StaticSuper {
    public static String staticGet() {
        return "Base staticGet()";
    }
    public String dynamicGet() {
        return "Base dynamicGet()";
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return "Derived staticGet()";
    }
    public String dynamicGet() {
        return "Derived dynamicGet()";
    }
}

public class StaticPolymorphism {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Generalización
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
} /* Output:
Base staticGet()
Derived dynamicGet()
*///:-
```

Los métodos estáticos están asociados con la clase y no con los objetos individuales.

Constructores y polimorfismo

Como suele suceder, los constructores difieren de los otros tipos de métodos, también en lo que respecta al polimorfismo. Aunque los constructores no son polimórficos (se trata realmente de métodos estáticos, pero la declaración **static** es implícita), tiene gran importancia comprender cuál es la forma en que funcionan los constructores dentro de las jerarquías complejas y en presencia de polimorfismo. Esta comprensión de los fundamentos nos ayudará a evitar errores desagradables.

Orden de las llamadas a los constructores

Hemos hablado brevemente del orden de las llamadas a los constructores en el Capítulo 5, *Inicialización y limpieza*, y también el Capítulo 7, *Reutilización de clases*, pero eso fue antes de introducir el concepto de polimorfismo.

El constructor de la clase base siempre se invoca durante el proceso de construcción correspondiente a una clase derivada. Esta llamada provoca un desplazamiento automático hacia arriba en la jerarquía de herencia, invocándose un constructor para todas las clases base. Esto tiene bastante sentido, porque el constructor tiene asignada una tarea especial: garantizar que el objeto se construye apropiadamente. Una clase derivada sólo tiene acceso a sus propios miembros y no a los de la clase base (aquellos miembros típicamente de tipo **private**). Sólo el constructor de la clase base dispone del conocimiento y del acceso adecuados para inicializar sus propios elementos. Por tanto, resulta esencial que se invoquen todos los constructores, en caso contrario, no podría construirse el método completo. Ésta es la razón por la que el compilador impone que se realice una llamada al constructor para cada parte de una clase derivada. Si no especificamos explícitamente una llamada a un constructor de la clase base dentro del cuerpo de la clase derivada, el compilador invocará de manera automática el constructor predeterminado. Si no hay ningún constructor predeterminado, el compilador generará un error (en aquellos casos en que una determinada clase no tenga ningún constructor, el compilador sintetizará automáticamente un constructor predeterminado).

Veamos un ejemplo que muestra los efectos de la composición, de la herencia y del polimorfismo sobre el orden de construcción:

```
//: polymorphism/Sandwich.java
// Orden de las llamadas a los constructores.
package polymorphism;
import static net.mindview.util.Print.*;

class Meal {
    Meal() { print("Meal()"); }
}

class Bread {
    Bread() { print("Bread()"); }
}

class Cheese {
    Cheese() { print("Cheese()"); }
}

class Lettuce {
    Lettuce() { print("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { print("PortableLunch()"); }
}

public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { print("Sandwich()"); }
    public static void main(String[] args) {
        new Sandwich();
    }
} /* Output:
```

```

Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
*///:-
```

Este ejemplo crea una clase compleja a partir de otras clases y cada una de estas clases dispone de un constructor que se anuncia a sí mismo. La clase importante es **Sandwich**, que refleja tres niveles de herencia (cuatro si contamos la herencia implícita a partir de **Object**) y tres objetos miembro. Podemos ver en **main()** la salida cuando se crea un objeto **Sandwich**. Esto quiere decir que el orden de llamada a los constructores para un objeto complejo es el siguiente:

1. Se invoca al constructor de la clase base. Este paso se repite de forma recursiva de modo que la raíz de la jerarquía se construye en primer lugar, seguida de la siguiente clase derivada, etc., hasta alcanzar la clase situada en el nivel más profundo de la jerarquía.
2. Los inicializadores de los miembros se invocan según el orden de declaración.
3. Se invoca el cuerpo del constructor de la clase derivada.

El orden de las llamadas a los constructores es importante. Cuando utilizamos los mecanismos de herencia, sabemos todo acerca de la clase base y podemos acceder a los miembros de tipo **public** y **protected** de la misma. Esto quiere decir que debemos poder asumir que todos los demás miembros de la clase base son válidos cuando los encontramos en la clase derivada. En un método normal, el proceso de construcción ya ha tenido lugar, de modo que todos los miembros de todas las partes del objeto habrán sido construidos. Sin embargo, dentro del constructor debemos poder estar seguros de que todos los miembros que utilicemos hayan sido construidos. La única forma de garantizar esto es invocando primero al constructor de la clase base. Entonces, cuando nos encontramos dentro del constructor de la clase derivada, todos los miembros de la clase base a los que queremos acceder ya habrán sido inicializados. Saber que todos los miembros son válidos dentro del constructor es también la razón de que, siempre que sea posible, se deban inicializar todos los objetos miembro (los objetos incluidos en la clase mediante los mecanismos de composición) en su punto de definición dentro de la clase (por ejemplo, **b**, **c** y **l** en el ejemplo anterior). Si se ajusta a esta práctica a la hora de programar, le será más fácil garantizar que todos los miembros de la clase base y objetos miembro del objeto actual hayan sido inicializados. Lamentablemente, este sistema no nos permite gestionar todos los casos, como veremos en la siguiente sección.

Ejercicio 11: (1) Añada una clase **Pickle** a **Sandwich.java**.

Herencia y limpieza

Cuando se utilizan los mecanismos de composición y de herencia para crear una nueva clase, la mayor parte de las veces no tenemos que preocuparnos por las tareas de limpieza; los subobjetos pueden normalmente dejarse para que los procese el depurador de memoria. Sin embargo, si hay algún problema relativo a la limpieza, es necesario actuar con diligencia y crear un método **dispose()** (éste es el nombre que yo he seleccionado, pero usted puede utilizar cualquier otro que indique que estamos deshaciéndonos del objeto) en la nueva clase. Y, con la herencia, es necesario sustituir **dispose()** en la clase derivada si necesitamos realizar alguna tarea de limpieza especial que tenga que tener lugar como parte de la depuración de memoria. Cuando se sustituya **dispose()** en una clase heredada, es importante acordarse de invocar la versión de **dispose()** de la clase base, ya que en caso contrario las tareas de limpieza propias de la clase base no se llevarán a cabo. El siguiente ejemplo ilustra esta situación:

```

//: polymorphism/Frog.java
// Limpieza y herencia.
package polymorphism;
import static net.mindview.util.Print.*;

class Characteristic {
    private String s;
    Characteristic(String s) {
        this.s = s;
```

```

        print("Creating Characteristic " + s);
    }
    protected void dispose() {
        print("disposing Characteristic " + s);
    }
}

class Description {
    private String s;
    Description(String s) {
        this.s = s;
        print("Creating Description " + s);
    }
    protected void dispose() {
        print("disposing Description " + s);
    }
}

class LivingCreature {
    private Characteristic p =
        new Characteristic("is alive");
    private Description t =
        new Description("Basic Living Creature");
    LivingCreature() {
        print("LivingCreature()");
    }
    protected void dispose() {
        print("LivingCreature dispose");
        t.dispose();
        p.dispose();
    }
}

class Animal extends LivingCreature {
    private Characteristic p =
        new Characteristic("has heart");
    private Description t =
        new Description("Animal not Vegetable");
    Animal() { print("Animal()"); }
    protected void dispose() {
        print("Animal dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

class Amphibian extends Animal {
    private Characteristic p =
        new Characteristic("can live in water");
    private Description t =
        new Description("Both water and land");
    Amphibian() {
        print("Amphibian()");
    }
    protected void dispose() {
        print("Amphibian dispose");
        t.dispose();
    }
}

```

```

    p.dispose();
    super.dispose();
}

}

public class Frog extends Amphibian {
    private Characteristic p = new Characteristic("Croaks");
    private Description t = new Description("Eats Bugs");
    public Frog() { print("Frog()"); }
    protected void dispose() {
        print("Frog dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        Frog frog = new Frog();
        print("Bye!");
        frog.dispose();
    }
} /* Output:
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
disposing Description Eats Bugs
disposing Characteristic Croaks
Amphibian dispose
disposing Description Both water and land
disposing Characteristic can live in water
Animal dispose
disposing Description Animal not Vegetable
disposing Characteristic has heart
LivingCreature dispose
disposing Description Basic Living Creature
disposing Characteristic is alive
*//*:-

```

Cada clase de la jerarquía también contiene objetos miembro de los tipos **Characteristic** y **Description**, que también habrá que borrar. El orden de borrado debe ser el inverso del orden de inicialización, por si acaso uno de los subobjetos depende del otro. Para los campos, esto quiere decir el inverso del orden de declaración (puesto que los campos se inicializan en el orden de declaración). Para las clases base (siguiendo la norma utilizada en C++ para los destructores), debemos realizar primero las tareas de limpieza de la clase derivada y luego las de la clase base. La razón es que esas tareas de limpieza de la clase derivada tuvieron que invocar algunos métodos de la clase base que requieran que los componentes de la clase base continúen siendo accesibles, así que no debemos destruir esos componentes prematuramente. Analizando la salida podemos ver que se borran todas las partes del objeto **Frog** en orden inverso al de creación.

A partir de este ejemplo, podemos ver que aunque no siempre es necesario realizar tareas de limpieza, cuando se llevan a cabo es preciso hacerlo con un gran cuidado y una gran atención.

Ejercicio 12: (3) Modifique el Ejercicio 9 para que se muestre el orden de inicialización de las clases base y de las clases derivadas. Ahora añada objetos miembro a las clases base y derivadas, y muestre el orden en que se lleva a cabo la inicialización durante el proceso de construcción.

Observe también en el ejemplo anterior que un objeto **Frog** “posee” sus objetos miembro: crea esos objetos miembro y sabe durante cuánto tiempo tienen que existir (tanto como dure el objeto **Frog**), de modo que sabe cuándo invocar el método **dispose()** para borrar los objetos miembro. Sin embargo, si uno de estos objetos miembro es compartido con otros objetos, el problema se vuelve más complejo y no podemos simplemente asumir que basta con invocar **dispose()**. En estos casos, puede ser necesario un *recuento de referencias* para llevar la cuenta del número de objetos que siguen pudiendo acceder a un objeto compartido. He aquí un ejemplo:

```
//: polymorphism/ReferenceCounting.java
// Limpieza de objetos miembro compartidos.
import static net.mindview.util.Print.*;

class Shared {
    private int refcount = 0;
    private static long counter = 0;
    private final long id = counter++;
    public Shared() {
        print("Creating " + this);
    }
    public void addRef() { refcount++; }
    protected void dispose() {
        if(--refcount == 0)
            print("Disposing " + this);
    }
    public String toString() { return "Shared " + id; }
}

class Composing {
    private Shared shared;
    private static long counter = 0;
    private final long id = counter++;
    public Composing(Shared shared) {
        print("Creating " + this);
        this.shared = shared;
        this.shared.addRef();
    }
    protected void dispose() {
        print("disposing " + this);
        shared.dispose();
    }
    public String toString() { return "Composing " + id; }
}

public class ReferenceCounting {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Composing[] composing = { new Composing(shared),
            new Composing(shared), new Composing(shared),
            new Composing(shared), new Composing(shared) };
        for(Composing c : composing)
            c.dispose();
    }
} /* Output:
Creating Shared 0
Creating Composing 0
Creating Composing 1
```

```

Creating Composing 2
Creating Composing 3
Creating Composing 4
disposing Composing 0
disposing Composing 1
disposing Composing 2
disposing Composing 3
disposing Composing 4
Disposing Shared 0
*///:-
```

El contador **static long counter** lleva la cuenta del número de instancias de **Shared** que son creadas y también crea un valor para **id**. El tipo de **counter** es **long** en lugar de **int**, para evitar el desbordamiento (se trata sólo de una buena práctica de programación; es bastante improbable que esos desbordamientos de contadores puedan producirse en ninguno de los ejemplos de este libro). La variable **id** es de tipo **final** porque no esperamos que cambie de valor durante el tiempo de vida del objeto.

Cuando se asocia el objeto compartido a la clase, hay que acordarse de invocar **addRef()**, pero el método **dispose()** llevará la cuenta del número de referencias y decidirá cuándo hay que proceder con las tareas de limpieza. Esta técnica requiere un cierta diligencia por nuestra parte, pero si estamos compartiendo objetos que necesiten que se lleve a cabo una determinada tarea de limpieza, no son muchas las opciones que tenemos.

Ejercicio 13: (3) Añada un método **finalize()** a **ReferenceCounting.java** para verificar la condición de terminación (véase el Capítulo 5, *Inicialización y limpieza*).

Ejercicio 14: (4) Modifique el Ejercicio 12 para que uno de los objetos miembro sea un objeto compartido. Utilice el método de recuento del número de referencias y demuestre que funciona adecuadamente.

Comportamiento de los métodos polimórficos dentro de los constructores

La jerarquía de llamada a constructores plantea un dilema interesante. ¿Qué sucede si estamos dentro de un constructor e invocamos un método con acoplamiento dinámico del objeto que esté siendo construido?

Dentro de un método normal, la llamada con acoplamiento dinámico se resuelve en tiempo de ejecución, porque el objeto no puede saber si pertenece a la clase en la que se encuentra el método o a alguna de las clases derivadas de la misma.

Si invocamos un método con acoplamiento dinámico dentro de un constructor, también se utiliza la definición sustituida de dicho método (es decir, la definición del método que se encuentra en la clase actual). Sin embargo, el efecto de esta llamada puede ser inesperado, porque el método sustituido será invocado antes de que el objeto haya sido completamente construido. Esto puede hacer que queden ocultos algunos errores realmente difíciles de detectar.

Conceptualmente, la tarea del constructor es hacer que el objeto comience a existir (lo que no es una tarea trivial). Dentro de cualquier constructor, puede que el objeto completo sólo esté formado parcialmente, ya que de lo único que podemos estar seguros es de que los objetos de la clase base han sido inicializados. Si el constructor es sólo uno de los pasos a la hora de construir un objeto de una clase que haya sido derivada de la clase correspondiente a dicho constructor, las partes derivadas no habrán sido todavía inicializadas en el momento en que se invoque al constructor actual. Sin embargo, una llamada a un método con acoplamiento dinámico se "adentra" en la jerarquía de herencia, invocando un método dentro de una clase derivada. Si hacemos esto dentro de un constructor, podríamos estar invocando un método que manipulara miembros que todavía no han sido inicializados, lo cual constituye una receta segura para que se produzca un desastre.

Podemos ver el problema en el siguiente ejemplo:

```

//: polymorphism/PolyConstructors.java
// Los constructores en presencia de polimorfismo
// pueden no producir los resultados esperados.
import static net.mindview.util.Print.*;

class Glyph {
    void draw() { print("Glyph.draw()"); }
    Glyph() {
```

```

        print("Glyph() before draw()");
        draw();
        print("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        print("RoundGlyph.RoundGlyph()", radius = " + radius);
    }
    void draw() {
        print("RoundGlyph.draw()", radius = " + radius);
    }
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
} /* Output:
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
*///:-

```

Glyph.draw() está diseñado para ser sustituido, lo que se produce en **RoundGlyph**. Pero el constructor de **Glyph** invoca este método y la llamada termina en **RoundGlyph.draw()**, que parece que fuera la intención original. Pero si examinamos la salida, podemos ver que cuando el constructor de **Glyph** invoca **draw()**, el valor de **radius** no es ni siquiera el valor inicial predeterminado de 1, sino que es 0. Esto provocará, probablemente, que se dibuje en la pantalla un punto, o nada en absoluto, con lo que el programador se quedará contemplándolo tratando de imaginar por qué no funciona el programa.

El orden de inicialización descrito en la sección anterior no está completo del todo, y ahí es donde radica la clave para resolver el misterio. El proceso real de inicialización es:

1. El almacenamiento asignado al objeto se inicializa con ceros binarios antes de que suceda ninguna otra cosa.
2. Los constructores de las clases base se invocan tal y como hemos descrito anteriormente. En este punto se invoca el método sustituido **draw()** (si, se invoca *antes* de que llame al constructor de **RoundGlyph**) y éste descubre que el valor de **radius** es cero, debido al Paso 1.
3. Los inicializadores de los miembros se invocan según el orden de declaración.
4. Se invoca el cuerpo del constructor de la clase derivada.

La parte buena de todo esto es que todo se inicializa al menos con cero (o con lo que cero signifique para ese tipo de datos concreto) y no simplemente con datos aleatorios. Esto incluye las referencias a objetos que han sido incluidas en una clase a través del mecanismo de composición, que tendrán el valor **null**. Por tanto, si nos olvidamos de inicializar esa referencia, se generará una excepción en tiempo de ejecución. Todo lo demás tomará el valor cero, lo que usualmente nos sirve como pista a la hora de examinar la salida.

Por otro lado, es posible que el programador se quede horrorizado al ver la salida de este programa: hemos hecho algo perfectamente lógico, a pesar de lo cual el comportamiento es misteriosamente erróneo, sin que el compilador se haya quejado (C++ produce un comportamiento más racional en esta situación). Los errores de este tipo podrían quedar ocultos fácilmente, necesitándose una gran cantidad de tiempo para descubrirlos.

Como resultado, una buena directriz a la hora de implementar los constructores es: "Haz lo menos posible para garantizar que el objeto se encuentre en un estado correcto y, siempre que puedas evitarlo, no invoques ningún otro método de esta clase". Los únicos métodos seguros que se pueden invocar dentro de un constructor son aquellos de tipo **final** en la clase

base (esto también se aplica a los métodos privados, que son automáticamente de tipo **final**). Estos métodos no pueden ser sustituidos y no pueden, por tanto, darnos este tipo de sorpresas. Puede que no siempre seamos capaces de seguir esta directriz, pero al menos debemos tratar de cumplirla.

Ejercicio 15: (2) Añada una clase **RectangularGlyph** a **PolyConstructors.java** e ilustre el problema descrito en esta sección.

Tipos de retorno covariantes

Java SE5 añade los denominados *tipos de retorno covariantes*, lo que quiere decir que un método sustituido en una clase derivada puede devolver un tipo derivado del tipo devuelto por el método de la clase base:

```
//: polymorphism/CovariantReturn.java

class Grain {
    public String toString() { return "Grain"; }
}

class Wheat extends Grain {
    public String toString() { return "Wheat"; }
}

class Mill {
    Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
    Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
    public static void main(String[] args) {
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);
        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
} /* Output:
Grain
Wheat
*///:-
```

La diferencia clave entre Java SE5 y las versiones anteriores es que en éstas se obligaría a que la versión sustituida de **process()** devolviera **Grain**, en lugar de **Wheat**, a pesar de que **Wheat** deriva de **Grain** y sigue siendo, por tanto, un tipo de retorno legítimo. Los tipos de retorno covariantes permiten utilizar el tipo de retorno **Wheat** más específico.

Diseño de sistemas con herencia

Una vez que sabemos un poco sobre el polimorfismo, puede llegar a parecernos que todo debería heredarse, ya que el polimorfismo es una herramienta tan inteligente. Pero la realidad es que esto puede complicar nuestros diseños innecesariamente, de hecho, si decidimos utilizar la herencia como primera opción a la hora de utilizar una clase existente con el fin de formar otra nueva, las cosas pueden volverse innecesariamente complicadas.

Una técnica mejor consiste en tratar de utilizar primero la composición, especialmente cuando no resulte obvio cuál de los dos mecanismos debería emplearse. La composición no hace que el diseño tenga que adoptar una jerarquía de herencia. Pero, asimismo, la composición es más flexible, porque permite seleccionar dinámicamente un tipo (y por tanto un compo-

tamiento), mientras que la herencia exige que se conozca un tipo exacto en tiempo de compilación. El siguiente ejemplo ilustra esto:

```
//: polymorphism/Transmogrify.java
// Modificación dinámica del comportamiento de un objeto
// mediante la composición (el patrón de diseño basado en estados).
import static net.mindview.util.Print.*;

class Actor {
    public void act() {}
}

class HappyActor extends Actor {
    public void act() { print("HappyActor"); }
}

class SadActor extends Actor {
    public void act() { print("SadActor"); }
}

class Stage {
    private Actor actor = new HappyActor();
    public void change() { actor = new SadActor(); }
    public void performPlay() { actor.act(); }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage stage = new Stage();
        stage.performPlay();
        stage.change();
        stage.performPlay();
    }
} /* Output:
HappyActor
SadActor
****/-
```

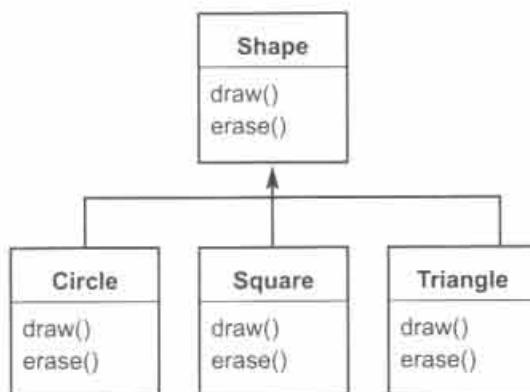
Un objeto **Stage** contiene una referencia a un objeto **Actor**, que se inicializa para que apunte a un objeto **HappyActor**. Esto significa que **performPlay()** produce un comportamiento concreto. Pero, como una referencia puede redirigirse a un objeto distinto en tiempo de ejecución, podríamos almacenar una referencia a un objeto **SadActor** en **actor**, y entonces el comportamiento producido por **performPlay()** variaría. Por tanto, obtenemos una mayor flexibilidad dinámica en tiempo de ejecución (esto se denomina también *patrón de diseño basado en estados*, consulte *Thinking in Patterns (with Java)* en www.MindView.net). Por contraste, no podemos decidir realizar la herencia de forma diferente en tiempo de ejecución, el mecanismo de herencia debe estar perfectamente determinado en tiempo de compilación.

Una regla general sería: “Utilice la herencia para expresar las diferencias en comportamiento y los campos para expresar las variaciones en el estado”. En el ejemplo anterior se utilizan ambos mecanismos; definimos mediante herencia dos clases distintas para expresar la diferencia en el método **act()** y **Stage** utiliza la composición para permitir que su estado sea modificado. Dicho cambio de estado, en este caso, produce un cambio de comportamiento.

Ejercicio 16: (3) Siguiendo el ejemplo de **Transmogrify.java**, cree una clase **Starship** que contenga una referencia **AlertStatus** que pueda indicar tres estados distintos. Incluya métodos para verificar los estados.

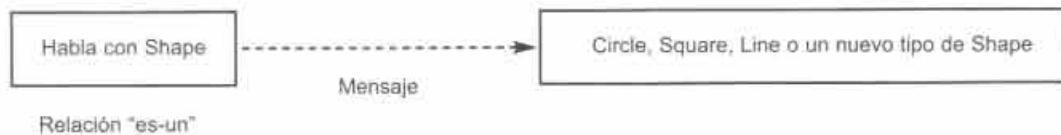
Sustitución y extensión

Podría parecer que la forma más limpia de crear una jerarquía de herencia sería adoptar un enfoque “puro”; es decir, sólo los métodos que hayan sido establecidos en la clase base serán sustituidos en la clase derivada, como puede verse en este diagrama:



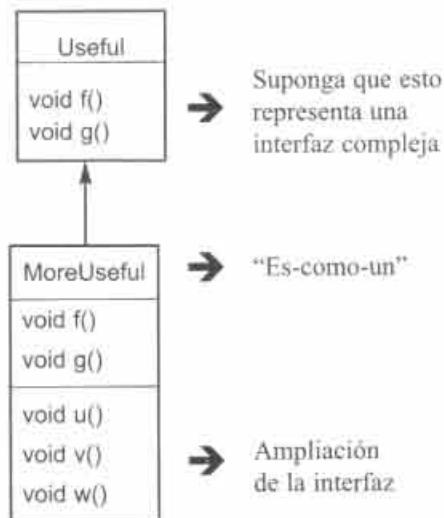
Esto podría decirse que es una relación de tipo “es-un” porque la interfaz de una clase establece lo que dicha clase es. La herencia garantiza que cualquier clase derivada tendrá la interfaz de la clase base y nada más. Si seguimos este diagrama, las clases derivadas *no tendrán nada más* que lo que la interfaz de la clase base ofrece.

Esto podría considerarse como una *sustitución pura*, porque podemos sustituir perfectamente un objeto de la clase base o un objeto de una clase derivada y no nos hace falta conocer ninguna información adicional acerca de las subclases a la hora de utilizarlas:

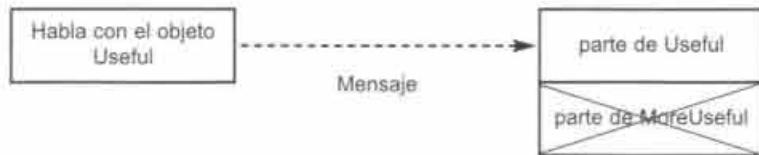


En otras palabras, la clase base puede recibir cualquier mensaje que envíemos a la clase derivada, porque las dos tienen exactamente la misma interfaz. Debido a esto lo que tenemos que hacer es generalizar a partir de la clase derivada, sin tener que preocuparnos de ver cuál es el tipo exacto del objeto con el que estemos tratando. Todo se maneja mediante el polimorfismo.

Cuando vemos las cosas de esta forma, debe parecer que las relaciones puras de tipo “es-un” son la forma más lógica de implementar las cosas, y qué cualquier otro tipo de diseño resulta confuso por comparación. Pero esta forma de pensar es un error. Tan pronto comencemos a pensar de esta forma, miraremos a nuestro alrededor y descubriremos que ampliar la interfaz (mediante la palabra clave **extends**) es la perfecta solución para un problema concreto. Este tipo de solución podría denominarse relación de tipo “es-como-un”, porque la clase derivada *es como* la clase base: tiene la misma interfaz elemental y tiene, además, otras características que requieren métodos adicionales para implementarlas:



Aunque este enfoque también resulta útil y lógico (dependiendo de la situación) tiene una desventaja. La parte ampliada de la interfaz en la clase derivada no está disponible en la clase base, por lo que, una vez que efectuemos una generalización no podremos invocar los nuevos métodos:



Si no estamos haciendo generalizaciones, no debe haber ningún problema, pero a menudo nos encontraremos en situaciones en las que necesitamos descubrir el tipo exacto del objeto para poder acceder a los métodos ampliados de dicho tipo. En la siguiente sección se explica cómo hacer esto.

Especialización e información de tipos en tiempo de ejecución

Puesto que perdemos la información específica del tipo mediante el proceso de *generalización* (*upcast*, que consiste en moverse hacia arriba por la jerarquía de herencia), tiene bastante sentido que para extraer la información de tipos; es decir, para volver a descender por la jerarquía de herencia, utilicemos un proceso de *especialización* (*downcast*). Sin embargo, sabemos que una generalización siempre es segura, porque la clase base no puede tener una interfaz más amplia que la clase derivada; por tanto, se garantiza que todo mensaje que envíemos a través de la interfaz de la clase base será aceptado. Pero con una especialización no sabemos realmente si una determinada forma, por ejemplo, es un círculo u otra cosa: también podría ser un triángulo, un cuadrado o algún otro tipo de forma.

Para resolver este problema, tiene que haber alguna manera de garantizar que la especialización se efectúe de forma correcta, de modo que no hagamos accidentalmente una proyección sobre el tipo inadecuado y luego envíemos un mensaje que el objeto no pueda aceptar. Si no podemos garantizar que la especialización se efectúe de manera correcta, nuestro programa no será muy seguro.

En algunos lenguajes (como C++) es necesario realizar una operación especial para poder llevar a cabo una especialización de tipos de forma correcta, pero en Java *todas* las proyecciones de tipos se comprueban. Por tanto, aunque parezca que estamos utilizando simplemente una proyección de tipos normal, usando paréntesis, dicha proyección se comprueba en tiempo de ejecución para garantizar que se trate, de hecho, del tipo que creemos que es. Si no lo es, se obtiene una excepción **ClassCastException**. Este acto de comprobación de tipos en tiempo de ejecución se denomina *información de tipos en tiempo de ejecución* (RTTI, *runtime type information*). El siguiente ejemplo ilustra el comportamiento de RTTI:

```

//: polymorphism/RTTI.java
// Especialización en información de tipos en tiempo de ejecución (RTTI).
// {ThrowsException}

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
        Useful[] x = {
            new Useful(),
            new MoreUseful()
        };
    }
}
  
```

```

x[0].f();
x[1].g();
// Tiempo de compilación: método no encontrado en Useful:
//! x[1].u();
((MoreUseful)x[1]).u(); // Especialización/RTTI
t((MoreUseful)x[0]).u(); // Excepción generada
}
} //;-

```

Como en el diagrama anterior, **MoreUseful** amplia la interfaz de **Useful**. Pero, como se trata de una clase heredada también puede generalizarse a **Useful**. Podemos ver esta generalización en acción durante la inicialización de la matriz **x** en **main()**. Puesto que ambos objetos de la matriz son de clase **Useful**, podemos enviar los métodos **f()** y **g()** a ambos, mientras que si tratamos de invocar **u()** (que sólo existe en **MoreUseful**), obtendremos un mensaje de error en tiempo de compilación.

Si queremos acceder a la interfaz ampliada de un objeto **MoreUseful**, podemos tratar de efectuar una especialización. Si se trata del tipo correcto, la operación tendrá éxito. En caso contrario, obtendremos una excepción **ClassCastException**. No es necesario escribir ningún código especial para esta excepción, ya que indica un error del programador que puede producirse en cualquier lugar del programa. La etiqueta de comentario **{ThrowsException}** le dice al sistema de construcción de los ejemplos de este libro que cabe esperar que este programa genere una excepción al ejecutarse.

El mecanismo RTTI es más complejo de lo que este ejemplo de proyección simple permite intuir. Por ejemplo, existe una forma de ver cuál es el tipo con el que estamos tratando *antes* de efectuar la especialización. El Capítulo 14, *Información de tipos* está dedicado al estudio de los diferentes aspectos de la información de tipos en tiempo de ejecución en Java.

Ejercicio 17: (2) Utilizando la jerarquía **Cycle** del Ejercicio 1, añada un método **balance()** a **Unicycle** y **Bicycle**, pero no a **Tricycle**. Cree instancias de los tres tipos y generalicelas para formar una matriz de objetos **Cycle**. Trate de invocar **balance()** en cada elemento de la matriz y observe los resultados. Realice una especialización e invoque **balance()** y observe lo que sucede.

Resumen

Polimorfismo significa "diferentes formas". En la programación orientada a objetos, tenemos una misma interfaz definida en la clase base y diferentes formas que utilizan dicha interfaz: las diferentes versiones de los métodos dinámicamente acoplados.

Hemos visto en este capítulo que resulta imposible comprender, o incluso crear, un ejemplo de polimorfismo sin utilizar la abstracción de datos y la herencia. El polimorfismo es una característica que no puede analizarse de manera aislada (a diferencia, por ejemplo, del análisis de la instrucción **switch**), sino que funciona de manera concertada, como parte del esquema global de relaciones de clases.

Para usar el polimorfismo, y por tanto las técnicas de orientación a objetos, de manera efectiva en los programas, es necesario ampliar nuestra visión del concepto de programación, para incluir no sólo los miembros de una clase individual, sino también los aspectos comunes de las distintas clases y las relaciones que tienen entre sí. Aunque esto requiere un esfuerzo significativo, se trata de un esfuerzo que merece la pena. Los resultados serán una mayor velocidad a la hora de desarrollar programas, una mejor organización del código y la posibilidad de disponer de programas ampliables, y un mantenimiento del código más eficiente.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Interfaces

9

Las interfaces y las clases abstractas proporcionan una forma más estructurada de separar la interfaz de la implementación.

Dichos mecanismos no son tan comunes en los lenguajes de programación. C++, por ejemplo, sólo tiene soporte indirecto para estos conceptos. El hecho de que existan palabras clave del lenguaje en Java para estos conceptos indica que esas ideas fueron consideradas lo suficientemente importantes como para proporcionar un soporte directo.

En primer lugar, vamos a examinar el concepto de *clase abstracta*, que es una clase de término medio entre una clase normal y una interfaz. Aunque nuestro primer impulso pudiera ser crear una interfaz, la clase abstracta constituye una herramienta importante y necesaria para construir clases que tengan algunos métodos no implementados. No siempre podemos utilizar una interfaz pura.

Clases abstractas y métodos abstractos

En todos los ejemplos de “instrumentos musicales” del capítulo anterior, los métodos de la clase base **Instrument** eran siempre “ficticios”. Si estos métodos llegan a ser invocados, es que hemos hecho algo mal. La razón es que **Instrument** no tiene otro sentido que crear una *interfaz común* para todas las clases derivadas de ella.

En dichos ejemplos, la única razón para establecer esta interfaz común es para poder expresarla de manera diferente para cada uno de los distintos subtipos. Esa interfaz establece una forma básica, de modo que podemos expresar todo aquello que es común para todas las clases derivadas. Otra forma de decir esto sería decir que **Instrument** es una *clase base abstracta*, o simplemente una *clase abstracta*.

Si tenemos una clase abstracta como **Instrument**, los objetos de dicha clase específica no tienen ningún significado propio casi nunca. Creamos una clase abstracta cuando queremos manipular un conjunto de clases a través de su interfaz común. Por tanto, el propósito de **Instrument** consiste simplemente en expresar la interfaz y no en una implementación concreta, por lo que no tiene sentido crear un objeto **Instrument** y probablemente convenga impedir que el usuario pueda hacerlo. Podemos impedirlo haciendo que todos los métodos de **Instrument** generen errores, pero eso retarda la información hasta el momento de la ejecución y requiere que el usuario realice pruebas exhaustivas y fiables. Generalmente, resulta preferible detectar los problemas en tiempo de compilación.

Java proporciona un mecanismo para hacer esto denominado *método abstracto*.¹ Se trata de un método que es incompleto; sólo tiene una declaración, y no dispone de un cuerpo. He aquí la sintaxis para la declaración de un método abstracto:

```
abstract void f();
```

Una clase que contenga métodos abstractos se denomina *clase abstracta*. Si una clase contiene uno o más métodos abstractos, la propia clase debe calificarse como **abstract**, (en caso contrario, el compilador generará un mensaje de error).

Si una clase abstracta está incompleta, ¿qué es lo que se supone que el compilador debe hacer cuando alguien trate de instanciar un objeto de esa clase? El compilador no puede crear de manera segura un objeto de una clase abstracta, por lo que

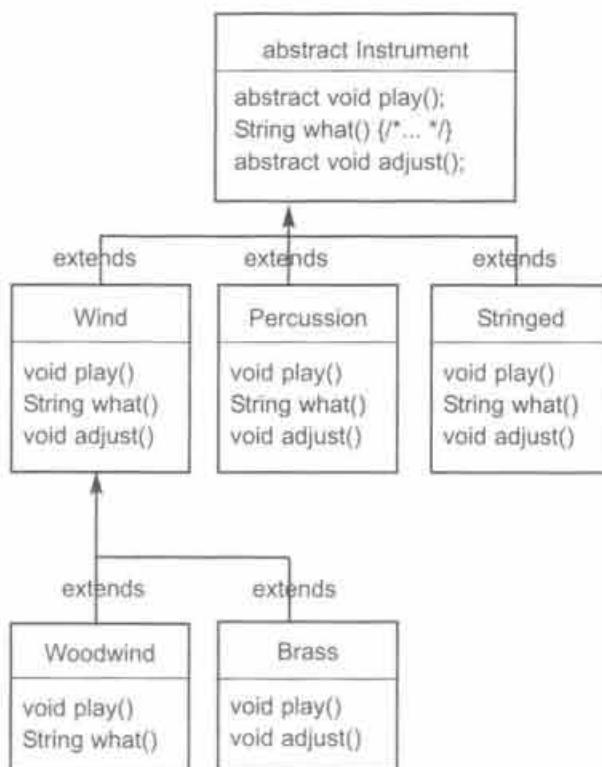
¹ Para los programadores de C++, se trata del análogo a las *funciones virtuales puras* de C++.

generará un mensaje de error. De esta forma, el compilador garantiza la pureza de la clase abstracta y no es necesario preocuparse de si se la va a utilizar correctamente.

Si definimos una clase heredada de una clase abstracta y queremos construir objetos del nuevo tipo, deberemos proporcionar definiciones de métodos para todos los métodos abstractos de la clase base. Si no lo hacemos (y podemos decidir no hacerlo), entonces la clase derivada será también abstracta, y el compilador nos obligará a calificar *esa* clase con la palabra clave **abstract**.

Resulta posible definir una clase como abstracta sin incluir ningún método abstracto. Esto resulta útil cuando tenemos una clase en la que no tiene sentido tener ningún método abstracto y, sin embargo, queremos evitar que se generen instancias de dicha clase.

La clase **Instrument** del capítulo anterior puede transformarse fácilmente en una clase abstracta. Sólo algunos de los métodos serán abstractos, ya que definir una clase como abstracta no obliga a que todos los métodos sean abstractos. He aquí el ejemplo modificado:



He aquí el ejemplo de la orquesta modificado para utilizar clases y métodos abstractos:

```

// interfaces/music4/Music4.java
// Clases y métodos abstractos.
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

abstract class Instrument {
    private int i; // Storage allocated for each
    public abstract void play(Note n);
    public String what() { return "Instrument"; }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play(Note n) {
        print("Wind playing " + Note.name(n));
    }
}
  
```

```

        print("Wind.play() " + n);
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play(Note n) {
        print("Percussion.play() " + n);
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
    public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
    public void play(Note n) {
        print("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    // No me preocupa el tipo, por lo que los nuevos tipos
    // añadidos al sistema seguirán funcionando:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Generalización durante la inserción en la matriz:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
}

```

```

    } /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:-

```

Podemos ver que no se ha efectuado ningún cambio, salvo en la clase base.

Resulta útil crear clases y métodos abstractos porque hacen que la abstracción de una clase sea explícita, e informan tanto al usuario como al compilador acerca de cómo se pretende que se utilice esa clase. Las clases abstractas también resultan útiles como herramientas de rediseño, ya que permiten mover fácilmente los métodos comunes hacia arriba en la jerarquía de herencia.

- Ejercicio 1:** (1) Modifique el Ejercicio 9 del capítulo anterior de modo que **Rodent** sea una clase abstracta. Defina los métodos de **Rodent** como abstractos siempre que sea posible.
- Ejercicio 2:** (1) Cree una clase abstracta sin incluir ningún método abstracto y verifique que no pueden crearse instancias de esa clase.
- Ejercicio 3:** (2) Cree una clase base con un método **print()** abstracto que se sustituye en una clase derivada. La versión sustituida del método debe imprimir el valor de una variable **int** definida en la clase derivada. En el punto de definición de esta variable, proporcione un valor distinto de cero. En el constructor de la clase base, llame a este método. En **main()**, cree un objeto del tipo derivado y luego invoque su método **print()**. Explique los resultados.
- Ejercicio 4:** (3) Cree una clase abstracta sin métodos. Defina una clase derivada y añádale un método. Cree un método estático que tome una referencia a la clase base, especialícelo para que apunte a la clase derivada e invoque el método. En **main()**, demuestre que este mecanismo funciona. Ahora, incluya la declaración abstracta del método en la clase base, eliminando así la necesidad de la especialización.

Interfaces

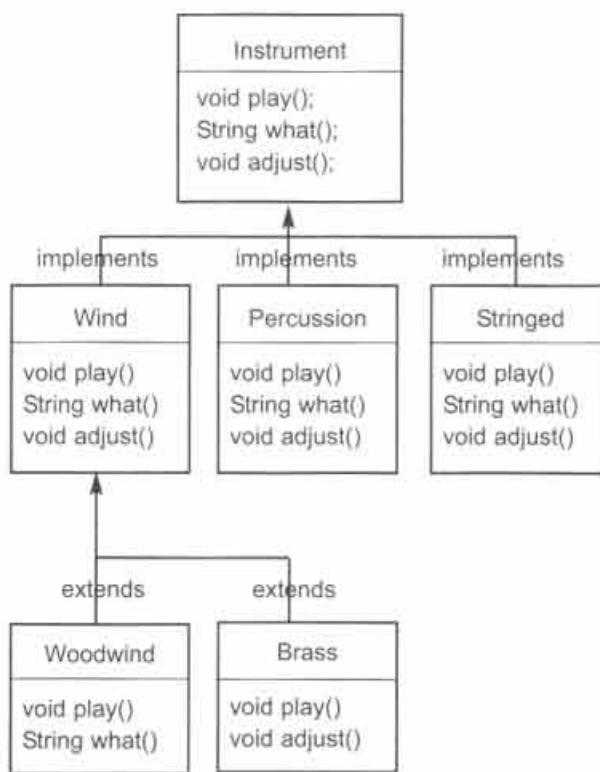
La palabra clave **interface** lleva el concepto de abstracción un paso más allá. La palabra clave **abstract** permite crear uno o más métodos no definidos dentro de una clase: proporcionamos parte de la interfaz, pero sin proporcionar la implementación correspondiente. La implementación se proporciona de las clases que hereden de la clase actual. La palabra clave **interface** produce una clase completamente abstracta, que no proporciona ninguna implementación en absoluto. Las interfaces permiten al creador determinar los nombres de los métodos, las listas de argumentos y los tipos de retorno, pero sin especificar ningún cuerpo de ningún método. Una interfaz proporciona simplemente una forma, sin ninguna implementación.

Lo que las interfaces hacen es decir: "Todas las clases que implementen esta interfaz concreta tendrán este aspecto". Por tanto, cualquier código que utilice una interfaz concreta sabrá qué métodos pueden invocarse para dicha interfaz y eso es todo. Por tanto, la interfaz se utiliza para establecer un "protocolo" entre las clases (algunos lenguajes de programación orientados a objetos disponen de una palabra clave denominada *protocol* para hacer lo mismo).

Sin embargo, una interfaz es algo más que simplemente una clase abstracta llevada hasta el extremo, ya que permite realizar una variante del mecanismo de "herencia múltiple" creando una clase que pueda generalizarse a más de un tipo base.

Para crear una interfaz, utilice la palabra clave **interface** en lugar de **class**. Al igual que con una clase, puede añadir la palabra clave **public** antes de **interface** (pero sólo si dicha interfaz está definida en un archivo del mismo nombre). Si no incluimos la palabra clave **public**, obtendremos un acceso de tipo paquete, porque la interfaz sólo será utilizable dentro del mismo paquete. Una interfaz también puede contener campos, pero esos campos serán implicitamente de tipo **static** y **final**.

Para definir una clase que se adapte a una interfaz concreta (o a un grupo de interfaces concretas), utilice la palabra clave **implements** que quiere decir: "La interfaz especifica cuál es el aspecto, pero ahora vamos a decir cómo *funciona*". Por lo demás, la definición de la clase derivada se asemeja al mecanismo normal de herencia. El diagrama para el ejemplo de los instrumentos musicales sería el siguiente:



Podemos ver en las clases **Woodwind** y **Brass** que una vez que hemos implementado la interfaz, la implementación pasa a ser una clase normal que puede ampliarse de la forma usual.

Podemos declarar explícitamente los métodos de una interfaz como **public**, pero esos métodos serán públicos aún cuando no lo especifiquemos. Por tanto, cuando implementemos una interfaz, los métodos de esa interfaz *deben* estar definidos como públicos. En caso contrario, se revertiría de forma predeterminada al acceso de tipo paquete, con lo que estariamos reduciendo la accesibilidad de los métodos durante la herencia, cosa que el compilador de Java no permite.

Podemos ver esto en la versión modificada del ejemplo **Instrument**. Observe que todos los métodos de la interfaz son estrictamente una declaración, que es lo único que el compilador permite. Además, ninguno de los métodos de **Instrument** se declara como **public**, pero de todos modos son públicos de manera automática:

```

//: interfaces/music5/Music5.java
// Interfaces.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
    // Constante de tiempo de compilación:
    int VALUE = 5; // static & final
    // No puede tener definiciones de métodos:
    void play(Note n); // Automáticamente público
    void adjust();
}

class Wind implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Wind"; }
    public void adjust() { print(this + ".adjust()"); }
}
  
```

```

class Percussion implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Percussion"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Stringed"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
    public String toString() { return "Brass"; }
}

class Woodwind extends Wind {
    public String toString() { return "Woodwind"; }
}

public class Music5 {
    // No le preocupa el tipo, por lo que los nuevos tipos
    // que se añaden al sistema seguirán funcionando:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Generalización durante la inserción en la matriz:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:-

```

En esta versión del ejemplo hemos hecho otro cambio: el método `what()` ha sido cambiado a `toString()`, dado que esa era la forma en que se estaba utilizando el método. Puesto que `toString()` forma parte de la clase raíz `Object`, no necesita aparecer en la interfaz.

El resto del código funciona de la misma manera. Observe que no importa si estamos generalizando a una clase “normal” denominada `Instrument`, a una clase abstracta llamada `Instrument`, o a una interfaz denominada `Instrument`. El compor-

tamiento es siempre el mismo. De hecho, podemos ver en el método `tune()` que no existe ninguna evidencia acerca de si `Instrument` es una clase "normal", una clase abstracta o una interfaz.

- Ejercicio 5:** (2) Cree una interfaz que contenga tres métodos en su propio paquete. Implemente la interfaz en un paquete diferente.
- Ejercicio 6:** (2) Demuestre que todos los métodos de una interfaz son automáticamente públicos.
- Ejercicio 7:** (1) Modifique el Ejercicio 9 del Capítulo 8, *Polimorfismo*, para que `Rodent` sea una interfaz.
- Ejercicio 8:** (2) En `polymorphism.Sandwich.java`, cree una interfaz denominada `FastFood` (con los métodos apropiados) y cambie `Sandwich` de modo que también implemente `FastFood`.
- Ejercicio 9:** (3) Rediseñe `Music5.java` moviendo los métodos comunes de `Wind`, `Percussion` y `Stringed` a una clase abstracta.
- Ejercicio 10:** (3) Modifique `Music5.java` añadiendo una interfaz `Playable`. Mueva la declaración de `play()` de `Instrument` a `Playable`. Añada `Playable` a las clases derivadas incluyéndola en la lista `implements`. Modifique `tune()` de modo que acepte un objeto `Playable` en lugar de un objeto `Instrument`.

Desacoplamiento completo

Cuando un método funciona con una clase en lugar de con una interfaz, estamos limitados a utilizar dicha clase o sus subclases. Si quisieramos aplicar ese método a una clase que no se encontrara en esa jerarquía, no podríamos. Las interfaces relajan esta restricción considerablemente. Como resultado, permiten escribir código más reutilizable.

Por ejemplo, suponga que disponemos de una clase `Processor` que tiene sendos métodos `name()` y `process()` que toman una cierta entrada, la modifican y generan una salida. La clase base se puede ampliar para crear diferentes tipos de objetos `Processor`. En este caso, los subtipos de `Processor` modifican objetos de tipo `String` (observe que los tipos de retorno pueden ser covariantes, pero no los tipos de argumentos):

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
    public String name() {
        return getClass().getSimpleName();
    }
    Object process(Object input) { return input; }
}

class Upcase extends Processor {
    String process(Object input) { // Retorno covariante
        return ((String)input).toUpperCase();
    }
}

class Downcase extends Processor {
    String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends Processor {
    String process(Object input) {
        // El método split() divide una cadena en fragmentos:
        return Arrays.toString(((String)input).split(" "));
    }
}
```

```

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
    public static String s =
        "Disagreement with beliefs is by definition incorrect";
    public static void main(String[] args) {
        process(new Upcase(), s);
        process(new Downcase(), s);
        process(new Splitter(), s);
    }
} /* Output:
Using Processor Upcase
DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
Using Processor Downcase
disagreement with beliefs is by definition incorrect
Using Processor Splitter
[Disagreement, with, beliefs, is, by, definition, incorrect]
*///:-

```

El método **Apply.process()** toma cualquier tipo de objeto **Processor** y lo aplica a un objeto **Object**, imprimiendo después los resultados. La creación de un método que se comporte de forma diferente dependiendo del objeto argumento que se le pase es lo que se denomina el patrón de diseño basado en estrategias. El método contiene la parte fija del algoritmo que hay que implementar, mientras que la estrategia contiene la parte que varía. La estrategia es el objeto que pasamos, y que contiene el código que hay que ejecutar. Aquí, el objeto **Processor** es la estrategia y en **main()** podemos ver cómo se aplican tres estrategias diferentes a la cadena de caracteres **s**.

El método **split()** es parte de la clase **String**; toma el objeto **String** y lo divide utilizando el argumento como frontera, y devolviendo una matriz **String[]**. Se utiliza aquí como forma abreviada de crear una matriz de objetos **String**.

Ahora suponga que descubrimos un conjunto de filtros electrónicos que pudieran encajar en nuestro método **Apply.process()**:

```

//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Waveform " + id; }
} //://:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
    public String name() {
        return getClass().getSimpleName();
    }
    public Waveform process(Waveform input) { return input; }
} //://:~

//: interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
    double cutoff;
    public LowPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) {
        return input; // Dummy processing
    }
} //://:-

```

```

//: interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
    double cutoff;
    public HighPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) { return input; }
} //:-

//: interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
    double lowCutoff, highCutoff;
    public BandPass(double lowCut, double highCut) {
        lowCutoff = lowCut;
        highCutoff = highCut;
    }
    public Waveform process(Waveform input) { return input; }
} //:-
```

Filter tiene los mismos elementos de interfaz que **Processor**, pero puesto que no hereda de **Processor** (puesto que el creador de la clase **Filter** no tenía ni idea de que podríamos querer usar esos objetos como objetos **Processor**), no podemos utilizar un objeto **Filter** con el método **Apply.process()**, a pesar de que funcionaría. Básicamente, el acoplamiento entre **Apply.process()** y **Processor** es más fuerte de lo necesario y esto impide que el código de **Apply.process()** pueda reutilizarse en lugares que sería útil. Observe también que las entradas y salidas son en ambos casos de tipo **Waveform**.

Sin embargo, si **Processor** es una interfaz, las restricciones se relajan lo suficiente como para poder reutilizar un método **Apply.process()** que acepte dicha interfaz. He aquí las versiones modificadas de **Processor** y **Apply**:

```

//: interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
    String name();
    Object process(Object input);
} //:-

//: interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
} //:-
```

La primera forma en que podemos reutilizar el código es si los programadores de clientes pueden escribir sus clases para que se adapten a la interfaz, como por ejemplo:

```

//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
    public String name() {
        return getClass().getSimpleName();
    }
    public abstract String process(Object input);
    public static String s =
        "If she weighs the same as a duck, she's made of wood";
```

```

public static void main(String[] args) {
    Apply.process(new Upcase(), s);
    Apply.process(new Downcase(), s);
    Apply.process(new Splitter(), s);
}
}

class Upcase extends StringProcessor {
    public String process(Object input) { // Retorno covariante
        return ((String)input).toUpperCase();
    }
}

class Downcase extends StringProcessor {
    public String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends StringProcessor {
    public String process(Object input) {
        return Arrays.toString(((String)input).split(" "));
    }
} /* Output:
Using Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Using Processor Downcase
if she weighs the same as a duck, she's made of wood
Using Processor Splitter
[If, she, weighs, the, same, as, a, duck., she's, made, of, wood]
*///:-
```

Sin embargo, a menudo nos encontramos en una situación en la que no podemos modificar las clases que queremos usar. En el caso de los filtros electrónicos, por ejemplo, la correspondiente biblioteca la hemos descubierto, en lugar de desarrollarla. En estos casos, podemos utilizar el patrón de diseño *adaptador*. Con dicho patrón de diseño, lo que hacemos es escribir código para tomar la interfaz de la que disponemos y producir la que necesitamos, como por ejemplo:

```

//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
    Filter filter;
    public FilterAdapter(Filter filter) {
        this.filter = filter;
    }
    public String name() { return filter.name(); }
    public Waveform process(Object input) {
        return filter.process((Waveform)input);
    }
}

public class FilterProcessor {
    public static void main(String[] args) {
        Waveform w = new Waveform();
        Apply.process(new FilterAdapter(new LowPass(1.0)), w);
        Apply.process(new FilterAdapter(new HighPass(2.0)), w);
        Apply.process(
            new FilterAdapter(new BandPass(3.0, 4.0)), w);
    }
}
```

```

} /* Output:
Using Processor LowPass
Waveform 0
Using Processor HighPass
Waveform 0
Using Processor BandPass
Waveform 0
*///:r-

```

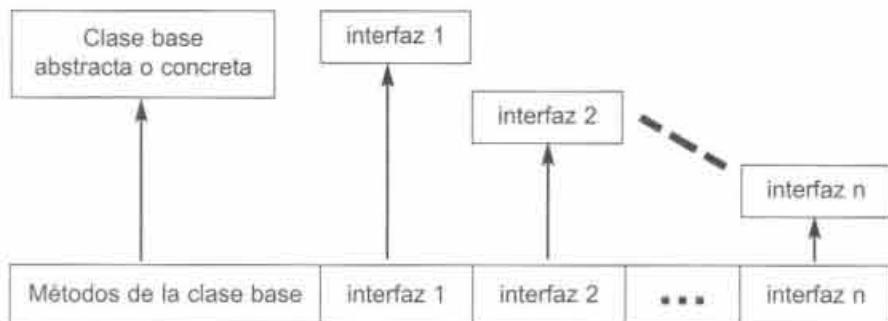
En esta aplicación, el patrón de diseño de adaptación, el constructor **FilterAdapter**, toma la interfaz que tenemos (**Filter**) y produce un objeto que tiene la interfaz **Processor** que necesitamos. Observe también la utilización del mecanismo de delegación en la clase **FilterAdapter**.

Desacoplar la interfaz de la implementación permite aplicar las interfaces a múltiples implementaciones diferentes, con lo que el código es más reutilizable.

Ejercicio 11: (4) Cree una clase con un método que tome como argumento un objeto **String** y produzca un resultado en el que se intercambie cada pareja de caracteres contenida en el argumento. Adapte la clase para que funcione con **interfaceprocessor.Apply.process()**.

“Herencia múltiple” en Java

Puesto que una interfaz no dispone de implementación (es decir, no hay ningún almacenamiento asociado con una interfaz) no hay nada que impida combinar varias interfaces. Esto resulta muy útil en ocasiones, como por ejemplo cuando queremos implementar el concepto “una **a** es una **a** y una **b** y una **c**”. En C++, este acto de combinar múltiples interfaces de clase se denomina *herencia múltiple*, y puede llegar a resultar muy completo, porque cada clase puede tener una implementación. En Java, podemos hacer lo mismo, pero sólo una de las clases puede tener una implementación, por lo que los problemas de C++ no aparecen en Java cuando se combinan múltiples interfaces:



En una clase derivada, no estamos obligados a tener una clase base que sea abstracta o concreta (una que no tenga métodos abstractos). Pero si realizamos la herencia de algo que no sea una interfaz, sólo podemos heredar de una de esas clases; los restantes elementos base deberán ser interfaces. Hay que colocar todos los nombres de interfaz detrás de la palabra clave **implements** y separarlos mediante comas. Podemos incluir tantas interfaces como queramos y podemos realizar generalizaciones (*upcast*) a cada interfaz, porque cada una de esas interfaces representa un tipo independiente. El siguiente ejemplo muestra una clase concreta que se combina con varias interfaces para producir una nueva clase:

```

//: interfaces/Adventure.java
// Interfaces múltiples.

interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
}

```

```

    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
} //:-)

```

Puede ver que **Hero** combina la clase concreta **ActionCharacter** con las interfaces **CanFight**, **CanSwim** y **CanFly**. Cuando se combina una clase concreta con interfaces de esta forma, la clase concreta debe expresarse en primer lugar y las interfaces indicarse a continuación (en caso contrario, el compilador nos dará un error).

La signatura de **fight()** es igual en la interfaz **CanFight** y en la clase **ActionCharacter**. Asimismo, a **fight()** no se le proporciona una definición en **Hero**. Podemos ampliar una interfaz, pero lo que obtenemos entonces será otra interfaz. Cuando queramos crear un objeto, todas las definiciones deberán haber sido ya proporcionadas. Aunque **Hero** no proporciona explícitamente una definición para **fight()**, dicha definición está incluida en **ActionCharacter**; por tanto, es posible crear objetos **Hero**.

En la clase **Adventure**, podemos ver que hay cuatro métodos que toman argumentos de las distintas interfaces y de la clase concreta. Cuando se crea un objeto **Hero**, se le puede pasar a cualquiera de estos métodos, lo que significa que estará siendo generalizado en cada caso a cada una de las interfaces. Debido a la forma en que se diseñan las interfaces en Java, este mecanismo funciona sin que el programador tenga que preocuparse de nada.

Recuerde que una de las principales razones para utilizar interfaces es la que se ilustra en el ejemplo anterior: para realizar generalizaciones a más de un tipo base (y poder disfrutar de la flexibilidad que esto proporciona). Sin embargo, una segunda razón para utilizar interfaces coincide con la razón por la que utilizamos clases base abstractas: para impedir que el programador de clientes cree un objeto de esta clase y para establecer que sólo se trata de una interfaz.

Esto hace que surja una cuestión: ¿debemos utilizar una interfaz o una clase abstracta? Si resulta posible crear nuestra clase base sin ninguna definición de método y sin ninguna variable miembro, siempre son preferibles las interfaces a las clases abstractas. De hecho, si sabemos que algo va a ser una clase base, podemos considerar si resultaría conveniente transformarla en interfaz (hablaremos más sobre este tema en el resumen del capítulo).

Ejercicio 12: (2) En **Adventure.java**, añada una interfaz llamada **CanClimb**, siguiendo el patrón de las otras interfaces.

Ejercicio 13: (2) Cree una interfaz y herede de ella otras dos nuevas interfaces. Defina, mediante herencia múltiple, una tercera interfaz a partir de estas otras dos.²

² Este ejemplo muestra cómo las interfaces evitan el denominado "problema del rumbo", que se presenta en el mecanismo de herencia múltiple de C++.

Ampliación de la interfaz mediante herencia

Podemos añadir fácilmente nuevas declaraciones de métodos a una interfaz utilizando los mecanismos de herencia, y también podemos combinar varias interfaces mediante herencia para crear una nueva interfaz. En ambos casos, obtendremos una interfaz nueva, como se ve en el siguiente ejemplo:

```
//: interfaces/HorrorShow.java
// Ampliación de una interfaz mediante herencia.

interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}

public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    static void w(Lethal l) { l.kill(); }
    public static void main(String[] args) {
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
} //:-
```

`DangerousMonster` es una extensión simple de `Monster` que produce una nueva interfaz. Esta se implementa en `DragonZilla`.

La sintaxis empleada en **Vampire** sólo funciona cuando se heredan interfaces. Normalmente, sólo podemos utilizar **extends** con una única clase, pero **extends** puede hacer referencia a múltiples interfaces base a la hora de construir una nueva interfaz. Como puede ver, los nombres de interfaz están simplemente separados por comas.

Ejercicio 14: (2) Cree tres interfaces, cada una de ellas con dos métodos. Defina mediante herencia una nueva interfaz que combine las tres, añadiendo un nuevo método. Cree una clase implementando la nueva interfaz y que también herede de una clase concreta. A continuación, escriba cuatro métodos, cada uno de los cuales tome una de las cuatro interfaces como argumento. En **main()**, cree un objeto de esa clase y páselo a cada uno de los métodos.

Ejercicio 15: (2) Modifique el ejercicio anterior creando una clase abstracta y haciendo que la clase derivada herede de ella.

Colisiones de nombres al combinar interfaces

Podemos encontrarnos con un pequeño problema a la hora de implementar múltiples interfaces. En el ejemplo anterior, tanto **CanFight** como **ActionCharacter** tienen sendos métodos idénticos **void fight()**. El que haya dos métodos idénticos no resulta problemático, pero ¿qué sucede si los métodos difieren en cuanto a **signatura** o en cuanto a **tipo de retorno**? He aquí un ejemplo:

```
//: interfaces/InterfaceCollision.java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // sobrecargado
}

class C3 extends C implements I2 {
    public int f(int i) { return 1; } // sobrecargado
}

class C4 extends C implements I3 {
    // Idéntico. No hay problema:
    public int f() { return 1; }
}

// Los métodos sólo difieren en el tipo de retorno:
// class C5 extends C implements I1 {}
// interface I4 extends I1, I3 {} //:-
```

La dificultad surge porque los mecanismos de anulación, de implementación y de sobrecarga se entremezclan de forma compleja. Asimismo, los métodos sobrecargados no pueden diferir sólo en cuanto al tipo de retorno. Si quitamos la marca de comentario de las dos últimas líneas, los mensajes de error nos informan del problema:

```
InterfaceCollision.java:23: f() in C cannot implement f() in I1: attempting to use incompatible return type
found: int
required: void
InterfaceCollision.java:24: Interfaces I3 and I1 are incompatible; both define f(), but with different return
type
```

Asimismo, utilizar los mismos nombres de método en diferentes interfaces que vayan a ser combinadas suele aumentar, generalmente, la confusión en lo que respecta a la legibilidad del código. Trate de evitar la utilización de nombres de método idénticos.

Adaptación a una interfaz

Una de las razones más importantes para utilizar interfaces consiste en que con ellas podemos disponer de múltiples implementaciones para una misma interfaz. En los casos más simples, esto se lleva a la práctica empleando un método que acepta una interfaz, lo que nos deja total libertad y responsabilidad para implementar dicha interfaz y pasar nuestro objeto a dicho método.

Por tanto, uno de los usos más comunes para las interfaces es el patrón de diseño basado en estrategia del que ya hemos hablado: escribimos un método que realice ciertas operaciones y dicho método toma como argumento una interfaz que especificaremos. Básicamente, lo que estamos diciendo es: "Puedes utilizar mi método con cualquier objeto que quieras, siempre que éste se adapte a mi interfaz". Esto hace que el método sea más flexible, general y reutilizable.

Por ejemplo, el constructor para la clase **Scanner** de Java SE5 (de la que hablaremos más en detalle en el Capítulo 13, *Cadenas de caracteres*) admite una interfaz **Readable**. Como veremos, **Readable** no es un argumento de ningún otro método de la biblioteca estándar de Java, fue creado pensando específicamente en **Scanner**, de modo que **Scanner** no tenga que restringir su argumento para que sea una clase determinada. De esta forma, podemos hacer que **Scanner** funcione con más tipos de datos. Si creamos una nueva clase y queremos poder usarla con **Scanner**, basta con que la hagamos de tipo **Readable**, como por ejemplo:

```
//: interfaces/RandomWords.java
// Implementación de una interfaz para adaptarse a un método.
import java.io.*;
import java.util.*;

public class RandomWords implements Readable {
    private static Random rand = new Random(47);
    private static final char[] capitals =
        "ABCDEFGHIJKLMNPQRSTUVWXYZ".toCharArray();
    private static final char[] lowers =
        "abcdefghijklmnopqrstuvwxyz".toCharArray();
    private static final char[] vowels =
        "aeiou".toCharArray();
    private int count;
    public RandomWords(int count) { this.count = count; }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1; // Indica el final de la entrada
        cb.append(capitals[rand.nextInt(capitals.length)]);
        for(int i = 0; i < 4; i++) {
            cb.append(vowels[rand.nextInt(vowels.length)]);
            cb.append(lowers[rand.nextInt(lowers.length)]);
        }
        cb.append(" ");
        return 10; // Número de caracteres añadidos
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new RandomWords(10));
        while(s.hasNext())
            System.out.println(s.next());
    }
} /* Output:
Yazeruyac
Fowenucor
Goeazimom
Raeuuácio
Nuoadesiw
Hageaikux
Rugicibui
Numasetih
```

```
Kuuuuozog
Waqizeyoy
*///:-
```

La interfaz **Readable** sólo requiere que se implemente un método **read()**. Dentro de **read()**, añadimos la información al argumento **CharBuffer** (hay varias formas de hacer esto, consulte la documentación de **CharBuffer**), o devolvemos **-1** cuando ya no haya más datos de entrada.

Supongamos que disponemos de una clase base que aún no implementa **Readable**, en este caso, ¿cómo podemos hacer que funcione con **Scanner**? He aquí un ejemplo de una clase que genera números en coma flotante aleatorios.

```
//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
    private static Random rand = new Random(47);
    public double next() { return rand.nextDouble(); }
    public static void main(String[] args) {
        RandomDoubles rd = new RandomDoubles();
        for(int i = 0; i < 7; i++)
            System.out.print(rd.next() + " ");
    }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:-
```

De nuevo, podemos utilizar el patrón de diseño adaptador, pero en este caso la clase adaptada puede crearse heredando e implementando la interfaz **Readable**. Por tanto, si utilizamos la herencia pseudo-múltiple proporcionada por la palabra clave **interface**, produciremos una nueva clase que será a la vez **RandomDoubles** y **Readable**:

```
//: interfaces/AdaptedRandomDoubles.java
// Creación de un adaptador mediante herencia.
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
    private int count;
    public AdaptedRandomDoubles(int count) {
        this.count = count;
    }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1;
        String result = Double.toString(next()) + " ";
        cb.append(result);
        return result.length();
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new AdaptedRandomDoubles(7));
        while(s.hasNextDouble())
            System.out.print(s.nextDouble() + " ");
    }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:-
```

Puesto que podemos añadir de esta forma una interfaz a cualquier clase existente, podemos deducir que un método que tome como argumento una interfaz nos permitirá adaptar cualquier clase para que funcione con dicho método. Aquí radica la verdadera potencia de utilizar interfaces en lugar de clases.

Ejercicio 16: (3) Cree una clase que genere una secuencia de caracteres. Adapte esta clase para que pueda utilizarse como entrada a un objeto **Scanner**.

Campos en las interfaces

Puesto que cualquier campo que incluyamos en una interfaz será automáticamente de tipo **static** y **final**, la interfaz constituye una herramienta conveniente para crear grupos de valores constantes. Antes de Java SE5, ésta era la única forma de producir el mismo efecto que con la palabra clave **enum** en C o C++. Por tanto, resulta habitual encontrarse con código anterior a la versión Java SE5 que presenta el aspecto siguiente:

```
//: interfaces/Months.java
// Uso de interfaces para crear grupos de constantes.
package interfaces;

public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} //:-
```

Observe la utilización del estilo Java, en el que todas las letras están en mayúsculas (con guiones bajos para separar las distintas palabras que formen un determinado identificador) en los casos de valores estáticos finales con inicializadores constantes. Los campos de una interfaz son automáticamente públicos, así que el atributo **public** no se especifica explícitamente.

Con Java SE5, ahora disponemos de la palabra clave **enum**, mucho más potente y flexible, por lo que rara vez tendrá sentido que utilicemos interfaces para definir constantes. Sin embargo, quizás se encuentre en muchas ocasiones con esta técnica antigua a la hora de leer código heredado (los suplementos de este libro disponibles en www.MindView.net proporcionan una descripción completa de la técnica previa a Java SE5 para producir tipos enumerados utilizando interfaces). Puede encontrar más detalles sobre el uso de la palabra clave **enum** en el Capítulo 19, *Tipos enumerados*.

Ejercicio 17: (2) Demuestre que los campos de una interfaz son implicitamente de tipo **static** y **final**.

Inicialización de campos en las interfaces

Los campos definidos en las interfaces no pueden ser valores “finales en blanco”, pero pueden inicializarse con expresiones no constantes. Por ejemplo:

```
//: interfaces/RandVals.java
// Inicialización de campos de interfaz con
// inicializadores no constantes.
import java.util.*;

public interface RandVals {
    Random RAND = new Random(47);
    int RANDOM_INT = RAND.nextInt(10);
    long RANDOM_LONG = RAND.nextLong() * 10;
    float RANDOM_FLOAT = RAND.nextLong() * 10;
    double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} //:-
```

Puesto que los campos son estáticos, se inicializan cuando se carga por primera vez la clase, lo que tiene lugar cuando se accede por primera vez a cualquiera de los campos. He aquí una prueba simple:

```
//: interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
```

```

public static void main(String[] args) {
    print(RandVals.RANDOM_INT);
    print(RandVals.RANDOM_LONG);
    print(RandVals.RANDOM_FLOAT);
    print(RandVals.RANDOM_DOUBLE);
}
} /* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*//*/-

```

Los campos, por supuesto, no forman parte de la interfaz. Los valores se almacenan en el área de almacenamiento estático correspondiente a dicha interfaz.

Anidamiento de interfaces

Las interfaces pueden anidarse dentro de clases y dentro de otras interfaces.³ Esto nos revela una serie de características interesantes:

```

//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
    interface B {
        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
    public interface C {
        void f();
    }
    class CImp implements C {
        public void f() {}
    }
    private class CImp2 implements C {
        public void f() {}
    }
    private interface D {
        void f();
    }
    public class DImp implements D {
        public void f() {}
    }
    public class DImp2 implements D {
        public void f() {}
    }
    public D getD() { return new DImp2(); }
    private D dRef;
    public void receiveD(D d) {
        dRef = d;
        dRef.f();
    }
}

```

³ Gracias a Martin Danner por hacer una pregunta a este respecto en un seminario.

```

}

interface E {
    interface G {
        void f();
    }
    // "public" redundante:
    public interface H {
        void f();
    }
    void g();
    // No puede ser private dentro de una interfaz:
    //! private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }
    class CImp implements A.C {
        public void f() {}
    }
    // No se puede implementar una interfaz privada excepto
    // dentro de la clase definitoria de dicha interfaz:
    //! class DImp implements A.D {
    //!     public void f() {}
    //! }
    class EIImp implements E {
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
    class EIImp2 implements E {
        public void g() {}
        class EG implements E.G {
            public void f() {}
        }
    }
    public static void main(String[] args) {
        A a = new A();
        // No se puede acceder a A.D:
        //! A.D ad = a.getD();
        // Sólo puede devolver a A.D:
        //! A.DImp2 di2 = a.getD();
        // No se puede acceder a un miembro de la interfaz:
        //! a.getD().f();
        // Sólo otra A puede utilizar getD():
        A a2 = new A();
        a2.receiveD(a.getD());
    }
} //!

```

La sintaxis para anidar una interfaz dentro de una clase es razonablemente amplia. Al igual que las interfaces no anidadas, las anidadas pueden tener visibilidad pública o con acceso de paquete.

Como característica adicional, las interfaces también pueden ser privadas, como podemos ver en **A.D** (se necesita la misma sintaxis de cualificación para las interfaces anidadas que para las clases anidadas). ¿Para qué sirve una interfaz anidada privada? Podemos suponer que sólo puede implementarse como clase interna privada, como en **DImp**, pero **A.DImp2** mues-

tra que también puede implementarse como clase pública. Sin embargo, **A.DImp2** sólo puede utilizarse como ella misma. No se nos permite mencionar el hecho de que implementa la interfaz privada **D**, por lo que implementar interfaces privadas es una forma de forzar la definición de los métodos de dicha interfaz sin añadir ninguna información de tipos (es decir, sin permitir ninguna generalización).

El método **getD()** nos revela un dato adicional acerca de las interfaces privadas: se trata de un método público que devuelve una referencia a un interfaz privada. ¿Qué podemos hacer con el valor de retorno de este método? En **main()**, podemos ver varios intentos de utilizar el valor de retorno, todos los cuales fallan. La única cosa que funciona es entregar el valor de retorno a un objeto que tenga permiso para usarlo, que en este caso es otro objeto **A**, a través del método **receiveD()**.

La interfaz **E** muestra que podemos anidar unas interfaces dentro de otras. Sin embargo, las reglas acerca de las interfaces, en particular, que todos los elementos de la interfaz tienen que ser públicos, se imponen aquí de manera estricta, por lo que una interfaz anidada dentro de otra será automáticamente pública y no puede nunca definirse como privada.

NestingInterfaces muestra las diversas formas en que pueden implementarse las interfaces anidadas. En particular, observe que, cuando implementamos una interfaz, no estamos obligados a implementar ninguna de las interfaces anidadas dentro de ella. Asimismo, las interfaces privadas no pueden implementarse fuera de sus clases definitorias.

Inicialmente, pudiera parecer que estas características sólo se hubieran añadido para garantizar la coherencia sintáctica, pero mi experiencia es que una vez que se conoce una característica siempre se descubren ocasiones en las que puede resultar útil.

Interfaces y factorías

El objeto principal de una interfaz es permitir la existencia de múltiples implementaciones, y una forma típica de producir objetos que encajen con una interfaz es el denominado patrón de diseño de *método factoría*. En lugar de llamar a un constructor directamente, invocamos un método de creación en un objeto factoría que produce una implementación de la interfaz; de esta forma, en teoría, nuestro código estará completamente aislado de la implementación de la interfaz, haciendo así posible intercambiar de manera transparente una implementación por otra. He aquí un ejemplo que muestra la estructura del método factoría:

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    Implementation1() {} // Package access
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
    public Service getService() {
        return new Implementation1();
    }
}

class Implementation2 implements Service {
    Implementation2() {} // Acceso de paquete
    public void method1() {print("Implementation2 method1");}
}
```

```

    public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
    public Service getService() {
        return new Implementation2();
    }
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(new Implementation1Factory());
        // Las implementaciones son completamente intercambiables:
        serviceConsumer(new Implementation2Factory());
    }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:-
```

Sin el método `factory`, nuestro código tendría que especificar en algún lugar el tipo exacto de objeto `Service` que se estuviera creando, para poder invocar el constructor apropiado.

¿Para qué sirve añadir este nivel adicional de indirección? Una razón común es para crear un marco de trabajo para el desarrollo. Suponga que estamos creando un sistema para juegos que permita, por ejemplo, jugar tanto al ajedrez como a las damas en un mismo tablero.

```

//: interfaces/Games.java
// Un marco de trabajo para juegos utilizando métodos factoría.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
}

class CheckersFactory implements GameFactory {
    public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
```

```

    }

    class ChessFactory implements GameFactory {
        public Game getGame() { return new Chess(); }
    }

    public class Games {
        public static void playGame(GameFactory factory) {
            Game s = factory.getGame();
            while(s.move())
        }
    }

    public static void main(String[] args) {
        playGame(new CheckersFactory());
        playGame(new ChessFactory());
    }
} /* Output:
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*///:-
```

Si la clase **Games** representa un fragmento complejo de código, esta técnica permite reutilizar dicho código con diferentes tipos de juegos. Podemos fácilmente imaginar otros juegos más elaborados que pudieran beneficiarse a la hora de desarrollar este diseño.

En el siguiente capítulo, veremos una forma más elegante de implementar las factorías utilizando clases internas anónimas.

Ejercicio 18: (2) Cree una interfaz **Cycle**, con implementaciones **Unicycle**, **Bicycle** y **Tricycle**. Cree factorías para cada tipo de **Cycle** y el código necesario que utilicen estas factorías.

Ejercicio 19: (3) Cree un marco de trabajo utilizando métodos factoría que permita simular las operaciones de lanzar una moneda y lanzar un dado.

Resumen

Resulta bastante tentador concluir que las interfaces resultan útiles y que, por tanto, siempre son preferibles a las clases concretas. Por supuesto, casi siempre que creemos una clase, podemos crear en su lugar una interfaz y una factoría.

Mucha gente ha caído en esta tentación creando interfaces y factorías siempre que era posible. La lógica subyacente a este enfoque es que a lo mejor podemos necesitar en el futuro una implementación diferente, por lo que añadimos siempre dicho nivel de abstracción. Esta técnica ha llegado a convertirse en una especie de optimización de diseño prematura.

La realidad es que todas las abstracciones deben estar motivadas por una necesidad real. Las interfaces deben ser algo que utilicemos cuando sea necesario para optimizar el código, en lugar de incluir ese nivel adicional de indirección en todas partes, ya que ello hace que aumente la complejidad. Esa complejidad adicional es significativa, y hacer que alguien trate de comprender ese código tan complejo sólo para descubrir al final que hemos añadido las interfaces “por si acaso” y sin una razón real, esa persona sospechará, con motivo, de todos los diseños que realicemos.

Una directriz apropiada es la que señala que *las clases resultan preferibles a las interfaces*. Comience con clases y, si está claro que las interfaces son necesarias, rediseñe el código. Las interfaces son una herramienta muy conveniente, pero está bastante generalizada la tendencia a utilizarlas en demasia.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Clases internas

10

Resulta posible situar la definición de una clase dentro de la definición de otra. Dichas clases se llaman **clases internas**.

Las clases internas constituyen una característica muy interesante, porque nos permite agrupar clases relacionadas y controlar la visibilidad mutua de esas clases. Sin embargo, es importante comprender que las clases internas son algo totalmente distinto al mecanismo de composición del que ya hemos hablado.

A primera vista, las clases internas parecen un simple mecanismo de ocultación de código: colocamos las clases dentro de otras clases. Sin embargo, como veremos, las clases internas sirven para algo más que eso: la clase interna conoce los detalles de la clase contenedora y puede comunicarse con ella. Asimismo, el tipo de código que puede escribirse con las clases internas es más elegante y claro (aunque no en todas las ocasiones, por supuesto).

Inicialmente, las clases internas pueden parecer extrañas y se requiere cierto tiempo para llegar a sentirse cómodo al utilizarlas en los diseños. La necesidad de las clases internas no siempre resulta obvia, pero después de describir la sintaxis básica y la semántica de las clases internas, la sección “¿Para qué sirven las clases internas?” debería permitir que el lector se haga una idea de los beneficios de emplear este tipo de clases.

Después de dicha sección, el resto del capítulo contiene un análisis más detallado de la sintaxis de las clases internas. Estas características se proporcionan con el fin de cubrir por completo el lenguaje, pero puede que no tengamos que usarlas nunca, o al menos no al principio. Así pues, puede que el lector sólo necesite consultar las partes iniciales del capítulo dejando los análisis más detallados como material de referencia.

Creación de clases internas

Para crear una clase interna, el procedimiento que se utiliza es el que cabría suponer: la definición de la clase se incluye dentro de otra clase contenedora:

```
//: innerclasses/Parcell.java
// Creación de clases internas.

public class Parcell {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // La utilización de clases internas se asemeja
    // a la de cualquier otra clase, dentro de Parcell:
```

```

public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
}
public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tasmania");
}
/* Output:
Tasmania
*///:-
```

Las clases internas utilizadas dentro de `ship()` parecen clases normales. Aquí, la única diferencia práctica es que los nombres están anidados dentro de `Parcel1`. Pronto veremos que esta diferencia no es la única.

Lo más normal es que una clase externa tenga un método que devuelva una referencia a una clase interna, como puede verse en los métodos `to()` y `contents()`:

```

//: innerclasses/Parcel2.java
// Devolución de una referencia a una clase interna.

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents contents() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = contents();
        Destination d = to(dest);
        System.out.println(d.readLabel());
    }
}
public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tasmania");
    Parcel2 q = new Parcel2();
    // Definición de referencias a clases internas:
    Parcel2.Contents c = q.contents();
    Parcel2.Destination d = q.to("Borneo");
}
/* Output:
Tasmania
*///:-
```

Si queremos construir un objeto de la clase interna en cualquier lugar que no sea dentro de un método no estático de la clase externa, debemos especificar el tipo de dicho objeto como `NombreClaseExterna.NombreClaseInternra`, como puede verse en `main()`.

Ejercicio 1: (1) Escriba una clase denominada **Outer** que contenga una clase interna llamada **Inner**. Añada un método a **Outer** que devuelva un objeto de tipo **Inner**. En **main()**, cree e inicialice una referencia a un objeto **Inner**.

El enlace con la clase externa

Hasta ahora, parece que las clases internas son simplemente un esquema de organización de código y de ocultación de nombres, lo cual resulta útil pero no especialmente necesario. Sin embargo, las cosas son más complejas de lo que parecen, cuando se crea una clase interna, cada objeto de esa clase interna dispone de un *enlace al objeto contenedor que lo ha creado*, por lo cual puede acceder a los miembros de dicho objeto contenedor sin utilizar ninguna cualificación especial. Además, las clases internas tienen derechos de acceso a todos los elementos de la clase contenedora.¹ El siguiente ejemplo ilustra esta característica:

```
//: innerclasses/Sequence.java
// Almacena una secuencia de objetos.

interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private Object[] items;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
        if(next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector {
        private int i = 0;
        public boolean end() { return i == items.length; }
        public Object current() { return items[i]; }
        public void next() { if(i < items.length) i++; }
    }
    public Selector selector() {
        return new SequenceSelector();
    }
    public static void main(String[] args) {
        Sequence sequence = new Sequence(10);
        for(int i = 0; i < 10; i++)
            sequence.add(Integer.toString(i));
        Selector selector = sequence.selector();
        while(!selector.end()) {
            System.out.print(selector.current() + " ");
            selector.next();
        }
    }
} /* Output:
0 1 2 3 4 5 6 7 8 9
*///:-
```

¹ Esto difiere significativamente del diseño de *clases anidadas* en C++, que simplemente se trata de un mecanismo de ocultación de nombres. No hay ningún enlace al objeto contenedor ni ningún tipo de permisos implícitos en C++.

La secuencia **Sequence** es simplemente una matriz de tamaño fijo de objetos **Object** con una clase envoltorio. Invocamos **add()** para añadir un nuevo objeto al final de la secuencia (si queda sitio). Para extraer cada uno de los objetos de la secuencia, hay una interfaz denominada **Selector**. Éste es un ejemplo del patrón de diseño iterador del que hablaremos más en detalle posteriormente en el libro. Un **Selector** permite ver si nos encontramos al final de la secuencia [**end()**], acceder al objeto actual [**current()**] y desplazarse al objeto siguiente [**next()**] de la secuencia. Como **Selector** es una interfaz, otras clases pueden implementar la interfaz a su manera y otros métodos pueden tomar la interfaz como argumento, para crear código de propósito más general.

Aquí, **SequenceSelector** es una clase privada que proporciona la funcionalidad **Selector**. En **main()**, podemos ver la creación de una secuencia, seguida de la adición de una serie de objetos de tipo **String**. A continuación, se genera un objeto **Selector** con una llamada a **selector()**, y este objeto se utiliza para desplazarse a través de la secuencia y seleccionar cada elemento.

A primera vista, la creación de **SequenceSelector** se asemeja a la de cualquier otra clase interna. Pero examinemos el ejemplo en más detalle. Observe que cada uno de los métodos [**end()**, **current()** y **next()**] hace referencia a **items**, que es una referencia que no forma parte de **SequenceSelector**, sino que se encuentra en un campo privado dentro de la clase contenedora. Sin embargo, la clase interna puede acceder a los métodos y campos de la clase contenedora como si fueran de su propiedad. Esta característica resulta muy cómoda, como puede verse en el ejemplo anterior.

Así pues, una clase interna tiene acceso automático a los miembros de la clase contenedora. ¿Cómo puede suceder esto? La clase interna captura en secreto una referencia al objeto concreto de la clase contenedora que sea responsable de su creación. Entonces, cuando hacemos referencia a un miembro de la clase contenedora, dicha referencia se utiliza para seleccionar dicho miembro. Afortunadamente, el compilador se encarga de resolver todos estos detalles por nosotros, pero resulta evidente que sólo podrá crearse un objeto de la clase interna en asociación con otro objeto de la clase contenedora (cuando, como veremos pronto, la clase interna sea no estática). La construcción del objeto de la clase interna necesita de una referencia al objeto de la clase contenedora y el compilador se quejará si no puede acceder a dicha referencia. La mayor parte de las veces todo este mecanismo funciona sin que el programador tenga que intervenir para nada.

Ejercicio 2: (1) Cree una clase que almacene un objeto **String** y que disponga de un método **toString()** que muestre esa cadena de caracteres. Añada varias instancias de la nueva clase a un objeto **Sequence** y luego visualícelas.

Ejercicio 3: (1) Modifique el Ejercicio 1 para que **Outer** tenga un campo **private String** (initializado por el constructor) e **Inner** tenga un método **toString()** que muestre este campo. Cree un objeto de tipo **Inner** y visualícelo.

Utilización de **this** y **.new**

Si necesita generar la referencia al objeto de la clase externa, basta con indicar el nombre de la clase externa seguido de un punto y de la palabra clave **this**. La referencia resultante tendrá automáticamente el tipo correcto, que se conoce y se comprueba en tiempo de compilación, por lo que no hay ningún gasto adicional en tiempo de procesamiento. He aquí un ejemplo que muestra cómo utilizar **this**:

```
//: innerclasses/DotThis.java
// Cualificación del acceso al objeto de la clase externa.

public class DotThis {
    void f() { System.out.println("DotThis.f()"); }
    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // Un "this" haría referencia al "this" de Inner
        }
    }
    public Inner inner() { return new Inner(); }
    public static void main(String[] args) {
        DotThis dt = new DotThis();
        DotThis.Inner dti = dt.inner();
```

```

        dti.outer().f();
    }
} /* Output:
DotThis.f()
*/:-
```

Algunas veces, necesitamos decir a un objeto que cree otro objeto de una de sus clases internas. Para hacer esto es necesario proporcionar una referencia al objeto de la clase externa en la expresión `new`, utilizando la sintaxis `.new`, como en el siguiente ejemplo:

```

//: innerclases/DotNew.java
// Creación de una clase interna directamente utilizando la sintaxis .new.

public class DotNew {
    public class Inner {}
    public static void main(String[] args) {
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
} /*:-
```

Para crear un objeto de la clase interna directamente, no se utiliza esta misma forma haciendo referencia al nombre de la clase externa `DotNew` como cabría esperar, sino que en su lugar es necesario utilizar un *objeto* de la clase externa para crear un objeto de la clase interna, como podemos ver en el ejemplo anterior. Esto resuelve también las cuestiones relativas a los ámbitos de los nombres en la clase interna, por lo que nunca escribiríamos (porque, de hecho, *no se puede*) `dn.new DotNew.Inner()`.

No es posible crear un objeto de la clase interna a menos que ya se disponga de un objeto de la clase externa. Esto se debe a que el objeto de la clase interna se conecta de manera transparente al de la clase externa que lo haya creado. Sin embargo, si definimos una *clase anidada*, (una clase interna estática), entonces no será necesaria la referencia al objeto de la clase externa.

A continuación puede ver cómo se aplicaría `.new` al ejemplo “Parcel”:

```

//: innerclases/Parcel3.java
// Utilización de .new para crear instancias de clases internas.

public class Parcel3 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        // Hay que usar una instancia de la clase externa
        // para crear una instancia de la clase interna:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d = p.new Destination("Tasmania");
    }
} /*:-
```

Ejercicio 4: (2) Añada un método a la clase `Sequence.SequenceSelector` que genere la referencia a la clase externa `Sequence`.

Ejercicio 5: (1) Cree una clase con una clase interna. En otra clase separada, cree una instancia de la clase interna.

Clases internas y generalización

Las clases internas muestran su utilidad real cuando comenzamos a generalizar a una clase base y, en particular, a una interfaz. (El efecto de generar una referencia a una interfaz a partir de un objeto que la implemente es prácticamente el mismo que el de realizar una generalización a una clase base). La razón es que entonces la clase interna (la implementación de la interfaz) puede ser no visible y estar no disponible, lo cual resulta muy útil para ocultar la implementación. Lo único que se obtiene es una referencia a la clase base o a la interfaz.

Podemos crear interfaces para los ejemplos anteriores:

```
//: innerclasses/Destination.java
public interface Destination {
    String readLabel();
} //:-

//: innerclasses/Contents.java
public interface Contents {
    int value();
} //:-
```

Ahora **Contents** y **Destination** representan interfaces disponibles para el programador de clientes. Recuerde que una interfaz hace que todos sus miembros sean automáticamente públicos.

Cuando obtenemos una referencia a la clase base o a la interfaz, es posible que no podamos averiguar el tipo exacto, como se muestra en el siguiente ejemplo:

```
//: innerclasses/TestParcel.java

class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}

public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("Tasmania");
        // Ilegal -- no se puede acceder a la clase privada:
        // Parcel4.PContents pc = p.new PContents();
    }
} //:-
```

En **Parcel4** hemos añadido algo nuevo. La clase interna **PContents** es **private**, así que sólo puede acceder a ella **Parcel4**. Las clases normales (no internas) no pueden ser privadas o protegidas; sólo pueden tener acceso público o de paquete. **PDestination** es protegida, por lo que sólo pueden acceder a ella **Parcel4**, las clases contenidas en el mismo paquete (ya

que **protected** también proporciona acceso de paquete) y las clases que hereden de **Parcel4**. Esto quiere decir que el programador de clientes tiene un conocimiento de estos miembros y un acceso a los mismos restringido. De hecho, no podemos ni siquiera realizar una especialización a una clase interna privada (ni a una clase interna protegida, a menos que estemos usando una clase que herede de ella), porque no se puede acceder al nombre, como podemos ver en **class TestParcel**. Por tanto, las clases internas privadas proporcionan una forma para que los diseñadores de clases eviten completamente las dependencias de la codificación de tipos y oculten totalmente los detalles relativos a la implementación. Además, la extensión de una interfaz resulta inútil desde la perspectiva del programador de clientes, ya que éste no puede acceder a ningún método adicional que no forme parte de la interfaz pública. Esto también proporciona una oportunidad para que el compilador de Java genere código más eficiente.

- Ejercicio 6:** (2) Cree una interfaz con al menos un método, dentro de su propio paquete. Cree una clase en un paquete separado. Añada una clase interna protegida que implemente la interfaz. En un tercer paquete, defina una clase que herede de la anterior y, dentro de un método, devuelva un objeto de la clase interna protegida, efectuando una generalización a la interfaz durante el retorno.
- Ejercicio 7:** (2) Cree una clase con un campo privado y un método privado. Cree una clase interna con un método que modifique el campo de la clase externa e invoque el método de la clase externa. En un segundo método de la clase externa, cree un objeto de la clase interna e invoque su método, mostrando a continuación el efecto que esto tenga sobre el objeto de la clase externa.
- Ejercicio 8:** (2) Determine si una clase externa tiene acceso a los elementos privados de su clase interna.

Clases internas en los métodos y ámbitos

Lo que hemos visto hasta ahora son los usos típicos de las clases internas. En general, el código que escribamos y el que podamos leer donde aparezcan clases internas estará compuesto por clases internas “simples” que resulten fáciles de comprender. Sin embargo, las sintaxis de las clases internas abarca varias otras técnicas más complejas. Las clases internas pueden crearse dentro de un método o incluso dentro de un ámbito arbitrario. Existen dos razones para hacer esto:

1. Como hemos visto anteriormente, podemos estar implementando una interfaz de algún tipo para poder crear y devolver una referencia.
2. Podemos estar tratando de resolver un problema complicado y queremos crear una clase que nos ayude a encontrar la solución, pero sin que la clase esté públicamente disponible.

En los siguientes ejemplos, vamos a modificar el código anterior para utilizar:

1. Una clase definida dentro de un método
2. Una clase definida dentro de un ámbito en el interior de un método
3. Una clase *anónima* que implemente una interfaz
4. Una clase anónima que amplie una clase que disponga de un constructor no predeterminado
5. Una clase anónima que se encargue de la inicialización de campos
6. Una clase anónima que lleve a cabo la construcción utilizando el mecanismo de inicialización de instancia (las clases internas anónimas no pueden tener constructores).

El primer ejemplo muestra la creación de una clase completa dentro del ámbito de un método (en lugar de dentro del ámbito de otra clase). Esto se denomina *clase interna local*:

```
//: innerclasses/Parcel5.java
// Anidamiento de una clase dentro de un método.

public class Parcel5 {
    public Destination destination(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public void go() {
                System.out.println("Going to " + label);
            }
        }
        return new PDestination(s);
    }
}
```

```

        }
        public String readLabel() { return label; }
    }
    return new PDestination(s);
}
public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    Destination d = p.destination("Tasmania");
}
} //:-.

```

La clase **PDestination** es parte de **destination()** en lugar de ser parte de **Parcel5**. Por tanto, no se puede acceder a **PDestination** fuera de **destination()**. Observe la generalización que tiene lugar en la instrucción **return**: lo único que sale de **destination()** es una referencia a **Destination**, que es la clase base. Por supuesto, el hecho de que el nombre de la clase **PDestination** se coloque dentro de **destination()** no quiere decir que **PDestination** no sea un objeto válido una vez que **destination()** termina.

Podemos utilizar el identificador de clase **PDestination** para nombrar cada clase interna dentro de un mismo subdirectorío sin que se produzcan colisiones de clases.

El siguiente ejemplo muestra cómo podemos anidar clases dentro de un ámbito arbitrario.

```

//: innerclasses/Parcel6.java
// Anidamiento de una clase dentro de un ámbito.

public class Parcel6 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
        // ¡No se puede usar aquí! Fuera de ámbito:
        // TrackingSlip ts = new TrackingSlip("x");
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        p.track();
    }
} //:-.

```

La clase **TrackingSlip** está anidada dentro del ámbito de una instrucción **if**. Esto no quiere decir que la *clase* se cree condicionalmente; esa clase se compila con todo el resto del código. Sin embargo, la clase no está disponible fuera del ámbito en que está definida. Por lo demás, se asemeja a una clase normal.

Ejercicio 9: (1) Cree una interfaz con al menos un método e implemente dicha interfaz definiendo una clase interna dentro de un método que devuelva una referencia a la interfaz.

Ejercicio 10: (1) Repita el ejercicio anterior, pero definiendo la clase interna dentro de un ámbito en el interior de un método.

Ejercicio 11: (2) Cree una clase interna privada que implemente una interfaz pública. Escriba un método que devuelva una referencia a una instancia de la clase interna privada, generalizada a la interfaz. Demuestre que la clase interna está completamente oculta, tratando de realizar una especialización sobre la misma.

Clases internas anónimas

El siguiente ejemplo puede parecer un tanto extraño:

```
//: innerclasses/Parcel7.java
// Devolución de una instancia de una clase interna anónima.

public class Parcel7 {
    public Contents contents() {
        return new Contents() { // Inserte una definición de clase
            private int i = 11;
            public int value() { return i; }
        }; // En este caso hace falta el punto y coma
    }
    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Contents c = p.contents();
    }
} ///:-
```

El método **contents()** combina la creación del valor de retorno con la definición de la clase que representa dicho valor de retorno. Además, la clase es *anónima*, es decir, no tiene nombre. Para complicar aún más las cosas, parece como si estuviéramos empezando a crear un objeto **Contents**, y que entonces, antes de llegar al punto y coma, dijéramos "Un momento: voy a introducir una definición de clase".

Lo que esta extraña sintaxis significa es: "Crea un objeto de una clase anónima que herede de **Contents**". La referencia devuelta por la expresión **new** se generalizará automáticamente a una referencia de tipo **Contents**. La sintaxis de la clase interna anónima es una abreviatura de:

```
//: innerclasses/Parcel7b.java
// Versión expandida de Parcel7.java

public class Parcel7b {
    class MyContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    public Contents contents() { return new MyContents(); }
    public static void main(String[] args) {
        Parcel7b p = new Parcel7b();
        Contents c = p.contents();
    }
} ///:-
```

En la clase interna anónima, **Contents** se crea utilizando un constructor predeterminado.

El siguiente código muestra lo que hay que hacer si la clase base necesita un constructor con un argumento:

```
//: innerclasses/Parcel8.java
// Invocación del constructor de la clase base.

public class Parcel8 {
    public Wrapping wrapping(int x) {
        // Llamada al constructor de la clase base:
        return new Wrapping(x); // Pasar argumento del constructor.
        public int value() {
            return super.value() * 47;
        }
    }; // Punto y coma necesario
}
public static void main(String[] args) {
```

```

Parcel8 p = new Parcel8();
Wrapping w = p.wrapping(10);
}
} //:-

```

Es decir, simplemente pasamos el argumento apropiado al constructor de la clase base, como sucede aquí con la `x` que se pasa en `new Wrapping(x)`. Aunque se trata de una clase normal con una implementación, `Wrapping` se está usando también como “interfaz” con sus clases derivadas:

```

//: innerclasses/Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
} //:-

```

Como puede observar, `Wrapping` tiene un constructor que requiere un argumento, para que las cosas sean un poco más interesantes.

El punto y coma situado al final de la clase interna anónima no marca el final del cuerpo de la clase, sino el final de la expresión que contenga a la clase anónima. Por tanto, es una utilización idéntica al uso del punto y coma en cualquier otro lugar.

También se puede realizar la inicialización cuando se definen los campos en una clase anónima:

```

//: innerclasses/Parcel9.java
// Una clase interna anónima que realiza la
// inicialización. Versión más breve de Parcel5.java.

public class Parcel9 {
    // El argumento debe ser final para poder utilizarlo
    // dentro de la clase interna anónima:
    public Destination destination(final String dest) {
        return new Destination() {
            private String label = dest;
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.destination("Tasmania");
    }
} //:-

```

Si estamos definiendo una clase interna anónima y queremos usar un objeto que está definido fuera de la clase interna anónima, el compilador requiere que la referencia al argumento sea `final`, como puede verse en el argumento de `destination()`. Si nos olvidamos de hacer esto, obtendremos un mensaje de error en tiempo de compilación.

Mientras que estemos simplemente realizando una asignación a un campo, la técnica empleada en este ejemplo resulta adecuada. ¿Pero qué sucede si necesitamos realizar algún tipo de actividad similar a la de los constructores? No podemos disponer de un constructor nominado dentro de una clase anónima (ya que la clase no tiene ningún nombre), pero con el mecanismo de *inicialización de instancia*, podemos, en la práctica, crear un constructor para una clase interna anónima, de la forma siguiente:

```

//: innerclasses/AnonymousConstructor.java
// Creación de un constructor para una clase interna anónima.
import static net.mindview.util.Print.*;

abstract class Base {
    public Base(int i) {
        print("Base constructor, i = " + i);
    }
    public abstract void f();
}

```

```

public class AnonymousConstructor {
    public static Base getBase(int i) {
        return new Base(i) {
            { print("Inside instance initializer"); }
            public void f() {
                print("In anonymous f()");
            }
        };
    }
    public static void main(String[] args) {
        Base base = getBase(47);
        base.f();
    }
} /* Output:
Base constructor, i = 47
Inside instance initializer
In anonymous f()
*///:-

```

En este caso, la variable `i` no tenía porqué haber sido final. Aunque se pasa `i` al constructor base de la clase anónima, nunca se utiliza esa variable *dentro* de la clase anónima.

He aquí un ejemplo con inicialización de instancia. Observe que los argumentos de `destination()` deben ser de tipo final, puesto que se los usa dentro de la clase anónima:

```

//: innerclasses/Parcel10.java
// Uso de "inicialización de instancia" para realizar
// la construcción de una clase interna anónima.

public class Parcel10 {
    public Destination
    destination(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Inicialización de instancia para cada objeto:
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Over budget!");
            }
            private String label = dest;
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel10 p = new Parcel10();
        Destination d = p.destination("Tasmania", 101.395F);
    }
} /* Output:
Over budget!
*///:-

```

Dentro del inicializador de instancia, podemos ver código que no podría ejecutarse como parte de un inicializador de campo (es decir, la instrucción `if`). Por tanto, en la práctica, un inicializador de instancia es el constructor de una clase interna anónima. Por supuesto, esta solución está limitada: no se pueden sobrecargar los inicializadores de instancia, así que sólo podemos disponer de uno de estos constructores.

Las clases internas anónimas están en cierta medida limitadas si las comparamos con el mecanismo normal de herencia, porque tienen que extender una clase o implementar una interfaz, pero no pueden hacer ambas cosas al mismo tiempo. Y, si implementamos una interfaz, sólo podemos implementar una.

Ejercicio 12: (1) Repita el Ejercicio 7 utilizando una clase interna anónima.

Ejercicio 13: (1) Repita el Ejercicio 9 utilizando una clase interna anónima.

Ejercicio 14: (1) Modifique `interfaces/HorrorShow.java` para implementar `DangerousMonster` y `Vampire` utilizando clases anónimas.

Ejercicio 15: (2) Cree una clase con un constructor no predeterminado (uno que tenga argumentos) y sin ningún constructor predeterminado (es decir, un constructor sin argumentos). Cree una segunda clase que tenga un método que devuelva una referencia a un objeto de la primera clase. Cree el objeto que hay que devolver definiendo una clase interna anónima que herede de la primera clase.

Un nuevo análisis del método factoría

Observe que el ejemplo `interfaces/Factories.java` es mucho más atractivo cuando se utilizan clases internas anónimas:

```
//: innerclasses/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    private Implementation1() {}
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation1();
            }
        };
}

class Implementation2 implements Service {
    private Implementation2() {}
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation2();
            }
        };
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(Implementation1.factory);
    }
}
```

```

    // Las implementaciones son completamente intercambiables:
    serviceConsumer(Implementation2.factory);
}
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:-
```

Ahora, los constructores de **Implementation1** e **Implementation2** pueden ser privados y no hay ninguna necesidad de crear una clase nominada como factoría. Además, a menudo sólo hace falta un único objeto factoría, de modo que aquí lo hemos creado como un campo estático en la implementación de **Service**. Asimismo, la sintaxis resultante es más clara.

También podemos mejorar el ejemplo **interfaces/Games.java** utilizando clases internas anónimas:

```

//: innerclases/Games.java
// Utilización de clases internas anónimas con el marco de trabajo Game.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private Checkers() {}
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Checkers(); }
    };
}

class Chess implements Game {
    private Chess() {}
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Chess(); }
    };
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
    public static void main(String[] args) {
        playGame(Checkers.factory);
        playGame(Chess.factory);
    }
} /* Output:
```

```

Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*///:-

```

Recuerde el consejo que hemos dado al final del último capítulo: *Utilice las clases con preferencia a las interfaces*. Si su diseño necesita una interfaz, ya se dará cuenta de ello. En caso contrario, no emplee una interfaz a menos que se vea obligado.

Ejercicio 16: (1) Modifique la solución del Ejercicio 18 del Capítulo 9, *Interfaces* para utilizar clases internas anónimas.

Ejercicio 17: (1) Modifique la solución del Ejercicio 19 del Capítulo 9, *Interfaces* para utilizar clases internas anónimas.

Clases anidadas

Si no es necesario disponer de una conexión entre el objeto de la clase interna y el objeto de la clase externa, podemos definir la clase interna como estática. Esto es lo que comúnmente se denomina una *clase anidada*.² Para comprender el significado de la palabra clave **static** cuando se la aplica a las clases internas, hay que recordar que el objeto de una clase interna normal mantiene implicitamente una referencia al objeto de la clase contenedora que lo ha creado. Sin embargo, esto no es cierto cuando definimos una clase interna como estática. Por tanto, una clase anidada significa:

1. Que no es necesario un objeto de la clase externa para crear un objeto de la clase anidada.
2. Que no se puede acceder a un objeto no estático de la clase externa desde un objeto de una clase anidada.

Las clases anidadas difieren de las clases internas ordinarias también en otro aspecto. Los campos y los métodos en las clases internas normales sólo pueden encontrarse en el nivel externo de una clase, por lo que las clases internas normales no pueden tener datos estáticos, campos estáticos o clases anidadas. Sin embargo, las clases anidadas pueden tener cualquiera de estos elementos:

```

//: innerclasses/Parcel11.java
// Clases anidadas (clases internas estáticas).

public class Parcel11 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected static class ParcelDestination
        implements Destination {
        private String label;
        private ParcelDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
        // Las clases anidadas pueden contener otros elementos estáticos:
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
            public static void f() {}
            static int x = 10;
        }
    }
}

```

² Guarda cierto parecido con las clases anidadas de C++, salvo porque dichas clases no permiten acceder a miembros privados a diferencia de lo que sucede en Java.

```

public static Destination destination(String s) {
    return new ParcelDestination(s);
}
public static Contents contents() {
    return new ParcelContents();
}
public static void main(String[] args) {
    Contents c = contents();
    Destination d = destination("Tasmania");
}
} //:-/

```

En **main()**, no es necesario ningún objeto de **Parcel11**; en su lugar, utilizamos la sintaxis normal para seleccionar un miembro estático con el que invocar los métodos que devuelven referencias a **Contents** y **Destination**.

Como hemos visto anteriormente en el capítulo, en una clase interna normal (no estática), el vínculo con el objeto de clase externa se utiliza empleando una referencia **this** especial. Una clase anidada no tiene referencia **this** especial, lo que hace que sea análoga a un método estático.

Ejercicio 18: (1) Cree una clase que contenga una clase anidada. En **main()**, cree una instancia de la clase anidada.

Ejercicio 19: (2) Cree una clase que contenga una clase interna que a su vez contenga otra clase interna. Repita el proceso utilizando clases anidadas. Observe los nombres de los archivos **.class** generados por el compilador.

Clases dentro de interfaces

Normalmente, no podemos incluir cualquier código dentro de una interfaz, pero una clase anidada *puede* ser parte de una interfaz. Cualquier clase que coloquemos dentro de una interfaz será automáticamente pública y estática. Puesto que la clase es estática no viola las reglas de las interfaces; simplemente, la clase anidada se incluye dentro del espacio de nombres de la interfaz. Podemos incluso implementar la interfaz contenedora dentro de la clase contenedora de la forma siguiente, como por ejemplo en:

```

//: innerclasses/ClassInInterface.java
// {main: ClassInInterface$Test}

public interface ClassInInterface {
    void howdy();
}
class Test implements ClassInInterface {
    public void howdy() {
        System.out.println("Howdy!");
    }
    public static void main(String[] args) {
        new Test().howdy();
    }
}
} /* Output:
Howdy!
*///:-

```

Resulta bastante útil anidar una clase dentro de una interfaz cuando queremos crear un código común que haya que emplear con todas las diferentes implementaciones de dicha interfaz.

Anteriormente en el libro ya sugerímos incluir un método **main()** en todas las clases, con el fin de utilizarlo como mecanismo de prueba de estas clases. Una desventaja de esta técnica es la cantidad de código compilado adicional con la que hay que trabajar. Si esto representa un problema, pruebe a utilizar una clase anidada para incluir el código de prueba:

```

//: innerclasses/TestBed.java
// Inclusión del código de prueba en una clase anidada.
// {main: TestBed$Tester}

public class TestBed {

```

```

public void f() { System.out.println("f()"); }
public static class TestBed {
    public static void main(String[] args) {
        TestBed t = new TestBed();
        t.f();
    }
}
/* Output:
f()
*///:-

```

Esto genera una clase separada denominada **TestBed\$Tester** (para ejecutar el programa, escribiríamos `java TestBed$Tester`). Puede utilizar esta clase para las pruebas, pero no necesitará incluirla en el producto final, bastará con borrar **TestBed\$Tester.class** antes de crear el producto definitivo.

Ejercicio 20: (1) Cree una interfaz que contenga una clase anidada. Implemente esta interfaz y cree una instancia de la clase anidada.

Ejercicio 21: (2) Cree una interfaz que contenga una clase anidada en la que haya un método estático que invoque los métodos de la interfaz y muestre los resultados. Implemente la interfaz y pase al método una instancia de la implementación.

Acceso al exterior desde una clase múltiplemente anidada

No importa con qué profundidad pueda estar anidada una clase interna: la clase anidada podrá acceder transparentemente a todos los miembros de todas las clases dentro de las cuales esté anidada, como podemos ver en el siguiente ejemplo:³

```

//: innerclasses/MultiNestingAccess.java
// Las clases anidadas pueden acceder a todos los miembros de
// todos los niveles de las clases en las que está anidada.

class MNA {
    private void f() {}
    class A {
        private void g() {}
        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}

public class MultiNestingAccess {
    public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
    }
} //:-

```

Puede ver que en **MNA.A.B**, los métodos **g()** y **f()** son invocables sin necesidad de ninguna cualificación (a pesar del hecho de que son privados). Este ejemplo también ilustra la sintaxis necesaria para crear objetos de clases internas múltiplemente anidadas cuando se crean los objetos en una clase diferente. La sintaxis `“.new”` genera el ámbito correcto, por lo que no hace falta cualificar el nombre de la clase dentro de la llamada al constructor.

³ Gracias de nuevo a Martin Danner.

¿Para qué se usan las clases internas?

Hasta este momento, hemos analizado buena parte de los detalles sintácticos y semánticos que describen la forma de funcionar de las clases internas, pero esto no responde a la pregunta de para qué sirven las clases internas. ¿Por qué los diseñadores de Java se tomaron tantas molestias para añadir esta característica fundamental al lenguaje?

Normalmente, la clase interna hereda de otra clase o implementa una interfaz y el código incluido en la clase interna manipula el objeto de la clase externa dentro del cual hubiera sido creado. Así pues, podríamos decir que una clase interna proporciona una especie de ventana hacia la clase externa.

Una de las cuestiones fundamentales acerca de las clases internas es la siguiente: si simplemente necesitamos una referencia a una interfaz, ¿por qué no hacemos simplemente que la clase externa implemente dicha interfaz? La respuesta es que: "Si eso es todo lo que necesita, entonces esa es la manera de hacerlo". Por tanto, ¿qué es lo que distingue una clase interna que implementa una interfaz de una clase externa que implementa la misma interfaz? La respuesta es que no siempre disponemos de la posibilidad de trabajar con interfaces, sino que en ocasiones nos vemos forzados a trabajar con implementaciones. Por tanto, la razón más evidente para utilizar clases internas es la siguiente:

Cada clase interna puede heredar de una implementación de manera independiente. Por tanto, la clase interna no está limitada por el hecho de si la clase externa ya está heredando de una implementación.

Sin la capacidad de las clases internas para heredar, en la práctica, de más de una clase concreta o abstracta, algunos problemas de diseño y de programación serían intratables. Por tanto, una forma de contemplar las clases internas es decir que representan el resto de la solución del problema de la herencia múltiple. Las interfaces resuelven parte del problema, pero las clases internas permiten en la práctica una "herencia de múltiples implementaciones". En otras palabras, las clases internas nos permiten en la práctica heredar de varios elementos que no sean interfaces.

Para analizar esto con mayor detalle, piense en una situación en la que tuviéramos dos interfaces que deban de alguna forma ser implementadas dentro de una clase. Debido a la flexibilidad de las interfaces, tenemos dos opciones: una única clase o una clase interna.

```
//: innerclasses/MultiInterfaces.java
// Dos formas de implementar múltiples interfaces con una clase.
package innerclasses;

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Clase interna anónima:
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
} ///:-
```

Por supuesto, esto presupone que la estructura del código tenga sentido en ambos casos desde el punto de vista lógico. Sin embargo, normalmente dispondremos de algún tipo de directriz, extraída de la propia naturaleza del problema, que nos indicará si debemos utilizar una única clase o una clase interna, pero en ausencia de cualquier otra restricción, la técnica utilizada en el ejemplo anterior no presenta muchas diferencias desde el punto de vista de la implementación. Ambas soluciones funcionan adecuadamente.

Sin embargo, si tenemos clases abstractas o concretas en lugar de interfaces, nos veremos obligados a utilizar clases internas si nuestra clase debe implementar de alguna forma las otras clases de las que se quiere heredar:

```
//: innerclasses/MultiImplementation.java
// Con clases abstractas o concretas, las clases
// internas son la única forma de producir el efecto
// de la "herencia de múltiples implementaciones".
package innerclasses;

class D {}
abstract class E {}

class Z extends D {
    E makeE() { return new E(); }
}

public class MultiImplementation {
    static void takesD(D d) {}
    static void takesE(E e) {}
    public static void main(String[] args) {
        Z z = new Z();
        takesD(z);
        takesE(z.makeE());
    }
} //EZ-
```

Si no necesitáramos resolver el problema de la “herencia de múltiples implementaciones”, podríamos escribir el resto del programa sin necesidad de utilizar clases internas. Pero con las clases internas tenemos, además, las siguientes características adicionales:

1. La clase interna puede tener múltiples instancias, cada una con su propia información de estado que es independiente de la información contenida en el objeto de la clase externa.
2. En una única clase externa, podemos tener varias clases internas, cada una de las cuales implementa la misma interfaz o hereda de la misma clase en una forma diferente. En breve mostraremos un ejemplo de este caso.
3. El punto de creación del objeto de la clase interna no está ligado a la creación del objeto de la clase externa.
4. No existe ninguna relación de tipo “es-un” potencialmente confusa con la clase interna, se trata de una entidad separada.

Por ejemplo, si **Sequence.java** no utilizara clases internas, estaríamos obligados a decir que “un objeto **Sequence** es un objeto **Selector**”, y sólo podría existir un objeto **Selector** para cada objeto **Sequence** concreto. Sin embargo, podríamos fácilmente pensar en definir un segundo método, **reverseSelector()**, que produjera un objeto **Selector** que se desplazara en sentido inverso a través de la secuencia. Este tipo de flexibilidad sólo está disponible con las clases internas.

Ejercicio 22: (2) Implemente **reverseSelector()** en **Sequence.java**.

Ejercicio 23: (4) Cree una interfaz **U** con tres métodos. Cree una clase **A** con un método que genere una referencia a **U**, definiendo una clase interna anónima. Cree una segunda clase **B** que contenga una matriz de **U**. **B** debe tener un método que acepte y almacene una referencia a **U** en la matriz, un segundo método que configure una referencia en la matriz (especificada mediante el argumento del método) con el valor **null**, y un tercer método que se desplace a través de la matriz e invoque los métodos de **U**. En **main()**, cree un grupo de objetos **A** y un único **B**. Rellene el objeto **B** con referencias a **U** generadas por los objetos **A**. Utilice el objeto **B** para realizar llamadas a todos los objetos **A**. Elimine algunas de las referencias **U** del objeto **B**.

Cierres y retrollamada

Un *cierre (closure)* es un objeto invocable que retiene información acerca del ámbito en que fue creado. Teniendo en cuenta esta definición, podemos ver que una clase interna es un cierre orientado a objetos, porque no contiene simplemente cada elemento de información del objeto de la clase externa ("el ámbito en que fue creado"), sino que almacena de manera automática una referencia que apunta al propio objeto de la clase externa, en el cual tiene permiso para manipular todos los miembros, incluso aunque sea privados.

Uno de los argumentos más sólidos que se proporcionaron para incluir algún mecanismo de punteros en Java era el de permitir las *retrollamadas (callbacks)*. Con una retrollamada, se proporciona a algún otro objeto un elemento de información que le permite llamar al objeto original en un momento posterior. Se trata de un concepto muy potente, como veremos más adelante. Sin embargo, si se implementa una retrollamada utilizando un puntero, nos veremos forzados a confiar en que el programador se comporte correctamente y no haga un mal uso del puntero. Como hemos visto hasta el momento, el lenguaje Java tiende a ser bastante más precavido, por lo que no se han incluido punteros en el lenguaje.

El cierre proporcionado por la clase interna es una buena solución, bastante más flexible y segura que la basada en punteros. Veamos un ejemplo:

```
//: innerclasses/Callbacks.java
// Utilización de clases internas para las retrollamadas
package innerclasses;
import static net.mindview.util.Print.*;

interface Incrementable {
    void increment();
}

// Muy simple para limitarse a implementar la interfaz:
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        print(i);
    }
}

class MyIncrement {
    public void increment() { print("Other operation"); }
    static void f(MyIncrement mi) { mi.increment(); }
}

// Si nuestra clase debe implementar increment() de
// alguna otra forma, es necesario utilizar una clase interna:
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        print(i);
    }
    private class Closure implements Incrementable {
        public void increment() {
            // Especifique un método de la clase externa; en caso
            // contrario, se produciría una recursión infinita:
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
}
```

```

}

class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) { callbackReference = cbh; }
    void go() { callbackReference.increment(); }
}

public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 = new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
} /* Output:
Other operation
1
1
2
Other operation
2
Other operation
3
*///:-*

```

Esto muestra también una distinción adicional entre el hecho de implementar una interfaz en una clase externa y el hecho de hacerlo en una clase interna. **Callee1** es claramente la solución más simple en términos de código. **Callee2** hereda de **MyIncrement**, que ya dispone de un método **increment()** diferente que lleva a cabo alguna tarea que no está relacionada con la que la interfaz **Incrementable** espera. Cuando se hereda **MyIncrement** en **Callee2**, **increment()** no puede ser sustituido para que lo utilice **Incrementable**, por lo que estamos obligados a proporcionar una implementación separada mediante una clase interna. Observe también que cuando se crea una clase interna no se añade nada a la interfaz de la clase externa ni se la modifica de ninguna manera.

Todo en **Callee2** es privado salvo **getCallbackReference()**. Para permitir *algún tipo* de conexión con el mundo exterior, la interfaz **Incrementable** resulta esencial. Con este ejemplo podemos ver que las interfaces permiten una completa separación entre la interfaz y la implementación.

La clase interna **Closure** implementa **Incrementable** para proporcionar un enganche con **Callee2**, pero un enganche que sea lo suficientemente seguro. Cualquiera que obtenga la referencia a **Incrementable** sólo puede por supuesto invocar **increment()** y no tiene a su disposición ninguna otra posibilidad (a diferencia de un puntero, que nos permitiría hacer cualquier cosa).

Caller toma una referencia a **Incrementable** en su constructor (aunque la captura de la referencia de retrollamada podría tener lugar en cualquier instante) y luego en algún momento posterior utiliza la referencia para efectuar una retrollamada a la clase **Callee**.

El valor de las retrollamadas radica en su flexibilidad; podemos decidir de manera dinámica qué métodos van a ser invocados en tiempo de ejecución. Las ventajas de esta manera de proceder resultarán evidentes en el Capítulo 22, *Interfaces gráficas de usuario*, en el que emplearemos de manera intensiva las retrollamadas para implementar la funcionalidad GUI (*Graphical User Interface*).

Clases internas y marcos de control

Un ejemplo más concreto del uso de clases internas es el que puede encontrarse en lo que denominamos *marco de control*.

Un *marco de trabajo de una aplicación* es una clase o un conjunto de clases diseñado para resolver un tipo concreto de problema. Para aplicar un marco de trabajo de una aplicación, lo que normalmente se hace es heredar de una o más clases y sustituir algunos de los métodos. El código que escribamos en los métodos sustituidos sirve para personalizar la solución general proporcionada por dicho marco de trabajo de la aplicación, con el fin de resolver nuestros problemas específicos. Se trata de un ejemplo del patrón de diseño basado en el *método de plantillas* (véase *Thinking in Patterns (with Java)* en www.MindView.net). El método basado en plantillas contiene la estructura básica del algoritmo, e invoca uno o más métodos sustituibles con el fin de completar la acción que el algoritmo dictamina. Los patrones de diseño separan las cosas que no cambian de las cosas que sí que sufren modificación y en este caso el método basado en plantillas es la parte que permanece invariable, mientras que los métodos sustituibles son los elementos que se modifican.

Un marco de control es un tipo particular de marco de trabajo de aplicación, que está dominado por la necesidad de responder a cierto suceso. Los sistemas que se dedican principalmente a responder a sucesos se denominan *sistemas dirigidos por sucesos*. Un problema bastante común en la programación de aplicaciones es la interfaz gráfica de usuario (GUI), que está casi completamente dirigida por sucesos. Como veremos en el Capítulo 22, *Interfaces gráficas de usuario*, la biblioteca Swing de Java es un marco de control que resuelve de manera elegante el problema de las interfaces GUI y que utiliza de manera intensiva las clases internas.

Para ver la forma en que las clases internas permiten crear y utilizar de manera sencilla marcos de control, considere un marco de control cuyo trabajo consista en ejecutar sucesos cada vez que dichos sucesos estén "listos". Aunque "listos" podría significar cualquier cosa, en este caso nos basaremos en el dato de la hora actual. El ejemplo que sigue es un marco de control que no contiene ninguna información específica acerca de qué es aquello que está controlando. Dicha información se suministra mediante el mecanismo de herencia, cuando se implementa la parte del algoritmo correspondiente al método `action()`.

En primer lugar, he aquí la interfaz que describe los sucesos de control. Se trata de una clase abstracta, en lugar de una verdadera interfaz, porque el comportamiento predeterminado consiste en llevar a cabo el control dependiendo del instante actual. Por tanto, parte de la implementación se incluye aquí:

```
//: innerclasses/controller/Event.java
// Los métodos comunes para cualquier suceso de control.
package innerclasses.controller;

public abstract class Event {
    private long eventTime;
    protected final long delayTime;
    public Event(long delayTime) {
        this.delayTime = delayTime;
        start();
    }
    public void start() { // Permite la reinicialización
        eventTime = System.nanoTime() + delayTime;
    }
    public boolean ready() {
        return System.nanoTime() >= eventTime;
    }
    public abstract void action();
} ///:-
```

El constructor captura el tiempo (medido desde el instante de creación del objeto) cuando se quiere ejecutar el objeto `Event`, y luego invoca `start()`, que toma el instante actual y añade el retardo necesario, con el fin de generar el instante en el que el suceso tendrá lugar. En lugar de incluirlo en el constructor, `start()` es un método independiente. De esta forma, se puede reiniciar el temporizador después de que el suceso haya caducado, de manera que el objeto `Event` puede reutilizarse. Por ejemplo, si queremos un suceso repetitivo, podemos invocar simplemente `start()` dentro del método `action()`.

`ready()` nos dice cuándo es el momento de ejecutar el método `action()`. Por supuesto, `ready()` puede ser sustituido en una clase derivada, con el fin de basar el suceso `Event` en alguna otra cosa distinta del tiempo.

El siguiente archivo contiene el marco de control concreto que gestiona y dispara los sucesos. Los objetos `Event` se almacenan dentro de un objeto contenedor de tipo `List<Event>` (una lista de sucesos), que es un tipo de objeto que analizare-

mos en más detalle en el Capítulo 11, *Almacenamiento de objetos*. Pero ahora lo único que necesitamos saber es que `add()` añade un objeto `Event` al final de la lista `List`, que `size()` devuelve el número de elementos de `List`, que la sintaxis `foreach` permite extraer objetos `Event` sucesivos de `List`, y que `remove()` elimina el objeto `Event` especificado de `List`.

```
//: innerclasses/controller/Controller.java
// El marco de trabajo reutilizable para sistemas de control.
package innerclasses.controller;
import java.util.*;

public class Controller {
    // Una clase de java.util para almacenar los objetos Event:
    private List<Event> eventList = new ArrayList<Event>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0)
            // Hacer una copia para no modificar la lista
            // mientras se están seleccionando sus elementos:
            for(Event e : new ArrayList<Event>(eventList))
                if(e.ready())
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
    }
}
} //:-
```

El método `run()` recorre en bucle una copia de `eventList`, buscando un objeto `Event` que esté listo para ser ejecutado. Para cada uno que encuentra, imprime información utilizando el método `toString()` del objeto, invoca el método `action()` y luego elimina el objeto `Event` de la lista.

Observe que en este diseño, hasta ahora, no sabemos nada acerca de *qué es* exactamente lo que un objeto `Event` hace. Y éste es precisamente el aspecto fundamental del diseño: la manera en que “separa las cosas que cambian de las cosas que permanecen iguales”. O, por utilizar un término que a mí personalmente me gusta, el “vector de cambio” está compuesto por las diferentes acciones de los objetos `Event`, y podemos expresar diferentes acciones creando distintas subclases de `Event`.

Aquí es donde entran en juego las clases internas. Estas clases nos permiten dos cosas:

1. La implementación completa de un marco de control se crea en una única clase, encapsulando de esa forma todas aquellas características distintivas de dicha implementación. Las clases internas se usan para expresar los múltiples tipos distintos de acciones [`action()`] necesarias para resolver el problema.
2. Las clases internas evitan que esta implementación sea demasiado confusa, ya que podemos acceder fácilmente a cualquiera de los miembros de la clase externa. Sin esta capacidad, el código podría llegar a ser tan complejo que terminaríamos tratando de buscar una alternativa.

Considere una implementación concreta del marco de control diseñado para regular las funciones de un invernadero.⁴ Cada acción es totalmente distinta: encender y apagar las luces, iniciar y detener el riego, apagar y encender los termostatos, hacer sonar alarmas y reiniciar el sistema. Pero el marco de control está diseñado de tal manera que se aíslan fácilmente estas distintas secciones del código. Las clases internas permiten disponer de múltiples versiones derivadas de la misma clase base, `Event`, dentro de una misma clase. Para cada tipo de acción, heredamos una nueva clase interna `Event` y escribimos el código de control en la implementación de `action()`.

Como puede suponer por los marcos de trabajo para aplicaciones, la clase `GreenhouseControls` hereda de `Controller`:

```
//: innerclasses/GreenhouseControls.java
// Genera una aplicación específica del sistema
```

⁴ Por alguna razón, este problema siempre me ha resultado bastante grato de resolver; proviene de mi anterior libro *C++ Inside & Out*, pero Java permite obtener una solución más elegante.

```

// de control, dentro de una única clase. Las clases
// internas permiten encapsular diferente funcionalidad
// para cada tipo de suceso.
import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) { super(delayTime); }
        public void action() {
            // Poner código de control del hardware aquí
            // para encender físicamente las luces.
            light = true;
        }
        public String toString() { return "Light is on"; }
    }
    public class LightOff extends Event {
        public LightOff(long delayTime) { super(delayTime); }
        public void action() {
            // Poner código de control del hardware aquí
            // para apagar físicamente las luces.
            light = false;
        }
        public String toString() { return "Light is off"; }
    }
    private boolean water = false;
    public class WaterOn extends Event {
        public WaterOn(long delayTime) { super(delayTime); }
        public void action() {
            // Poner el código de control del hardware aquí.
            water = true;
        }
        public String toString() {
            return "Greenhouse water is on";
        }
    }
    public class WaterOff extends Event {
        public WaterOff(long delayTime) { super(delayTime); }
        public void action() {
            // Poner el código de control del hardware aquí.
            water = false;
        }
        public String toString() {
            return "Greenhouse water is off";
        }
    }
    private String thermostat = "Day";
    public class ThermostatNight extends Event {
        public ThermostatNight(long delayTime) {
            super(delayTime);
        }
        public void action() {
            // Poner el código de control del hardware aquí.
            thermostat = "Night";
        }
        public String toString() {
            return "Thermostat on night setting";
        }
    }
}

```

```

public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Poner el código de control del hardware aquí.
        thermostat = "Day";
    }
    public String toString() {
        return "Thermostat on day setting";
    }
}
// Un ejemplo de action() que inserta un
// nuevo ejemplar de sí misma en la línea de sucesos:
public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
        addEvent(new Bell(delayTime));
    }
    public String toString() { return "Bing!"; }
}
public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
        super(delayTime);
        this.eventList = eventList;
        for(Event e : eventList)
            addEvent(e);
    }
    public void action() {
        for(Event e : eventList) {
            e.start(); // Re-ejecutar cada suceso.
            addEvent(e);
        }
        start(); // Re-ejecutar cada suceso
        addEvent(this);
    }
    public String toString() {
        return "Restarting system";
    }
}
public static class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating"; }
}
} //:-.

```

Observe que `light`, `water` y `thermostat` pertenecen a la clase externa `GreenhouseControls`, a pesar de lo cual las clases internas pueden acceder a dichos campos sin ninguna cualificación y sin ningún permiso especial. Asimismo, los métodos `action()` suelen requerir algún tipo de control del hardware.

La mayoría de las clases `Event` parecen similares, pero `Bell` y `Restart` son especiales. `Bell` hace sonar una alarma y luego añade un nuevo objeto `Bell` a la lista de sucesos, para que vuelva a sonar posteriormente. Observe cómo las clases internas *casi parecen* un verdadero mecanismo de herencia múltiple. `Bell` y `Restart` tienen todos los métodos de `Event` y también parecen tener todos los métodos de la clase externa `GreenhouseControls`.

A `Restart` se le proporciona una matriz de objetos `Event` y aquélla se encarga de añadirla al controlador. Puesto que `Restart()` es simplemente otro objeto `Event`, también se puede añadir un objeto `Restart` dentro de `Restart.action()` para que el sistema se reinicie a sí mismo de manera periódica.

La siguiente clase configura el sistema creando un objeto **GreenhouseControls** y añadiendo diversos tipos de objetos **Event**. Esto es un ejemplo del patrón de diseño *Command*: cada objeto de **eventList** es una solicitud encapsulada en forma de objeto:

```
//: innerclasses/GreenhouseController.java
// Configurar y ejecutar el sistema de control de invernadero.
// {Args: 5000}
import innerclasses.controller.*;

public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // En lugar de fijar los valores, podríamos analizar
        // información de configuración incluida
        // en un archivo de texto;
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(
                new GreenhouseControls.Terminate(
                    new Integer(args[0])));
        gc.run();
    }
} /* Output:
Bing!
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Thermostat on day setting
Restarting system
Terminating
*/:-
```

Esta clase inicializa el sistema, para que añada todos los sucesos apropiados. El suceso **Restart** se ejecuta repetidamente y carga cada vez la lista **eventList** en el objeto **GreenhouseControls**. Si proporcionamos un argumento de línea de comandos que indique los milisegundos, **Restart** terminará el programa después de ese número de milisegundos especificado (esto se usa para las pruebas).

Por supuesto, resulta más flexible leer los sucesos de un archivo en lugar de leerlos en el código. Uno de los ejercicios del Capítulo 18, *E/S*, pide, precisamente, que modifiquemos este ejemplo para hacer eso.

Este ejemplo debería permitir apreciar cuál es el valor de las clases internas, especialmente cuando se las usa dentro de un marco de control. Sin embargo, en el Capítulo 22, *Interfaces gráficas de usuario*, veremos de qué forma tan elegante se utilizan las clases internas para definir las acciones de una interfaz gráfica de usuario. Al terminar ese capítulo, espero haberle convencido de la utilidad de ese tipo de clases.

Ejercicio 24: (2) En **GreenhouseControls.java**, añada una serie de clases internas **Event** que permitan encender y apagar una serie de ventiladores. Configure **GreenhouseController.java** para utilizar estos nuevos objetos **Event**.

Ejercicio 25: (3) Herede de **GreenhouseControls** en **GreenhouseControls.java** para añadir clases internas **Event** que permitan encender y apagar una serie de vaporizadores. Escriba una nueva versión de **GreenhouseController.java** para utilizar estos nuevos objetos **Event**.

Cómo heredar de clases internas

Puesto que el constructor de la clase interna debe efectuar la asociación como una referencia al objeto de la clase contenedora, las cosas se complican ligeramente cuando tratamos de heredar de una clase interna. El problema es que la referencia "secreta" al objeto de la clase contenedora *debe inicializarse*, a pesar de lo cual en la clase derivada no hay ningún objeto predeterminado con el que asociarse. Es necesario utilizar una sintaxis especial para que dicha asociación se haga de forma explícita:

```
//: innerclasses/InheritInner.java
// Heredando de una clase interna.

class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    //! InheritInner() {} // No se compilará
    InheritInner(WithInner wi) {
        wi.super();
    }
    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
} //:-
```

Puede ver que **InheritInner** sólo amplía la clase interna, no la externa. Pero cuando llega el momento de crear un constructor, el predeterminado no sirve y no podemos limitarnos a pasar una referencia a un objeto contenedor. Además, es necesario utilizar la sintaxis:

```
enclosingClassReference.super();
```

dentro del constructor. Esto proporciona la referencia necesaria y el programa podrá así compilarse.

Ejercicio 26: (2) Cree una clase con una clase interna que tenga un constructor no predeterminado (uno que tome argumentos). Cree una segunda clase con una clase interna que herede de la primera clase interna.

¿Pueden sustituirse las clases internas?

¿Qué sucede cuando creamos una clase interna, luego heredamos de la clase contenedora y redefinimos la clase interna? En otras palabras, ¿es posible "sustituir" la clase interna completa? Podría parecer que esta técnica resultaría muy útil, pero el "sustituir" una clase interna como si fuera otro método cualquiera de la clase externa no tiene, en realidad, ningún efecto:

```
//: innerclasses/BigEgg.java
// No se puede sustituir una clase interna como si fuera un método.
import static net.mindview.util.Print.*;

class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() { print("Egg.Yolk()"); }
    }
    public Egg() {
        print("New Egg()");
        y = new Yolk();
    }
}
```

```

    }

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() { print("BigEgg.Yolk()"); }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} /* Output:
New Egg()
Egg.Yolk()
*///:-

```

El compilador sintetiza automáticamente el constructor predeterminado, y éste invoca al constructor predeterminado de la clase base. Podríamos pensar que puesto que se está creando un objeto **BigEgg**, se utilizará la versión “sustituida” de **Yolk**, pero esto no es así, como podemos ver analizando la salida.

Este ejemplo muestra que no hay ningún mecanismo mágico adicional relacionado con las clases internas que entre en acción al heredar de la clase externa. Las dos clases internas son entidades completamente separadas, cada una con su propio espacio de nombres. Sin embargo, lo que sigue siendo posible es heredar explícitamente de la clase interna:

```

//: innerclasses/BigEgg2.java
// Herencia correcta de una clase interna.
import static net.mindview.util.Print.*;

class Egg2 {
    protected class Yolk {
        public Yolk() { print("Egg2.Yolk()"); }
        public void f() { print("Egg2.Yolk.f()"); }
    }
    private Yolk y = new Yolk();
    public Egg2() { print("New Egg2()"); }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { print("BigEgg2.Yolk()"); }
        public void f() { print("BigEgg2.Yolk.f()"); }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} /* Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*///:-

```

Ahora, **BigEgg2.Yolk** amplía explícitamente **extends Egg2.Yolk** y sustituye sus métodos. El método **insertYolk()** permite que **BigEgg2** generalice uno de sus propios objetos **Yolk** a la referencia **y** en **Egg2**, por lo que **g()** invoca **y.f()**, se utiliza la versión sustituida de **f()**. La segunda llamada a **Egg2.Yolk()** es la llamada que el constructor de la clase base hace al constructor de **BigEgg2.Yolk**. Como puede ver, cuando se llama a **g()** se utiliza la versión sustituida de **f()**.

Clases internas locales

Como hemos indicado anteriormente, también pueden crearse clases internas dentro de bloques de código, normalmente dentro del cuerpo de un método. Una clase interna local no puede tener un especificador de acceso, porque no forma parte de la clase externa, pero sí que tiene acceso a las variables finales del bloque de código actual y a todos los miembros de la clase contenedora. He aquí un ejemplo donde se compara la creación de una clase interna local con la de una clase interna anónima:

```
//: innerclasses/LocalInnerClass.java
// Contiene una secuencia de objetos.
import static net.mindview.util.Print.*;

interface Counter {
    int next();
}

public class LocalInnerClass {
    private int count = 0;
    Counter getCounter(final String name) {
        // Una clase interna local:
        class LocalCounter implements Counter {
            public LocalCounter() {
                // La clase interna local puede tener un constructor
                print("LocalCounter()");
            }
            public int next() {
                printnb(name); // Acceso a variable local final
                return count++;
            }
        }
        return new LocalCounter();
    }
    // Lo mismo con una clase interna anónima:
    Counter getCounter2(final String name) {
        return new Counter() {
            // La clase interna anónima no puede tener un constructor
            // nominado, sino sólo un inicializador de instancia:
            {
                print("Counter()");
            }
            public int next() {
                printnb(name); // Acceso a una variable local final
                return count++;
            }
        };
    }
    public static void main(String[] args) {
        LocalInnerClass lic = new LocalInnerClass();
        Counter
            c1 = lic.getCounter("Local inner "),
            c2 = lic.getCounter2("Anonymous inner ");
        for(int i = 0; i < 5; i++)
            print(c1.next());
        for(int i = 0; i < 5; i++)
            print(c2.next());
    }
} /* Output:
LocalCounter()
Counter()
```

```

Local inner 0
Local inner 1
Local inner 2
Local inner 3
Local inner 4
Anonymous inner 5
Anonymous inner 6
Anonymous inner 7
Anonymous inner 8
Anonymous inner 9
*///:-

```

Counter devuelve el siguiente valor de una secuencia. Está implementado como una clase local y como una clase interna anónima, teniendo ambas los mismos comportamientos y capacidades. Puesto que el nombre de la clase interna local no es accesible fuera del método, la única justificación para utilizar una clase interna local en lugar de una clase interna anónima es que necesitemos un constructor nominado y/o un constructor sobrecargado, ya que una clase interna anónima sólo puede utilizar un mecanismo de inicialización de instancia.

Otra razón para definir una clase interna local en lugar de una clase interna anónima es que necesitemos construir más de un objeto de dicha clase.

Identificadores de una clase interna

Puesto que todas las clases generan un archivo `.class` que almacena toda la información relativa a cómo crear objetos de dicho tipo (esta información genera una “metaclasa” denominada objeto **Class**), podemos imaginar fácilmente que las clases internas también deberán producir archivos `.class` para almacenar la información de sus objetos **Class**. Los nombres de estos archivos /classes responden a una fórmula estricta: el nombre de la clase contenedora, seguido de un signo ‘\$’, seguido del nombre de la clase interna. Por ejemplo, los archivos `.class` creados por **LocalInnerClass.java** incluyen:

```

Counter.class
LocalInnerClass$1.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class

```

Si las clases internas son anónimas, el compilador simplemente genera una serie de números para que actúen como identificadores de las clases internas. Si las clases internas están anidadas dentro de otras clases internas, sus nombres se añaden simplemente después de un ‘\$’ y del identificador o identificadores de las clases externas.

Aunque este esquema de generación de nombres internos resulta simple y directo, también es bastante robusto y permite tratar la mayoría de las situaciones.⁵ Puesto que se trata del esquema estándar de denominación para Java, los archivos generados son automáticamente independientes de la plataforma (tenga en cuenta que el compilador Java modifica las clases internas de múltiples maneras para hacer que funcionen adecuadamente).

Resumen

Las interfaces y las clases internas son conceptos bastante más sofisticados que los que se pueden encontrar en muchos lenguajes de programación orientada a objetos; por ejemplo, no podremos encontrar nada similar en C++. Ambos conceptos resuelven, conjuntamente, el mismo problema que C++ trata de resolver con su mecanismo de herencia múltiple. Sin embargo, el mecanismo de herencia múltiple en C++ resulta bastante difícil de utilizar, mientras que las interfaces y las clases internas de Java son, por comparación, mucho más accesibles.

Aunque estas funcionalidades son, por sí mismas, razonablemente sencillas, el uso de las mismas es una de las cuestiones fundamentales de diseño, de forma similar a lo que ocurre con el polimorfismo. Con el tiempo, aprenderá a reconocer

⁵ Por otro lado, ‘\$’ es un metacarácter de la *shell* Unix, por lo que en ocasiones podemos encontrarnos con problemas a la hora de listar los archivos `.class`. Resulta un tanto extraño este problema, dado que el lenguaje Java ha sido definido por Sun, una empresa volcada en el mercado Unix. Supongo que no tuvieron en cuenta el problema, pensando en que los programadores se centrarian principalmente en los archivos de código fuente.

aquellas situaciones en las que debe utilizarse una interfaz o una clase interna, o ambas cosas. Pero al menos, en este punto del libro, sí que el lector debería sentirse cómodo con la sintaxis y la semántica aplicables. A medida que vaya viendo cómo se aplican estas funcionalidades, terminará por interiorizarlas.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Almacenamiento de objetos

11

Es un programa bastante simple que sólo dispone de una cantidad de objetos con tiempos de vida conocidos.

En general, los programas siempre crearán nuevos objetos basándose en algunos criterios que sólo serán conocidos en tiempo de ejecución. Antes de ese momento, no podemos saber la cantidad ni el tipo exacto de los objetos que necesitamos. Para resolver cualquier problema general de programación, necesitamos poder crear cualquier número de objetos en cualquier momento y en cualquier lugar. Por tanto, no podemos limitarnos a crear una referencia nominada para almacenar cada uno de los objetos:

```
MiTipo unaReferencia;
```

ya que no podemos saber cuántas de estas referencias vamos a necesitar.

La mayoría de los lenguajes proporciona alguna manera de resolver este importante problema. Java dispone de varias formas para almacenar objetos (o más bien, referencias a objetos). El tipo soportado por el compilador es la matriz, de la que ya hemos hablado antes. Una matriz es la forma más eficiente de almacenar un grupo de objetos, y recomendamos utilizar esta opción cada vez que se quiera almacenar un grupo de primitivas. Pero una matriz tiene un tamaño fijo y, en el caso más general, no podemos saber en el momento de escribir el programa cuántos objetos vamos a necesitar o si hará falta una forma más sofisticada de almacenar los objetos, por lo que la restricción relativa al tamaño fijo de una matriz es demasiado limitante.

La biblioteca **java.util** tiene un conjunto razonablemente completo de *clases contenedoras* para resolver este problema, siendo los principales tipos básicos **List**, **Set**, **Queue** y **Map** (lista, conjunto, cola y mapa). Estos tipos de objetos también se conocen con el nombre de *clases de colección*, pero como la biblioteca Java utiliza el nombre **Collection** para hacer referencia a un subconjunto concreto de la biblioteca, en este texto utilizaremos el término más general de "contenedor". Los contenedores proporcionan formas sofisticadas de almacenar los objetos, y con ellos podemos resolver un sorprendente número de problemas.

Además de tener otras características (**Set**, por ejemplo, sólo almacena un objeto de cada valor mientras que **Map** es una *matriz asociativa* que permite asociar objetos con otros objetos), las clases contenedoras de Java tienen la funcionalidad de cambiar automáticamente de tamaño. Por tanto, a diferencia de las matrices, podemos almacenar cualquier número de objetos y no tenemos que preocuparnos, mientras estemos escribiendo el programa, del tamaño que tenga que tener el contenedor.

Aún cuando no tienen soporte directo mediante palabras clave de Java,¹ las clases contenedoras son herramientas fundamentales que incrementan de manera significativa nuestra potencia de programación. En este capítulo vamos a aprender los aspectos básicos de la biblioteca de contenedores de Java poniendo el énfasis en la utilización típica de los contenedores. Aquí, vamos a centrarnos en los contenedores que se utilizan de manera cotidiana en las tareas de programación. Posteriormente, en el Capítulo 17, *Análisis detallado de los contenedores*, veremos el resto de los contenedores y una serie de detalles acerca de su funcionalidad y de cómo utilizarlos.

¹ Diversos lenguajes como Perl, Python y Ruby tienen soporte nativo para los contenedores.

Genéricos y contenedores seguros respecto al tipo

Uno de los problemas de utilizar los contenedores anteriores a Java SE5 era que el compilador permitía insertar un tipo incorrecto dentro de un contenedor. Por ejemplo, considere un contenedor de objetos **Apple** que utilice el contenedor más básico general, **ArrayList**. Por ahora, podemos considerar que **ArrayList** es “una matriz que se expande automáticamente”. La utilización de una matriz **ArrayList** es bastante simple: basta con crear una, insertar objetos utilizando **add()** y acceder a ellos mediante **get()**, utilizando un índice: es lo mismo que hacemos con las matrices, pero sin emplear corchetes.² **ArrayList** también dispone de un método **size()** que nos permite conocer cuántos elementos se han añadido, para no utilizar inadvertidamente índices que estén más allá del contenedor que provoquen un error (generando una *excepción de tiempo de ejecución*; hablaremos de las excepciones en el Capítulo 12, *Tratamiento de errores mediante excepciones*).

En este ejemplo, insertamos en el contenedor objetos **Apple** y **Orange** y luego los extraemos. Normalmente, el compilador Java proporcionará una advertencia, debido a que el ejemplo *no usa genéricos*. Aquí, empleamos una *anotación* de Java SE5 para suprimir esas advertencias del compilador. Las anotaciones comienzan con un signo de '@', y admiten un argumento; esta anotación en concreto es **@SuppressWarnings** y el argumento indica que sólo deben suprimirse las advertencias no comprobadas (“unchecked”):

```
//: holding/ApplesAndOrangesWithoutGenerics.java
// Ejemplo simple de contenedor (produce advertencias del compilador).
// {ThrowsException}
import java.util.*;

class Apple {
    private static long counter;
    private final long id = counter++;
    public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList apples = new ArrayList();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // No se impide añadir un objeto Orange:
        apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            ((Apple)apples.get(i)).id();
        // Orange sólo se detecta en tiempo de ejecución
    }
} /* (Execute to see output) */:-
```

Hablaremos más en detalle de las anotaciones Java SE5 en el Capítulo 20, *Anotaciones*.

Las clases **Apple** y **Orange** son diferentes; no tienen nada en común salvo que ambas heredan de **Object** (recuerde que si no se indica explícitamente de qué clase se está heredando, se hereda automáticamente de **Object**). Puesto que **ArrayList** almacena objetos de tipo **Object**, no sólo se pueden añadir objetos **Apple** a este contenedor utilizando el método **add()** de **ArrayList**, sino que también se pueden añadir objetos **Orange** sin que se produzca ningún tipo de advertencia ni en tiempo de compilación ni en tiempo de ejecución. Cuando luego tratamos de extraer lo que pensamos que son objetos **Apple** utilizando el método **get()** de **ArrayList**, obtenemos una referencia a un objeto de tipo **Object** que deberemos proyectar sobre un objeto **Apple**. Entonces, necesitaremos encerrar toda la expresión entre paréntesis para forzar la evaluación de la proyección antes de invocar el método **id()** de **Apple**; en caso contrario, se producirá un error de sintaxis.

² Éste es uno de los casos en los que la sobrecarga de operadores resultaría conveniente. Las clases contenedoras de C++ y C# producen una sintaxis más limpia utilizando la sobrecarga de operadores.

En tiempo de ejecución, al intentar proyectar el objeto **Orange** sobre un objeto **Apple**, obtendremos un error en la forma de la excepción que antes hemos mencionado.

En el Capítulo 20, *Genéricos*, veremos que la *creación* de clases utilizando los genéricos de Java puede resultar muy compleja. Sin embargo, la *aplicación* de clases genéricas predefinidas suele resultar sencilla. Por ejemplo, para definir un contenedor **ArrayList** en el que almacenar objetos **Apple**, tenemos que indicar **ArrayList<Apple>** en lugar de sólo **ArrayList**. Los corchetes angulares rodean los *parámetros de tipo* (puede haber más de uno), que especifican el tipo o tipos que pueden almacenarse en esa instancia del contenedor.

Con los genéricos evitamos, *en tiempo de compilación*, que se puedan introducir objetos de tipo incorrecto en un contenedor.³ He aquí de nuevo el ejemplo utilizando genéricos:

```
//: holding/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Error en tiempo de compilación:
        // apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            System.out.println(apples.get(i).id());
        // Utilización de foreach:
        for(Apple c : apples)
            System.out.println(c.id());
    }
} /* Output:
0
1
2
0
1
2
*///:-
```

Ahora el compilador evitara que introduzcamos un objeto **Orange** en **apples**, convirtiendo el error en tiempo de ejecución en un error de tiempo de compilación.

Observe también que la operación de proyección ya no es necesaria a la hora de extraer los elementos de la lista. Puesto que la lista conoce qué tipos de objetos almacena, ella misma se encarga de realizar la proyección por nosotros cuando invocamos **get()**. Por tanto, con los genéricos no sólo podemos estar seguros de que el compilador comprobará el tipo de los objetos que introduzcamos en un contenedor, sino que también obtendremos una sintaxis más limpia a la hora de utilizar los objetos almacenados en el contenedor.

El ejemplo muestra también que, si no necesitamos utilizar el índice de cada elemento, podemos utilizar la sintaxis *foreach* para seleccionar cada elemento de la lista.

No estamos limitados a almacenar el tipo exacto de objeto dentro de un contenedor cuando especificamos dicho tipo como un parámetro genérico. La operación de generalización (*upcasting*) funciona de igual forma con los genéricos que con los demás tipos:

```
//: holding/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
```

³ Al final del Capítulo 15, *Genéricos*, se incluye una explicación sobre la gravedad de este problema. Sin embargo, dicho capítulo también explica que los genéricos de Java resultan útiles para otras cosas además de definir contenedores que sean seguros con respecto al tipo de los datos.

```

class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        apples.add(new GrannySmith());
        apples.add(new Gala());
        apples.add(new Fuji());
        apples.add(new Braeburn());
        for(Apple c : apples)
            System.out.println(c);
    }
} /* Output: (Sample)
GrannySmith@7d772e
Gala@11b86e7
Fuji@35ce36
Braeburn@757aef
*///:-*/

```

Por tanto, podemos añadir un subtipo de **Apple** a un contenedor que hayamos especificado que va a almacenar objetos **Apple**.

La salida se produce utilizando el método **toString()** predeterminado de **Object**, que imprime el nombre de la clase seguido de una representación hexadecimal sin signo del código *hash* del objeto (generado por el método **hashCode()**). Veremos más detalles acerca de los códigos *hash* en el Capítulo 17, *Análisis detallado de los contenedores*.

Ejercicio 1: (2) Cree una nueva clase llamada **Gerbil** con una variable **int gerbilNumber** que se inicializa en el constructor. Añada un método denominado **hop()** que muestre el valor almacenado en esa variable entera. Cree una lista **ArrayList** y añada objetos **Gerbil** a la lista. Ahora, utilice el método **get()** para desplazarse a través de la lista e invoque el método **hop()** para cada objeto **Gerbil**.

Conceptos básicos

La biblioteca de contenedores Java toma la idea de “almacenamiento de los objetos” y la divide en dos conceptos distintos, expresados mediante las interfaces básicas de la biblioteca:

1. **Collection:** una secuencia de elementos individuales a los que se aplica una o más reglas. Un contenedor **List** debe almacenar los elementos en la forma en la que fueron insertados, un contenedor **Set** no puede tener elementos duplicados y un contenedor **Queue** produce los elementos en el orden determinado por una *disciplina de cola* (que normalmente es el mismo orden en el que fueron insertados).
2. **Map:** un grupo de parejas de objetos clave-valor, que permite efectuar búsquedas de valores utilizando una clave. Un contenedor **ArrayList** permite buscar un objeto utilizando un número, por lo que en un cierto sentido sirve para asociar números con los objetos. Un *mapa* permite buscar un objeto utilizando *otro objeto*. También se le denomina *matriz asociativa*, (porque asocia objetos con otros objetos) o *diccionario* (porque se utiliza para buscar un objeto valor mediante un objeto clave, de la misma forma que buscamos una definición utilizando una palabra). Los contenedores **Map** son herramientas de programación muy potentes.

Aunque no siempre es posible, deberíamos tratar de escribir la mayor parte del código para que se comunique con estas interfaces; asimismo, el único lugar en el que deberíamos especificar el tipo concreto que estemos usando es el lugar de la creación del contenedor. Así, podemos crear un contenedor **List** de la forma siguiente:

```
List<Apple> apples = new ArrayList<Apple>();
```

Observe que el contenedor **ArrayList** ha sido generalizado a un contenedor **List**, por contraste con la forma en que lo habíamos tratado en los ejemplos anteriores. La idea de utilizar la interfaz es que, si posteriormente queremos cambiar la implementación, lo único que tendremos que hacer es efectuar la modificación en el punto de creación, de la forma siguiente:

```
List<Apple> apples = new LinkedList<Apple>();
```

Así, normalmente crearemos un objeto de una clase concreta, lo generalizaremos a la correspondiente interfaz y luego utilizaremos la interfaz en el resto del código.

Esta técnica no siempre sirve, porque algunas clases disponen de funcionalidad adicional. Por ejemplo, **LinkedList** tiene métodos adicionales que no forman parte de la interfaz **List** mientras que **TreeMap** tiene métodos que no se encuentran en la interfaz **Map**. Si necesitamos usar esos métodos, no podremos efectuar la operación de generalización a la interfaz más general.

La interfaz **Collection** generaliza la idea de *secuencia*: una forma de almacenar un grupo de objetos. He aquí un ejemplo simple que rellena un contenedor **Collection** (representado aquí mediante un contenedor **ArrayList**) con objetos **Integer** y luego imprime cada elemento en el contenedor resultante:

```
//: holding/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<Integer>();
        for(int i = 0; i < 10; i++)
            c.add(i); // Autoboxing
        for(Integer i : c)
            System.out.print(i + ", ");
    }
} /* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*///:-
```

Puesto que este ejemplo sólo utiliza métodos **Collection**, cualquier objeto que herede de una clase de **Collection** funcionará, pero **ArrayList** representa el tipo más básico de secuencia.

El nombre del método **add()** sugiere que ese método introduce un nuevo elemento en el contenedor **Collection**. Sin embargo, la documentación indica expresamente que **add()** “garantiza que este objeto **Collection** contenga el elemento especificado”. Esto es así para permitir la existencia de contenedores **Set**, que añaden el elemento sólo si éste no se encuentra ya en el contenedor. Con un objeto **ArrayList**, o con cualquier tipo de **List**, **add()** siempre significa “insertar el elemento”, porque las listas no se preocupan de si existen duplicados.

Todas las colecciones pueden recorrerse utilizando la sintaxis *foreach*, como hemos mostrado aquí. Más adelante en el capítulo aprenderemos acerca de un concepto más flexible denominado *Iterador*.

Ejercicio 2: (1) Modifique **SimpleCollection.java** para utilizar un contenedor **Set** para **c**.

Ejercicio 3: (2) Modifique **innerclasses/Sequence.java** de modo que se pueda añadir cualquier número de elementos.

Adición de grupos de elementos

Existen métodos de utilidad en las clases de matrices y colecciones de **java.util** que añaden grupos de elementos a una colección. El método **Arrays.asList()** toma una matriz y una lista de elementos separados por comas (utilizando *varargs*) y lo transforma en un objeto **List**. **Collections.addAll()** toma un objeto **Collection** y una matriz o una lista separada por comas y añade los elementos a la colección. He aquí un ejemplo donde se ilustran ambos métodos, así como el método **addAll()** más convencional que forma parte de todos los tipos **Collection**:

```
//: holding/AddingGroups.java
// Adición de grupos de elementos a objetos Collection.
import java.util.*;

public class AddingGroups {
    public static void main(String[] args) {
        Collection<Integer> collection =
            new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
        Integer[] moreInts = { 6, 7, 8, 9, 10 };
        collection.addAll(Arrays.asList(moreInts));
        for(Integer i : collection)
            System.out.print(i + " ");
    }
}
```

```

collection.addAll(Arrays.asList(moreInts));
// Se ejecuta bastante más rápido, pero no se puede
// construir una colección de esta forma:
Collections.addAll(collection, 11, 12, 13, 14, 15);
Collections.addAll(collection, moreInts);
// Produce una lista "respaldada" en una matriz:
List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
list.set(1, 99); // OK -- modificar un elemento
// list.add(21); // Error de ejecución porque la matriz
// subyacente no se puede cambiar de tamaño.
}
} //:-)

```

El constructor de una colección puede aceptar otra colección con el fin de utilizarla para inicializarse a sí misma, así que podemos emplear `Arrays.asList()` para generar la entrada para el constructor. Sin embargo, `Collections.addAll()` se ejecuta mucho más rápido y resulta igualmente sencillo construir el objeto `Collection` sin ningún elemento y luego invocar `Collections.addAll()`, por lo que ésta es la técnica que más se suele utilizar.

El método miembro `Collection.addAll()` sólo puede tomar como argumento otro objeto `Collection`, por lo que no es tan flexible como `Arrays.asList()` o `Collections.addAll()`, que utilizan listas de argumentos variables.

También es posible utilizar directamente la salida de `Arrays.asList()` como una lista, pero la representación subyacente en este caso es la matriz, que no se puede cambiar de tamaño. Si tratamos de añadir o borrar elementos en dicha lista, eso implicaría cambiar el tamaño de la matriz, por lo que se obtiene un error “Unsupported Operation” en tiempo de ejecución.

Una limitación de `Arrays.asList()` es que dicho método trata de adivinar cuál es el tipo de lista resultante, sin prestar atención a la variable a la que la estemos asignando. En ocasiones, esto puede causar un problema:

```

//: holding/AsListInference.java
// Arrays.asList() determina el tipo en sí mismo.
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
    public static void main(String[] args) {
        List<Snow> snow1 = Arrays.asList(
            new Crusty(), new Slush(), new Powder());

        // No se compilará:
        // List<Snow> snow2 = Arrays.asList(
        //     new Light(), new Heavy());
        // El compilador dirá:
        // found   : java.util.List<Powder>
        // required: java.util.List<Snow>

        // Collections.addAll() no se confunde:
        List<Snow> snow3 = new ArrayList<Snow>();
        Collections.addAll(snow3, new Light(), new Heavy());

        // Proporcionar una indicación utilizando una
        // especificación explícita del tipo del argumento:
        List<Snow> snow4 = Arrays.<Snow>asList(
            new Light(), new Heavy());
    }
} //:-)

```

Al tratar de crear **snow2**, **Arrays.asList()** sólo dispone del tipo **Powder**, por lo que crea un objeto **List<Powder>** en lugar de **List<Snow>**, mientras que **Collections.addAll()** funciona correctamente, porque sabe, a partir del primer argumento, cuál es el tipo de destino.

Como puede ver por la creación de **snow4**, es posible insertar una “indicación” en mitad de **Arrays.asList()**, para decirle al compilador cuál debería ser el tipo de destino real para el objeto **List** producido por **Arrays.asList()**. Esto se denomina *especificación explícita del tipo de argumento*.

Los mapas son más complejos como tendremos oportunidad de ver, y la biblioteca estándar de Java no proporciona ninguna forma para inicializarlos de manera automática, salvo a partir de los contenidos de otro objeto **Map**.

Impresión de contenedores

Es necesario utilizar **Arrays.toString()** para generar una representación imprimible de una matriz, pero los contenedores se imprimen adecuadamente sin ninguna medida especial. He aquí un ejemplo que también nos va a permitir presentar los contenedores básicos de Java:

```
//: holding/PrintingContainers.java
// Los contenedores se imprimen automáticamente.
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
    static Collection fill(Collection<String> collection) {
        collection.add("rat");
        collection.add("cat");
        collection.add("dog");
        collection.add("dog");
        return collection;
    }
    static Map fill(Map<String, String> map) {
        map.put("rat", "Fuzzy");
        map.put("cat", "Rags");
        map.put("dog", "Bosco");
        map.put("dog", "Spot");
        return map;
    }
    public static void main(String[] args) {
        print(fill(new ArrayList<String>()));
        print(fill(new LinkedList<String>()));
        print(fill(new HashSet<String>()));
        print(fill(new TreeSet<String>()));
        print(fill(new LinkedHashSet<String>()));
        print(fill(new HashMap<String, String>()));
        print(fill(new TreeMap<String, String>()));
        print(fill(new LinkedHashMap<String, String>()));
    }
} /* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[dog, cat, rat]
[cat, dog, rat]
[rat, cat, dog]
{dog=Spot, cat=Rags, rat=Fuzzy}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*///:-
```

Este ejemplo muestra las dos categorías principales en la biblioteca de contenedores de Java. La distinción entre ambas se basa en el número de elementos que se almacenan en cada “posición” del contenedor. La categoría **Collection** sólo almacena

na un elemento en cada posición; esta categoría incluye el objeto **List**, que almacena un grupo de elementos en una secuencia especificada, el objeto **Set**, que no permite añadir un elemento idéntico a otro que ya se encuentre dentro del conjunto y el objeto **Queue**, que sólo permite insertar objetos en un “extremo” del contenedor y extraer los objetos del otro “extremo” (en lo que a este ejemplo respecta se trataría simplemente de otra forma de manipular una secuencia, por lo que no lo hemos incluido). Un objeto **Map**, por su parte, almacena dos objetos, una *clave* y un *valor* asociado, en cada posición.

A la salida, podemos ver que el comportamiento de impresión predeterminado (que se implementa mediante el método **toString()** de cada contenedor) genera resultados razonablemente legibles. Una colección se imprime rodeada de corchetes, estando cada elemento separado por una coma. Un mapa estará rodeado de llaves, asociándose las claves y los valores mediante un signo igual (las claves a la izquierda y los valores a la derecha).

El primer método **fill()** funciona con todos los tipos de colección, cada uno de los cuales se encarga de implementar el método **add()** para incluir nuevos elementos.

ArrayList y **LinkedList** son tipos de listas y, como puede ver a la salida, ambos almacenan los elementos en el mismo orden en el que fueron insertados. La diferencia entre estos dos tipos de objeto no está sólo en la velocidad de ciertos tipos de operaciones, sino también en que la clase **LinkedList** contiene más operaciones que **ArrayList**. Hablaremos más en detalle de estas operaciones posteriormente en el capítulo.

HashSet, **TreeSet** y **LinkedHashSet** son tipos de conjuntos. La salida muestra que los objetos **Set** sólo permiten almacenar una copia de cada elemento (no se puede introducir dos veces el mismo elemento), pero también muestra que las diferentes implementaciones de **Set** almacenan los elementos de manera distinta. **HashSet** almacena los elementos utilizando una técnica bastante compleja que analizaremos en el Capítulo 17, *Análisis detallado de los contenedores*, lo único que necesitamos saber en este momento es que dicha técnica representa la forma más rápida de extraer elementos y, como resultado, el orden de almacenamiento puede parecer bastante extraño (a menudo, lo único que nos preocupa es si un cierto objeto forma parte de un conjunto, y no el orden en que aparecen los objetos). Si el orden de almacenamiento fuera importante, podemos utilizar un objeto **TreeSet**, que mantiene los objetos en orden de comparación ascendente, o un objeto **LinkedHashSet**, que mantiene los objetos en el orden en que fueron añadidos.

Un mapa (también denominado *matriz asociativa*) permite buscar un objeto utilizando una *clave*, como si fuera una base de datos simple. El objeto asociado se denomina *valor*. Si tenemos un mapa que asocia los estados americanos con sus capitales y queremos saber la capital de Ohio, podemos buscarla utilizando “Ohio” como clave, de forma muy similar al proceso de acceso a una matriz utilizando un índice. Debido a este comportamiento, un objeto mapa sólo admite una copia de cada clave (no se puede introducir dos veces la misma clave).

Map.put(key, value) añade un valor (el elemento deseado) y lo asocia con una clave (aquellos con lo que buscaremos el elemento). **Map.get(key)** devuelve el valor asociado con una clave. En el ejemplo anterior sólo se añaden parejas de clave-valor, sin realizar ninguna búsqueda. Ilustraremos el proceso de búsqueda más adelante.

Observe que no es necesario especificar (ni preocuparse por ello) el tamaño del mapa, porque éste cambia de tamaño automáticamente. Asimismo, los mapas saben cómo imprimirse, mostrando la asociación existente entre claves y valores. En el orden en que se mantienen las claves y valores dentro de un objeto **Map** no es el orden de inserción, porque la implementación **HashMap** utiliza un algoritmo muy rápido que se encarga de controlar dicho orden.

En el ejemplo se utilizan las tres versiones básicas de **Map**: **HashMap**, **TreeMap** y **LinkedHashMap**. Al igual que **HashSet**, **HashMap** proporciona la técnica de búsqueda más rápida, no almacenando los elementos en ningún orden aparente. Un objeto **TreeMap** mantiene las claves en un orden de comparación ascendente, mientras que **LinkedHashMap** tiene las claves en orden de inserción sin dejar, por ello, de ser igual de rápido que **HashMap** a la hora de realizar búsquedas.

Ejercicio 4: (3) Cree una clase *generadora* que devuelva nombres de personajes (como objetos **String**) de su película favorita cada vez que invoque **next()**, y que vuelva al principio de la lista de personajes una vez que haya acabado con todos los nombres. Utilice este generador para llenar una matriz, un **ArrayList**, un **LinkedList**, un **HashSet**, un **LinkedHashSet** y un **TreeSet**, y luego imprima cada contenedor.

List

Las listas garantizan que los elementos se mantengan en una secuencia concreta. La interfaz **List** añade varios métodos a **Collection** que permiten la inserción y la eliminación de elementos en mitad de una lista.

Existen dos tipos de objetos **List**:

- El objeto básico **ArrayList**, que es el que mejor permite acceder a los elementos de forma aleatoria, pero que resulta más lento a la hora de insertar y eliminar elementos en mitad de una lista.
- El objeto **LinkedList**, que proporciona un acceso secuencial óptimo, siendo las inserciones y borrados en mitad de una lista enormemente rápidos. **LinkedList** resulta relativamente lento para los accesos aleatorios, pero tiene muchas más funcionalidades que **ArrayList**.

El siguiente ejemplo se adelanta un poco dentro de la estructura del libro, utilizando una biblioteca del Capítulo 14, *Información de tipos* para importar **typeinfo.pets**. Se trata de una biblioteca que contiene una jerarquía de clases **Pet** (mascota), junto algunas herramientas para generar aleatoriamente objetos **Pet**. No es necesario entender todos los detalles en este momento, sino que basta con saber que existe: (1) una clase **Pet** y varios subtipos de **Pet** y (2) que el método **Pets.arrayList()** estático devuelve un método **ArrayList** lleno de objetos **Pet** aleatoriamente seleccionados:

```
//: holding/ListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListFeatures {
    public static void main(String[] args) {
        Random rand = new Random(47);
        List<Pet> pets = Pets.arrayList(7);
        print("1: " + pets);
        Hamster h = new Hamster();
        pets.add(h); // Cambio de tamaño automático
        print("2: " + pets);
        print("3: " + pets.contains(h));
        pets.remove(h); // Eliminar objeto a objeto
        Pet p = pets.get(2);
        print("4: " + p + " " + pets.indexOf(p));
        Pet cymric = new Cymric();
        print("5: " + pets.indexOf(cymric));
        print("6: " + pets.remove(cymric));
        // Debe ser el objeto exacto:
        print("7: " + pets.remove(p));
        print("8: " + pets);
        pets.add(3, new Mouse()); // Insertar en un determinado índice
        print("9: " + pets);
        List<Pet> sub = pets.subList(1, 4);
        print("subList: " + sub);
        print("10: " + pets.containsAll(sub));
        Collections.sort(sub); // Ordenación de la colección
        print("sorted subList: " + sub);
        // El orden no es importante en containsAll():
        print("11: " + pets.containsAll(sub));
        Collections.shuffle(sub, rand); // Mezclar los elementos
        print("shuffled subList: " + sub);
        print("12: " + pets.containsAll(sub));
        List<Pet> copy = new ArrayList<Pet>(pets);
        sub = Arrays.asList(pets.get(1), pets.get(4));
        print("sub: " + sub);
        copy.retainAll(sub);
        print("13: " + copy);
        copy = new ArrayList<Pet>(pets); // Obtener una nueva copia
        copy.remove(2); // Eliminar según un índice
        print("14: " + copy);
        copy.removeAll(sub); // Sólo elimina objetos exactos
        print("15: " + copy);
    }
}
```

```

        copy.set(1, new Mouse()); // Sustituir un elemento
        print("16: " + copy);
        copy.addAll(2, sub); // Insertar una lista en el medio
        print("17: " + copy);
        print("18: " + pets.isEmpty());
        pets.clear(); // Eliminar todos los elementos
        print("19: " + pets);
        print("20: " + pets.isEmpty());
        pets.addAll(Pets.arrayList(4));
        print("21: " + pets);
        Object[] o = pets.toArray();
        print("22: " + o[3]);
        Pet[] pa = pets.toArray(new Pet[0]);
        print("23: " + pa[3].id());
    }
} /* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:-
```

Hemos numerado las líneas de impresión para poder establecer la relación de la salida con el código fuente. La primera línea de salida muestra la lista original de objetos **Pet**. A diferencia de las matrices, un objeto **List** permite añadir o eliminar elementos después de haberlo creado y el objeto cambia automáticamente de tamaño. Ésa es su característica fundamental: se trata de una secuencia modificable. En la línea de salida 2 podemos ver el resultado de añadir un objeto **Hamster**. El objeto se ha añadido al final de la lista.

Podemos averiguar si un objeto se encuentra dentro de la lista utilizando el método **contains()**. Si queremos eliminar un objeto, podemos pasar la referencia a dicho objeto al método **remove()**. Asimismo, si disponemos de una referencia a un objeto, podemos ver en qué número de índice está almacenado ese objeto dentro de la lista utilizando **indexOff()**, como puede verse en la línea de salida 4.

A la hora de determinar si un elemento forma parte de una lista, a la hora de descubrir el índice de un elemento y a la hora de eliminar un elemento de una lista a partir de su referencia, se utiliza el método **equals()** (que forma parte de la clase raíz **Object**). Cada objeto **Pet** se define como un objeto único, por lo que, aunque existan dos objetos **Cymric** en la lista, si creamos un nuevo objeto **Cymric** y lo pasamos a **indexOff()**, el resultado será **-1** (indicando que no se ha encontrado el obje-

to); asimismo, si tratamos de eliminar el objeto con `remove()`, el valor devuelto será `false`. Para otras clases, el método `equals()` puede estar definido de forma diferente; dos objetos `String`, por ejemplo, serán iguales si los contenidos de las cadenas de caracteres son idénticos. Así que, para evitar sorpresas, es importante ser consciente de que el comportamiento de un objeto `List` varía dependiendo del comportamiento del método `equals()`.

En las líneas de salida 7 y 8, podemos ver que se puede eliminar perfectamente un objeto que se corresponda exactamente con otro objeto de la lista.

También resulta posible insertar un elemento en mitad de la lista, como puede verse en la línea de salida 9 y en el código que la precede, pero esta operación nos permite resaltar un potencial problema de rendimiento: para un objeto `LinkedList`, la inserción y eliminación en mitad de una lista son operaciones muy poco costosas (salvo por, en este caso, el propio acceso aleatorio en mitad de la lista), mientras que para un objeto `ArrayList` se trata de una operación bastante costosa. ¿Quiere esto decir que nunca deberíamos insertar elementos en mitad de una lista `ArrayList`, y que por el contrario, deberíamos emplear un objeto `LinkedList` en caso de tener que llevar a cabo esa operación? De ninguna manera: simplemente significa que debemos tener en cuenta el potencial problema, y que si comenzamos a hacer numerosas inserciones en mitad de un objeto `ArrayList` y nuestro programa comienza a ralentizarse, podemos sospechar que el posible culpable es la implementación de la lista concreta que hemos elegido (la mejor forma de descubrir uno de estos cuellos de botella, como podrá ver en el suplemento <http://MindView.net/Books/BetterJava>, consiste en utilizar un perfilador). El de la optimización es un problema bastante complicado, y lo mejor es no preocuparse por él hasta que veamos que es absolutamente necesario (aunque comprender los posibles problemas siempre resulta útil).

El método `subList()` permite crear fácilmente una sublista a partir de otra lista de mayor tamaño, y esto produce de forma natural un resultado `true` cuando se pasa la sublista a `containsAll()` para ver si los elementos se encuentran en esa lista de mayor tamaño. También merece la pena recalcar que el orden no es importante: puede ver en las líneas de salida 11 y 12 que al invocar los métodos `Collections.sort()` y `Collections.shuffle()` (que ordenan y aleatorizan, respectivamente, los elementos) con la sublista `sub`, el resultado de `containsAll()` no se ve afectado. `subList()` genera una lista respaldada por la lista original. Por tanto, los cambios efectuados en la lista devuelta se verán reflejados en la lista original, y viceversa.

El método `retainAll()` es, en la práctica, una operación de “intersección de conjuntos”, que en este caso conserva todos los elementos de `copy` que se encuentren también en `sub`. De nuevo, el comportamiento resultante dependerá del método `equals()`.

La línea de salida 14 muestra el resultado de eliminar un elemento utilizando su número índice, lo cual resulta bastante más directo que eliminarlo mediante la referencia al objeto, ya que no es necesario preocuparse acerca del comportamiento de `equals()` cuando se utilizan índices.

El método `removeAll()` también opera de manera distinta dependiendo del método `equals()`. Como su propio nombre sugiere, se encarga de eliminar de la lista todos los objetos que estén en el argumento de tipo `List`.

El nombre del método `set()` no resulta muy adecuado, debido a la posible confusión con la clase `Set`, un mejor nombre habría sido “replace” (sustituir) porque este método se encarga de sustituir el elemento situado en el índice indicado (el primer argumento) con el segundo argumento.

La línea de salida 17 muestra que, para las listas, existe un método `addAll()` sobrecargado que nos permite insertar la nueva lista en mitad de la lista original, en lugar de limitarnos a añadirla al final con el método `addAll()` incluido en `Collection`.

Las líneas de salida 18-20 muestran el efecto de los métodos `isEmpty()` y `clear()`.

Las líneas de salida 22 y 23 muestran cómo puede convertirse cualquier objeto `Collection` en una matriz utilizando `toArray()`. Se trata de un método sobrecargado, la versión sin argumentos devuelve una matriz de `Object`, pero si se pasa una matriz del tipo de destino a la versión sobrecargada, se generará una matriz del tipo especificado (suponiendo que los mecanismos de comprobación de tipos no detecten ningún error). Si la matriz utilizada como argumento resulta demasiado pequeña para almacenar todos los objetos de la lista `List` (como sucede en este ejemplo), `toArray()` creará una nueva matriz del tamaño apropiado. Los objetos `Pet` tienen un método `id()`, pudiendo ver en el ejemplo cómo se invoca dicho método para uno de los objetos de la matriz resultante.

Ejercicio 5: Modifique `ListFeatures.java` para utilizar enteros (recuerde la característica de *autoboxing*) en lugar de objetos `Pet`, y explique las diferencias que haya en los resultados.

Ejercicio 6: (2) Modifique `ListFeatures.java` para utilizar cadenas de caracteres en lugar de objetos `Pet`, y explique las diferencias que haya en los resultados.

Ejercicio 7: (3) Cree una clase y construya luego una matriz inicializada de objetos de dicha clase. Rellene una lista a partir de la matriz. Cree un subconjunto de la lista utilizando **subList()**, y luego elimine dicho subconjunto de la lista.

Iterator

En cualquier contenedor, tenemos que tener una forma de insertar elementos y de volver a extraerlos. Después de todo, esa es la función principal de un contenedor: almacenar cosas. En una lista, **add()** es una de las formas de insertar elementos y **get()** es una de las formas de extraerlos.

Si queremos razonar a un nivel más alto, nos encontramos con un problema: necesitamos desarrollar el programa con el tipo exacto de contenedor para poder utilizarlo. Esto puede parecer una desventaja a primera vista, pero ¿qué sucede si escribimos código para una lista y posteriormente descubrimos que sería conveniente aplicar ese mismo código a un conjunto? Suponga que quisiéramos escribir desde el principio, un fragmento de código de propósito general, que no supiera con qué tipo de contenedor está trabajando, para poderlo utilizar con diferentes tipos de contenedores sin reescribir dicho código. ¿Cómo podríamos hacer esto?

Podemos utilizar el concepto de *Iterator* (otro patrón de diseño) para conseguir este grado de abstracción. Un iterador es un objeto cuyo trabajo consiste en desplazarse a través de una secuencia y seleccionar cada uno de los objetos que la componen, sin que el programa cliente tenga que preocuparse acerca de la estructura subyacente a dicha secuencia. Además, un iterador es lo que usualmente se denomina un *objeto ligero*: un objeto que resulta barato crear. Por esa razón, a menudo nos encontramos con restricciones aparentemente extrañas que afectan a los iteradores; por ejemplo, un objeto **Iterator** de Java sólo se puede desplazar en una dirección. No son muchas las cosas que podemos hacer con un objeto **Iterator** salvo:

1. Pedir a una colección que nos devuelva un iterador utilizando un método **iterator()**. Dicho iterador estará preparado para devolver el primer elemento de la secuencia.
2. Obtener el siguiente objeto de la secuencia mediante **next()**.
3. Ver si hay más objetos en la secuencia utilizando el método **hasNext()**.
4. Eliminar el último elemento devuelto por el iterador mediante **remove()**.

Para ver cómo funciona, podemos volver a utilizar las herramientas de la clase **Pet** que hemos tomado prestadas del Capítulo 14, *Información de tipos*:

```
//: holding/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(12);
        Iterator<Pet> it = pets.iterator();
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
        // Un enfoque más simple, siempre que sea posible:
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
        // Un iterador también permite eliminar elementos:
        it = pets.iterator();
        for(int i = 0; i < 6; i++) {
            it.next();
            it.remove();
        }
        System.out.println(pets);
    }
}
```

```

    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*///:-
```

Con un objeto **Iterator**, no necesitamos preocuparnos acerca del número de elementos que haya en el contenedor. Los métodos **hasNext()** y **next()** se encargan de dicha tarea por nosotros.

Si simplemente nos estamos desplazando hacia adelante por la lista y no pretendemos modificar el propio objeto **List**, la sintaxis **foreach** resulta más sucinta.

Un iterador permite también eliminar el último elemento generado por **next()**, lo que quiere decir que es necesario invocar a **next()** antes de llamar a **remove()**.⁴

Esta idea de tomar un contenedor de objetos y recorrerlo para realizar una cierta operación con cada uno resulta muy potente y haremos un extenso uso de ella a lo largo de todo el libro.

Ahora consideremos la creación de un método **display()** que sea neutral con respecto al contenedor utilizado:

```

//: holding/CrossContainerIteration.java
import typeinfo.pets.*;
import java.util.*;

public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
*///:-
```

Observe que **display()** no contiene ninguna información acerca del tipo de secuencia que está recorriendo, lo cual nos muestra la verdadera potencia del objeto **Iterator**: la capacidad de separar la operación de recorrer una secuencia de la estructura subyacente de recorrer dicha secuencia. Por esta razón, decimos en ocasiones que los iteradores *unifican el acceso a los contenedores*.

⁴ **remove()** es un método “opcional” (existen otros métodos también opcionales), lo que significa que no todas las implementaciones de **Iterator** deben implementarlo. Este tema se trata en el Capítulo 17, *Análisis detallado de los contenedores*. Los contenedores de la biblioteca estándar de Java sí implementan el método **remove()**, por lo que no es necesario preocuparse de este tema hasta que lleguemos a este capítulo.

- Ejercicio 8:** (1) Modifique el Ejercicio 1 para que utilice un iterador para recorrer la lista mientras se invoca **hop()**.
- Ejercicio 9:** (4) Modifique **innerclasses/Sequence.java** para que **Sequence** funcione con un objeto **Iterator** en lugar de un objeto **Selector**.
- Ejercicio 10:** (2) Modifique el Ejercicio 9 del Capítulo 8, *Polimorfismo* para utilizar un objeto **ArrayList** para almacenar los objetos **Rodent** y un iterador para recorrer la secuencia de objetos **Rodent**.
- Ejercicio 11:** (2) Escriba un método que utilice un iterador para recorrer una colección e imprima el resultado de **toString()** para cada objeto del contenedor. Rellene todos los diferentes tipos de colecciones con una serie de objetos y aplique el método que haya diseñado a cada contenedor.

ListIterator

ListIterator es un subtipo más potente de **Iterator** que sólo se genera mediante las clases **List**. Mientras que **Iterator** sólo se puede desplazar hacia adelante, **ListIterator** es bidireccional. También puede generar los índices de los elementos siguiente y anterior, en relación al lugar de la lista hacia el que el iterador está apuntando, permite sustituir el último elemento listado utilizando el método **set()**. Podemos generar un iterador **ListIterator** que apunte al principio de la lista invocando **listIterator()**, y también podemos crear un iterador **ListIterator** que comience apuntando a un índice **n** de la lista invocando **listIterator(n)**. He aquí un ejemplo que ilustra estas capacidades:

```
//: holding/ListIteration.java
import typeinfo.pets.*;
import java.util.*;

public class ListIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(8);
        ListIterator<Pet> it = pets.listIterator();
        while(it.hasNext()) {
            System.out.print(it.next() + ", " + it.nextInt() +
                ", " + it.previousIndex() + "; ");
            System.out.println();
        }
        // Hacia atrás:
        while(it.hasPrevious())
            System.out.print(it.previous().id() + " ");
        System.out.println();
        System.out.println(pets);
        it = pets.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.set(Pets.randomPet());
        }
        System.out.println(pets);
    }
} /* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4;
Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster, EgyptianMau]
*///:-
```

El método **Pets.randomPet()** se utiliza para sustituir todos los objetos **Pet** de la lista desde la posición 3 en adelante.

- Ejercicio 12:** (3) Cree y rellene un objeto **List<Integer>**. Cree un segundo objeto **List<Integer>** del mismo tamaño que el primero y utilice sendos objetos **ListIterator** para leer los elementos de la primera lista e insertarlos en la segunda en orden inverso (pruebe a explorar varias formas distintas de resolver este problema).

LinkedList

LinkedList también implementa la interfaz **List** básica, como **ArrayList**, pero realiza ciertas operaciones de forma más eficiente que **ArrayList** (la inserción y la eliminación en la mitad de la lista). A la inversa, resulta menos eficiente para las operaciones de acceso aleatorio.

LinkedList también incluye métodos que permiten usar este tipo de objetos como una pila, como una cola o como una cola bidireccional.

Algunos de estos métodos son alias o ligeramente variantes de los otros, con el fin de disponer de nombres que resulten más familiares dentro del contexto de un uso específico (en particular en el caso de los objetos **Queue**, que se usan para implementar colas). Por ejemplo, **getFirst()** y **element()** son idénticos; devuelven la cabecera (primer elemento) de la lista sin eliminarlo y generan la excepción **NoSuchElementException** si la lista está vacía. **peek()** es una variante de estos métodos que devuelve **null** si la lista está vacía.

removeFirst() y **remove()** también son idénticos; eliminan y devuelven la cabecera de la lista, generando la excepción **NoSuchElementException** para una lista vacía; **poll()** es una variante que devuelve **null** si la lista está vacía.

addFirst() inserta un elemento al principio de la lista.

offer() es el mismo método que **add()** y **addLast()**. Todos ellos añaden un elemento al final de la lista.

removeLast() elimina y devuelve el último elemento de la lista.

He aquí un ejemplo que muestra las similitudes y diferencias básicas entre todas estas funcionalidades. Este ejemplo no repite aquellos comportamientos que ya han sido ilustrados en **ListFeatures.java**:

```
//: holding/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class LinkedListFeatures {
    public static void main(String[] args) {
        LinkedList<Pet> pets =
            new LinkedList<Pet>(Pets.arrayList(5));
        print(pets);
        // Idénticos:
        print("pets.getFirst(): " + pets.getFirst());
        print("pets.element(): " + pets.element());
        // Sólo difiere en el comportamiento con las listas vacías:
        print("pets.peek(): " + pets.peek());
        // Idénticos; elimina y devuelve el primer elemento:
        print("pets.remove(): " + pets.remove());
        print("pets.removeFirst(): " + pets.removeFirst());
        // Sólo difiere en el comportamiento con las listas vacías:
        print("pets.poll(): " + pets.poll());
        print(pets);
        pets.addFirst(new Rat());
        print("After addFirst(): " + pets);
        pets.offer(Pets.randomPet());
        print("After offer(): " + pets);
        pets.add(Pets.randomPet());
        print("After add(): " + pets);
        pets.addLast(new Hamster());
        print("After addLast(): " + pets);
        print("pets.removeLast(): " + pets.removeLast());
    }
} /* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
```

```

pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Cymric]
After add(): [Rat, Mutt, Pug, Cymric, Pug]
After addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
pets.removeLast(): Hamster
*///:-
```

El resultado de `Pets.arrayList()` se entrega al constructor de `LinkedList` con el fin de rellenar la lista enlazada. Si analiza la interfaz `Queue`, podrá ver los métodos `element()`, `offer()`, `peek()`, `poll()` y `remove()` que han sido añadidos a `LinkedList` para poder disponer de la implementación de una cola. Más adelante en el capítulo se incluyen ejemplos completos del manejo de colas.

Ejercicio 13: (3) En el ejemplo `innerclasses/GreenhouseController.java`, la clase `Controller` utiliza un objeto `ArrayList`. Cambie el código para utilizar en su lugar un objeto `LinkedList` y emplee un iterador para recorrer el conjunto de sucesos.

Ejercicio 14: (3) Cree un objeto vacío `LinkedList<Integer>`. Utilizando un iterador `ListIterator`, añada valores enteros a la lista insertándolos siempre en mitad de la misma.

Stack

Una pila (*stack*) se denomina en ocasiones “contenedor de tipo LIFO” (*last-in, first-out*, el último en entrar es el primero en salir). El último elemento que pongamos en la “parte superior” de la pila será el primero que tengamos que sacar de la misma, como si se tratara de una pila de platos en una cafetería.

`LinkedList` tiene métodos que implementan de forma directa la funcionalidad de pila, por lo que también podríamos usar una lista enlazada `LinkedList` en lugar de definir una clase con las características de una pila. Sin embargo, definir una clase a propósito permite en ocasiones clarificar las cosas:

```

//: net/mindview/util/Stack.java
// Definición de una pila a partir de una lista enlazada.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void push(T v) { storage.addFirst(v); }
    public T peek() { return storage.getFirst(); }
    public T pop() { return storage.removeFirst(); }
    public boolean empty() { return storage.isEmpty(); }
    public String toString() { return storage.toString(); }
} ///:-
```

Esto nos permite introducir el ejemplo más simple posible de definición de una clase mediante genéricos. La `<T>` que sigue al nombre de la clase le dice al compilador que se trata de un *tipo parametrizado* y que el parámetro de tipo (que será sustituido por un tipo real cuando se utilice la clase) es `T`. Básicamente, lo que estamos diciendo es: “Estamos definiendo una pila `Stack` que almacena objetos de tipo `T`”. La pila se implementa utilizando un objeto `LinkedList`, y también se define dicho objeto `LinkedList` para que almacene el tipo `T`. Observe que `push()` (el método que introduce objetos en la pila) toma un objeto de tipo `T`, mientras que `peek()` y `pop()` devuelven el objeto de tipo `T`. El método `peek()` devuelve el elemento superior de la pila sin eliminarlo, mientras que `pop()` extrae y devuelve dicho elemento superior.

Si lo único que queremos es disponer del comportamiento de pila, el mecanismo de herencia resulta inapropiado, porque generaría una clase que incluiría el resto de los métodos de `LinkedList` (en el Capítulo 17, *Análisis detallado de los contenedores*, podrá ver que los diseñadores de Java 1.0 cometieron este error al crear `java.util.Stack`).

He aquí una sencilla demostración de esta nueva clase **Stack**:

```
//: holding/StackTest.java
import net.mindview.util.*;

public class StackTest {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
    }
} /* Output:
fleas has dog My
*///:-
```

Si quiere utilizar esta clase **Stack** en su propio código, tendrá que especificar completamente el paquete (o cambiar el nombre de la clase) cuando cree una pila; en caso contrario, probablemente entre en colisión con la clase **Stack** del paquete **java.util**. Por ejemplo, si importamos **java.util.*** en el ejemplo anterior, deberemos usar los nombres de los paquetes para evitar las colisiones.

```
//: holding/StackCollision.java
import net.mindview.util.*;

public class StackCollision {
    public static void main(String[] args) {
        net.mindview.util.Stack<String> stack =
            new net.mindview.util.Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
        System.out.println();
        java.util.Stack<String> stack2 =
            new java.util.Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack2.push(s);
        while(!stack2.empty())
            System.out.print(stack2.pop() + " ");
    }
} /* Output:
fleas has dog My
fleas has dog My
*///:-
```

Las dos clases **Stack** tienen la misma interfaz, pero no existe ninguna interfaz común **Stack** en **java.util**, probablemente porque la clase original **java.util.Stack**, que estaba diseñada de una forma inadecuada, ya tenía ocupado el nombre. Aunque **java.util.Stack** existe, **LinkedList** permite obtener una clase **Stack** mejor, por lo que resulta preferible la técnica basada en **net.mindview.util.Stack**.

También podemos controlar la selección de la implementación **Stack** “preferida” utilizando una instrucción de importación explícita:

```
import net.mindview.util.Stack;
```

Ahora cualquier referencia a **Stack** hará que se seleccione la versión de **net.mindview.util**, mientras que para seleccionar **java.util.Stack** es necesario emplear una cualificación completa.

Ejercicio 15: (4) Las pilas se utilizan a menudo para evaluar expresiones en lenguajes de programación. Utilizando **net.mindview.util.Stack**, evalúe la siguiente expresión, donde ‘+’ significa “introducir la letra siguiente en la pila” mientras que ‘-’ significa “extraer la parte superior de la fila e imprimirlo”: “+U+n+c---+e+r+t---+a-+j+n+t+y---+ -r+u--+l+e+s---”

Set

Los objetos de tipo **Set** (conjuntos) no permiten almacenar más de una instancia de cada objeto. Si tratamos de añadir más de una instancia de un mismo objeto, **Set** impide la duplicación. El uso más común de **Set** consiste en comprobar la pertenencia, para poder preguntar de una manera sencilla si un determinado objeto se encuentra dentro de un conjunto. Debido a esto, la operación más importante de un conjunto suele ser la de búsqueda, así que resulta habitual seleccionar la implementación **HashSet**, que está optimizada para realizar búsquedas rápidamente.

Set tiene la misma interfaz que **Collection**, por lo que no existe ninguna funcionalidad adicional, a diferencia de los dos tipos distintos de **List**. En lugar de ello, **Set** es exactamente un objeto **Collection**, salvo porque tiene un comportamiento distinto (éste es un ejemplo ideal del uso de los mecanismos de herencia y de polimorfismo: permiten expresar diferentes comportamientos). Un objeto **Set** determina la pertenencia basándose en el “valor” de un objeto, lo cual constituye un tema relativamente complejo del que hablaremos en el Capítulo 17, *Análisis detallado de los contenedores*.

He aquí un ejemplo que utiliza un conjunto **HashSet** con objetos **Integer**:

```
//: holding/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intset = new HashSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26, 11, 18, 3, 12, 27, 17, 2, 13,
28, 20, 25, 10, 5, 0]
*///:-
```

En el ejemplo, se añaden diez mil números aleatorios entre 0 y 29 al conjunto, por lo que cabe imaginar que cada valor tendrá muchos duplicados. A pesar de ello, podemos ver que sólo aparece una instancia de cada valor en los resultados.

Observará también que la salida no tiene ningún orden específico. Esto se debe a que **HashSet** utiliza el mecanismo de *hash* para acelerar las operaciones; este mecanismo se analiza en detalle en el Capítulo 17, *Análisis detallado de los contenedores*. El orden mantenido por un conjunto **HashSet** es diferente del que se mantiene en un **TreeSet** o en un **LinkedHashSet**, ya que cada implementación almacena los elementos de forma distinta. **TreeSet** mantiene los elementos ordenados en una estructura de datos de tipo de árbol rojo-negro, mientras que **HashSet** utiliza una función de *hash*. **LinkedHashSet** también emplea una función *hash* para acelerar las búsquedas, pero parece mantener los elementos en orden de inserción utilizando una lista enlazada.

Si queremos que los resultados estén ordenados, una posible técnica consiste en utilizar un conjunto **TreeSet** en lugar de **HashSet**:

```
//: holding/SortedSetOfInteger.java
import java.util.*;

public class SortedSetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> intset = new TreeSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:-
```

Una de las operaciones más comunes que tendremos que realizar es comprobar la pertenencia al conjunto de un determinado miembro usando `contains()`, pero hay otras operaciones que nos recuerdan a los diagramas Venn que enseñan en el colegio:

```
//: holding/SetOperations.java
import java.util.*;
import static net.mindview.util.Print.*;

public class SetOperations {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<String>();
        Collections.addAll(set1,
            "A B C D E F G H I J K L".split(" "));
        set1.add("M");
        print("H: " + set1.contains("H"));
        print("N: " + set1.contains("N"));
        Set<String> set2 = new HashSet<String>();
        Collections.addAll(set2, "H I J K L".split(" "));
        print("set2 in set1: " + set1.containsAll(set2));
        set1.remove("H");
        print("set1: " + set1);
        print("set2 in set1: " + set1.containsAll(set2));
        set1.removeAll(set2);
        print("set2 removed from set1: " + set1);
        Collections.addAll(set1, "X Y Z".split(" "));
        print("'X Y Z' added to set1: " + set1);
    }
} /* Output:
H: true
N: false
set2 in set1: true
set1: [D, K, C, B, L, G, I, M, A, F, J, E]
set2 in set1: false
set2 removed from set1: [D, C, B, G, M, A, F, E]
'X Y Z' added to set1: [Z, D, C, B, G, M, A, F, Y, X, E]
*///:-
```

Los nombres de los métodos resultan bastante descriptivos, y existen unos cuantos métodos adicionales que podrá encontrar en el JDK.

Generar una lista de elementos diferentes puede resultar muy útil en determinadas ocasiones. Por ejemplo, suponga que quiera enumerar todas las palabras contenidas en el archivo `SetOperations.java` anterior. Con la utilidad `net.mindview.TextFile` que presentaremos más adelante en el libro, podremos abrir un archivo y almacenar su contenido en un objeto `Set`:

```
//: holding/UniqueWords.java
import java.util.*;
import net.mindview.util.*;

public class UniqueWords {
    public static void main(String[] args) {
        Set<String> words = new TreeSet<String>(
            new TextFile("SetOperations.java", "\\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L, M, N, Output, Print, Set,
SetOperations, String, X, Y, Z, add, addAll, added, args, class, contains, containsAll,
false, from, holding, import, in, java, main, mindview, net, new, print, public, remove,
removeAll, removed, set1, set2, split, static, to, true, util, void]
*///:-
```

TextFile hereda de **List<String>**. El constructor de **TextFile** abre el archivo y lo descompone en palabras de acuerdo con la expresión regular “\W+”, que significa “una o más letras” (las expresiones regulares se presentan en el Capítulo 13, *Cadenas de caracteres*). El resultado se entrega al constructor de **TreeSet**, que añade el contenido del objeto **List** al conjunto. Puesto que se trata de un objeto **TreeSet**, el resultado está ordenado. En este caso, la reordenación se realiza *lexicográficamente* de modo que las letras mayúsculas y minúsculas se encuentran en grupos separados. Si desea realizar una ordenación *alfabética*, puede pasar el comparador **String.CASE_INSENSITIVE_ORDER** (un *comparador* es un objeto que establece un orden) al constructor **TreeSet**:

```
//: holding/UniqueWordsAlphabetic.java
// Generación de un listado alfabético.
import java.util.*;
import net.mindview.util.*;

public class UniqueWordsAlphabetic {
    public static void main(String[] args) {
        Set<String> words =
            new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        words.addAll(
            new TextFile("SetOperations.java", "\\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, add, addAll, added, args, B, C, class, Collections, contains, containsAll, D, E, F,
false, from, G, H, HashSet, holding, I, import, in, J, java, K, L, M, main, mindview, N,
net, new, Output, Print, public, remove, removeAll, removed, Set, set1, set2,
SetOperations, split, static, String, to, true, util, void, X, Y, Z]
*///:-
```

Los comparadores se analizarán en el Capítulo 16, *Matrices*.

Ejercicio 16: (5) Cree un objeto **Set** con todas las vocales. Utilizando el archivo **UniqueWords.java**, cuente y muestre el número de vocales en cada palabra de entrada, y muestre también el número total de vocales en el archivo de entrada.

Map

La posibilidad de establecer correspondencias entre unos objetos y otros puede ser enormemente potente a la hora de resolver ciertos problemas de programación. Por ejemplo, consideremos un programa que permite examinar la aleatoriedad de la clase **Random**. Idealmente, **Random** debería producir la distribución perfectamente aleatoria de números, pero para comprobar si esto es así debería generar numerosos números aleatorios y llevar la cuenta de cuáles caen dentro de cada uno de los rangos definidos. Un objeto **Map** nos permite resolver fácilmente el problema; en este caso, la clase será el número generado por **Random** y el valor será el número de veces que ese número ha aparecido:

```
//: holding/Statistics.java
// Ejemplo simple de HashMap.
import java.util.*;

public class Statistics {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Map<Integer, Integer> m =
            new HashMap<Integer, Integer>();
        for(int i = 0; i < 10000; i++) {
            // Generar un número entre 0 y 20:
            int r = rand.nextInt(20);
            Integer freq = m.get(r);
            m.put(r, freq == null ? 1 : freq + 1);
        }
        System.out.println(m);
    }
}
```

```

    }
} /* Output:
{15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478, 3=508, 7=471, 12=521, 17=509,
2=489, 13=506, 9=549, 6=519, 1=502, 14=477, 10=513, 5=503, 0=481}
*///:-
```

En `main()`, la característica de *autoboxing* convierte el valor `int` aleatoriamente generado en una referencia a `Integer` que puede utilizarse con el mapa `HashMap` (no pueden utilizarse primitivas en los contenedores). El método `get()` devuelve `null` si la clave no se encuentra ya en el contenedor (lo que quiere decir que es la primera vez que se ha encontrado ese número concreto). En caso contrario, el método `get()` devuelve el valor `Integer` asociado con esa clave, el cual tendremos que incrementar (de nuevo, la característica de *autoboxing* simplifica la expresión, pero en la práctica se llevan a cabo las necesarias conversiones hacia y desde `Integer`).

He aquí un ejemplo que nos permite utilizar la descripción de `String` para buscar objetos `Pet`. También nos muestra cómo podemos comprobar si un determinado objeto `Map` contiene una clave o un valor utilizando los métodos `containsKey()` y `containsValue()`:

```

//: holding/PetMap.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetMap {
    public static void main(String[] args) {
        Map<String,Pet> petMap = new HashMap<String,Pet>();
        petMap.put("My Cat", new Cat("Molly"));
        petMap.put("My Dog", new Dog("Ginger"));
        petMap.put("My Hamster", new Hamster("Bosco"));
        print(petMap);
        Pet dog = petMap.get("My Dog");
        print(dog);
        print(petMap.containsKey("My Dog"));
        print(petMap.containsValue(dog));
    }
} /* Output:
{My Cat=Cat Molly, My Hamster=Hamster Bosco, My Dog=Dog Ginger}
Dog Ginger
true
true
*///:-
```

Los mapas, al igual que las matrices y las colecciones, pueden expandirse fácilmente para que sean multidimensionales; basta con definir un objeto `Map` cuyos valores sean también mapas (y los valores de *esos otros mapas* pueden ser, a su vez, otros contenedores o incluso otros mapas). Así, resulta bastante fácil combinar los contenedores para generar estructuras de datos muy potentes. Por ejemplo, suponga que queremos llevar una lista de personas que tienen múltiples mascotas; en ese caso, lo único que necesitaremos es un objeto `Map<Person, List<Pet>>`:

```

//: holding/MapOfList.java
package holding;
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
    public static Map<Person, List<? extends Pet>>
        petPeople = new HashMap<Person, List<? extends Pet>>();
    static {
        petPeople.put(new Person("Dawn"),
            Arrays.asList(new Cymric("Molly"),new Mutt("Spot")));
        petPeople.put(new Person("Kate"),

```

```

        Arrays.asList(new Cat("Shackleton"),
                      new Cat("Elsie May"), new Dog("Margrett")));
    petPeople.put(new Person("Marilyn"),
                  Arrays.asList(
                      new Pug("Louie aka Louis Snorkelstein Dupree"),
                      new Cat("Stanford aka Stinky el Negro"),
                      new Cat("Pinkola")));
    petPeople.put(new Person("Luke"),
                  Arrays.asList(new Rat("Fuzzy"), new Rat("Fizzy")));
    petPeople.put(new Person("Isaac"),
                  Arrays.asList(new Rat("Freckly")));
}
public static void main(String[] args) {
    print("People: " + petPeople.keySet());
    print("Pets: " + petPeople.values());
    for(Person person : petPeople.keySet()) {
        print(person + " has:");
        for(Pet pet : petPeople.get(person))
            print("    " + pet);
    }
}
/* Output:
People: [Person Luke, Person Marilyn, Person Isaac, Person Dawn, Person Kate]
Pets: [[Rat Fuzzy, Rat Fizzy], [Pug Louie aka Louis Snorkelstein Dupree, Cat Stanford aka
Stinky el Negro, Cat Pinkola], [Rat Freckly], [Cymric Molly, Mutt Spot], [Cat Shackleton,
Cat Elsie May, Dog Margrett]]
Person Luke has:
    Rat Fuzzy
    Rat Fizzy
Person Marilyn has:
    Pug Louie aka Louis Snorkelstein Dupree
    Cat Stanford aka Stinky el Negro
    Cat Pinkola
Person Isaac has:
    Rat Freckly
Person Dawn has:
    Cymric Molly
    Mutt Spot
Person Kate has:
    Cat Shackleton
    Cat Elsie May
    Dog Margrett
*///:-
```

Un mapa puede devolver un objeto **Set** con sus claves, un objeto **Collection** con sus valores o un objeto **Set** con las parejas clave-valor que tiene almacenadas. El método **keySet()** genera un conjunto de todas las claves de **petPeople**, que se utiliza en la instrucción **foreach** para iterar a través del mapa.

Ejercicio 17: (2) Tome la clase **Gerbil** del Ejercicio 1 y cambie el ejemplo para incluirla en su lugar en un objeto **Map**, asociando el nombre de cada objeto **Gerbil** (por ejemplo, "Fuzzy" o "Spot") como si fuera una cadena de caracteres (la clave) de cada objeto **Gerbil** (el valor) que incluyamos en la tabla. Genere un iterador para el conjunto de claves **keySet()** y utilicelo para desplazarse a lo largo del mapa, buscando el objeto **Gerbil** correspondiente a cada clave, imprimiendo la clave e invocando el método **hop()** del objeto **Gerbil**.

Ejercicio 18: (3) Rellene un mapa **HashMap** con parejas clave-valor. Imprima los resultados para mostrar la ordenación según el código **hash**. Ordene las parejas, extraiga la clave e introduzca el resultado en un mapa **LinkedHashMap**. Demuestre que se mantiene el orden de inserción.

Ejercicio 19: (2) Repita el ejercicio anterior con sendos conjuntos **HashSet** y **LinkedHashSet**.

Ejercicio 20: (3) Modifique el Ejercicio 16 para llevar la cuenta de cuántas veces ha aparecido cada vocal.

- Ejercicio 21:** (3) Utilizando `Map<String, Integer>`, siga el ejemplo de `UniqueWords.java` para crear un programa que lleve la cuenta del número de apariciones de cada palabra en un archivo. Ordene los resultados utilizando `Collections.sort()` proporcionando como segundo argumento `String.CASE_INSENSITIVE_ORDER` (para obtener una ordenación alfabética), y muestre los resultados.
- Ejercicio 22:** (5) Modifique el ejercicio anterior para que utilice una clase que contenga un campo de tipo `String` y un campo contador para almacenar cada una de las diferentes palabras, así como un conjunto `Set` de estos objetos con el fin de mantener la lista de palabras.
- Ejercicio 23:** (4) Partiendo de `Statistics.java`, cree un programa que ejecute la prueba repetidamente y compruebe si hay algún número que tienda a aparecer más que los otros en los resultados.
- Ejercicio 24:** (2) Rellene un mapa `LinkedHashMap` con claves de tipo `String` y objetos del tipo que prefiera. Ahora extraiga las parejas, ordénelas según las claves y vuelva a insertarlas en el mapa.
- Ejercicio 25:** (3) Cree un objeto `Map<String, ArrayList<Integer>>`. Utilice `net.mindview.TextFile` para abrir un archivo de texto y lea el archivo de palabra en palabra (utilice "`\W+`" como segundo argumento del constructor `TextFile`). Cuente las palabras a medida que lee el archivo, y para cada palabra de un archivo, anote en la matriz `ArrayList<Integer>` el contador asociado con dicha palabra; esto es, en la práctica, la ubicación dentro del archivo en la que encontró dicha palabra.
- Ejercicio 26:** (4) Tome el mapa resultante del ejercicio anterior y ordene de nuevo las palabras, tal como aparecían en el archivo original.

Queue

Una *cola* (*queue*) es normalmente un contenedor de tipo FIFO (*first-in, first-out*, el primero en entrar es el primero en salir). En otras palabras, lo que hacemos es insertar elementos por uno de los extremos y extraerlos por el otro, y el orden en que insertemos los elementos coincidirá con el orden en que estos serán extraídos. Las colas se utilizan comúnmente como un mecanismo fiable para transferir objetos desde un área de un programa a otro. Las colas son especialmente importantes en la programación concurrente, como veremos en el Capítulo 21, *Concurrencia*, porque permiten transferir objetos con seguridad de una a otra tarea.

`LinkedList` dispone de métodos para soportar el comportamiento de una cola e implementa la interfaz `Queue`, por lo que un objeto `LinkedList` puede utilizarse como implementación de `Queue`. Generalizando un objeto `LinkedList` a `Queue`, este ejemplo utiliza los métodos específicos de gestión de colas de la interfaz `Queue`:

```
//: holding/QueueDemo.java
// Generalización de un objeto LinkedList a un objeto Queue.
import java.util.*;

public class QueueDemo {
    public static void printQ(Queue queue) {
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            queue.offer(rand.nextInt(i + 10));
        printQ(queue);
        Queue<Character> qc = new LinkedList<Character>();
        for(char c : "Brontosaurus".toCharArray())
            qc.offer(c);
        printQ(qc);
    }
} /* Output:
```

```

8 1 1 1 6 14 3 1 0 1
Brontosaurus
*///:-

```

`offer()` es uno de los métodos específicos de `Queue`; este método inserta un elemento al final de la cola, siempre que sea posible, o bien devuelve el valor `false`. Tanto `peek()` como `element()` devuelven la cabecera de la cola *sin eliminarla*, pero `peek()` devuelve `null` si la cola está vacía y `element()` genera `NoSuchElementException`. Tanto `poll()` como `remove()` eliminan y devuelven la cabecera de la cola, pero `poll()` devuelve `null` si la cola está vacía, mientras que `remove()` genera `NoSuchElementException`.

La característica de *autoboxing* convierte automáticamente el resultado `int` de `nextInt()` en el objeto `Integer` requerido por `queue`, y el valor `char` en el objeto `Character` requerido por `qc`. La interfaz `Queue` limita el acceso a los métodos de `LinkedList` de modo que sólo están disponibles los métodos apropiados, con lo que estaremos menos tentados de utilizar los métodos de `LinkedList` (aqui, podríamos proyectar de nuevo `queue` para obtener un objeto `LinkedList`, pero al menos nos resultará bastante más complicado utilizar esos métodos).

Observe que los métodos específicos de `Queue` proporcionan una funcionalidad completa y autónoma. Es decir, podemos disponer de una cola utilizable sin ninguno de los métodos que se encuentran en `Collection`, que es de donde se ha heredado.

Ejercicio 27: (2) Escriba una clase denominada `Command` que contenga un objeto `String` y que tenga un método `operation()` que imprima la cadena de caracteres. Escriba una segunda clase con un método que rellene un objeto `Queue` con objetos `Command` y devuelva la cola rellena. Pase el objeto `Queue` relleno a un método de una tercera clase que consuma los objetos de la cola e invoque sus métodos `operation()`.

PriorityQueue

El mecanismo FIFO (*First-in, first-out*) describe la *disciplina de gestión de colas* más común. La disciplina de gestión de colas es lo que decide, dado un grupo de elementos existentes en la cola, cuál es el que va a continuación. La disciplina FIFO dice que el siguiente elemento es aquel que haya estado esperando durante más tiempo.

Por el contrario, una *cola con prioridad* implica que el elemento que va a continuación será aquel que tenga una necesidad mayor (la prioridad más alta). Por ejemplo, en un aeropuerto, puede que un cliente que está en medio de la cola pase a ser atendido inmediatamente si su avión está a punto de salir. Si construimos un sistema de mensajería, algunos mensajes serán más importantes que otros y será necesario tratar esos mensajes antes, independientemente de cuándo hayan llegado. El contenedor `PriorityQueue` ha sido añadido en Java SE5 para proporcionar una implementación automática de este tipo de comportamiento.

Cuando ofrecemos, como método `offer()`, un objeto a una cola de tipo `PriorityQueue`, dicho objeto se ordena dentro de la cola.⁵ El mecanismo de ordenación predeterminado utiliza el *orden natural* de los objetos de la cola, pero podemos modificar dicho elemento proporcionando nuestro propio objeto `Comparator`. La clase `PriorityQueue` garantiza que cuando se invocuen los métodos `peek()`, `poll()` o `remove()`, el elemento que se obtenga será aquél que tenga la prioridad más alta.

Resulta trivial implementar una cola de tipo `PriorityQueue` que funcione con tipos predefinidos como `Integer`, `String` o `Character`. En el siguiente ejemplo, el primer conjunto de valores son los valores aleatorios del ejemplo anterior, con lo cual podemos ver cómo se los extrae de manera diferente de la cola `PriorityQueue`:

```

//: holding/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<Integer> priorityQueue =
            new PriorityQueue<Integer>();
        Random rand = new Random(47);

```

⁵ Esto depende, de hecho, de la implementación. Los algoritmos de colas con prioridad suelen realizar la ordenación durante la inserción (manteniendo una estructura de memoria que se conoce con el nombre de círculo), pero también puede perfectamente seleccionarse el elemento más importante en el momento de la extracción. La elección del algoritmo podría tener su importancia si la prioridad de los objetos puede modificarse mientras estos están esperando en la cola.

```

for(int i = 0; i < 10; i++)
    priorityQueue.offer(rand.nextInt(i + 10));
QueueDemo.printQ(priorityQueue);

List<Integer> ints = Arrays.asList(25, 22, 20,
    18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
priorityQueue = new PriorityQueue<Integer>(ints);
QueueDemo.printQ(priorityQueue);
priorityQueue = new PriorityQueue<Integer>(
    ints.size(), Collections.reverseOrder());
priorityQueue.addAll(ints);
QueueDemo.printQ(priorityQueue);

String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
List<String> strings = Arrays.asList(fact.split(""));
PriorityQueue<String> stringPQ =
    new PriorityQueue<String>(strings);
QueueDemo.printQ(stringPQ);
stringPQ = new PriorityQueue<String>(
    strings.size(), Collections.reverseOrder());
stringPQ.addAll(strings);
QueueDemo.printQ(stringPQ);

Set<Character> charSet = new HashSet<Character>();
for(char c : fact.toCharArray())
    charSet.add(c); // Autoboxing
PriorityQueue<Character> characterPQ =
    new PriorityQueue<Character>(charSet);
QueueDemo.printQ(characterPQ);
}
} /* Output:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 23 21 21 20 18 18 14 14 9 9 3 3 2 1 1
    A A B C C C D D E E E F H H I I L N N O O O O S S S T T U U U W
    W U U U T T S S S O O O O N N L I I H H F E E E D D C C C B A A
    A B C D E F H I L N O S T U W
*/

```

Como puede ver, se permiten los duplicados, los valores menores tienen la prioridad más alta (en el caso de **String**, los espacios también cuentan como valores y tienen una prioridad superior a la de las letras). Para ver cómo podemos verificar la ordenación proporcionando nuestro propio objeto comparador, la tercera llamada al constructor de **PriorityQueue<Integer>** y la segunda llamada a **PriorityQueue<String>** utilizan el comparador de orden inverso generado por **Collections.reverseOrder()** (añadido en Java SE5).

La última sección añade un conjunto **HashSet** para eliminar los objetos **Character** duplicados, simplemente con el fin de hacer las cosas un poco más interesantes.

Integer, **String** y **Character** funcionan con **PriorityQueue** porque estas clases ya tienen un orden natural predefinido. Si queremos utilizar nuestra propia clase en una cola **PriorityQueue**, deberemos incluir una funcionalidad adicional para generar una ordenación natural, o bien proporcionar nuestro objeto **Comparator**. En el Capítulo 17, *Análisis detallado de los contenedores* se proporciona un ejemplo más sofisticado en el que se ilustra este mecanismo.

Ejercicio 28: (2) Rellene la cola **PriorityQueue** (utilizando **offer()**) con valores de tipo **Double** creados utilizando **java.util.Random**, y luego elimine los elementos con **poll()** y visualicelos.

Ejercicio 29: (2) Cree una clase simple que herede de **Object** y que no contenga ningún nombre y demuestre que se pueden añadir múltiples elementos de dicha clase a una cola **PriorityQueue**. Este tema se explicará en detalle en el Capítulo 17, *Análisis detallado de los contenedores*.

Comparación entre Collection e Iterator

Collection es la interfaz raíz que describe las cosas que tienen en común todos los contenedores de secuencias. Podríamos considerarla como una especie de “interfaz incidental”, que surgió debido a la existencia de aspectos comunes entre las otras interfaces. Además, la clase **java.util.AbstractCollection** proporciona una implementación predeterminada de **Collection**, para poder crear un nuevo subtipo de **AbstractCollection** sin duplicar innecesariamente el código.

Un argumento en favor de disponer de una interfaz es que ésta nos permite crear código genérico. Escribiendo como una interfaz en lugar de como una implementación, nuestro código puede aplicarse a más tipos de objetos.⁶ Por tanto, si escribimos un método que admite un tipo **Collection**, dicho método podrá aplicarse a cualquier tipo que implemente **Collection**, y esto permite implementar **Collection** en cualquier clase nueva para poderla usar con el método que hayamos escrito. Resulta interesante resaltar, sin embargo, que la biblioteca estándar C++ no dispone de ninguna clase base común para sus contenedores: los aspectos comunes entre los contenedores se consiguen utilizando iteradores. En Java, podría parecer adecuado seguir la técnica utilizada en C++ y expresar los aspectos comunes de los contenedores utilizando un iterador en lugar de una colección. Sin embargo, ambos enfoques están entrelazados, ya que implementar **Collection** también implica que deberemos proporcionar un método **iterator()**:

```
//: holding/InterfaceVsIterator.java
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
    public static void display(Collection<Pet> pets) {
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        List<Pet> petList = Pets.arrayList(8);
        Set<Pet> petSet = new HashSet<Pet>(petList);
        Map<String,Pet> petMap =
            new LinkedHashMap<String,Pet>();
        String[] names = ("Ralph, Eric, Robin, Lacey, " +
            "Britney, Sam, Spot, Fluffy").split(", ");
        for(int i = 0; i < names.length; i++)
            petMap.put(names[i], petList.get(i));
        display(petList);
        display(petSet);
        display(petList.iterator());
        display(petSet.iterator());
        System.out.println(petMap);
        System.out.println(petMap.keySet());
        display(petMap.values());
        display(petMap.values().iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat

```

⁶ Algunas personas defienden la creación automática de una interfaz para toda posible combinación de métodos en una clase; en ocasiones, defienden que se haga esto para todas las clases. En mi opinión, una interfaz debería tener un significado mayor que una mera duplicación mecánica de combinaciones de métodos, por lo que suelo preferir esperar y ver qué valor añadiría una interfaz antes de crearla.

```

0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
(Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt, Britney=Pug, Sam=Cymric, Spot=Pug,
Fluffy=Manx)
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:-
```

Ambas versiones de `display()` funcionan con objetos `Map` y con subtipos de `Collection`, y tanto la interfaz `Collection` como `Iterator` permiten desacoplar los métodos `display()`, sin forzarles a conocer ningún detalle acerca de la implementación concreta del contenedor subyacente.

En este caso, los dos enfoques se combinan bien. De hecho, `Collection` lleva el concepto un paso más allá porque es de tipo `Iterable`, y por tanto, en la implementación de `display(Collection)` podemos usar la estructura `foreach`, lo que hace que el código sea algo más limpio.

El uso de `Iterator` resulta muy recomendable cuando se implementa una clase externa, es decir, una que no sea de tipo `Collection`, ya que en ese caso resultaría difícil o incómodo hacer que esa clase implementara la interfaz `Collection`. Por ejemplo, si creamos una implementación de `Collection` heredando de una clase que almacene objetos `Pet`, deberemos implementar todos los métodos de `Collection`, incluso aunque no necesitemos utilizarlos dentro del método `display()`. Aunque esto puede llevarse a cabo fácilmente heredando de `AbstractCollection`, estaremos forzados a implementar `iterator()` de todas formas, junto con `size()`, para proporcionar los métodos que no están implementados en `AbstractCollection`, pero son utilizados por los otros métodos de `AbstractCollection`:

```

//: holding/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

public class CollectionSequence
extends AbstractCollection<Pet> {
    private Pet[] pets = Pets.createArray(8);
    public int size() { return pets.length; }
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // No implementado
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        CollectionSequence c = new CollectionSequence();
        InterfaceVsIterator.display(c);
        InterfaceVsIterator.display(c.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:-
```

El método `remove()` es una “operación opcional”, de la que hablaremos más adelante en el Capítulo 17, *Análisis detallado de los contenedores*. Aquí, no es necesario implementarlo y, si lo invocamos, generará una excepción.

En este ejemplo, podemos ver que si implementamos `Collection`, también implementamos `iterator()`; además, vemos que implementar únicamente `iterator()` sólo requiere un esfuerzo ligeramente menor que heredar de `AbstractCollection`. Sin embargo, si nuestra clase ya hereda de otra clase, no podemos heredar de `AbstractCollection`. En tal caso, para implemen-

tar **Collection** sería necesario implementar todos los métodos de la interfaz. En este caso, resultaría mucho más sencillo heredar y añadir la posibilidad de crear un iterador:

```
//: holding/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

class PetSequence {
    protected Pet[] pets = Pets.createArray(8);
}

public class NonCollectionSequence extends PetSequence {
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // No implementado
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        NonCollectionSequence nc = new NonCollectionSequence();
        InterfaceVsIterator.display(nc.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:-
```

Generar un objeto **Iterator** es la forma con acoplamiento más débil para conectar una secuencia con un método que consume esa secuencia; además, se imponen muchas menos restricciones a la clase correspondiente a la secuencia que si implementamos **Collection**.

Ejercicio 30: (5) Modifique **CollectionSequence.java** para que no herede de **AbstractCollection**, sino que implemente **Collection**.

La estructura *foreach* y los iteradores

Hasta ahora, hemos utilizado principalmente la sintaxis *foreach* con las matrices, pero dicha sintaxis también funciona con cualquier objeto de tipo **Collection**. Hemos visto algunos ejemplos en los que empleaba **ArrayList**, pero he aquí una demostración general:

```
//: holding/ForEachCollections.java
// All collections work with foreach.
import java.util.*;

public class ForEachCollections {
    public static void main(String[] args) {
        Collection<String> cs = new LinkedList<String>();
        Collections.addAll(cs,
            "Take the long way home".split(" "));
        for(String s : cs)
            System.out.print("!" + s + "!");
    }
} /* Output:
!Take!the!long!way!home!
*///:-
```

Como es una colección, este código demuestra que todos los objetos **Collection** permiten emplear la estructura *foreach*.

La razón de que este método funcione es que en Java SE5 se ha introducido una nueva interfaz denominada **Iterable** que contiene un método **iterator()** para generar un objeto **Iterator**, y la interfaz **Iterable** es lo que la estructura *foreach* utiliza para desplazarse a través de una secuencia. Por tanto, si creamos cualquier clase que implemente **Iterable**, dicha clase podrá ser utilizada en una instrucción *foreach*:

```
//: holding/IterableClass.java
// Anything Iterable works with foreach.
import java.util.*;

public class IterableClass implements Iterable<String> {
    protected String[] words = {"And that is how " +
        "we know the Earth to be banana-shaped.").split(" ");
    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private int index = 0;
            public boolean hasNext() {
                return index < words.length;
            }
            public String next() { return words[index++]; }
            public void remove() { // No implemented
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        for(String s : new IterableClass())
            System.out.print(s + " ");
    }
} /* Output:
And that is how we know the Earth to be banana-shaped.
*///:-
```

El método **iterator()** devuelve una instancia de una implementación interna anónima de **Iterator<String>** que devuelve cada palabra de la matriz. En **main()**, podemos ver que **IterableClass** funciona perfectamente en una instrucción *foreach*.

En Java SE5, hay varias clases que se han definido como **Iterable**, principalmente todas las clases de tipo **Collection** (pero no las de tipo **Map**). Por ejemplo, este código muestra todas las variables del entorno del sistema operativo:

```
//: holding/EnvironmentVariables.java
import java.util.*;

public class EnvironmentVariables {
    public static void main(String[] args) {
        for(Map.Entry entry: System.getenv().entrySet()) {
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
} /* (Execute to see output) *///:-
```

System.getenv()⁷ devuelve un objeto **Map**, **entrySet()** produce un conjunto de elementos **Map.Entry** y un conjunto (**Set**) de tipo **Iterable**, por lo que se le puede usar en un bucle *foreach*.

La instrucción *foreach* funciona con una matriz o cualquier cosa de tipo **Iterable**, pero esto no quiere decir que una matriz sea automáticamente de tipo **Iterable**, ni tampoco que se produzca ningún tipo de mecanismo de *autoboxing*:

⁷ Esto no estaba disponible antes de Java SE5, porque se pensaba que estaba acoplado de una manera demasiado estrecha con el sistema operativo, lo que violaba la regla de "escribir los programas una vez y ejecutarlos en cualquier lugar". El hecho de que se haya incluido ahora sugiere que los diseñadores de Java han decidido ser más pragmáticos.

```
//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
    static <T> void test(Iterable<T> ib) {
        for(T t : ib)
            System.out.print(t + " ");
    }
    public static void main(String[] args) {
        test(Arrays.asList(1, 2, 3));
        String[] strings = { "A", "B", "C" };
        // Una matriz funciona con foreach, pero no es de tipo Iterable:
        //!! test(strings);
        // Hay que convertirla explícitamente al tipo Iterable:
        test(Arrays.asList(strings));
    }
} /* Output:
1 2 3 A B C
*///:-
```

Al tratar de pasar una matriz como argumento de tipo **Iterable** el programa falla. No hay ningún tipo de conversión automática a **Iterable**; sino que debe realizarse de forma manual.

Ejercicio 31: (3) Modifique **polymorphism/shape/RandomShapeGenerator.java** para hacerlo de tipo **Iterable**. Tendrá que añadir un constructor que admita el número de elementos que queremos que el iterador genere antes de pararse. Verifique que el programa funciona.

El método basado en adaptadores

¿Qué sucede si tenemos una clase existente que sea de tipo **Iterable**, y queremos añadir una o más formas nuevas de utilizar esta clase en una instrucción *foreach*? Por ejemplo, suponga que queremos poder decidir si hay que iterar a través de una lista de palabras en dirección directa o inversa. Si nos limitamos a heredar de la clase y sustituimos el método **iterator()**, estaremos sustituyendo el método existente y no dispondremos de la capacidad de opción.

Una solución es utilizar lo que yo denomino *Método basado en adaptadores*. Cuando se dispone de una interfaz y nos hace falta otra, podemos resolver el problema escribiendo un adaptador. Aquí, lo que queremos es *añadir* la capacidad de generar un iterador inverso, pero sin perder el iterador directo predeterminado, así que no podemos limitarnos a sustituir el método. En su lugar, lo que hacemos es añadir un método que genere un objeto **Iterable** que pueda entonces utilizarse en la instrucción *foreach*. Como puede ver en el ejemplo siguiente, esto nos permite proporcionar múltiples formas de usar *foreach*:

```
//: holding/AdapterMethodIdiom.java
// El método basado en adaptadores permite utilizar
// foreach con tipos adicionales de objetos iterables.
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
    public ReversibleArrayList(Collection<T> c) { super(c); }
    public Iterable<T> reversed() {
        return new Iterable<T>() {
            public Iterator<T> iterator() {
                return new Iterator<T>() {
                    int current = size() - 1;
                    public boolean hasNext() { return current > -1; }
                    public T next() { return get(current--); }
                    public void remove() { // No implementado
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}
```

```

        }
    );
}

public class AdapterMethodIdiom {
    public static void main(String[] args) {
        ReversibleArrayList<String> ral =
            new ReversibleArrayList<String>(
                Arrays.asList("To be or not to be".split(" ")));
        // Toma el iterador normal vía iterator():
        for(String s : ral)
            System.out.print(s + " ");
        System.out.println();
        // Entregar el objeto Iterable de nuestra elección
        for(String s : ral.reversed())
            System.out.print(s + " ");
    }
} /* Output:
To be or not to be
be to not or be To
*///:-
```

Si nos limitamos a poner el objeto **ral** dentro de la instrucción *foreach*, obtenemos el iterador directo (predeterminado). Pero si invocamos **reversed()** sobre el objeto, el comportamiento será distinto.

Utilizando esta técnica, podemos añadir dos métodos adaptadores al ejemplo **IterableClass.java**:

```

//: holding/MultiIterableClass.java
// Adición de varios métodos adaptadores.
import java.util.*;

public class MultiIterableClass extends IterableClass {
    public Iterable<String> reversed() {
        return new Iterable<String>() {
            public Iterator<String> iterator() {
                return new Iterator<String>() {
                    int current = words.length - 1;
                    public boolean hasNext() { return current > -1; }
                    public String next() { return words[current--]; }
                    public void remove() { // No implementado
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }

    public Iterable<String> randomized() {
        return new Iterable<String>() {
            public Iterator<String> iterator() {
                List<String> shuffled =
                    new ArrayList<String>(Arrays.asList(words));
                Collections.shuffle(shuffled, new Random(47));
                return shuffled.iterator();
            }
        };
    }

    public static void main(String[] args) {
        MultiIterableClass mic = new MultiIterableClass();
        for(String s : mic.reversed())
            System.out.print(s + " ");
```

```

        System.out.println();
        for(String s : mic.randomized())
            System.out.print(s + " ");
        System.out.println();
        for(String s : mic)
            System.out.print(s + " ");
    }
} /* Output:
banana-shaped. be to Earth the know we how is that And
is banana-shaped. Earth that how the be And we know to
And that is how we know the Earth to be banana-shaped.
*///:-
```

Observe que el segundo método, `random()`, no crea su propio **Iterator** sino que se limita a devolver el correspondiente a la lista **List** mezclada.

Puede ver a la salida que el método `Collections.shuffle()` no afecta a la matriz original, sino que sólo mezcla las referencias en **shuffled**. Esto es así porque el método `randomized()` empaqueta en el nuevo objeto **ArrayList** el resultado de `Arrays.asList()`. Si mezclaríamos directamente la lista generada por `Arrays.asList()` se modificaría la matriz subyacente, como puede ver a continuación:

```

//: holding/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        List<Integer> list1 =
            new ArrayList<Integer>(Arrays.asList(ia));
        System.out.println("Before shuffling: " + list1);
        Collections.shuffle(list1, rand);
        System.out.println("After shuffling: " + list1);
        System.out.println("array: " + Arrays.toString(ia));

        List<Integer> list2 = Arrays.asList(ia);
        System.out.println("Before shuffling: " + list2);
        Collections.shuffle(list2, rand);
        System.out.println("After shuffling: " + list2);
        System.out.println("array: " + Arrays.toString(ia));
    }
} /* Output:
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:-
```

En el primer caso, la salida de `Arrays.asList()` se entrega al constructor de `ArrayList()`, y esto crea una lista **ArrayList** que hace referencia a los elementos de **ia**. Mezclar estas referencias no modifica la matriz. Sin embargo, si utilizamos directamente el resultado de `Arrays.asList(ia)`, el mezclado modifica el orden de **ia**. Es importante tener en cuenta que `Arrays.asList()` genera un objeto **List** que utiliza la matriz subyacente como su implementación física. Si hacemos algo a ese objeto **List** que lo modifique y no queremos que la matriz original se vea afectada, entonces será necesario realizar una copia en otro contenedor.

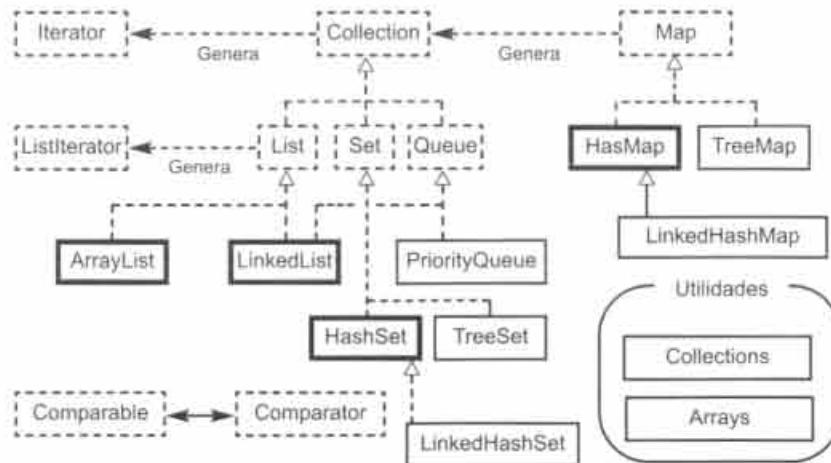
Ejercicio 32: (2) Siguiendo el ejemplo de **MultiliteralClass**, añada métodos `reversed()` y `randomized()` a **NonCollectionSequence.java**. Haga también que **NonCollectionSequence** implemente **Iterable** y muestre que las distintas técnicas funcionan en las instrucciones `foreach`.

Resumen

Java proporciona varias formas de almacenar objetos:

1. Las matrices asocian índices numéricos con los objetos. Almacenan objetos de un tipo conocido, así que no es necesario proyectar el resultado sobre ningún otro tipo a la hora de buscar un objeto. Pueden ser multidimensionales y también almacenar tipos primitivos. Sin embargo, su tamaño no puede modificarse después de haberlas creado.
2. Las colecciones (**Collection**) almacenan elementos independientes, mientras que los mapas (**Map**) almacenan parejas asociadas. Utilizando los genéricos de Java, especificamos el tipo de objeto que hay que almacenar en los contenedores, con el fin de no introducir un tipo incorrecto en un contenedor y también para no tener que efectuar una proyección de los elementos en el momento de extraerlos de un contenedor. Tanto las colecciones como los mapas cambian automáticamente de tamaño a medida que añadimos más elementos. Los contenedores no permiten almacenar tipos primitivos, pero el mecanismo de *autoboxing* se encarga de producir las primitivas a los tipos envoltorio almacenados en el contenedor.
3. Al igual que una matriz, una lista (**List**) también asocia índices numéricos con objetos; por tanto, las matrices y las listas son contenedores ordenados.
4. Utilice una lista **ArrayList** si tiene que realizar numerosos accesos aleatorios; pero, si lo que va a hacer es un gran número de inserciones y de borrados en mitad de la lista, utilice **LinkedList**.
5. El comportamiento de las colas (**Queue**) y de las pilas se obtiene mediante **LinkedList**.
6. Un mapa (**Map**) es una forma de asociar los objetos no con valores enteros, sino con *otros objetos*. Los mapas **HashMap** están diseñados para un acceso rápido, mientras que un mapa **TreeMap** mantiene sus claves ordenadas y no es tan rápido como **HashMap**. Un mapa **LinkedHashMap** mantiene sus elementos en orden de inserción, pero proporciona un acceso rápido con mecanismos de *hash*.
7. Los conjuntos (**Set**) sólo aceptan objetos no duplicados. Los conjuntos **HashSet** proporcionan las búsquedas más rápidas, mientras que **TreeSet** mantiene los elementos en orden. **LinkedHashSet** mantiene los elementos en orden de inserción.
8. No hay necesidad de utilizar las clases antiguas **Vector**, **Hashtable** y **Stack** en los nuevos programas.

Resulta útil examinar un diagrama simplificado de los contenedores Java (sin las clases abstractas ni los componentes antiguos). En este diagrama se incluyen únicamente las interfaces y clases que podemos encontrar de forma habitual.



Taxonomía simple de los contenedores

Como puede ver, sólo hay realmente cuatro componentes contenedores básicos: **Map**, **List**, **Set** y **Queue**, y sólo dos o tres implementaciones de cada uno (las implementaciones de `java.util.concurrent` para **Queue** no están incluidas en este diagrama). Los contenedores que más habitualmente se utilizan son los que tienen líneas gruesas de color negro a su alrededor.

Los recuadros punteados representan **interfaces**, mientras que los recuadros de línea continua son clases normales (concretas). Las líneas de puntos con flechas huecas indican que una clase concreta está implementando una interfaz. Las flechas llenas muestran que una clase puede generar objetos de la clase a la que apunta la flecha. Por ejemplo, cualquier objeto **Collection** puede generar un objeto **Iterator**, y un objeto **List** puede generar un objeto **ListIterator** (además de un objeto **Iterator** normal, ya que **List** hereda de **Collection**).

He aquí un ejemplo que muestra la diferencia en métodos entre las distintas clases. El código concreto se ha tomado del Capítulo 15, *Genéricos*; simplemente lo incluimos aquí para poder generar la correspondiente salida. La salida también muestra las interfaces que se implementan en cada clase o interfaz.

```
//: holding/ContainerMethods.java
import net.mindview.util.*;

public class ContainerMethods {
    public static void main(String[] args) {
        ContainerMethodDifferences.main(args);
    }
} /* Output: (Sample)
Collection: [add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable, Serializable]
TreeSet extends Set, adds: [pollLast, navigableHeadSet, descendingIterator, lower, headSet, ceiling, pollFirst, subSet, navigableTailSet, comparator, first, floor, last, navigableSubSet, higher, tailSet]
Interfaces in TreeSet: [NavigableSet, Cloneable, Serializable]
List extends Collection, adds: [listIterator, indexOf, get, subList, set, lastIndexOf]
Interfaces in List: [Collection]
ArrayList extends List, adds: [ensureCapacity, trimToSize]
Interfaces in ArrayList: [List, RandomAccess, Cloneable, Serializable]
LinkedList extends List, adds: [pollLast, offer, descendingIterator, addFirst, peekLast, removeFirst, peekFirst, removeLast, getLast, pollFirst, pop, poll, addLast, removeFirstOccurrence, getFirst, element, peek, offerLast, push, offerFirst, removeLastOccurrence]
Interfaces in LinkedList: [List, Deque, Cloneable, Serializable]
Queue extends Collection, adds: [offer, element, peek, poll]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, containsKey, containsValue, entrySet, equals, get, hashCode, isEmpty, keySet, put, putAll, remove, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [subMap, comparator, firstKey, lastKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingEntrySet, subMap, pollLastEntry, lastKey, floorEntry, lastEntry, lowerKey, navigableHeadMap, navigableTailMap, descendingKeySet, tailMap, ceilingEntry, higherKey, pollFirstEntry, comparator, firstKey, floorKey, higherEntry, firstEntry, navigableSubMap, headMap, lowerEntry, ceilingKey]
Interfaces in TreeMap: [NavigableMap, Cloneable, Serializable]
*///:-
```

Como puede ver, todos los conjuntos, excepto `TreeSet`, tienen exactamente la misma interfaz que `Collection`. `List` y `Collection` difieren significativamente, aunque `List` requiere métodos que se encuentran en `Collection`. Por otro lado, los métodos de la interfaz `Queue` son autónomos; los métodos de `Collection` no son necesarios para crear una implementación de `Queue` funcional. Finalmente, la única intersección entre `Map` y `Collection` es el hecho de que un mapa puede generar colecciones utilizando los métodos `entrySet()` y `values()`.

Observe la interfaz de marcado `java.util.RandomAccess`, que está asociada a `ArrayList` pero no a `LinkedList`. Esta interfaz proporciona información para aquellos algoritmos que quieran modificar dinámicamente su comportamiento dependiendo del uso de un objeto `List` concreto.

Es verdad que esta organización parece un poco extraña en lo que respecta a las jerarquías orientadas a objetos. Sin embargo, a medida que conozca más detalles acerca de los contenedores en `java.util` (en particular, en el Capítulo 17, *Análisis detallado de los contenedores*), verá que existen otros problemas más importantes que esa estructura de herencia ligeramente extraña. Las bibliotecas de contenedores han constituido siempre un problema de diseño realmente difícil; resolver estos problemas implica tratar de satisfacer un conjunto de fuerzas que a menudo se oponen entre sí. Como consecuencia, debemos prepararnos para llegar a ciertos compromisos en algunos momentos.

A pesar de estos problemas, los contenedores de Java son herramientas fundamentales que se pueden utilizar de forma cotidiana para hacer nuestros programas más simples, más potentes y más efectivos. Puede que tardemos un poco en acostumbrarnos a algunos aspectos de la biblioteca, pero lo más probable es que el lector comience rápidamente a adquirir y utilizar las clases que componen esta biblioteca.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Tratamiento de errores mediante excepciones

12

La filosofía básica de Java es que “el código erróneo no será ejecutado”.

El momento ideal de detectar un error es en tiempo de compilación, antes incluso de poder ejecutar el programa. Sin embargo, no todos los errores pueden detectarse en tiempo de compilación. El resto de los problemas deberán ser gestionados en tiempo de ejecución, utilizando algún tipo de formalidad que permita que quien ha originado el error pase la información apropiada a un receptor que sabrá cómo hacerse cargo de las dificultades apropiadamente.

Una de las formas más potentes de incrementar la robustez del código es disponer de un mecanismo avanzado de recuperación de errores. La recuperación de errores es una de las preocupaciones principales de todos los programas que escribimos, pero resulta especialmente importante en Java, donde uno de los objetivos principales consiste en crear componentes de programas para que otros los utilicen. *Para crear un sistema robusto, cada componente tiene que ser robusto.* Al proporcionar un modelo coherente de informe de errores utilizando excepciones, Java permite que los componentes comuniquen los problemas de manera fiable al código cliente.

Los objetivos del tratamiento de excepciones en Java son simplificar la creación de programas fiables de gran envergadura utilizando menos código de lo habitual y llevar esto a cabo con una mayor confianza en que nuestra aplicación no va a encontrarse con errores no probados. El tema de las excepciones no resulta demasiado difícil de aprender y el mecanismo de excepciones es una de esas características que proporcionan beneficios inmediatos y de gran importancia a cualquier proyecto.

Puesto que el tratamiento de excepciones es la única forma oficial en la que Java informa acerca de los errores, y dado que dicho mecanismo de tratamiento de excepciones es impuesto por el compilador Java, no son demasiados los ejemplos que podríamos escribir en este libro sin antes estudiar el tratamiento de excepciones. En este capítulo, se presenta el código que es necesario escribir para gestionar las excepciones adecuadamente, mostrándose también cómo podemos generar nuestras propias excepciones si alguno de nuestros métodos se mete en problemas.

Conceptos

El lenguaje C y otros lenguajes anteriores disponían a menudo de múltiples esquemas de tratamiento de errores, y dichos esquemas se solían establecer por convenio y no como parte del lenguaje de programación. Normalmente, lo que se hacia era devolver un valor especial o configurar una variable indicadora, y se suponía que el receptor debía examinar el valor o la variable y determinar que algo iba mal. Sin embargo, a medida que fueron pasando los años, se descubrió que los programadores que utilizaban una biblioteca tendían a pensar en sí mismos como si fueran inmunes al error; era casi como si dijeran: “Sí, puede que a otros se les presenten errores, pero en *mi código* no hay errores”. Por tanto, de forma bastante natural, los programadores tendían a no comprobar las condiciones del error (y además, en ocasiones, esas condiciones de error eran demasiado tontas como para comprobarlas¹). Si fuéramos tan exhaustivos como para comprobar si se ha producido un error cada vez que invocamos un método, el código se convertiría en una pesadilla ilegible. Puesto que los programadores

¹ El programador en C puede, por ejemplo, consultar el valor de retorno de `printf()`.

podían, a pesar de todo, construir sus sistemas con estos lenguajes, se resistían a admitir la realidad: que esta técnica de gestión de errores representaba una limitación importante a la hora de crear programas de gran envergadura que fueran robustos y mantenibles.

La solución consiste en eliminar la naturaleza casual del tratamiento de errores e imponer una cierta formalidad. Este modo de proceder tiene, en la práctica, una larga historia, porque las implementaciones de los mecanismos del *tratamiento de excepciones* se remontan a los sistemas operativos de la década de 1960, e incluso a la instrucción “**on error goto**” de BASIC. Pero el mecanismo de tratamiento de excepciones de C++ estaba basado en Ada, y el de Java se basa principalmente en C++ (aunque se asemeja más al de Object Pascal).

La palabra “excepción” hace referencia a algo que no tiene lugar de la forma acostumbrada. En el lugar donde aparece un problema, puede que no sepamos qué hacer con él, pero de lo que si podemos estar seguros es de que no podemos continuar como si no hubiera pasado nada; es necesario pararse y alguien, en algún lugar, tiene que saber cómo responder al error. Sin embargo, no disponemos de suficiente información en el contexto actual como para poder corregir el problema, así que lo que hacemos es pasar dicho problema a otro contexto superior en el que haya alguien cualificado para tomar la decisión adecuada.

El otro beneficio importante derivado de las excepciones es que tienden a reducir la complejidad del código de gestión de errores. Sin las excepciones, es necesario comprobar si se ha producido un error concreto y resolverlo en múltiples lugares del programa. Sin embargo, con las excepciones ya no hace falta comprobar los errores en el punto donde se produce la llamada a un método, ya que la excepción garantiza que alguien detecte el error. Además, sólo hace falta tratar el problema en un único sitio, en lo que se denomina la *rutina de tratamiento de excepciones*. Esto nos ahorra código y permite también separar el código que describe lo que queremos hacer durante la ejecución normal, de ese otro código que se ejecuta cuando las cosas van mal. En general, la lectura, la escritura y la depuración del código se hacen mucho más claras con las excepciones que cuando se utiliza la antigua forma de tratar los errores.

Excepciones básicas

Una *condición excepcional* es un problema que impide la continuación del método o ámbito actuales. Resulta importante distinguir las condiciones excepcionales de los problemas normales, en los que disponemos de la suficiente información dentro del contexto actual, como para poder resolver de alguna manera la dificultad con que nos hayamos encontrado. Con una condición excepcional, no podemos continuar con el procesamiento, porque no disponemos de la información necesaria para tratar con el problema *en el contexto actual*. Lo único que podemos hacer es saltar fuera del contexto actual y pasar dicho problema a otro contexto de orden superior. Esto es lo que sucede cuando generamos una excepción.

La división representa un ejemplo sencillo: si estamos a punto de dividir por cero, merece la pena comprobar dicha condición, pero ¿qué implica que el denominador sea cero? Puede que sepamos, en el contexto del problema que estemos intentando resolver en ese método concreto, cómo tratar con un denominador cero. Pero si se trata de un valor inesperado, no podremos tratar con él, y deberemos por tanto generar una excepción en lugar de continuar con la ruta de ejecución en la que estuvieramos.

Cuando generamos una excepción suceden varias cosas. En primer lugar, se crea el objeto excepción de la misma forma que cualquier otro objeto Java: en el cúmulo utilizando la instrucción **new**. A continuación, se detiene la ruta actual de ejecución (aquella que ya no podemos continuar) y se extrae del contexto actual la referencia al objeto excepción. En este punto, el mecanismo de tratamiento de excepciones se hace cargo del problema y comienza a buscar un lugar apropiado donde continuar ejecutando el programa. Dicho lugar apropiado es la *rutina de tratamiento de excepciones*, cuya tarea consiste en recuperarse del problema de modo que el programa pueda intentar hacer otra cosa o simplemente continuar con lo que estuviera haciendo.

Como ejemplo simple de generación de excepciones vamos a considerar una referencia a un objeto denominada **t**. Es posible que nos pasen una referencia que no haya sido inicializada, así que podríamos tratar de comprobarlo antes de invocar un método en el que se utilice dicha referencia al objeto. Podemos enviar la información acerca del error a otro contexto de orden superior, creando un objeto que represente dicha información y extrayéndolo del contexto actual. Este proceso se denomina *generar una excepción*. He aquí el aspecto que tendría en el código:

```
if(t == null)
    throw new NullPointerException();
```

Esto genera la excepción, lo que nos permite, en el contexto actual, olvidarnos del problema: dicho problema será gestionado de manera transparente en algún otro lugar. En breve veremos *dónde* exactamente se gestiona la excepción.

Las excepciones nos permiten pensar en cada cosa que hagamos como si se tratara de una transacción, encargándose las excepciones de proteger esas transacciones: "...la premisa fundamental de las transacciones es que hacia falta un mecanismo de tratamiento de excepciones en la informática distribuida. Las transacciones son el equivalente informático de los contratos legales. Si algo va mal, detenemos todos los cálculos"². También podemos pensar en las transacciones como si se tratara de un sistema integrado para deshacer acciones, porque (con un cierto cuidado) podemos establecer varios puntos de recuperación en cualquier programa. Si una parte del programa falla, la excepción se encarga de "deshacer" las operaciones hasta un punto estable conocido dentro del programa.

Uno de los aspectos más importantes de las excepciones es, que si sucede algo realmente grave, no permiten que el programa continúe con la ruta de ejecución ordinaria. Este tema ha constituido un grave problema en lenguajes como C y C++, especialmente en C, donde no había ninguna forma de impedir que el programa continuara por una ruta de ejecución en caso de aparecer un problema, de modo que era posible ignorar los problemas durante un largo tiempo y acabar en un estado totalmente inadecuado. Las excepciones nos permiten (entre otras cosas) obligar al programa a detenerse y a informarnos de qué es lo que ha pasado o, idealmente, obligar al programa a resolver el problema y volver a un estado estable.

Argumentos de las excepciones

Al igual que sucede con cualquier objeto en Java, las excepciones siempre se crean en el cúmulo de memoria utilizando **new**, lo que asigna el correspondiente espacio de almacenamiento e invoca un constructor. Existen dos constructores en todas las excepciones estándar. El primero es el constructor predeterminado y el segundo toma un argumento de cadena de caracteres para poder aportar la información pertinente a la excepción:

```
throw new NullPointerException("t = null");
```

Esta cadena de caracteres puede posteriormente extraerse utilizando varios métodos, como tendremos oportunidad de ver.

La palabra clave **throw** produce una serie de resultados interesantes. Después de crear un objeto excepción mediante **new**, le proporcionamos la referencia resultante a **throw**. En la práctica, el objeto es "devuelto" desde el método, aun cuando dicho tipo de objeto no es normalmente lo que estaba diseñado que el método devolviera. Una forma bastante simplificadora de considerar el mecanismo de tratamiento de excepciones es como si fuera un tipo distinto de mecanismo de retorno, aunque no debemos caer en la tentación de llevar esa analogía demasiado lejos, porque podríamos meternos en problemas. También podemos salir de ámbitos de ejecución ordinarios generando una excepción. En cualquiera de los casos, se devuelve un objeto excepción y se sale del método o ámbito donde la excepción se haya producido.

Las similitudes con el proceso normal de devolución de resultados por parte de un método terminan aquí, porque *el lugar al que se vuelve* es completamente distinto de aquel al que volvemos en una llamada normal a un método (volvemos a una rutina apropiada de tratamiento de excepciones que puede estar alejada muchos niveles dentro de la pila del lugar en el que se ha generado la excepción).

Además, podemos generar cualquier tipo de objeto **Throwable**, que es la clase raíz de las excepciones. Normalmente, lo que haremos será generar una clase de excepción distinta para cada tipo diferente de error. La información acerca del error está representada tanto dentro del objeto excepción como, implicitamente, en el nombre de la clase de excepción, por lo que alguien en el contexto de orden superior puede determinar qué es lo que hay que hacer con la excepción (a menudo, la única información es el tipo de excepción, no almacenándose ninguna información significativa dentro del objeto excepción).

Detección de una excepción

Para ver cómo se detecta una excepción, primero es necesario entender el concepto de *región protegida*. Se trata de una sección de código que puede generar excepciones que esté seguida por el código necesario para tratar dichas excepciones.

² Jim Gray, ganador del premio Turing Award por las contribuciones que su equipo ha realizado al tema de las transacciones. Las palabras están tomadas de una entrevista publicada en www.acmqueue.org.

El bloque try

Si nos encontramos dentro de un método y generamos una excepción (o si otro método al que invoquemos dentro de éste genera una excepción), dicho método terminará al generarse la excepción. Si no queremos generar (con **throw**) la excepción para salir del método, podemos definir un bloque especial dentro de dicho método para capturar la excepción. Este bloque se denomina *bloque try* (probar) porque lo que hacemos en la práctica es “probar” dentro de ese bloque las diversas llamadas a métodos. El bloque **try** es un ámbito de ejecución ordinario precedido por la palabra clave **try**:

```
try {
    // Código que podría generar excepciones
}
```

Si quisieramos comprobar cuidadosamente los errores en un lenguaje de programación que no tuviera mecanismos de tratamiento de excepciones, tendríamos que rodear todas las llamadas a métodos con código de preparación y de comprobación de errores, incluso si invocáramos el mismo método en varias ocasiones. Con el tratamiento de excepciones incluimos todo dentro del bloque **try** y capturamos todas las excepciones en un único lugar. Esto significa que el código es mucho más fácil de escribir y de leer, porque el objetivo del código no se ve confundido con los mecanismos de comprobación de errores.

Rutinas de tratamiento de excepciones

Por supuesto, la excepción generada debe acabar siendo tratada en algún lugar. Dicho “lugar” es la *rutina de tratamiento de excepciones*, existiendo una de dichas rutinas por cada tipo de excepción que queramos capturar. Las rutinas de tratamiento de excepciones están situadas inmediatamente a continuación del bloque **try** y se denotan mediante la palabra clave **catch**:

```
try {
    // Código que podría generar excepciones
} catch(Tipo1 id1) {
    // Tratamiento de las excepciones de Tipo1
} catch(Tipo2 id2) {
    // Tratamiento de las excepciones de Tipo2
} catch(Tipo3 id3) {
    // Tratamiento de las excepciones de Tipo3
}

// etc...
```

Cada cláusula **catch** (rutina de tratamiento de excepciones) es como un pequeño método que toma un único argumento de un tipo concreto. El identificador (**id1**, **id2**, etc.) puede utilizarse dentro de la rutina de tratamiento de excepciones exactamente igual que un argumento de un método. En ocasiones, nunca utilizamos el identificador, porque el tipo de la excepción nos proporciona ya suficiente información como para poder tratar con ella, pero a pesar de todo el identificador debe estar presente.

Las rutinas de tratamiento de excepciones deben aparecer inmediatamente después del bloque **try**. Si se genera una excepción, el mecanismo de tratamiento de excepciones trata de buscar la primera rutina de tratamiento cuyo argumento se ajuste al tipo de la excepción. A continuación, entra en la cláusula **catch**, y la excepción se considera tratada. La búsqueda de rutinas de tratamiento se detiene en cuanto finalizada la cláusula **catch**. Sólo se ejecutará la cláusula **catch** que se ajuste al tipo de la excepción; estas cláusulas no son como las instrucciones **switch**, en las que hace falta una instrucción **break** después de cada cláusula **case** para impedir que las restantes cláusulas se ejecuten.

Observe que, dentro del bloque **try**, puede haber distintas llamadas a métodos que generen la misma excepción, pero sólo hace falta una única rutina de tratamiento.

Terminación y reanudación

Existen dos modelos básicos en la teoría de tratamiento de excepciones. Java soporta el modelo denominado de *terminación*,³ en el que asumimos que el error es tan crítico que no hay forma de volver al lugar en el que se generó la excepción.

³ Como la mayoría de los lenguajes, incluyendo C++, C#, Python, D, etc.

Quienquiera que generara la excepción, decidió que no había ninguna manera de salvar la situación, por lo que *no desea* que volvamos a ese punto en el que la excepción fue generada.

La alternativa se denomina *reanudación*. Esta alternativa implica que la rutina de tratamiento de excepciones debe hacer algo para rectificar la situación, después de lo cual se vuelve a intentar ejecutar el método fallido, presumiendo que esa segunda vez tendrá éxito. Si queremos utilizar la técnica de reanudación, quiere decir que esperamos podemos continuar con la ejecución después de que la excepción sea tratada.

Si quiere disponer de un comportamiento de reanudación en Java, no genere una excepción cuando se encuentre con error. En lugar de ello, invoque un método que corrija el problema. Alternativamente, coloque el bloque **try** dentro de un bucle **while** que continúe volviendo a entrar en el bloque **try** hasta que el resultado sea satisfactorio.

Históricamente, los programadores que utilizaban sistemas operativos donde se admitía el tratamiento de excepciones con reanudación terminaron utilizando código basado en el mecanismo de terminación y evitando emplear la reanudación. Por tanto, aunque la reanudación pueda parecer atractiva a primera vista, no resulta demasiado útil en la práctica. La razón principal es, probablemente, el acoplamiento resultante: las rutinas de tratamiento con reanudación necesitan saber dónde se ha generado la excepción, y necesitan también contener código no genérico específico de cada ubicación donde la excepción se genere. Esto hace que el código sea difícil de escribir y de mantener, especialmente en sistemas grandes en los que las excepciones puedan generarse en muchos puntos distintos.

Creación de nuestras propias excepciones

No tenemos porqué limitarnos a utilizar las excepciones Java existentes. La jerarquía de excepciones de Java no puede prever todos los errores de los que vayamos a querer informar, así que podemos crear nuestros propios errores para indicar un problema especial con el que nuestra biblioteca pueda encontrarse.

Para crear nuestra propia clase de excepción, debemos heredar de una clase de excepción existente, preferiblemente de una cuyo significado esté próximo al de nuestra propia excepción (aunque a menudo esto no es posible). La forma más trivial de crear un nuevo tipo de excepción consiste, simplemente, en permitir que el compilador cree el constructor predeterminado por nosotros, por lo que no hace falta prácticamente ningún código:

```
//: exceptions/InheritingExceptions.java
// Creación de nuestras propias excepciones.

class SimpleException extends Exception {}

public class InheritingExceptions {
    public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        InheritingExceptions sed = new InheritingExceptions();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.out.println("Caught it!");
        }
    }
} /* Output:
Throw SimpleException from f()
Caught it!
*///:-
```

El compilador crea un constructor predeterminado, que de forma automática (e invisible) invoca al constructor predeterminado de la clase base. Por supuesto, en este caso no obtenemos un constructor **SimpleException(String)**, pero en la práctica dicho constructor no se usa demasiado. Como veremos, lo más importante acerca de una excepción es el nombre de la clase, por lo que la mayor parte de las veces una excepción como la que aquí se muestra será perfectamente adecuada.

Aquí, el resultado se imprime en la consola, donde se captura y se comprueba automáticamente utilizando el sistema de visualización de la salida de programas de este libro. Sin embargo, también podríamos enviar la información de error a la salida de *error estándar* escribiendo en **System.err**. Normalmente, suele ser mejor enviar aquí la información de error que a **System.out**, que puede estar redirigido. Si se envía la salida a **System.err**, no será redirigida junto con **System.out**, por lo que es más probable que el usuario vea la información.

También podemos crear una clase de excepción que tenga un constructor con un argumento **String**:

```
//: exceptions/FullConstructors.java

class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) { super(msg); }
}

public class FullConstructors {
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }

    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }

    public static void main(String[] args) {
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
    }
} /* Output:
Throwing MyException from f()
MyException
    at FullConstructors.f(FullConstructors.java:11)
    at FullConstructors.main(FullConstructors.java:19)
Throwing MyException from g()
MyException: Originated in g()
    at FullConstructors.g(FullConstructors.java:15)
    at FullConstructors.main(FullConstructors.java:24)
*///:-
```

El código añadido es de pequeño tamaño: dos constructores que definen la forma en que se crea **MyException**. En el segundo constructor, se invoca explícitamente el constructor de la clase base con un argumento **String** utilizando la palabra clave **super**.

En las rutinas de tratamiento, se invoca uno de los métodos **Throwable** (de donde se hereda **Exception**): **printStackTrace()**. Como puede ver a la salida, esto genera información acerca de la secuencia de métodos que fueron invocados hasta llegar al punto en que se generó la excepción. Aquí, la información se envía a **System.out**, siendo automáticamente capturada y mostrada a la salida. Sin embargo, si invocamos la versión predeterminada:

```
e.printStackTrace();
```

la información va a la salida de error estándar.

- Ejercicio 1:** (2) Cree una clase con un método **main()** que genere un objeto de la clase **Exception** dentro de un bloque **try**. Proporcione al constructor de **Exception** un argumento **String**. Capture la excepción dentro de una cláusula **catch** e imprima el argumento **String**. Añada una cláusula **finally** e imprima un mensaje para demostrar que pasó por allí.
- Ejercicio 2:** (1) Defina una referencia a un objeto e inicialicela con **null**. Trate de invocar un método a través de esta referencia. Ahora rodee el código con una cláusula **try-catch** para capturar la excepción.
- Ejercicio 3:** (1) Escriba código para generar y capturar una excepción **ArrayIndexOutOfBoundsException** (índice de matriz fuera de límites).
- Ejercicio 4:** (2) Cree su propia clase de excepción utilizando la palabra clave **extends**. Escriba un constructor para dicha clase que tome un argumento **String** y lo almacene dentro del objeto como una referencia de tipo **String**. Escriba un método que muestre la cadena de caracteres almacenada. Cree una cláusula **try-catch** para probar la nueva excepción.
- Ejercicio 5:** (3) Defina un comportamiento de tipo reanudación utilizando un bucle **while** que se repita hasta que se deje de generar una excepción.

Excepciones y registro

También podemos *registrar* la salida utilizando **java.util.logging**. Aunque los detalles completos acerca de los mecanismos de registro se presentan en el suplemento que puede encontrarse en la dirección <http://MindView.net/Books/BetterJava>, los mecanismos de registro básicos son lo suficientemente sencillos como para poder utilizarlos aquí.

```
//: exceptions/LoggingExceptions.java
// Una excepción que proporciona la información a través de un registro.
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
    private static Logger logger =
        Logger.getLogger("LoggingException");
    public LoggingException() {
        StringWriter trace = new StringWriter();
        printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
}

public class LoggingExceptions {
    public static void main(String[] args) {
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
    }
} /* Output: (85% match)
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
        at LoggingExceptions.main(LoggingExceptions.java:19)

Caught LoggingException
```

```

Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
        at LoggingExceptions.main(LoggingExceptions.java:24)

Caught LoggingException
*///:-

```

El método estático **Logger.getLogger()** crea un objeto **Logger** asociado con el argumento **String** (usualmente, el nombre del paquete y la clase a la que se refieren los errores) que envía su salida a **System.err**. La forma más fácil de escribir en un objeto **Logger** consiste simplemente en invocar el método asociado con el nivel de mensaje de registro; aquí utilizamos **severe()**. Para producir el objeto **String** para el mensaje de registro, convendría disponer de la traza de la pila correspondiente al lugar donde se generó la excepción, pero **printStackTrace()** no genera un objeto **String** de forma predeterminada. Para obtener un objeto **String**, necesitamos usar el método sobrecargado **printStackTrace()** que toma un objeto **java.io.PrintWriter** como argumento (todo esto se explicará con detalle en el Capítulo 18, *E/S*). Si entregamos al constructor de **PrintWriter** un objeto **java.io.StringWriter**, la salida puede extraerse como un objeto **String** invocando **toString()**.

Aunque la técnica utilizada por **LoggingException** resulta muy cómoda, porque integra toda la infraestructura de registro dentro de la propia excepción, y funciona por tanto automáticamente sin intervención del programador de clientes, lo más común es que nos veamos en la situación de capturar y registrar la excepción generada por algún otro, por lo que es necesario generar el mensaje de registro en la propia rutina de tratamiento de excepciones:

```

//: exceptions/LoggingExceptions2.java
// Registro de las excepciones capturadas.
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
    private static Logger logger =
        Logger.getLogger("LoggingExceptions2");
    static void logException(Exception e) {
        StringWriter trace = new StringWriter();
        e.printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
    public static void main(String[] args) {
        try {
            throw new NullPointerException();
        } catch(NullPointerException e) {
            logException(e);
        }
    }
} /* Output: (90% match)
Aug 30, 2005 4:07:54 PM LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
        at LoggingExceptions2.main(LoggingExceptions2.java:16)
*///:-

```

El proceso de creación de nuestras propias excepciones puede llevarse un paso más allá. Podemos añadir constructores y miembros adicionales:

```

//: exceptions/ExtraFeatures.java
// Mejora de las clases de excepción.
import static net.mindview.util.Print.*;

class MyException2 extends Exception {
    private int x;
    public MyException2() {}
    public MyException2(String msg) { super(msg); }
    public MyException2(String msg, int x) {
        super(msg);

```

```

    this.x = x;
}
public int val() { return x; }
public String getMessage() {
    return "Detail Message: "+ x + " " + super.getMessage();
}
}

public class ExtraFeatures {
    public static void f() throws MyException2 {
        print("Throwing MyException2 from f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {
        print("Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }
    public static void h() throws MyException2 {
        print("Throwing MyException2 from h()");
        throw new MyException2("Originated in h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
            System.out.println("e.val() = " + e.val());
        }
    }
}

/* Output:
Throwing MyException2 from f()
MyException2: Detail Message: 0 null
    at ExtraFeatures.f(ExtraFeatures.java:22)
    at ExtraFeatures.main(ExtraFeatures.java:34)
Throwing MyException2 from g()
MyException2: Detail Message: 0 Originated in g()
    at ExtraFeatures.g(ExtraFeatures.java:26)
    at ExtraFeatures.main(ExtraFeatures.java:39)
Throwing MyException2 from h()
MyException2: Detail Message: 47 Originated in h()
    at ExtraFeatures.h(ExtraFeatures.java:30)
    at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
*///:-
```

Se ha añadido un campo `x` junto con un método que lee dicho valor y un constructor adicional que lo inicializa. Además, `Throwable.getMessage()` ha sido sustituido para generar un mensaje de detalle más interesante. `getMessage()` es un método similar a `toString()` que se utiliza para las clases de excepción.

Puesto que una excepción no es más que otro tipo de objeto, podemos continuar este proceso de mejora de nuestras clases de extensión. Recuerde, sin embargo, que todo este trabajo adicional puede no servir para nada en los programas de cliente que utilicen nuestros paquetes, ya que dichos programas pueden que simplemente miren cuál es la excepción que se ha generado y nada más (ésa es la forma en la que se utilizan la mayoría de las excepciones de la biblioteca Java).

Ejercicio 6: (1) Cree dos clases de excepción, cada una de las cuales realice su propia tarea de registro automáticamente. Demuestre que dichas clases funcionan.

Ejercicio 7: (1) Modifique el Ejercicio 3 para que la cláusula **catch** registre los resultados.

La especificación de la excepción

En Java, debemos tratar de informar al programador de clientes, que invoque nuestros métodos, acerca de las excepciones que puedan ser generadas por cada método. Esto resulta conveniente porque quien realiza la invocación puede saber así exactamente qué código necesita escribir para capturar todas las excepciones potenciales. Por supuesto, si el código fuente está disponible, el programador de clientes podría examinarlo y buscar las instrucciones **throw**, pero puede que una biblioteca se distribuya sin el correspondiente código fuente. Para evitar que esto constituya un problema, Java proporciona una sintaxis (*y nos obliga* a usar esa sintaxis) para permitirnos informar al programador de clientes acerca de cuáles son las excepciones que el método genera, de modo que el programador pueda tratarlas. Se trata de la *especificación de excepciones* que forma parte de la declaración del método y que aparece después de la lista de argumentos.

La especificación de excepciones utiliza una palabra clave adicional, **throws**, seguida de todos los tipos potenciales de excepciones. Por tanto, la definición de un método podría tener el aspecto siguiente:

```
void f() throws TooBig, TooSmall, DivZero { //...}
```

Sin embargo, si decimos

```
void f() { // ...}
```

significa que el método no genera ninguna excepción (*salvo por las excepciones heredadas de RuntimeException*, que pueden generarse en cualquier lugar sin necesidad de que exista una especificación de excepciones; hablaremos de estas excepciones más adelante).

No podemos proporcionar información falsa acerca de la especificación de excepciones. Si el código dentro del método provoca excepciones y ese método no las trata, el compilador lo detectará y nos informará de que debemos tratar la excepción o indicar, mediante una especificación de excepción, que dicha excepción puede ser devuelta por el método. Al imponer que se utilicen especificaciones de excepción en todos los lugares, Java garantiza en tiempo de compilación que exista una cierta corrección en la definición de las excepciones.

Sólo hay una manera de que podamos proporcionar información falsa: podemos declarar que generamos una excepción que en realidad no vamos a generar. El compilador aceptará lo que le digamos y forzará a los usuarios de nuestro método a tratar esa excepción como si realmente fuera a ser generada. La única ventaja de hacer esto es disponer por adelantado de una forma de añadir posteriormente la excepción; si más adelante decidimos comenzar a generar esa excepción, no tendremos que realizar cambios en el código existente. También resulta importante esta característica para crear interfaces y clases bases abstractas cuyas clases derivadas o implementaciones puedan necesitar generar excepciones.

Las excepciones que se comprueban y se imponen en tiempo de compilación se denominan *excepciones comprobadas*.

Ejercicio 8: (1) Escriba una clase con un método que genere una excepción del tipo creado en el Ejercicio 4. Trate de compilarlo sin incluir una especificación de excepción para ver qué es lo que dice el compilador. Añada la especificación de excepción apropiada. Pruebe la clase y su correspondiente excepción dentro de una cláusula **try-catch**.

Cómo capturar una excepción

Resulta posible crear una rutina de tratamiento que capture cualquier tipo de excepción. Para hacer esto, podemos capturar el tipo de excepción de la clase base **Exception** (existen otros tipos de excepciones base, pero **Exception** es la base que resulta apropiada para casi todas las actividades de programación):

```
    catch(Exception e) {
        System.out.println("Caught an exception");
    }
}
```

Esto permitirá capturar cualquier excepción, por lo que si usamos este mecanismo convendrá ponerlo al *final* de la lista de rutinas de tratamiento, con el fin de evitar que se impida entrar en acción a otras rutinas de tratamiento de excepciones que puedan estar situadas a continuación.

Puesto que la clase **Exception** es la base de todas las clases de excepción que tienen importancia para el programador, no vamos a obtener demasiada información específica acerca de la excepción concreta que hayamos capturado, pero podemos invocar los métodos incluidos en *su tipo base Throwable*:

String getMessage()

String getLocalizedMessage()

Obtiene el mensaje de detalle o un mensaje ajustado para la configuración de idioma utilizada.

String toString()

Devuelve una descripción corta del objeto **Throwable**, incluyendo el mensaje de detalle, si es que existe uno.

void printStackTrace()

void printStackTrace(PrintStream)

void printStackTrace(java.io.PrintWriter)

Imprime el objeto **Throwable** y la traza de pila de llamadas de dicho objeto. La pila de llamadas muestra la secuencia de llamadas a métodos que nos han conducido hasta el lugar donde la excepción fue generada. La primera versión imprime en la salida de error estándar, mientras que la segunda y la tercera imprimen en cualquier salida que elijamos (en el Capítulo 18, E/S, veremos por qué existen dos tipos de flujos de salida).

Throwable fillInStackTrace()

Registra información dentro de este objeto **Throwable** acerca del estado actual de los marcos de la pila. Resulta útil cuando una aplicación está volviendo a generar un error o una excepción (hablaremos de esto en breve).

Además, también tenemos acceso a otros métodos del tipo base de **Throwable** que es **Object** (que es el tipo base de todos los objetos). El que más útil puede resultar para las excepciones es **getClass()**, que devuelve un objeto que representa la clase de este objeto. A su vez, podemos consultar este objeto **Class** para obtener su nombre mediante **getName()**, que incluye información o **getSimpleName()**, que sólo devuelve el nombre de la clase.

He aquí un ejemplo que muestra el uso de los métodos básicos de **Exception**:

```
//: exceptions/ExceptionMethods.java
// Ilustración de los métodos de Exception.
import static net.mindview.util.Print.*;

public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("My Exception");
        } catch(Exception e) {
            print("Caught Exception");
            print("getMessage(): " + e.getMessage());
            print("getLocalizedMessage(): " +
                  e.getLocalizedMessage());
            print("toString(): " + e);
            print("printStackTrace():");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
Caught Exception
getMessage(): My Exception
getLocalizedMessage(): My Exception
toString(): java.lang.Exception: My Exception
```

```

printStackTrace();
java.lang.Exception: My Exception
    at ExceptionMethods.main(ExceptionMethods.java:8)
*///:-

```

Podemos ver que los métodos proporcionan sucesivamente más información, de hecho, cada uno de ellos es un superconjunto del anterior.

Ejercicio 9: (2) Cree tres nuevos tipos de excepciones. Escriba una clase con un método que genera las tres. En **main()**, llame al método pero utilice una única cláusula **catch** que permita capturar los tres tipos de excepciones.

La traza de la pila

También puede accederse directamente a la información proporcionada por **printStackTrace()**, utilizando **getStackTrace()**. Este método devuelve una matriz de elementos de traza de la pila, cada uno de los cuales representa un marco de pila. El elemento cero es la parte superior de la pila y se corresponde con la última invocación de método de la secuencia (el punto en que este objeto **Throwable** fue generado). El último elemento de la matriz (la parte inferior de la pila) es la primera invocación de método de la secuencia. Este programa proporciona una ilustración simple:

```

//: exceptions/WhoCalled.java
// Acceso mediante programa a la información de traza de la pila.

public class WhoCalled {
    static void f() {
        // Generar una excepción para rellenar la traza de pila.
        try {
            throw new Exception();
        } catch (Exception e) {
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("-----");
        g();
        System.out.println("-----");
        h();
    }
} /* Output:
f
main
-----
f
g
main
-----
f
g
h
main
*///:-

```

Aquí, nos limitamos a imprimir el nombre del método, pero también podríamos imprimir el objeto completo **StackTraceElement**, que contiene información adicional.

Regeneración de una excepción

En ocasiones, conviene regenerar una excepción que hayamos capturado, particularmente cuando se utiliza **Exception** para capturar todas las excepciones. Puesto que ya disponemos de la referencia a la excepción actual, podemos simplemente volver a generar dicha referencia:

```
catch(Exception e) {
    System.out.println("An exception was thrown");
    throw e;
}
```

La regeneración de una excepción hace que ésta pase a las rutinas de tratamiento de excepciones del siguiente contexto de nivel superior. Las cláusulas **catch** adicionales incluidas en el mismo bloque **try** seguirán ignorándose. Además, se preserva toda la información acerca del objeto de excepción, por lo que la rutina de tratamiento en el contexto de nivel superior que capture ese tipo excepción específico podrá extraer toda la información de dicho objeto.

Si nos limitamos a regenerar la excepción actual, la información que imprimamos acerca de dicha excepción en **printStackTrace()** será la relativa al origen de la excepción, no la relativa al lugar donde la hayamos regenerado. Si queremos insertar nueva información de traza de la pila podemos hacerlo invocando **fillInStackTrace()**, que devuelve un objeto **Throwable** que habrá generado insertando la información de pila actual en el antiguo objeto de excepción. He aquí un ejemplo:

```
//: exceptions/Rethrowing.java
// Ilustración de fillInStackTrace()

public class Rethrowing {
    public static void f() throws Exception {
        System.out.println("originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println("Inside g(),e.printStackTrace()");
            e.printStackTrace(System.out);
            throw e;
        }
    }
    public static void h() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println("Inside h(),e.printStackTrace()");
            e.printStackTrace(System.out);
            throw (Exception)e.fillInStackTrace();
        }
    }
    public static void main(String[] args) {
        try {
            g();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
}
```

```

        }
    }
} /* Output:
originating the exception in f()
Inside g(),e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.g(Rethrowing.java:11)
    at Rethrowing.main(Rethrowing.java:29)
main: printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.g(Rethrowing.java:11)
    at Rethrowing.main(Rethrowing.java:29)
originating the exception in f()
Inside h(),e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:7)
    at Rethrowing.h(Rethrowing.java:20)
    at Rethrowing.main(Rethrowing.java:35)
main: printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.h(Rethrowing.java:24)
    at Rethrowing.main(Rethrowing.java:35)
*///:-
```

La línea donde se invoca `fillInStackTrace()` será el nuevo punto de origen de la excepción.

También resulta posible volver a generar una excepción distinta de aquella que hayamos capturado. Si hacemos esto, obtenemos un efecto similar a cuando usamos `fillInStackTrace()`: la información acerca del lugar de origen de la excepción se pierde, y lo que nos queda es la información correspondiente a la nueva instrucción `throw`:

```

//: exceptions/RethrowNew.java
// Regeneración de un objeto distinto de aquél que fue capturado.

class OneException extends Exception {
    public OneException(String s) { super(s); }
}

class TwoException extends Exception {
    public TwoException(String s) { super(s); }
}

public class RethrowNew {
    public static void f() throws OneException {
        System.out.println("originating the exception in f()");
        throw new OneException("thrown from f()");
    }
    public static void main(String[] args) {
        try {
            try {
                f();
            } catch(OneException e) {
                System.out.println(
                    "Caught in inner try, e.printStackTrace()");
                e.printStackTrace(System.out);
                throw new TwoException("from inner try");
            }
        } catch(TwoException e) {
            System.out.println(
```

```

        "Caught in outer try, e.printStackTrace());
        e.printStackTrace(System.out);
    }
}
} /* Output:
originating the exception in f()
Caught in inner try, e.printStackTrace()
OneException: thrown from f()
    at RethrowNew.f(RethrowNew.java:15)
    at RethrowNew.main(RethrowNew.java:20)
Caught in outer try, e.printStackTrace()
TwoException: from inner try
    at RethrowNew.main(RethrowNew.java:25)
*///:-
```

La excepción final sólo sabe que proviene del bloque `try` interno y no de `f()`.

Nunca hay que preocuparse acerca de borrar la excepción anterior, ni ninguna otra excepción. Se trata de objetos basados en el cúmulo de memoria que se crean con `new`, por lo que el depurador de memoria se encargará automáticamente de borrarlos.

Encadenamiento de excepciones

A menudo, nos interesa capturar una excepción y generar otra, pero manteniendo la información acerca de la excepción de origen; este procedimiento se denomina *encadenamiento de excepciones*. Antes de la aparición del JDK 1.4, los programadores tenían que escribir su propio código para preservar la información de excepción original, pero ahora todas las subclases de `Throwable` tienen la opción de admitir un objeto *causa* dentro de su constructor. Lo que se pretende es que la *causa* represente la excepción original, y al pasársela lo que hacemos es mantener la traza de la pila correspondiente al origen de la excepción, incluso aunque estemos generando una nueva excepción.

Resulta interesante observar que las únicas subclases de `Throwable` que proporcionan el argumento *causa* en el constructor son las tres clases de excepción fundamentales `Error` (utilizada por la máquina virtual JVM para informar acerca de los errores del sistema), `Exception` y `RuntimeException`. Si queremos encadenar cualquier otro tipo de excepción, tenemos que hacerlo utilizando el método `initCause()`, en lugar del constructor.

He aquí un ejemplo que nos permite añadir dinámicamente campos a un objeto `DynamicFields` en tiempo de ejecución:

```

//: exceptions/DynamicFields.java
// Una clase que añade dinámicamente campos a sí misma.
// Ilustra el mecanismo de encadenamiento de excepciones.
import static net.mindview.util.Print.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
    private Object[][] fields;
    public DynamicFields(int initialSize) {
        fields = new Object[initialSize][2];
        for(int i = 0; i < initialSize; i++)
            fields[i] = new Object[] { null, null };
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Object[] obj : fields) {
            result.append(obj[0]);
            result.append(": ");
            result.append(obj[1]);
            result.append("\n");
        }
        return result.toString();
    }
}
```

```

    }
    private int hasField(String id) {
        for(int i = 0; i < fields.length; i++)
            if(id.equals(fields[i][0]))
                return i;
        return -1;
    }
    private int
    getFieldNumber(String id) throws NoSuchFieldException {
        int fieldNum = hasField(id);
        if(fieldNum == -1)
            throw new NoSuchFieldException();
        return fieldNum;
    }
    private int makeField(String id) {
        for(int i = 0; i < fields.length; i++)
            if(fields[i][0] == null) {
                fields[i][0] = id;
                return i;
            }
        // No hay campos vacíos. Añadir uno:
        Object[][] tmp = new Object[fields.length + 1][2];
        for(int i = 0; i < fields.length; i++)
            tmp[i] = fields[i];
        for(int i = fields.length; i < tmp.length; i++)
            tmp[i] = new Object[] { null, null };
        fields = tmp;
        // Llamada recursiva con campos expandidos:
        return makeField(id);
    }
    public Object
    getField(String id) throws NoSuchFieldException {
        return fields[getFieldNumber(id)][1];
    }
    public Object setField(String id, Object value)
    throws DynamicFieldsException {
        if(value == null) {
            // La mayoría de las excepciones no tienen un constructor con "causa".
            // En estos casos es necesario emplear initCause(),
            // disponible en todas las subclases de Throwable.
            DynamicFieldsException dfe =
                new DynamicFieldsException();
            dfe.initCause(new NullPointerException());
            throw dfe;
        }
        int fieldNumber = hasField(id);
        if(fieldNumber == -1)
            fieldNumber = makeField(id);
        Object result = null;
        try {
            result = getField(id); // Obtener valor anterior
        } catch(NoSuchFieldException e) {
            // Utilizar constructor que admite la "causa":
            throw new RuntimeException(e);
        }
        fields[fieldNumber][1] = value;
        return result;
    }
    public static void main(String[] args) {

```

```

DynamicFields df = new DynamicFields(3);
print(df);
try {
    df.setField("d", "A value for d");
    df.setField("number", 47);
    df.setField("number2", 48);
    print(df);
    df.setField("d", "A new value for d");
    df.setField("number3", 11);
    print("df: " + df);
    print("df.getField(\"d\") : " + df.getField("d"));
    Object field = df.setField("d", null); // Excepción
} catch(NoSuchFieldException e) {
    e.printStackTrace(System.out);
} catch(DynamicFieldsException e) {
    e.printStackTrace(System.out);
}
}
} /* Output:
null: null
null: null
null: null

d: A value for d
number: 47
number2: 48

df: d: A new value for d
number: 47
number2: 48
number3: 11

df.getField("d") : A new value for d
DynamicFieldsException
    at DynamicFields.setField(DynamicFields.java:64)
    at DynamicFields.main(DynamicFields.java:94)
Caused by: java.lang.NullPointerException
    at DynamicFields.setField(DynamicFields.java:66)
    ... 1 more
*///:-
```

Cada objeto **DynamicFields** contiene una matriz de parejas **Object-Object**. El primer objeto es el identificador del campo (de tipo **String**), mientras que el segundo es el valor del campo, que puede ser de cualquier tipo, salvo una primitiva no envuelta en otro tipo de objeto. Cuando se crea el objeto, tratamos de adivinar cuántos campos vamos a necesitar. Cuando invocamos **setField()**, dicho método localiza el campo existente que tenga dicho nombre o crea un nuevo campo, colocando a continuación el valor en él. Si se queda sin espacio, se añade nuevo espacio creando una matriz de longitud igual a la anterior más uno y copiando en ella los antiguos elementos. Si tratamos de almacenar un valor **null**, se genera una excepción **DynamicFieldsException** creando una excepción y utilizando **initCause()** para insertar una excepción **NullPointerException** como la causa.

Como valor de retorno, **setField()** también extrae el antiguo valor situado en dicho campo utilizando **getField()**, que podría generar la excepción **NoSuchFieldException** (no existe un campo con dicho nombre). Si el programador de clientes invoca **getField()**, entonces será responsable de tratar la excepción **NoSuchFieldException**, pero si esta excepción se genera dentro de **setField()**, se tratará de un error de programación, por lo que **NoSuchFieldException** se convierte en una excepción **RuntimeException** utilizando el constructor que admite un argumento de *causa*.

Como podrá observar, **toString()** utiliza un objeto **StringBuilder** para crear su resultado. Hablaremos más en detalle acerca de **StringBuilder** en el Capítulo 13, *Cadenas de caracteres*, pero en general conviene utilizar este tipo de objetos cada vez que estemos escribiendo un método **toString()** que implique la utilización de bucles como es el caso aquí.

Ejercicio 10: (2) Cree una clase con dos métodos, `f()` y `g()`. En `g()`, genere una excepción de un nuevo tipo definido por el usuario. En `f()`, invoque a `g()`, capture su excepción y, en la cláusula `catch`, genere una excepción diferente (de un segundo tipo también definido por el usuario). Compruebe el código en `main()`.

Ejercicio 11: (1) Repita el ejercicio anterior, pero dentro de la cláusula `catch`, envuelva la excepción `g()` dentro de una excepción `RuntimeException`.

Excepciones estándar de Java

La clase Java `Throwable` describe todas las cosas que puedan generarse como una excepción. Existen dos tipos generales de objetos `Throwable` ("tipos de" = "que heredan de"). `Error` representa los errores de tiempo de compilación y del sistema de los que no tenemos que preocuparnos de capturar (salvo en casos muy especiales). `Exception` es el tipo básico que puede generarse desde cualquiera de los métodos de la biblioteca estándar de Java, así como desde nuestros propios métodos y también cuando se producen errores de ejecución. Por tanto, el tipo base que más interesa a los programadores de Java es usualmente `Exception`.

La mejor forma de obtener una panorámica de las excepciones consiste en examinar la documentación del JDK. Conviene hacer esto al menos una vez para tener una idea de las distintas excepciones, aunque si lo hace se dará cuenta pronto de que no existen diferencias muy grandes entre una excepción y otra, salvo en lo que se refiere al nombre. Asimismo, el número de excepciones en Java continua creciendo; es por ello que resulta absurdo tratar de imprimirlas todas en un libro. Cualquier nueva biblioteca que obtengamos de un fabricante de software dispondrá probablemente, asimismo, de sus propias excepciones. Lo importante es comprender el concepto y qué es lo que debe hacerse con las excepciones.

La idea básica es que el nombre de la excepción represente el problema que ha tenido lugar y que ese nombre de excepción trate de ser autoexplicativo. No todas las excepciones están definidas en `java.lang`; algunas se definen para dar soporte a otras bibliotecas como `util`, `net` e `io`, lo cual puede deducirse a partir del nombre completo de la clase correspondiente o de la clase de la que heredan. Por ejemplo, todas las excepciones de E/S heredan de `java.io.IOException`.

Caso especial: `RuntimeException`

El primer ejemplo de este capítulo era:

```
if(t == null)
    throw new NullPointerException();
```

Puede resultar un poco aterrador pensar que debemos comprobar si todas las referencias que se pasan a un método son iguales a `null` (dado que no podemos saber de antemano si el que ha realizado la invocación nos ha pasado una referencia válida). Afortunadamente, no es necesario realizar esa comprobación manualmente; dicha comprobación forma parte del sistema de comprobación estándar en tiempo de ejecución que Java aplica automáticamente, y si se realiza cualquier llamada a una referencia `null`, Java generará automáticamente la excepción `NullPointerException`. Por tanto, el anterior fragmento de código resulta siempre superfluo, aunque sí que puede resultar interesante realizar otras comprobaciones para protegerse frente a la aparición de una excepción `NullPointerException`.

Existe un conjunto completo de tipos de excepción que cae dentro de esta categoría. Se trata de excepciones que siempre son generadas de forma automática por Java y que no es necesario incluir en las especificaciones de excepciones. Afortunadamente, todas estas excepciones están agrupadas, dependiendo todas ellas de una única clase base denominada `RuntimeException`, que constituye un ejemplo perfecto de herencia: establece una familia de tipos que tienen determinadas características y comportamientos en común. Asimismo, nunca es necesario escribir una especificación de excepción que diga que un método puede generar `RuntimeException` (o cualquier tipo heredado de `RuntimeException`), porque se trata de excepciones *no comprobadas*. Puesto que estas excepciones indican errores de programación, normalmente no se suele capturar una excepción `RuntimeException`, sino que el sistema las trata automáticamente. Si nos viéramos obligados a comprobar la aparición de este tipo de excepciones, el código sería enormemente lioso. Pero, aunque normalmente no vamos a capturar excepciones `RuntimeException`, sí que podemos generar este tipo de excepciones en nuestros propios paquetes.

¿Qué sucede cuando no capturamos estas excepciones? Puesto que el compilador no obliga a incluir especificaciones de excepción para estas excepciones, resulta bastante posible que una excepción `RuntimeException` ascienda por toda la jerar-

queja de métodos sin ser capturada, hasta llegar al método `main()`. Para ver lo que sucede en este caso, trate de ejecutar el siguiente ejemplo:

```
//: exceptions/NeverCaught.java
// Lo que sucede al ignorar una excepción RuntimeException.
// {ThrowsException}
public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
    }
} //:-
```

Como puede ver, **RuntimeException** (o cualquier cosa que herede de ella) es un caso especial, ya que el compilador no requiere que incluyamos una especificación de excepción para estos tipos. La salida se envía a **System.err**:

```
Exception in thread "main" java.lang.RuntimeException: From f()
    at NeverCaught.f(NeverCaught.java:7)
    at NeverCaught.g(NeverCaught.java:10)
    at NeverCaught.main(NeverCaught.java:13)
```

Por tanto, la respuesta es: si una excepción **RuntimeException** llega hasta `main()` sin ser capturada, se invoca `printStackTrace()` para dicha excepción en el momento de salir del programa.

Recuerde que sólo las excepciones de tipo **RuntimeException** (y sus subclases) pueden ser ignoradas en nuestros programas, ya que el compilador obliga exhaustivamente a tratar todas las excepciones comprobadas. El razonamiento que explica esta forma de actuar es que **RuntimeException** representa un error de programación, que es:

1. Un error que no podemos anticipar. Por ejemplo, una referencia **null** que escapa a nuestro control.
2. Un error que nosotros, como programadores, deberíamos haber comprobado en nuestro código (como por ejemplo una excepción **ArrayIndexOutOfBoundsException**, que indica que deberíamos haber prestado atención al tamaño de una matriz). Una excepción que tiene lugar como consecuencia del punto 1 suele convertirse en un problema del tipo especificado en el punto 2.

Como puede ver, resulta enormemente beneficioso disponer de excepciones en este caso, ya que nos ayudan en el proceso de depuración.

Es interesante observar que el mecanismo de tratamiento de excepciones de Java no tiene un único objetivo. Por supuesto, está diseñado para tratar esos molestos errores de ejecución que tienen lugar debido a la acción de fuerzas que escapan al control de nuestro código, pero también resulta esencial para ciertos tipos de errores de programación que el compilador no puede detectar.

Ejercicio 12: (3) Modifique `innerclasses/Sequence.java` para que genere una excepción apropiada si tratamos de introducir demasiados elementos.

Realización de tareas de limpieza con `finally`

A menudo, existe algún fragmento de código que nos gustaría ejecutar independientemente de si la excepción ha sido generada dentro de un bloque `try`. Usualmente, ese fragmento de código se relaciona con alguna operación distinta de la de recuperación de memoria (ya que esta operación es realizada automáticamente por el depurador de memoria). Para conseguir este efecto, utilizamos una cláusula `finally`⁴ después de todas las rutinas de tratamiento de excepciones. La estructura completa de una sección de tratamiento de excepciones sería, por tanto:

⁴ El mecanismo de tratamiento de excepciones de C++ no dispone de la cláusula `finally`, porque depende de la utilización de destructores para llevar a cabo estas tareas de limpieza.

```

try {
    // La región protegida: actividades peligrosas
    // que pueden generar A, B o C
} catch(A a1) {
    // Rutina de tratamiento para la situación A
} catch(B b1) {
    // Rutina de tratamiento para la situación B
} catch(C c1) {
    // Rutina de tratamiento para la situación C
} finally {
    // Actividades que tienen lugar en todas las ocasiones
}

```

Para demostrar que la cláusula **finally** siempre se ejecuta, pruebe a ejecutar este programa:

```

//; exceptions/FinallyWorks.java
// La cláusula finally siempre se ejecuta.

class ThreeException extends Exception {}

public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // El post-incremento es cero la primera vez:
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("No exception");
            } catch(ThreeException e) {
                System.out.println("ThreeException");
            } finally {
                System.out.println("In finally clause");
                if(count == 2) break; // fuera del bucle "while"
            }
        }
    }
} /* Output:
ThreeException
In finally clause
No exception
In finally clause
*///:-*

```

Analizando la salida, podemos ver que la cláusula **finally** se ejecuta se haya generado o no una excepción.

Este programa también nos indica cómo podemos tratar con el hecho de que las excepciones en Java no nos permiten continuar con la ejecución a partir del punto donde se generó la excepción, como ya hemos indicado anteriormente. Si incluimos nuestro bloque **try** en un bucle, podremos establecer una condición que habrá que satisfacer antes de continuar con el programa. También podemos añadir un contador estático o algún otro tipo de elemento para permitir que el bucle pruebe con varias técnicas diferentes antes de darse por vencido. De esta forma, podemos proporcionar un mayor nivel de robustez a nuestros programas.

¿Para qué sirve **finally**?

En un lenguaje que no tenga depuración de memoria y que no tenga llamadas automáticas a destructores,⁵ la cláusula **finally** es importante porque permite al programador garantizar que se libera la memoria, independientemente de lo que suceda en

⁵ Un destructor es una función que siempre se invoca cuando un objeto deja de ser utilizado. Siempre sabemos exactamente dónde y cuándo se invoca al destructor. C++ dispone de llamadas automáticas a destructores, mientras que C#, que se parece más a Java, dispone de un mecanismo que hace posible que tenga lugar la destrucción automática.

el bloque **try**. Pero Java dispone de un depurador de memoria, por lo que la liberación de memoria casi nunca es un problema. Asimismo, no dispone de ningún destructor al que invocar. Por tanto, ¿cuándo es necesario utilizar **finally** en Java?

La cláusula **finally** es necesaria cuando tenemos que restaurar a su estado original *alguna otra cosa* distinta de la propia memoria. Se trata de algún tipo de tarea de limpieza que se encargue, por ejemplo, de cerrar un archivo abierto o una conexión de red, de borrar algo que hayamos dibujado en la pantalla o incluso de accionar un conmutador en el mundo exterior, tal como se ilustra en el siguiente ejemplo:

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;

public class Switch {
    private boolean state = false;
    public boolean read() { return state; }
    public void on() { state = true; print(this); }
    public void off() { state = false; print(this); }
    public String toString() { return state ? "on" : "off"; }
} //:-

//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} //:-

//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} //:-

//: exceptions/OnOffSwitch.java
// ¿Por qué usar finally?

public class OnOffSwitch {
    private static Switch sw = new Switch();
    public static void f()
    throws OnOffException1,OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Código que puede generar excepciones...
            f();
            sw.off();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
            sw.off();
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
            sw.off();
        }
    }
} /* Output:
on
off
*:-*/
```

Nuestro objetivo es asegurarnos de que el conmutador esté cerrado cuando se complete la ejecución de **main()**, por lo que situamos **sw.off()** al final del bloque **try** y al final de cada rutina de tratamiento de excepciones. Pero es posible que se genere alguna excepción que no sea capturada aquí, en cuyo caso no se ejecutaría **sw.off()**. Sin embargo, con **finally** podemos incluir el código de limpieza del bloque **try** en un único lugar:

```
//: exceptions/WithFinally.java
// Finally garantiza que se ejecuten las tareas de limpieza.

public class WithFinally {
```

```

static Switch sw = new Switch();
public static void main(String[] args) {
    try {
        sw.on();
        // Código que puede generar excepciones...
        OnOffSwitch.f();
    } catch(OnOffException1 e) {
        System.out.println("OnOffException1");
    } catch(OnOffException2 e) {
        System.out.println("OnOffException2");
    } finally {
        sw.off();
    }
}
} /* Output:
on
off
*///:-
```

Aquí, la llamada a **sw.off()** se ha desplazado incluyéndola en un único lugar, donde se garantiza que será realizada independientemente de lo que suceda.

Incluso en aquellos casos en que la excepción no es capturada en el conjunto actual de cláusulas **catch**, **finally** se ejecutaría antes de que el mecanismo de tratamiento de excepciones continúe buscando una rutina de tratamiento adecuada en el siguiente nivel de orden superior:

```

//: exceptions/AlwaysFinally.java
// Finally siempre se ejecuta.
import static net.mindview.util.Print.*;

class FourException extends Exception {}

public class AlwaysFinally {
    public static void main(String[] args) {
        print("Entering first try block");
        try {
            print("Entering second try block");
            try {
                throw new FourException();
            } finally {
                print("finally in 2nd try block");
            }
        } catch(FourException e) {
            System.out.println(
                "Caught FourException in 1st try block");
        } finally {
            System.out.println("finally in 1st try block");
        }
    }
} /* Output:
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
*///:-
```

La instrucción **finally** también se ejecutará en aquellas situaciones donde estén implicadas instrucciones **break** y **continue**. Observe que la cláusula **finally** junto con las instrucciones **break** y **continue** etiquetadas elimina la necesidad de una instrucción **goto** en Java.

- Ejercicio 13:** (2) Modifique el Ejercicio 9 añadiendo una cláusula **finally**. Verifique que la cláusula **finally** se ejecuta, incluso cuando se genera una excepción **NullPointerException**.
- Ejercicio 14:** (2) Demuestre que **OnOffSwitch.java** puede fallar, generando una excepción **RuntimeException** dentro del bloque **try**.
- Ejercicio 15:** (2) Demuestre que **WithFinally.java** no falla, generando una excepción **RuntimeException** dentro del bloque **try**.

Utilización de **finally** durante la ejecución de la instrucción **return**

Puesto que una cláusula **finally** siempre se ejecuta, resulta posible volver desde múltiples puntos dentro de un método sin dejar por ello de garantizar que se realicen las tareas de limpieza importantes:

```
//: exceptions/MultipleReturns.java
import static net.mindview.util.Print.*;

public class MultipleReturns {
    public static void f(int i) {
        print("Initialization that requires cleanup");
        try {
            print("Point 1");
            if(i == 1) return;
            print("Point 2");
            if(i == 2) return;
            print("Point 3");
            if(i == 3) return;
            print("End");
            return;
        } finally {
            print("Performing cleanup");
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 4; i++)
            f(i);
    }
} /* Output:
Initialization that requires cleanup
Point 1
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
*///:-
```

Podemos ver, en la salida del ejemplo, que no importa desde dónde volvamos, ya que siempre se ejecuta la cláusula **finally**.

Ejercicio 16: (2) Modifique `reusing/CADSystem.java` para demostrar que si volvemos desde un punto situado en la mitad de una estructura `try-finally` se seguirán ejecutando adecuadamente las tareas de limpieza.

Ejercicio 17: (3) Modifique `polymorphism/Frog.java` para que utilice la estructura `try-finally` con el fin de garantizar que se lleven a cabo las tareas de limpieza y demuestre que esto funciona incluso si ejecutamos una instrucción `return` en mitad de la estructura `try-finally`.

Un error: la excepción perdida

Lamentablemente, existe un fallo en la implementación del mecanismo de excepciones de Java. Aunque las excepciones son una indicación de que se ha producido una crisis en el programa y nunca deberían ignorarse, resulta posible que una excepción se pierda sin más. Esto sucede cuando se utiliza una configuración concreta con una cláusula `finally`:

```
//: exceptions/LostMessage.java
// Forma de perder una excepción.

class VeryImportantException extends Exception {
    public String toString() {
        return "A very important exception!";
    }
}

class HoHumException extends Exception {
    public String toString() {
        return "A trivial exception";
    }
}

public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args) {
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.f();
            } finally {
                lm.dispose();
            }
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
A trivial exception
*///:-
```

Podemos ver, analizando la salida, que no existe ninguna prueba de que se haya producido la excepción `VeryImportantException`, que es simplemente sustituida por la excepción `HoHumException` en la cláusula `finally`. Se trata de un fallo importante, puesto que implica que puede perderse completamente una excepción, y además puede perderse de una forma bastante más sutil y difícil de detectar que en el ejemplo anterior. Por contraste, C++ considera como un error de programación que se genere una segunda excepción antes de que la primera haya sido tratada. Quizá, una futura versión de Java solventará este problema (por otro lado, normalmente lo que haremos será encerrar cualquier método que genere una excepción, como es el caso de `dispose()` en el ejemplo anterior, dentro de una cláusula `try-catch`).

Una forma todavía más simple de perder una excepción consiste en volver con **return** desde dentro de una cláusula **finally**:

```
//: exceptions/ExceptionSilencer.java

public class ExceptionSilencer {
    public static void main(String[] args) {
        try {
            throw new RuntimeException();
        } finally {
            // La utilización de 'return' dentro de un bloque finally
            // hará que se pierda la excepción generada.
            return;
        }
    }
} //:-
```

Si ejecutamos este programa, veremos que no produce ninguna salida, a pesar de que se ha generado una excepción.

Ejercicio 18: (3) Añada un segundo nivel de pérdida de excepciones a **LostMessage.java** para que la propia excepción **HoHumException** sea sustituida por una tercera excepción.

Ejercicio 19: (2) Solucione el problema de **LostMessage.java** protegiendo la llamada contenida en la cláusula **finally**.

Restricciones de las excepciones

Cuando sustituimos un método, sólo podemos generar aquellas excepciones que hayan sido especificadas en la versión del método correspondiente a la clase base. Se trata de una restricción muy útil, ya que implica que el código que funcione con la clase base funcionará también automáticamente con cualquier objeto derivado de la clase base (lo cual, por supuesto, es un concepto fundamental dentro de la programación orientada a objetos), incluyendo las excepciones.

Este ejemplo ilustra los tipos de restricciones impuestas a las excepciones (en tiempo de compilación):

```
//: exceptions/StormyInning.java
// Los métodos sustituidos sólo pueden generar las excepciones
// especificadas en sus versiones de la clase base,
// o excepciones derivadas de las excepciones de la clase base.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
    public Inning() throws BaseballException {}
    public void event() throws BaseballException {
        // No tiene por qué generar nada
    }
    public abstract void atBat() throws Strike, Foul;
    public void walk() {} // No genera ninguna excepción comprobada
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
    public void event() throws RainedOut;
    public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements Storm {
```

En **Inning**, podemos ver que tanto el constructor como el método **event()** especifican que generan una excepción, pero que nunca lo hacen. Esto es legal, porque nos permite obligar al usuario a capturar cualquier excepción que podamos añadir en las versiones sustituidas de **event()**. La misma idea puede aplicarse a los métodos abstractos como podemos ver en **atBat()**.

La interfaz **Storm** es interesante porque contiene un método (`event()`) que está definido en **Inning**, y otro método que no lo está. Ambos métodos generan un nuevo tipo de excepción, **RainedOut**. Cuando **StormyInning** amplía (**extends**) **Inning** e implementa (**implements**) **Storm**, podemos ver que el método `event()` de **Storm** *no puede* cambiar la interfaz de excepciones de `event()` definida en **Inning**. De nuevo, esto tiene bastante sentido porque en caso contrario nunca sabriamos si estamos capturando el objeto correcto a la hora de trabajar con la clase base. Por supuesto, si un método descrito en una interfaz no se encuentra en la clase base, como es el caso de `rainHard()`, no existe ningún problema en cuanto a las excepciones que genere.

La restricción relativa a las excepciones no se aplica a los constructores. Podemos ver en **StormyInning** que un constructor puede generar todo aquello que desee, independientemente de lo que genere el constructor de la clase base. Sin embargo, puesto que siempre hay que invocar el constructor de la clase base de una forma o de otra (aqui se invoca el constructor predeterminado de manera automática), el constructor de la clase derivada deberá declarar todas las excepciones del constructor de la clase base en su propia especificación de excepciones.

Un constructor de la clase derivada no puede capturar las excepciones generadas por su constructor de la clase base.

La razón por la que **StormyInning.walk()** no podrá compilarse es que genera una excepción, mientras que **Inning.walk()** no lo hace. Si se permitiera esto, entonces podríamos escribir código que invocara a **Inning.walk()** y que no tuviera que tratar ninguna excepción, pero entonces, cuando efectuáramos una sustitución y empleáramos un objeto de una clase derivada de **Inning**, podrían generarse excepciones, con lo que nuestro código fallaría. Obligando a los métodos de la clase derivada a adaptarse a las especificaciones de excepciones de los métodos de la clase base, se mantiene la posibilidad de sustituir los objetos.

El método sustituido **event()** muestra que la versión de un método definido en la clase derivada puede elegir no generar ninguna excepción, incluso a pesar de que la versión de la clase sí las genere. De nuevo, no pasa nada por hacer esto, ya que no dejarán de funcionar aquellos programas que se hayan escrito bajo la suposición de que la versión de la clase base genera excepciones. Podemos aplicar una lógica similar a **atBat()**, que genera **PopFoul**, una excepción que deriva de la excepción **Foul** generada por la versión de la clase base de **atBat()**. De esta forma, si escribimos código que funcione con **Inning** y que invoque **atBat()**, deberemos capturar la excepción **Foul**. Puesto que **PopFoul** deriva de **Foul**, la rutina de tratamiento de excepciones también permitirá capturar **PopFoul**.

El último punto de interés se encuentra en **main()**. Aquí, podemos ver que, si estamos tratando con un objeto que sea exactamente del tipo **StormyInning**, el compilador nos obligará a capturar únicamente las excepciones que sean específicas de esa clase, pero si efectuamos una generalización al tipo base, entonces el compilador (correctamente) nos obligará a capturar las excepciones del tipo base. Todas estas restricciones permiten obtener un código de tratamiento de excepciones mucho más robusto⁶.

Aunque es el compilador el que se encarga de imponer las especificaciones de excepciones en los casos de herencia, esas especificaciones de excepciones no forman parte de la firma de un método, que está compuesta sólo por el nombre del método y los tipos de argumentos. Por tanto, no es posible sobrecargar los métodos basándose solamente en las especificaciones de excepciones. Además, el hecho de que exista una especificación de excepción en la versión de la clase base de un método no quiere decir que dicha especificación deba existir en la versión de la clase derivada del método. Esto difiere bastante de las reglas normales de herencia, según las cuales un método de la clase base deberá también existir en la clase derivada. Dicho de otra forma, la “interfaz de especificación de excepciones” de un método concreto puede estrecharse durante la herencia y cuando se realizan sustituciones, pero lo que no puede es ensancharse; se trata, precisamente, de la regla opuesta a la que se aplica a la interfaz de una clase durante la herencia.

Ejercicio 20: (3) Modifique **StormyInning.java** añadiendo un tipo excepción **UmpireArgument** y una serie de métodos que generen esta excepción. Compruebe la jerarquía modificada.

Constructores

Es importante que siempre nos hagamos la pregunta siguiente: “Si se produce una excepción, ¿se limpiará todo apropiadamente?” La mayor parte de las veces, podemos estar razonablemente seguros, pero con los constructores existe un problema. El constructor sitúa los objetos en un estado inicial seguro, pero puede realizar alguna operación (como por ejemplo abrir un archivo) que no revierta hasta que el usuario termine con el objeto e invoque un método de limpieza especial. Si generamos una excepción desde dentro de un constructor, puede que estas tareas de limpieza no se lleven a cabo apropiadamente. Esto quiere decir que debemos tener un especial cuidado a la hora de escribir los constructores.

Podíamos pensar que la cláusula **finally** es una solución. Pero las cosas no son tan simples, porque **finally** lleva a cabo las tareas de limpieza *todas las veces*. Si un constructor falla en mitad de la ejecución, puede que no haya tenido tiempo de crear alguna parte del objeto que será limpiado en la cláusula **finally**.

⁶ El estándar ISO C++ ha añadido unas restricciones similares, que obligan a que las excepciones de los métodos derivados sean iguales a las excepciones generadas por los métodos de la clase base, o al menos a que deriven de ellas. Éste es uno de los casos en los que C++ es capaz de comprobar las especificaciones de excepciones en tiempo de compilación.

En el siguiente ejemplo, se crea una clase denominada **InputFile** que abre un archivo y permite leerlo linea a linea. Utiliza las clases **FileReader** y **BufferedReader** de la biblioteca estándar E/S de Java que se analizará en el Capítulo 18, *E/S*. Estas clases son lo suficientemente simples como para que el lector no tenga ningún problema a la hora de comprender los fundamentos de su utilización:

```
//: exceptions/InputFile.java
// Hay que prestar a las excepciones en los constructores.
import java.io.*;

public class InputFile {
    private BufferedReader in;
    public InputFile(String fname) throws Exception {
        try {
            in = new BufferedReader(new FileReader(fname));
            // Otro código que pueda generar excepciones
        } catch(FileNotFoundException e) {
            System.out.println("Could not open " + fname);
            // No estará abierto, por lo que no hay que cerrarlo.
            throw e;
        } catch(Exception e) {
            // Todas las demás excepciones deben cerrarlo
            try {
                in.close();
            } catch(IOException e2) {
                System.out.println("in.close() unsuccessful");
            }
            throw e; // Regenerar
        } finally {
            // !!! No cerrarlo aquí!!!
        }
    }
    public String getLine() {
        String s;
        try {
            s = in.readLine();
        } catch(IOException e) {
            throw new RuntimeException("readLine() failed");
        }
        return s;
    }
    public void dispose() {
        try {
            in.close();
            System.out.println("dispose() successful");
        } catch(IOException e2) {
            throw new RuntimeException("in.close() failed");
        }
    }
} ///:-
```

El constructor de **InputFile** toma un argumento **String**, que representa el nombre del archivo que queremos abrir. Dentro de un bloque **try**, crea un objeto **FileReader** utilizando el nombre del archivo. Un objeto **FileReader** no resulta particularmente útil hasta que lo empleemos para crear otro objeto **BufferedReader** (para lectura con *buffer*). Uno de los beneficios de **InputFile** es que combina las dos acciones.

Si el constructor de **FileReader** falla, generará una excepción **FileNotFoundException**, que indicará que no se ha encontrado el archivo. Éste es el único caso en el cual no queremos cerrar el archivo, ya que no hemos llegado a poder abrirlo. *Cualquier otra cláusula catch* deberá cerrar el archivo, porque *estará abierto* en el momento de entrar en dicha cláusula **catch** (por supuesto, el asunto se complica si hay más de un método que pueda generar una excepción **FileNotFoundException**. En dicho caso, normalmente, habrá que descomponer las cosas en varios bloques **try**). El método

do **close()** puede generar una excepción, así que lo encerramos dentro de una cláusula **try** y tratamos de capturar la excepción aún cuando ese método se encuentre dentro del bloque de otra cláusula **catch**; para el compilador de Java se trata simplemente de un par adicional de símbolos de llave. Después de realizar las operaciones locales, la excepción se vuelve a generar, lo cual resulta apropiado porque este constructor ha fallado y no queremos que el método que ha hecho la invocación asuma que el objeto se ha creado apropiadamente y es válido.

En este ejemplo, la cláusula **finally** *no es*, en modo alguno, el lugar donde cerrar el archivo con **close()**, ya que eso haría que el archivo se cerrara cuando el constructor completara su ejecución. Lo que queremos es que el archivo continúe abierto mientras dure la vida útil del objeto **Inputfile**.

El método **getLine()** devuelve un objeto **String** que contiene la siguiente línea del archivo. Dicho método invoca a **readLine()**, que puede generar una excepción, pero dicha excepción es capturada, por lo que **getLine()** no genera excepción alguna. Uno de los problemas de diseño relativos a las excepciones es el de si debemos tratar una excepción completamente en este nivel, si sólo debemos tratarla parcialmente y pasar la misma excepción (u otra distinta) al nivel siguiente, o si debemos pasar la excepción directamente al siguiente nivel. Pasar la excepción directamente, siempre que sea apropiado, puede simplificar bastante el programa. En nuestro caso, el método **getLine()** convierte la excepción al tipo **RuntimeException** para indicar qué se ha producido un error de programación.

El método **dispose()** debe ser llamado por el usuario cuando ya no se necesite el método **Inputfile**. Esto hará que se liberen los recursos del sistema (como por ejemplo los descriptores de archivo) que estén siendo utilizados por los objetos **BufferedReader** y/o **FileReader**. Evidentemente, no queremos hacer esto hasta que hayamos terminado de utilizar el objeto **Inputfile**. Podríamos pensar en incluir dicha funcionalidad en un método **finalize()**, pero como hemos dicho en el Capítulo 5, *Inicialización y limpieza*, no siempre podemos estar seguros de que se vaya a llamar a **finalize()** (e, incluso si *estuvieramos* seguros de que va a ser llamado, lo que no sabemos es *cuando*). Ésta es una de las desventajas de Java: las tareas de limpieza (exceptuando las de memoria) no tienen lugar automáticamente, por lo que es preciso informar a los programadores de clientes de que ellos son los responsables.

La forma más segura de utilizar una clase que pueda generar una excepción durante la construcción y que requiera que se lleven a cabo tareas de limpieza consiste en emplear bloques **try** anidados:

```
//: exceptions/Cleanup.java
// Forma de garantizar la apropiada limpieza de un recurso.

public class Cleanup {
    public static void main(String[] args) {
        try {
            Inputfile in = new Inputfile("Cleanup.java");
            try {
                String s;
                int i = 1;
                while((s = in.getLine()) != null)
                    ; // Realizar aquí el procesamiento linea a linea...
            } catch(Exception e) {
                System.out.println("Caught Exception in main");
                e.printStackTrace(System.out);
            } finally {
                in.dispose();
            }
        } catch(Exception e) {
            System.out.println("Inputfile construction failed");
        }
    }
} /* Output:
dispose() successful
*///:-
```

Examine cuidadosamente la lógica utilizada: la construcción del objeto **Inputfile** se encuentra en su propio bloque **try**. Si dicha construcción falla, se entra la cláusula **catch** externa y no se invoca el método **dispose()**. Sin embargo, si la construcción tiene éxito, entonces hay que asegurarse de que el objeto se limpie, por lo que inmediatamente después de la construcción creamos un nuevo bloque **try**. La cláusula **finally** que lleva a cabo las tareas de limpieza está asociada con el bloque

try interno; de esta forma, la cláusula **finally** no se ejecuta si la construcción falla, mientras que *siempre* se ejecuta si la construcción tiene éxito.

Esta técnica general de limpieza debe utilizarse aún cuando el constructor no genere ninguna excepción. La regla básica es: justo después de crear un objeto que requiera limpieza, incluya una estructura **try-finally**:

```
//: exceptions/CleanupIdiom.java
// Cada objeto eliminable debe estar seguido por try-finally

class NeedsCleanup { // Construction can't fail
    private static long counter = 1;
    private final long id = counter++;
    public void dispose() {
        System.out.println("NeedsCleanup " + id + " disposed");
    }
}

class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
    // La construcción no puede fallar:
    public NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
    public static void main(String[] args) {
        // Sección 1:
        NeedsCleanup nc1 = new NeedsCleanup();
        try {
            // ...
        } finally {
            nc1.dispose();
        }

        // Sección 2:
        // Si la construcción no puede fallar, podemos agrupar los objetos:
        NeedsCleanup nc2 = new NeedsCleanup();
        NeedsCleanup nc3 = new NeedsCleanup();
        try {
            // ...
        } finally {
            nc3.dispose(); // Orden inverso al de construcción
            nc2.dispose();
        }

        // Sección 3:
        // Si la construcción puede fallar, hay que proteger cada uno:
        try {
            NeedsCleanup2 nc4 = new NeedsCleanup2();
            try {
                NeedsCleanup2 nc5 = new NeedsCleanup2();
                try {
                    // ...
                } finally {
                    nc5.dispose();
                }
            } catch(ConstructionException e) { // constructor de nc5
                System.out.println(e);
            } finally {
                nc4.dispose();
            }
        }
    }
}
```

```

        }
    } catch(ConstructionException e) { // constructor de nc4
        System.out.println(e);
    }
}
/* Output:
NeedsCleanup 1 disposed
NeedsCleanup 3 disposed
NeedsCleanup 2 disposed
NeedsCleanup 5 disposed
NeedsCleanup 4 disposed
*///:-
```

En `main()`, la sección 1 nos resulta bastante sencilla de entender: incluimos una estructura **try-finally** después de un objeto eliminable. Si la construcción del objeto no puede fallar, no es necesario incluir ninguna cláusula **catch**. En la sección 2, podemos ver que los objetos con constructores que no pueden fallar pueden agruparse tanto para las tareas de construcción como para las de limpieza.

La sección 3 muestra cómo tratar con aquellos objetos cuyos constructores pueden fallar y que necesitan limpieza. Para poder manejar adecuadamente esta situación, las cosas se complican, porque es necesario rodear cada construcción con su propia estructura **try-catch**, y cada construcción de objeto debe ir seguida de un **try-finally** para garantizar la limpieza.

Lo complicado del tratamiento de excepciones en este caso es un buen argumento en favor de la creación de constructores que no puedan fallar, aunque lamentablemente esto no siempre es posible.

Observe que si **dispose()** puede generar una excepción, entonces serán necesarios bloques **try** adicionales. Básicamente, lo que debemos hacer es pensar con cuidado en todas las posibilidades y protegernos frente a cada una.

Ejercicio 21: (2) Demuestre que un constructor de una clase derivada no puede capturar excepciones generadas por su constructor de la clase base.

Ejercicio 22: (2) Cree una clase denominada **FailingConstructor** con un constructor que pueda fallar en mitad del proceso de construcción y generar una excepción. En `main()`, escriba el código que permita protegerse apropiadamente frente a este fallo.

Ejercicio 23: (4) Añada una clase con un método **dispose()** al ejercicio anterior. Modifique **FailingConstructor** para que el constructor cree uno de estos objetos eliminables como un objeto miembro, después de lo cual el constructor puede generar una excepción y crear un segundo objeto miembro eliminable. Escriba el código necesario para protegerse adecuadamente contra los fallos y verifique en `main()` que están cubiertas todas las posibles situaciones de fallo.

Ejercicio 24: (3) Añada un método **dispose()** a la clase **FailingConstructor** y escriba el código necesario para utilizar adecuadamente esta clase.

Localización de excepciones

Cuando se genera una excepción, el sistema de tratamiento de excepciones busca entre las rutinas de tratamiento “más cercanas”, en el orden en que fueron escritas. Cuando encuentra una correspondencia, se considera que la excepción ha sido tratada y no se continúa con el proceso de búsqueda.

Localizar la excepción correcta no requiere que haya una correspondencia perfecta entre la excepción y su rutina de tratamiento. Todo objeto de una clase derivada se corresponderá con una rutina de tratamiento correspondiente a la clase base, como se muestra en este ejemplo:

```

//: exceptions/Human.java
// Captura de jerarquías de excepciones.

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
```

```

public static void main(String[] args) {
    // Capturar el tipo exacto:
    try {
        throw new Sneeze();
    } catch(Sneeze s) {
        System.out.println("Caught Sneeze");
    } catch(Annoyance a) {
        System.out.println("Caught Annoyance");
    }
    // Capturar el tipo base:
    try {
        throw new Sneeze();
    } catch(Annoyance a) {
        System.out.println("Caught Annoyance");
    }
}
} /* Output:
Caught Sneeze
Caught Annoyance
*///:-
```

La excepción **Sneeze** será capturada por la primera cláusula **catch** con la que se corresponda, que será por supuesto la primera. Sin embargo, si eliminamos la primera cláusula **catch**, dejando sólo la cláusula **catch** correspondiente a **Annoyance**, el código seguirá funcionando porque se está capturando la clase base de **Sneeze**. Dicho de otra forma, **catch(Annoyance a)** permitirá capturar una excepción **Annoyance** o cualquier clase derivada de ella. Esto resulta útil porque si decidimos añadir más excepciones derivadas a un método, no será necesario cambiar el código de los programas cliente, siempre y cuando el cliente capture las excepciones de la clase base.

Si tratamos de “enmascarar” las excepciones de la clase derivada, incluyendo primero la cláusula **catch** correspondiente a la clase base, como en el siguiente ejemplo:

```

try {
    throw new Sneeze();
} catch(Annoyance a) {
    // ...
} catch(Sneeze s) {
    // ...
}
```

el compilador nos dará un mensaje de error, ya que verá que la cláusula **catch** correspondiente a **Sneeze** nunca puede ejecutarse.

Ejercicio 25: (2) Cree una jerarquía de excepciones en tres niveles. Ahora cree una clase base **A** con un método que genere una excepción de la base de nuestra jerarquía. Herede una clase **B** de **A** y sustituya el método para que genere una excepción en el nivel dos de la jerarquía. Repita el proceso, heredando una clase **C** de **B**. En **main()**, cree un objeto **C** y generalicelo a **A**, invoque el método a continuación.

Enfoques alternativos

Un sistema de tratamiento de excepciones es un mecanismo especial que permite a nuestros programas abandonar la ejecución de la secuencia normal de instrucciones. Ese mecanismo especial se utiliza cuando tiene lugar una “condición excepcional”, tal que la ejecución normal ya no es posible o deseable. Las excepciones representan condiciones que el método actual no es capaz de gestionar. La razón por la que se desarrollaron los sistemas de tratamiento de excepciones es porque la técnica de gestionar cada posible condición de error producida por cada llamada a función era demasiado onerosa, lo que hacía que los programadores no la implementaran. Como resultado, se terminaba ignorando los errores en los programas. Merece la pena recalcar que incrementar la comodidad de los programadores a la hora de tratar los errores fue una de las principales motivaciones para desarrollar los sistemas de tratamiento de excepciones.

Una de las directrices de mayor importancia en el tratamiento de excepciones es “no captures una excepción a menos que sepa qué hacer con ella”. De hecho, uno de los *objetivos* más importantes del tratamiento de excepciones es quitar el código de tratamiento de errores del punto en el que los errores se producen. Esto nos permite concentrarnos en lo que queremos conseguir en cada sección del código, dejando la manera de tratar con los problemas para una sección separada del mismo código. Como resultado, el código principal no se ve oscurecido por la lógica de tratamiento de errores, con lo que resulta mucho más fácil de comprender y de mantener. Los mecanismos de tratamiento de excepciones también tienden a reducir la cantidad de código dedicado a estas tareas, permitiendo que una rutina de tratamiento de servicio a múltiples lugares posibles de generación de errores.

Las excepciones comprobadas complican un poco este escenario, porque nos fuerzan a añadir cláusulas **catch** en lugares en los que puede que no estemos listos para gestionar un error. Esto puede dar como resultado que no se traten ciertas excepciones:

```
try {
    // ... hacer algo útil
} catch(ObligatoryException e) {} // Glub!
```

Los programadores (incluido yo en la primera edición de este libro) tienden a hacer lo más simple y a capturar la excepción olvidándose luego de tratarla; esto se hace a menudo de forma inadvertida, pero una vez que lo hacemos, el compilador se queda satisfecho, de modo que si no nos acordamos de revisar y corregir el código, esa excepción se pierde. La excepción tiene lugar, pero desaparece todo rastro de la misma una vez que ha sido capturada y se deja sin tratar. Puesto que el compilador nos fuerza a escribir código desde el principio para tratar la excepción, incluir una rutina de tratamiento vacía parece la solución más simple, aunque en realidad es lo peor que podemos hacer.

Horrorizado al darme cuenta de que yo había hecho precisamente esto, en la segunda edición del libro “corregí” el problema imprimiendo la traza de la pila dentro de la rutina de tratamiento (como puede verse apropiadamente en varios ejemplos de este capítulo). Aunque resulta útil trazar el comportamiento de las excepciones, esta forma de proceder sigue indicando que realmente no sabemos qué hacer con esa excepción en dicho punto del código. En esta sección, vamos a estudiar algunos de los problemas y algunas de las complicaciones derivadas de las excepciones comprobadas, y vamos a repasar las opciones que tenemos a la hora de tratar con ellas.

El tema parece bastante simple, pero no sólo resulta complicado sino que también es motivo de controversia. Hay personas que sostienen firmemente los argumentos de ambos bandos y que piensan que la respuesta correcta (es decir, la suya) resulta tremadamente obvia. En mi opinión, la razón de que den estas posiciones tan vehementes es que resulta bastante obvia la ventaja que se obtiene al pasar de un lenguaje con un pobre tratamiento de los tipos, como el previo ANSI C a un lenguaje fuertemente tipado con tipos estáticos (es decir, comprobados en tiempo de compilación) como C++ o Java. Cuando se hace esa transición (como hice yo mismo), las ventajas resultan tan evidentes que puede parecer que la comprobación estática de tipos es siempre la mejor respuesta a la mayoría de los problemas. Mi esperanza con las líneas que siguen es que, al relatar mi propia evolución, el lector pueda ver que el valor *absoluto* de la comprobación estática de tipos es cuestionable; obviamente, resulta muy útil la mayor parte de las veces, pero hay un línea tremadamente difusa a partir de la cual esa comprobación estática de tipos comienza a ser un estorbo y un problema (una de mis citas favoritas es la que dice “todos los modelos son erróneos, aunque algunos de ellos resultan útiles”).

Historia

Los sistemas de tratamiento de excepciones tienen su origen en sistemas como PL/I y Mesa, y posteriormente se incorporaron en CLU, Smalltalk, Modula-3, Ada, Eiffel, C++, Python, Java y los lenguajes post-Java como Ruby y C#. El diseño de Java es similar a C++, excepto en aquellos lugares en los que los diseñadores de Java pensaron que la técnica usada en C++ podría causar problemas.

Para proporcionar a los programadores un marco de trabajo que estuvieran más dispuestos a utilizar para el tratamiento y la recuperación de errores, el sistema de tratamiento de excepciones se añadió a C++ bastante tarde en el proceso de estandarización, promovido por Bjarne Stroustrup, el autor original del lenguaje. El modelo de las excepciones de C++ proviene principalmente de CLU. Sin embargo, en aquel entonces existían otros lenguajes que también soportaban el tratamiento de excepciones: Ada, Smalltalk (ambos tienen excepciones, pero no tienen especificaciones de excepciones) y Modula-3 (que incluía tanto las excepciones como las especificaciones).

En su artículo pionero⁷ sobre el tema, Liskov y Snyder observaron que uno de los principales defectos de los lenguajes tipo C, que informan acerca de los errores de manera transitoria es que:

"...toda invocación debe ir seguida de una prueba incondicional para determinar cuál ha sido el resultado. Este requisito hace que se desarrollen programas difíciles de leer y que probablemente también son poco eficientes, lo que tiende a desanimar a los programadores a la hora de señalizar y tratar las excepciones."

Por tanto, uno de los motivos originales para desarrollar sistemas de tratamiento de excepciones era eliminar este requisito, pero con las excepciones comprobadas en Java nos encontramos precisamente con este tipo de código. Los autores continúan diciendo:

"...si se requiere que se asocie el texto de una rutina de tratamiento a la invocación que genera la excepción, el resultado serán programas poco legibles en los que las expresiones estarán descompuestas debido a la presencia de las rutinas de tratamiento."

Siguiendo el enfoque adoptado en CLU, Stroustrup afirmó, al diseñar las excepciones de C++, que el objetivo era reducir la cantidad de código requerida para recuperarse de los errores. En mi opinión, estaba partiendo de la observación de que los programadores no solían escribir código de tratamiento de errores en C debido a que la cantidad y la colocación de dicho código en los programas era muy difícil de manejar y tendía a distraer del objetivo principal del programa. Como resultado, los programadores solían abordar el problema de la misma manera que en C, ignorando los errores en el código y utilizando depuradores para localizar los problemas. Para usar las excepciones, había que convencer a estos programadores de C de que escribieran código "adicional", que normalmente no escribirían. Por tanto, para hacer que pasen a adoptar una forma más eficiente de tratar los errores, la cantidad de código que esos programadores deben "añadir" no debe ser excesiva. Resulta importante tener presente este objetivo de diseño inicial a la hora de eliminar los efectos que las excepciones comprobadas tienen en Java.

C++ tomó prestada de CLU una idea adicional: la especificación de excepción, mediante la cual se enuncian programáticamente en la firma del método las excepciones que pueden generarse como resultado de la llamada al método. La especificación de excepciones tiene, en realidad, dos objetivos. Puede querer decir: "Puedo generar esta excepción en mi código; encárgate de tratarla". Pero también puede significar: "Estoy ignorando esta excepción que puede producirse como resultado de mi código, encárgate de tratarla". Hasta ahora, nos estamos centrando en la parte que dice "encárgate de tratarla" a la hora de examinar la mecánica y las sintaxis de las excepciones, pero lo que en este momento concreto nos interesa es el hecho de que a menudo ignoramos las excepciones que se producen en nuestro código, y eso es precisamente lo que la especificación de excepciones puede indicar.

En C++, la especificación de excepciones no forma parte de la información de tipo de una función (la firma). La única comprobación que se realiza en tiempo de compilación consiste en garantizar que las especificaciones de excepciones se utilizan de manera coherente, por ejemplo, si una función o un método generan excepciones, entonces las versiones sobrecargadas o derivadas también deberán generar esas excepciones. A diferencia de Java, sin embargo, no se realiza ninguna comprobación en tiempo de compilación para determinar si la función o método va a generar en realidad dicha excepción, o si la especificación de excepciones está completa (es decir, si describe con precisión todas las excepciones que puedan ser generadas). Esta validación sí que se produce, pero sólo en tiempo de ejecución. Si se genera una excepción que viola la especificación de excepciones, el programa C++ invocará la función de la biblioteca estándar `unexpected()`.

Resulta interesante observar que, debido al uso de plantillas, las especificaciones de excepciones no se utilizan en absoluto en la biblioteca estándar de C++. En Java, existen una serie de restricciones que afectan a la forma en que pueden emplearse los genéricos Java con las especificaciones de excepciones.

Perspectivas

En primer lugar, merece la pena observar que es el lenguaje Java el que ha inventado las excepciones comprobadas (inspiradas claramente en las especificaciones de excepciones de C++ y en el hecho de que los programadores C++ no suelen ocuparse de las mismas). Sin embargo, se trata de un experimento que ningún lenguaje subsiguiente ha incorporado.

⁷ Barbara Liskov y Alan Snyder, *Exception Handling in CLU*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, Noviembre 1979. Este artículo no está disponible en Internet, sino sólo en copia impresa, por lo que tendrá que encargar una copia a través de su biblioteca.

En segundo lugar, las excepciones comprobadas parecen ser algo “evidentemente bueno” cuando se las contempla dentro de ejemplos de nivel introductorio y en pequeños programas. Según algunos autores, las dificultades más sutiles comienzan a aparecer en el momento en que los programas crecen de tamaño. Por supuesto, el tamaño de los programas no suele incrementarse de manera espectacular de la noche a la mañana, sino que lo más normal es que los programas vayan creciendo de tamaño poco a poco. Los lenguajes que puedan no ser adecuados para proyectos de gran envergadura, se utilizan sin problema para proyectos de pequeño tamaño. Pero esos proyectos crecen y, en algún punto, nos damos cuenta de que las cosas que antes eran manejables ahora son relativamente difíciles. A eso es a lo que me refería al comentar que los mecanismos de comprobación de tipos pueden llegar a hacerse engorrosos: en particular, cuando esos mecanismos se combinan con el concepto de excepciones comprobadas.

El tamaño del programa parece ser una de las cuestiones principales. Y esto es, en sí mismo, un problema porque la mayoría de los análisis tienden a utilizar como ilustración programas de pequeño tamaño. Uno de los diseñadores de C# escribió que:

“El examen de programas de pequeño tamaño nos lleva a la conclusión de que imponer el uso de especificaciones de excepciones podría mejorar tanto la productividad del desarrollador como la calidad de código, pero la experiencia con los grandes proyectos de desarrollo software sugiere un resultado completamente distinto: una menor productividad y un incremento en la calidad del código que es, como mucho, poco significativo.”⁸

En referencias a las excepciones no capturadas, los creadores de CLU escribían:

“Pensamos que era poco realista exigir al programador que proporcionara rutinas de tratamiento en aquellas situaciones en las que no es posible llevar a cabo ninguna acción con verdadero significado.”⁹

A la hora de explicar por qué una declaración de función sin ninguna especificación significa que la función pueda generar cualquier excepción en lugar de *ninguna* excepción, Stroustrup escribe:

“Sin embargo, eso requeriría que se incluyeran especificaciones de excepción para casi todas las funciones, haría que fuera necesario efectuar muchas recompilaciones y dificultaría la cooperación con el software escrito en otros lenguajes. Esto animaría a los programadores a subvertir los mecanismos de tratamiento de excepciones y a escribir código espurio para suprimir las excepciones. Proporcionaría un falso sentido de seguridad a las personas que no se hubieran dado cuenta de la excepción.”¹⁰

Precisamente, con las excepciones comprobadas en Java podemos ver que se produce precisamente esta reacción: tratar de subvertir las excepciones.

Martin Fowler (autor de *UML Distilled, Refactoring* y *Analysis Patterns*) escribía en cierta ocasión lo siguiente:

“...en conjunto, creo que las excepciones son buenas, pero las excepciones comprobadas en Java causan más problemas de los que resuelven.”

Actualmente, lo que opino es que el paso más importante dado por Java fue unificar el modelo de información de errores, de modo que de todos los errores se informa utilizando excepciones. Esto no sucedía en C++, porque, debido a la compatibilidad descendente con C, seguía estando disponible el modelo de limitarse a ignorar los errores. Pero, cuando disponemos de un mecanismo de informe de errores coherente con excepciones, las excepciones pueden utilizarse si se desea y, en caso contrario, se propagarán al siguiente nivel superior (la consola u otro programa contenedor). Cuando Java modificó el modelo C++ para que las excepciones fueran la única forma de informar de los errores, la imposición adicional relativa a las excepciones comprobadas puede que haya dejado de ser tan necesaria.

En el pasado, creía firmemente que tanto las excepciones comprobadas como la comprobación estática de tipos resultaban esenciales para el desarrollo de programas robustos. Sin embargo, tanto la información proporcionada por otros programa-

⁸ <http://discuss.develop.com/archives/wa.exe?A2=ind0011&L=DOTNET&P=R32820>

⁹ *Exception Handling in CLU*, Liskov & Snyder.

¹⁰ Bjarne Stroustrup, *The C++ Programming Language*, 3rd Edition (Addison-Wesley, 1997), p. 376.

dores como mi experiencia directa¹¹ con lenguajes que son más dinámicos que estáticos, me han hecho pensar que las mayores ventajas provienen en realidad de:

1. Un modelo unificado de informe de errores basado en excepciones, independientemente de si el programador está obligado por el compilador a tratarlas.
2. Mecanismos de comprobación de tipos, independientemente de cuándo tenía lugar esa comprobación. Es decir, en tanto que se imponga un uso apropiado de los tipos, a menudo no importa si eso sucede en tiempo de compilación o de ejecución.

Además, se puede aumentar muy significativamente la productividad si se reducen las restricciones impuestas al programador en tiempo de compilación. De hecho, se necesita algo de *reflexión* junto con los *genéricos*, para compensar la naturaleza demasiado restrictiva del mecanismo estático de tipos, como podrá ver en una serie de ejemplos a lo largo del libro.

Algunas personas me han dicho que lo que digo constituye una auténtica blasfemia y que al expresar estas ideas mi reputación quedará irremediablemente destruida, la civilización se vendrá abajo y un mayor porcentaje de los proyectos de programación fallarán. La creencia de que el compilador puede ser la salvación de nuestro proyecto, al indicarnos los errores en tiempo de compilación, se basa en evidencias bastante fuertes, pero es todavía más importante que nos demos cuenta de las limitaciones que afectan a lo que el compilador es capaz de hacer. En el suplemento disponible en <http://MindView.net/Books/BetterJava>, se hace un gran hincapié en el valor que tienen los procesos automatizados de construcción de programas y las pruebas de las unidades componentes de un programa, mecanismos ambos que nos ayudan mucho más que el tratar de convertirlo todo en un error sintáctico. Merece la pena recordar que:

*"Un buen lenguaje de programación es aquél que ayuda a los programadores a escribir buenos programas. Ningún lenguaje de programación podrá impedir siempre que sus usuarios escriban malos programas."*¹²

En cualquier caso, la probabilidad de que las excepciones comprobadas sean eliminadas de Java parece muy baja. Sería un cambio de lenguaje demasiado radical y los que se oponen a esa eliminación dentro de Sun parecen tener bastante fuerza. Sun tiene una larga historia (y una norma) de ser siempre compatible en sentido descendente; de hecho, casi todo el software de Sun se ejecuta en todo el hardware Sun, independientemente de lo antiguo que éste sea. Sin embargo, si llega a tener la sensación de que algunas excepciones comprobadas están empezando a ser engorrosas, y especialmente si se encuentra continuamente en la obligación de capturar excepciones para luego no saber qué hacer con ellas, existen ciertas alternativas.

Paso de las excepciones a la consola

En los programas simples, como muchos de los incluidos en este libro, la forma más fácil de preservar las excepciones sin escribir un montón de código consiste en pasárselas a la consola desde `main()`. Por ejemplo, si queremos abrir un archivo para lectura (que es algo que veremos cómo hacer en el Capítulo 18, *E/S*), tenemos que abrir y cerrar un objeto `FileInputStream`, que genera excepciones. En un programa simple, podemos hacer lo siguiente (podrá ver esta técnica utilizada en numerosos lugares de este libro):

```
//: exceptions/MainException.java
import java.io.*;
public class MainException {
    // Pasar todas las excepciones a la consola:
    public static void main(String[] args) throws Exception {
        // Abrir el archivo:
        FileInputStream file =
            new FileInputStream("MainException.java");
        // Usar el archivo...
        // Cerrar el archivo:
        file.close();
    }
}
```

¹¹ Indirectamente con Smalltalk a través de conversaciones con muchos programadores con experiencia en dicho lenguaje, directamente con Python (www.Python.org).

¹² Kees Koster, diseñador del lenguaje CDL, citado por Bertrand Meyer, diseñador del lenguaje Eiffel, [www.elf.com/elf/v1/n1/bm/right/](http://elf.com/elf/v1/n1/bm/right/).

Observe que **main()** es un método que también puede tener una especificación de excepciones, y en este ejemplo el tipo de la excepción es **Exception**, que es la clase raíz de todas las excepciones comprobadas. Pasando la excepción a la consola, nos vemos liberados de la obligación de incluir cláusulas **try-catch** dentro del cuerpo de **main()** (lamentablemente, la E/S de archivo es significativamente más compleja de lo que parece en este ejemplo, así que no se anime en exceso hasta que haya leído el Capítulo 18, *E/S*).

Ejercicio 26: (1) Cambie la cadena de caracteres que especifica el nombre del archivo en **MainException.java** para proporcionar un nombre de archivo que no exista. Ejecute el programa y anote el resultado.

Conversión de las excepciones comprobadas en no comprobadas

Pasar una excepción hacia el exterior desde **main()** resulta cómodo cuando estamos escribiendo programas simples para nuestro propio consumo, pero no resulta útil por regla general. El problema real surge cuando estamos escribiendo el cuerpo de un método normal, y entonces invocamos otro método y nos damos cuenta de lo siguiente: “no tengo ni idea de lo que hacer con esta excepción aquí, pero no quiero perderla ni imprimir simplemente un mensaje vanal”. Gracias al mecanismo de las excepciones encadenadas existe una solución nueva y simple, que consiste en limitarse a “envolver” una excepción comprobada dentro de otra de tipo **RuntimeException** pasándosela al constructor de **RuntimeException**, como en el siguiente ejemplo:

```
try {
    // ... hacer algo útil
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
    throw new RuntimeException(e);
}
```

Esto parece una solución ideal si queremos “desactivar” la excepción comprobada: no la perdemos y no tenemos porque incluirla en la especificación de excepciones de nuestro método; además, debido al encadenamiento de excepciones no perdemos ninguna información relativa a la excepción original.

Esta técnica proporciona la opción de ignorar la excepción y dejarla progresar por la pila de llamadas, sin vernos obligados a escribir cláusulas **try-catch** y/o especificaciones de excepciones. Sin embargo, seguimos pudiendo capturar y tratar la excepción específica, usando **getCause()**, como se muestra a continuación:

```
//: exceptions/TurnOffChecking.java
// "Desactivación" de excepciones comprobadas.
import java.io.*;
import static net.mindview.util.Print.*;

class WrapCheckedException {
    void throwRuntimeException(int type) {
        try {
            switch(type) {
                case 0: throw new FileNotFoundException();
                case 1: throw new IOException();
                case 2: throw new RuntimeException("Where am I?");
                default: return;
            }
        } catch(Exception e) { // Adaptar a no comprobada:
            throw new RuntimeException(e);
        }
    }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
    public static void main(String[] args) {
        WrapCheckedException wce = new WrapCheckedException();
        // Debemos invocar throwRuntimeException() sin un bloque try
```

```

// y dejar que las excepciones de tipo RuntimeException
// salgan del método;
wce.throwRuntimeException(3);
// O podemos elegir capturar las excepciones:
for(int i = 0; i < 4; i++)
    try {
        if(i < 3)
            wce.throwRuntimeException(i);
        else
            throw new SomeOtherException();
    } catch(SomeOtherException e) {
        print("SomeOtherException: " + e);
    } catch(RuntimeException re) {
        try {
            throw re.getCause();
        } catch(FileNotFoundException e) {
            print("FileNotFoundException: " + e);
        } catch(IOException e) {
            print("IOException: " + e);
        } catch(Throwable e) {
            print("Throwable: " + e);
        }
    }
}
/* Output:
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Where am I?
SomeOtherException: SomeOtherException
*///:-

```

WrapCheckedException.throwRuntimeException() contiene código que genera diferentes tipos de excepciones. Éstas se capturan y se envuelven dentro de objetos **RuntimeException**, así que se convierten en la “causa” de dichas excepciones.

En **TurnOffChecking**, podemos ver que es posible invocar **throwRuntimeException()** sin ningún bloque **try** porque el método no genera ninguna excepción comprobada. Sin embargo, cuando estemos listos para capturar las excepciones, seguiremos teniendo la posibilidad de capturar cualquier excepción que queramos poniendo nuestro código dentro de un bloque **try**. Comenzamos capturando todas las excepciones que sabemos explícitamente que pueden emergir del código incluido dentro del bloque **try**; en este caso, se captura primero **SomeOtherException**. Finalmente, se captura **RuntimeException** y se genera con **throw** el resultado de **getCause()** (la excepción envuelta). Esto extrae las excepciones de origen, que pueden ser entonces tratadas en sus propias cláusulas **catch**.

La técnica de envolver una excepción comprobada en otra de tipo **RuntimeException** se utilizará siempre que sea apropiado a lo largo del resto del libro. Otra solución consiste en crear nuestra propia subclase de **RuntimeException**. De esta forma, no es necesario capturarla, pero alguien puede hacerlo si así lo desea.

Ejercicio 27: (1) Modifique el Ejercicio 3 para convertir la excepción en otra de tipo **RuntimeException**.

Ejercicio 28: (1) Modifique el Ejercicio 4 de modo que la clase de excepción personalizada herede de **RuntimeException**, y muestre que el compilador nos permite no incluir el bloque **try**.

Ejercicio 29: (1) Modifique todos los tipos de excepción de **StormyInning.java** de modo que extiendan **RuntimeException**, y muestre que no son necesarias especificaciones de excepción ni bloques **try**. Elimine los comentarios ‘**!!**’ y muestre cómo pueden compilarse los métodos sin especificaciones.

Ejercicio 30: (2) Modifique **Human.java** de modo que las excepciones hereden de **RuntimeException**. Modifique **main()** de modo que se utilice la técnica de **TurnOffChecking.java** para tratar los diferentes tipos de excepciones.

Directrices relativas a las excepciones

Utilice las excepciones para:

1. Tratar los problemas en el nivel apropiado (evite capturar las excepciones a menos que sepa qué hacer con ellas).
2. Corregir el problema e invocar de nuevo el método que causó la excepción.
3. Corregir las cosas y continuar, sin volver a ejecutar el método.
4. Calcular algunos resultados alternativos en lugar de aquellos que se supone que el método debía producir.
5. Hacer lo que se pueda en el contexto actual y regenerar la *misma* excepción; entregándosela a un contexto de nivel superior.
6. Hacer lo que se pueda en el contexto actual y generar una excepción *diferente*, entregándosela a un contexto de nivel superior.
7. Terminar el programa.
8. Simplificar (si el esquema de excepciones utilizado hace que las cosas se vuelvan más complicadas, entonces será muy molesto de utilizar).
9. Hacer que la biblioteca y el programa sean más seguros (esto es una inversión a corto plazo de cara a la depuración y una inversión a largo plazo en lo que respecta a la robustez de la aplicación).

Resumen

Las excepciones son una parte fundamental de la programación Java; no es mucho lo que puede hacerse si no se sabe cómo trabajar con ellas. Por esa razón, hemos decidido introducir las excepciones en este punto del libro; hay muchas bibliotecas (como las de E/S, mencionadas anteriormente) que no pueden emplearse sin tratar las excepciones.

Una de las ventajas del tratamiento de excepciones es que nos permite concentrarnos en un cierto lugar en el problema que estemos tratando de resolver, y tratar con los errores relativos a dicho código en otro lugar. Y, aunque las excepciones se suelen explicar como herramientas que nos permiten *informar* acerca de los errores y *recuperarnos de ellos* en tiempo de ejecución, no es tan claro con cuánta frecuencia se implementa ese aspecto de “recuperación”, como tampoco está muy claro si resulta siempre posible. Mi percepción es que la recuperación es posible en no más del 10 por ciento de los casos, e incluso en esas situaciones sólo consiste en devolver la pila a un estado estable conocido, más que reanudar el procesamiento del programa. Pero, sea esto verdad o no, lo importante es que el valor fundamental de las excepciones radica en la función de “informe de errores”. El hecho de que Java insista en que se informe de todos los errores mediante las excepciones es lo que le proporciona a este lenguaje una gran ventaja respecto a otros lenguajes como C++, que permite informar de los errores de distintas manera o incluso no informar en absoluto. Disponer de un sistema coherente de informe de errores implica que no tenemos ya por qué hacernos la pregunta de “¿se nos está colando algún error por alguna parte?” cada vez que escribamos un fragmento de código (siempre y cuando, no capturemos las excepciones para luego dejarlas sin tratar).

Como podrá ver en futuros capítulos, al permitirnos olvidarnos de esta cuestión (aunque sea generando una excepción de tipo **RuntimeException**), los esfuerzos de diseño e implementación pueden centrarse en otras cuestiones más interesantes y complejas.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Cadenas de caracteres

13

La manipulación de las cadenas de caracteres es probablemente una de las actividades más comunes en la programación.

Esto resulta especialmente cierto en los sistemas web, en los que Java se utiliza ampliamente. En este capítulo, vamos a examinar más en detalle la que constituye, ciertamente, la clase más comúnmente utilizada de todo el lenguaje, **String**, junto con sus utilidades y clases asociadas.

Cadenas de caracteres inmutables

Los objetos de clase **String** son inmutables. Si examina la documentación del JDK referente a la clase **String**, verá que todos los métodos de la clase que parecen modificar una cadena de caracteres, lo que hacen, en realidad, es devolver un objeto **String** completamente nuevo que contiene dicha modificación. El objeto **String** original se deja sin modificar.

Considere el siguiente código:

```
//: strings/Immutable.java
import static net.mindview.util.Print.*;

public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy";
        print(q); // howdy
        String qq = upcase(q);
        print(qq); // HOWDY
        print(q); // howdy
    }
} /* Output:
howdy
HOWDY
howdy
*///:-
```

Cuando se pasa **q** a **upcase()** se trata en realidad de una copia de la referencia a **q**. El objeto al que esta referencia está conectado permanece en una única ubicación física. Las referencias se copian a medida que se las pasa de un sitio a otro.

Examinando la definición de **upcase()**, podemos ver que la referencia que se le pasa tiene un nombre **s**, y que dicha referencia existe sólo mientras que se ejecuta el cuerpo de **upcase()**. Cuando se completa **upcase()**, la referencia local **s** desaparece. **upcase()** devuelve el resultado, que es la cadena original con todos los caracteres en mayúscula. Por supuesto, lo que se devuelve en realidad es una referencia al resultado. Pero lo cierto es que la referencia que se devuelve apunta a un nuevo objeto, dejándose sin modificar el objeto al que apuntaba la referencia **q** original.

Este comportamiento es normalmente el que deseamos. Suponga que escribimos:

```
String s = "asdf";
String x = Immutable.toUpperCase(s);
```

¿Realmente queremos que el método `toUpperCase()` modifique el argumento? Para el lector del código, los argumentos suelen aparecer como fragmentos de información proporcionados al método, no como algo que haya que modificar. Esta garantía es importante, ya que hace que el código sea más fácil de escribir y comprender.

Comparación entre la sobrecarga de '+' y `StringBuilder`

Puesto que los objetos `String` son inmutables, podemos establecer tantos alias como queramos para un objeto `String` concreto. Puesto que un objeto `String` es de sólo lectura, no hay ninguna posibilidad de que una referencia modifique algo que pueda afectar a otras referencias.

La inmutabilidad puede presentar problemas de rendimiento. Un ejemplo claro es el operador '+' que está sobrecargado para los objetos `String`. La palabra "sobrecargado" significa que hay una operación a la que se le ha proporcionado un significado adicional cuando se la usa con una clase concreta (los operadores '+' y '+=' para objetos `String` son los únicos operadores sobrecargados en Java, y el lenguaje no permite que el programador sobrecargue ningún otro operador).¹

El operador '+' nos permite concatenar cadenas de caracteres:

```
//: strings/Concatenation.java

public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
} /* Output:
abcmangodef47
*///:-
```

Queremos tratar de imaginar *cómo sería la forma* en que este mecanismo funciona. El objeto `String` "abc" podría tener un método `append()` que creara un nuevo objeto `String` que contuviera "abc" concatenado con el contenido de `mango`. El nuevo objeto `String` crearía entonces otro objeto `String` que añadiera "def", etc.

Esto podría funcionar, pero requiere la creación de un montón de objetos `String` simplemente para componer esta nueva cadena de caracteres, lo que conduciría a que hubiera varios objetos `String` intermedios a los que habría que aplicar posteriormente los mecanismos de depuración de memoria. Sospecho que los diseñadores de Java intentaron en primer lugar esta solución (lo cual constituye una de las lecciones aplicándose al diseño software: en realidad no sabemos nada acerca de un sistema hasta que lo probamos con código y obtenemos algo que funcione). También sospecho que descubrieron que esta solución presentaba un rendimiento inaceptable.

Para ver lo que sucede en realidad podemos descompilar el código anterior utilizando la herramienta `javap`, que está incluida en el JDK. He aquí la línea de comandos necesaria:

```
javap -c Concatenation
```

El indicador `-c` genera el código intermedio JVM. Después de quitar las partes que no nos interesan y editar un poco los resultados, he aquí el código intermedio relevante:

```
public static void main(java.lang.String[]);
Code:
Stack=2, Locals=3, Args_size=1
0: ldc #2; //String mango
```

¹ C++ permite al programador sobrecargar los operadores a voluntad. Como esto puede ser a menudo bastante complicado (véase el Capítulo 10 de *Thinking in C++*, 2^a Edición, Prentice Hall, 2000), los diseñadores de Java consideraron que era una característica "indeseable" que no había que incluir en Java. Resulta gracioso que al final terminaran ellos mismos recurriendo a la sobrecarga de operadores y, lo que todavía resulta más irónico, se da la circunstancia de que la sobrecarga de operadores resultaría mucho más fácil en Java que en C++. Esto es lo que sucede en Python (véase www.Python.org) y C#, que disponen de mecanismos tanto de depuración de memoria como de sobrecarga sencilla de los operadores.

```

2:    astore_1
3:    new #3; //class StringBuilder
4:    dup
5:    invokespecial #4; //StringBuilder."<init>":()
10:   ldc #5; //String abc
12:   invokevirtual #6; //StringBuilder.append:(String)
15:   aload_1
16:   invokevirtual #6; //StringBuilder.append:(String)
19:   ldc #7; //String def
21:   invokevirtual #6; //StringBuilder.append:(String)
24:   bipush 47
26:   invokevirtual #8; //StringBuilder.append:(I)
29:   invokevirtual #9; //StringBuilder.toString:()
32:   astore_2
33:   getstatic #10; //Field System.out:PrintStream;
36:   aload_2
37:   invokevirtual #11; // PrintStream.println:(String)
40:   return

```

Si tiene experiencia con el lenguaje ensamblador, puede que este código le resulte familiar: las instrucciones como **dup** e **invokevirtual** son los equivalentes en la máquina virtual Java (JVM) al lenguaje ensamblador. Si nunca ha visto lenguaje ensamblador, no se preocupe: lo que hay que observar es que el compilador introduce la clase **java.lang.StringBuilder**. No había ninguna mención de **StringBuilder** en el código fuente, pero el compilador ha decidido utilizarlo por su cuenta, porque es mucho más eficiente.

En este caso, el compilador crea un objeto **StringBuilder** para crear la cadena de caracteres **s**, y llama a **append()** cuatro veces, una para cada uno de los fragmentos. Finalmente, invoca a **toString()** para generar el resultado, que almacena (con **astore_2**) como **s**.

Antes de dar por supuesto que lo que hay que hacer es utilizar cadenas de caracteres por todas partes y que el compilador se encargará de que todo sea eficiente, examinemos un poco más en detalle lo que el compilador está haciendo. He aquí un ejemplo que genera un resultado de tipo **String** de dos maneras: utilizando cadenas de caracteres y realizando la codificación de forma manual con **StringBuilder**:

```

//: strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
    public String implicit(String[] fields) {
        String result = "";
        for(int i = 0; i < fields.length; i++)
            result += fields[i];
        return result;
    }
    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < fields.length; i++)
            result.append(fields[i]);
        return result.toString();
    }
} //:-

```

Ahora, si ejecutamos **javap -c WitherStringBuilder**, podemos ver el código (simplificado) para los dos métodos diferentes. En primer lugar, **implicit()**:

```

public java.lang.String implicit(java.lang.String[]);
Code:
0:   ldc #2; //String
2:   astore_2
3:   iconst_0
4:   istore_3
5:   iload_3

```

```

6:      aload_1
7:      arraylength
8:      if_icmpge 38
11:     new #3; //class StringBuilder
14:     dup
15:     invokespecial #4; // StringBuilder."<init>":()
18:     aload_2
19:     invokevirtual #5; // StringBuilder.append:()
22:     aload_1
23:     iload_3
24:     aaload
25:     invokevirtual #5; // StringBuilder.append:()
28:     invokevirtual #6; // StringBuilder.toString:()
31:     astore_2
32:     iinc 3, 1
35:     goto 5
38:     aload_2
39:     areturn

```

Observe las líneas 8: y 35:, que forman un bucle. La línea 8: realiza una “comparación entera de tipo mayor o igual que” con los operandos de la pila y salta a la línea 38: cuando el bucle se ha terminado. La línea 35: es una instrucción de salto que vuelve al principio del bucle, en la línea 5:. Lo más importante que hay que observar es que la construcción del objeto **StringBuilder** tiene lugar *dentro* de este bucle, lo que quiere decir que obtenemos un nuevo objeto **StringBuilder** cada vez que pasemos a través del bucle.

He aquí el código intermedio correspondiente a **explicit()**:

```

public java.lang.String explicit(java.lang.String[]):
Code:
0:   new #3; //class StringBuilder
3:   dup
4:   invokespecial #4; // StringBuilder."<init>":()
7:   astore_2
8:   iconst_0
9:   istore_3
10:  iload_3
11:  aload_1
12:  arraylength
13:  if_icmpge 30
16:  aload_2
17:  aload_1
18:  iload_3
19:  aaload
20:  invokevirtual #5; // StringBuilder.append:()
23:  pop
24:  iinc 3, 1
27:  goto 10
30:  aload_2
31:  invokevirtual #6; // StringBuilder.toString:()
34:  areturn

```

No sólo es el código de bucle más corto y más simple, sino que además el método sólo crea un único objeto **StringBuilder**. La creación de un objeto **StringBuilder** explícito también nos permite preasignar su tamaño si disponemos de información adicional acerca de lo grande que debe ser, con lo cual no es necesario volver a reasignar constantemente el *buffer*.

Por tanto, cuando creamos un método **toString()**, si las operaciones son lo suficientemente simples como para que el compilador pueda figurarse el sólo cómo hacerlas, podemos generalmente confiar en que el compilador construirá el resultado de una forma razonable. Pero si hay bucles, conviene utilizar explícitamente un objeto **StringBuilder** en el método **toString()**, como se hace a continuación:

```
//: strings/UsingStringBuilder.java
```

```
import java.util.*;  
  
public class UsingStringBuilder {  
    public static Random rand = new Random(47);  
    public String toString() {  
        StringBuilder result = new StringBuilder("[");  
        for(int i = 0; i < 25; i++) {  
            result.append(rand.nextInt(100));  
            result.append(", ");  
        }  
        result.delete(result.length()-2, result.length());  
        result.append("]");  
        return result.toString();  
    }  
    public static void main(String[] args) {  
        UsingStringBuilder usb = new UsingStringBuilder();  
        System.out.println(usb);  
    }  
} /* Output:  
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9, 78, 98, 61, 20, 58, 16, 40,  
11, 22, 4]  
*///:-
```

Observe que cada parte del resultado se añade con una instrucción `append()`. Si tratamos de seguir un atajo y hacer algo como `append(a + ":" + c)`, el compilador saldrá a la palestra y comenzará a construir de nuevo más objetos `StringBuilder`.

Si tiene duda acerca de qué técnica utilizar, siempre puede ejecutar **javap** para comprobar los resultados.

Aunque `StringBuilder` dispone de un conjunto completo de métodos, incluyendo `insert()`, `replace()`, `substring()` e incluso `reverse()`, los que generalmente se usan son `append()` y `toString()`. Observe el uso de `delete()` para eliminar la última coma y el último espacio antes de añadir el corchete de cierre.

StringBuilder fue introducido en Java SE5. Antes de esta versión, Java utilizaba **StringBuffer**, que garantizaba la seguridad en lo que respecta a las hebras de programación (véase el Capítulo 21, *Concurrencia*) y era, por tanto, significativamente más caro en términos de recursos de procesamiento. Por tanto, las operaciones de manejo de caracteres en Java SE5/6 deberían ser más rápidas.

Ejercicio 1: (2) Analice SprinklerSystem.toString() en reusing/SprinklerSystem.java para descubrir si escribir el método `toString()` con un método `StringBuilder` explícito permitiría ahorrar operaciones de creación de objetos `StringBuilder`.

Recursión no intencionada

Puesto que los contenedores estándar Java (al igual que todas las demás clases) heredan en último término de **Object**, todos ellos contienen un método **toString()**. Este método ha sido sustituido para que los contenedores puedan generar una representación de tipo **String** de sí mismos, incluyendo los objetos que almacenan. **ArrayList.toString()**, por ejemplo, recorre los elementos del objeto **ArrayList** y llama a **toString()** para cada uno de ellos:

```
//: strings/ArrayListDisplay.java
import generics.coffee.*;
import java.util.*;

public class ArrayListDisplay {
    public static void main(String[] args) {
        ArrayList<Coffee> coffees = new ArrayList<Coffee>();
        for(Coffee c : new CoffeeGenerator(10))
            coffees.add(c);
        System.out.println(coffees);
    }
} /* Output:
```

```
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve 5,
Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
*/:-
```

Suponga que quisiéramos que el método `toString()` imprimiera la dirección de la clase. Parece que tendría bastante sentido hacer referencia simplemente a `this`:

```
//: strings/InfiniteRecursion.java
// Recursión accidental.
// [RunByHand]
import java.util.*;

public class InfiniteRecursion {
    public String toString() {
        return "InfiniteRecursion address: " + this + "\n";
    }
    public static void main(String[] args) {
        List<InfiniteRecursion> v =
            new ArrayList<InfiniteRecursion>();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} */:-
```

Si creamos un objeto `InfiniteRecursion` y luego lo imprimimos, obtendremos una secuencia muy larga de excepciones. Esto también se produce si colocamos los objetos `InfiniteRecursion` en un contenedor `ArrayList` e imprimimos dicho contenedor como aquí se muestra. Lo que está sucediendo es una conversión automática de tipos para las cadenas de caracteres. Cuando decimos:

```
"InfiniteRecursion address: " + this
```

El compilador ve un objeto `String` seguido de un símbolo '+' y algo que no es un objeto `String`, por lo que trata de convertir `this` a `String`. El compilador realiza esta conversión llamando a `toString()`, que es lo que produce una llamada recursiva.

Si queremos imprimir la dirección del objeto, la solución es llamar al método `toString()` de `Object`, y hace precisamente eso. Por tanto, en lugar de especificar `this`, lo que tendríamos que escribir es `super.toString()`.

Ejercicio 2: (1) Corrija el error de `InfiniteRecursion.java`.

Operaciones con cadenas de caracteres

He aquí algunos de los métodos básicos disponibles para objetos `String`. Los métodos sobrecargados se resumen en una única fila:

| Método | Argumentos, sobrecarga | Uso |
|---------------------------------------|--|---|
| Constructor | Sobrecargado: predeterminado, <code>String</code> , <code>StringBuilder</code> , <code>StringBuffer</code> , matrices <code>char</code> , matrices <code>byte</code> | Creación de objetos <code>String</code> . |
| <code>length()</code> | | Número de caracteres en el objeto <code>String</code> . |
| <code>charAt()</code> | Índice <code>int</code> | El carácter <code>char</code> en una posición dentro del objeto <code>String</code> . |
| <code>getChars(), getBytes()</code> | El principio y el final del que hay que copiar, la matriz a la que hay que copiar, un índice a la matriz de destino. | Copia caracteres o bytes en una matriz externa. |

| Método | Argumentos, sobrecarga | Uso |
|---|---|--|
| <code>toCharArray()</code> | | Genera un <code>char[]</code> que contiene los caracteres contenidos en el objeto <code>String</code> . |
| <code>equals(), equalsIgnoreCase()</code> | Un objeto <code>String</code> con el que comparar. | Una comprobación de igualdad de los contenidos de dos objetos <code>String</code> . |
| <code>compareTo()</code> | Un objeto <code>String</code> con el que comparar. | El resultado es negativo, cero o positivo dependiendo de la ordenación lexicográfica del objeto <code>String</code> y del argumento. ¡Las mayúsculas y minúsculas no son iguales! |
| <code>contains()</code> | Un objeto <code>CharSequence</code> que buscar. | El resultado es <code>true</code> si el argumento está contenido en el objeto <code>String</code> . |
| <code>contentEquals()</code> | Un objeto <code>CharSequence</code> o <code>StringBuffer</code> con el que comparar.. | El resultado es <code>true</code> si hay una correspondencia exacta con el argumento. |
| <code>equalsIgnoreCase()</code> | Un objeto <code>String</code> con el que comparar.. | El resultado es <code>true</code> si los contenidos son iguales, sin tener en cuenta la diferencia entre mayúsculas y minúsculas. |
| <code>regionMatches()</code> | Desplazamiento dentro de este objeto <code>String</code> , el otro objeto <code>String</code> junto con su desplazamiento y la longitud para comparar. La sobrecarga añade la posibilidad de "ignorar mayúsculas y minúsculas". | Un resultado booleano que indica si la región se corresponde. |
| <code>startsWith()</code> | Objeto <code>String</code> con el que puede comenzar. La sobrecarga añade el desplazamiento dentro de un argumento. | Un resultado booleano que indica si el objeto <code>String</code> comienza con el argumento. |
| <code>endsWith()</code> | Objeto <code>String</code> que puede ser sufijo de este objeto <code>String</code> . | Un resultado booleano que indica si el argumento es un sufijo. |
| <code>indexOf(), lastIndexOf()</code> | Sobrecargado: <code>char, char</code> e índice de inicio, <code>String, String</code> e índice de inicio. | Devuelve <code>-1</code> si el argumento no se encuentra dentro de este objeto <code>String</code> ; en caso contrario, devuelve el índice donde empieza el argumento. <code>lastIndexOf()</code> busca hacia atrás el final. |
| <code>substring()</code> (también <code>subSequence()</code>) | Sobrecargado: índice de inicio; índice de inicio + índice de fin. | Devuelve un nuevo objeto <code>String</code> que contiene el conjunto de caracteres especificado. |
| <code>concat()</code> | El objeto <code>String</code> que haya que concatenar. | Devuelve un nuevo objeto <code>String</code> que contiene los caracteres del objeto <code>String</code> original seguidos de los caracteres del argumento. |
| <code>replace()</code> | El antiguo carácter que hay que buscar, el nuevo carácter con el que hay que reemplazarlo. También puede reemplazar un objeto <code>CharSequence</code> con otro objeto <code>CharSequence</code> . | Devuelve un nuevo objeto <code>String</code> en el que se han efectuado las sustituciones. Utiliza el antiguo objeto <code>String</code> si no se encuentra ninguna correspondencia. |
| <code>toLowerCase()</code> <code>toUpperCase()</code> | | Devuelve un nuevo objeto <code>String</code> en el que se habrán cambiado todas las letras a minúsculas o mayúsculas. Utiliza el antiguo objeto <code>String</code> si no es necesario realizar ningún cambio. |
| <code>trim()</code> | | Devuelve un nuevo objeto <code>String</code> eliminando los espacios en blanco de ambos extremos. Utiliza el antiguo objeto <code>String</code> si no es necesario realizar ningún cambio. |

| Método | Argumentos, sobrecarga | Uso |
|-------------------------|---|--|
| <code>valueOf()</code> | Sobrecargado: <code>Object, char[], char[]</code> y desplazamiento y recuento, <code>boolean, char, int, long, float, double</code> . | Devuelve un objeto <code>String</code> que contiene una representación en forma de caracteres del argumento. |
| <code>intern()</code> | | Produce una y sólo una referencia a un objeto <code>String</code> por cada secuencia de caracteres distinta. |

Puede ver que todo método de `String` devuelve, cuidadosamente, un nuevo objeto `String` cuando es necesario cambiar los contenidos. Observe también que, si los contenidos no necesitan modificarse, el método se limita a devolver una referencia al objeto `String` original. Esto ahorra espacio de almacenamiento y recursos de procesamiento.

Los métodos `String` en los que están implicadas *expresiones regulares* se explican más adelante en este capítulo.

Formateo de la salida

Una de las características más esperadas que ha sido finalmente incorporada a Java SE5 es el formateo de la salida al estilo de la instrucción `printf()` de C. No sólo permite esto simplificar el código de salida, sino que también proporciona a los desarrolladores Java una gran capacidad de control sobre el formato a la alineación de esa salida.²

`printf()`

La función `printf()` de C no ensambla las cadenas de caracteres en la forma en que lo hace Java, sino que toma una única *cadena de formato* e inserta valores en ella, efectuando el formateo a medida que lo hace. En lugar de utilizar el operador sobrecargado '+' (que el lenguaje C no sobrecarga) para concatenar el texto entrecomillado y las variables, `printf()` utiliza marcadores especiales para indicar dónde deben insertarse los datos. Los argumentos que se insertan en la cadena de formato se indican a continuación mediante una lista separada por comas. Por ejemplo:

```
printf("Row 1: [%d %f]\n", x, y);
```

En tiempo de ejecución, el valor de `x` se inserta en `%d` y el valor de `y` se inserta en `%f`. Estos contenedores se denominan *especificadores de formato* y, además de decimos dónde se debe insertar el valor, también nos informan del tipo de variable que hay que insertar y de cómo hay que formatearla. Por ejemplo, el marcador '`%d`' anterior dice que `x` es un entero, mientras que '`%f`' dice que `y` es un valor de punto flotante (`float` o `double`).

`System.out.format()`

Java SE5 introdujo el método `format()`, disponible para los objetos `PrintStream` o `PrintWriter` (de los que hablaremos más en detalle en el Capítulo 18, *E/S*), entre los que se incluye `System.out`. El método `format()` está modelado basándose en la función `printf()` de C. Existe incluso un método `printf()` que pueden utilizar aquellos que se sientan nostálgicos, que simplemente invoca `format()`. He aquí un ejemplo simple:

```
// strings/SimpleFormat.java

public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // A la antigua usanza:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // A la nueva usanza:
        System.out.format("Row 1: [%d %f]\n", x, y);
    }
}
```

² Mark Welsh ha ayudado en la creación de esta sección, así como en la sección "Análisis de la entrada".

```

        System.out.printf("Row 1: [%d %f]\n", x, y);
    }
} /* Output:
Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
*///:-

```

Puede ver que **format()** y **printf()** son equivalentes. En ambos casos, hay una única cadena de formato, seguida de un argumento para cada especificador de formato.

La clase Formatter

Toda la nueva funcionalidad de formateo de Java es gestionada por la clase **Formatter** del paquete **java.util**. Podemos considerar **Formatter** como una especie de traductor que convierte la cadena de formato y los datos al resultado deseado. Cuando se crea un objeto **Formatter**, se le indica a dónde queremos que se manden los resultados, pasando esa información al constructor:

```

//: strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
    private String name;
    private Formatter f;
    public Turtle(String name, Formatter f) {
        this.name = name;
        this.f = f;
    }
    public void move(int x, int y) {
        f.format("%s The Turtle is at (%d,%d)\n", name, x, y);
    }
    public static void main(String[] args) {
        PrintStream outAlias = System.out;
        Turtle tommy = new Turtle("Tommy",
            new Formatter(System.out));
        Turtle terry = new Turtle("Terry",
            new Formatter(outAlias));
        tommy.move(0,0);
        terry.move(4,8);
        tommy.move(3,4);
        terry.move(2,5);
        tommy.move(3,3);
        terry.move(3,3);
    }
} /* Output:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*///:-

```

Toda la salida representada por **tommy** va a **System.out** mientras que la representa por **terry** va a un alias de **System.out**. El constructor está sobrecargado para admitir diversas ubicaciones de salida, pero las más útiles son **PrintStream** (como en el ejemplo), **OutputStream** y **File**. Veremos más detalles sobre esto en el Capítulo 18, *Entrada/salida*.

Ejercicio 3: (1) Modifique **Turtle.java** de modo que envíe toda la salida a **System.err**.

El ejemplo anterior utiliza un nuevo especificador de formato, '%s'. Este marcador indica un argumento de tipo **String** y es un ejemplo del tipo más simple de especificador de formato: uno que sólo tiene un tipo de conversión.

Especificadores de formato

Para controlar el espaciado y la alineación cuando se insertan los datos, hacen falta especificadores de formato más elaborados. He aquí la sintaxis más general:

```
%[índice_argumento] [indicadores] [anchura] [.precisión] conversión
```

A menudo, será necesario controlar el tamaño mínimo de un campo. Esto puede realizarse especificando una *anchura*. El objeto **Formatter** garantiza que un campo tenga al menos una anchura de un cierto número de caracteres, rellenándolo con espacios en caso necesario. De manera predeterminada, los datos se justifican a la derecha, pero esto puede cambiarse incluyendo '-' en la sección de indicadores.

Lo contrario de la *anchura* es la *precisión*, que se utiliza para especificar un máximo. A diferencia de la *anchura*, que es aplicable a todos los tipos de conversión de datos y se comporta de la misma manera con cada uno de ellos, *precisión* tiene un significado distinto para los diferentes tipos. Para las cadenas de caracteres, la *precisión* especifica el número máximo de caracteres del objeto **String** que hay que imprimir. Para los números en coma flotante, *precisión* especifica el número de posiciones decimales que hay que mostrar (el valor predeterminado es 6), efectuando un redondeo si hay más dígitos o añadiendo más ceros al final si hay pocos. Puesto que los enteros no tienen parte fraccionaria, *precisión* no es aplicable a ellos y se generará una excepción si se utiliza el argumento de precisión con un tipo de conversión entero.

El siguiente ejemplo utiliza especificadores de formato para imprimir una factura de la compra:

```
//: strings/Receipt.java
import java.util.*;

public class Receipt {
    private double total = 0;
    private Formatter f = new Formatter(System.out);
    public void printTitle() {
        f.format("%-15s %5s %10s\n", "Item", "Qty", "Price");
        f.format("%-15s %5s %10s\n", "----", "----", "-----");
    }
    public void print(String name, int qty, double price) {
        f.format("%-15.15s %5d %10.2f\n", name, qty, price);
        total += price;
    }
    public void printTotal() {
        f.format("%-15s %5s %10.2f\n", "Tax", "", total*0.06);
        f.format("%-15s %5s %10s\n", "----", "----", "-----");
        f.format("%-15s %5s %10.2f\n", "Total", "", total * 1.06);
    }
    public static void main(String[] args) {
        Receipt receipt = new Receipt();
        receipt.printTitle();
        receipt.print("Jack's Magic Beans", 4, 4.25);
        receipt.print("Princess Peas", 3, 5.1);
        receipt.print("Three Bears Porridge", 1, 14.29);
        receipt.printTotal();
    }
} /* Output:
   Item          Qty      Price
   ----         ----     -----
   Jack's Magic Be    4      4.25
   Princess Peas    3      5.10
   Three Bears Por    1     14.29
   Tax                  1.42
```

```

Total: 25.06
* * * *

```

Como puede ver, el objeto **Formatter** proporciona un considerable grado de control entre el espaciado y la alineación, con una notación bastante concisa. Aquí, las cadenas de formato se copian simplemente con el fin de producir el espaciado apropiado.

Ejercicio 4: (3) Modifique **Receipt.java** para que todas las anchuras estén controladas por un único conjunto de valores constantes. El objetivo es poder cambiar fácilmente una anchura modificando un único valor en un determinado lugar.

Conversiones posibles con Formatter

Estas son las conversiones con las que más frecuentemente nos podremos encontrar:

| Caracteres de conversión | |
|--------------------------|---|
| d | Entero (como decimal) |
| c | Carácter Unicode |
| b | Valor booleano |
| s | Cadena de caracteres |
| f | Coma flotante (como decimal) |
| e | Coma flotantes (en notación científica) |
| x | Entero (como hexadecimal) |
| h | Código <i>hash</i> (as hexadecimal) |
| % | Literal "%" |

He aquí un ejemplo que muestra estas conversiones en acción:

```

// strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
    public static void main(String[] args) {
        Formatter f = new Formatter(System.out);

        char u = 'a';
        System.out.println("u = " + u);
        f.format("s: %s\n", u);
        // f.format("d: %d\n", u);
        f.format("c: %c\n", u);
        f.format("b: %b\n", u);
        // f.format("f: %f\n", u);
        // f.format("e: %e\n", u);
        // f.format("x: %x\n", u);
        f.format("h: %h\n", u);

        int v = 121;
        System.out.println("v = " + v);
    }
}

```

```

f.format("d: %d\n", v);
f.format("c: %c\n", v);
f.format("b: %b\n", v);
f.format("s: %s\n", v);
// f.format("f: %f\n", v);
// f.format("e: %e\n", v);
f.format("x: %x\n", v);
f.format("h: %h\n", v);

BigInteger w = new BigInteger("5000000000000000");
System.out.println(
    "w = new BigInteger(\"5000000000000000\")");
f.format("d: %d\n", w);
// f.format("c: %c\n", w);
f.format("b: %b\n", w);
f.format("s: %s\n", w);
// f.format("f: %f\n", w);
// f.format("e: %e\n", w);
f.format("x: %x\n", w);
f.format("h: %h\n", w);

double x = 179.543;
System.out.println("x = 179.543");
// f.format("d: %d\n", x);
// f.format("c: %c\n", x);
f.format("b: %b\n", x);
f.format("s: %s\n", x);
f.format("f: %f\n", x);
f.format("e: %e\n", x);
// f.format("x: %x\n", x);
f.format("h: %h\n", x);

Conversion y = new Conversion();
System.out.println("y = new Conversion()");
// f.format("d: %d\n", y);
// f.format("c: %c\n", y);
f.format("b: %b\n", y);
f.format("s: %s\n", y);
// f.format("f: %f\n", y);
// f.format("e: %e\n", y);
// f.format("x: %x\n", y);
f.format("h: %h\n", y);

boolean z = false;
System.out.println("z = false");
// f.format("d: %d\n", z);
// f.format("c: %c\n", z);
f.format("b: %b\n", z);
f.format("s: %s\n", z);
// f.format("f: %f\n", z);
// f.format("e: %e\n", z);
// f.format("x: %x\n", z);
f.format("h: %h\n", z);
}

} /* Output: (Sample)
u = 'a'
s: a
c: a
b: true

```

```

h: 61
v = 121
d: 121
c: Y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("5000000000000000")
d: 5000000000000000
b: true
s: 5000000000000000
x: 2d79883d2000
h: 8842ala7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
h: true
s: Conversion@9cab16
h: 9cab16
z = false
b: false
s: false
h: 4d5
*///:-
```

Las líneas comentadas muestran conversiones que no son válidas para ese tipo de variables concreto, ejecutarlas harían que se generaría una excepción.

Observe que la conversión ‘b’ funciona para cada una de las variables anteriores. Aunque es válida para cualquier tipo de argumento, puede que no se comporte como cabría esperar. Para las primitivas **boolean** o los objetos de tipo **boolean**, el resultado será **true** o **false**, según corresponda. Sin embargo, para cualquier otro argumento, siempre que el tipo de argumento no sea **null**, el resultado será siempre **true**. Incluso el valor numérico de cero, que es sinónimo de **false** en muchos lenguajes (incluyendo C), generará **true**, de modo que tenga cuidado cuando utilice esta conversión con tipos no booleanos.

Existen otros tipos de conversión y otras opciones de especificador de formato más extraños. Puede consultarlos en la documentación del JDK para la clase **Formatter**.

Ejercicio 5: (5) Para cada uno de los tipos básicos de conversión de la tabla anterior, escriba la expresión de formateo más compleja posible. Es decir, utilice todos los posibles especificadores de formato disponibles para dicho tipo de conversión.

String.format()

Java SE5 también ha tomado prestado de C la idea de **sprintf()**, que es un método que se utiliza para crear cadenas de caracteres. **String.format()** es un método estático que toma los mismos argumentos que el método **format()** de **Formatter** pero devuelve un objeto **String**. Puede resultar útil cuando sólo se necesita invocar una vez a **format()**:

```

//: strings/DatabaseException.java

public class DatabaseException extends Exception {
    public DatabaseException(int transactionID, int queryID,
        String message) {
        super(String.format("(t%d, q%d) %s", transactionID,
            queryID, message));
    }
}
```

```

public static void main(String[] args) {
    try {
        throw new DatabaseException(3, 7, "Write failed");
    } catch(Exception e) {
        System.out.println(e);
    }
}
} /* Output:
DatabaseException: (t3, q7) Write failed
*///:-

```

Entre bastidores, todo lo que **String.format()** hace es instanciar el objeto **Formatter** y pasarle los argumentos que hayamos proporcionado, pero utilizar este método puede resultar a menudo más claro y más fácil que hacerlo de forma manual.

Una herramienta de volcado hexadecimal

Como segundo ejemplo, a menudo nos interesa examinar los bytes que componen un archivo binario utilizando formato hexadecimal. He aquí una pequeña utilidad que muestra una matriz binaria de bytes en un formato hexadecimal legible, utilizando **String.format()**:

```

//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;

public class Hex {
    public static String format(byte[] data) {
        StringBuilder result = new StringBuilder();
        int n = 0;
        for(byte b : data) {
            if(n % 16 == 0)
                result.append(String.format("%05X: ", n));
            result.append(String.format("%02X ", b));
            n++;
            if(n % 16 == 0) result.append("\n");
        }
        result.append("\n");
        return result.toString();
    }
    public static void main(String[] args) throws Exception {
        if(args.length == 0)
            // Comprobar mostrando este archivo de clase:
            System.out.println(
                format(BinaryFile.read("Hex.class")));
        else
            System.out.println(
                format(BinaryFile.read(new File(args[0]))));
    }
} /* Output: (Sample)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*///:-

```

Para abrir y leer el archivo binario, este programa presenta otra utilidad que se presentará en el Capítulo 18, *Entrada/salida: net.mindview.util.BinaryFile*. El método **read()** devuelve el archivo completo como una matriz de tipo **byte**.

Ejercicio 6: (2) Cree una clase que contenga campos **int**, **long**, **float** y **double**. Cree un método **toString()** para esta clase que utilice **String.format()**, y demuestre que la clase funciona correctamente.

Expresiones regulares

Las *expresiones regulares* han sido durante mucho tiempo parte integrante de las utilidades estándar Unix como sed y awk, y de lenguajes como Python y Perl (algunas personas piensan incluso que las expresiones regulares son la principal razón del éxito de Perl). Las herramientas de manipulación de cadenas de caracteres estaban anteriormente delegadas a las clases **String**, **StringBuffer** y **StringTokenizer** de Java, que disponían de funcionalidades relativamente simples si las comparamos con las expresiones regulares.

Las expresiones regulares son herramientas de procesamiento de texto potentes y flexibles. Nos permiten especificar, mediante programas, patrones complejos de texto que pueden buscarse en una cadena de entrada. Una vez descubiertos estos patrones, podemos reaccionar a su aparición de la forma que deseemos. Aunque la sintaxis de las expresiones regulares puede resultar intimidante al principio, proporcionan un lenguaje compacto y dinámico que puede emplearse para resolver todo tipo de tareas de procesamiento, comparación, selección, edición y verificación de cadenas de una forma general.

Fundamentos básicos

Una expresión regular es una forma de describir cadenas de caracteres en términos generales, de modo que podemos decir: “Si una cadena de caracteres contiene estos elementos, entonces se corresponde con lo que estoy buscando”. Por ejemplo, para decir que un número puede estar o no precedido por un signo menos, escribimos el signo menos seguido de un signo de interrogación de la forma siguiente:

-?

Para describir un entero, diremos que está compuesto de uno o más dígitos. En las expresiones regulares, un dígito se describe mediante ‘\d’. Si tiene experiencia con las expresiones regulares en otros lenguajes, observará inmediatamente la diferencia en la forma de gestionar las barras inclinadas. En otros lenguajes, ‘\W’ significa: “Quiero insertar una barra inclinada a la izquierda normal y corriente (literal) en la expresión regular. No le asigne ningún significado especial”. En Java, ‘\W’ significa: “Estoy insertando una barra inclinada de expresión regular, por lo que el siguiente carácter tiene un significado especial”. Por ejemplo, si queremos indicar un dígito, la cadena de la expresión regular será ‘\d’. Si deseamos insertar una barra inclinada literal, tendremos que escribir ‘\\W’. Sin embargo, elementos tales como de nueva linea y de tabulación utilizan una barra inclinada simple: ‘\n\t’.

Para indicar “una o más apariciones de la expresión precedente”, se utiliza un símbolo ‘+’. Por tanto, para decir “posiblemente un signo menos seguido de uno o más dígitos”, escribiríamos:

-?\d+

La forma más simple de utilizar las expresiones regulares consiste en utilizar la funcionalidad incluida dentro de la clase **String**. Por ejemplo, podemos comprobar si un objeto **String** se corresponde con la expresión regular anterior:

```
//: strings/IntegerMatch.java

public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-?\d+"));
        System.out.println("5678".matches("-?\d+"));
        System.out.println("+911".matches("-?\d+"));
        System.out.println("+911".matches("( - | + ) ? \d+"));
    }
} /* Output:
true
true
false
true
****:+
```

Las primeras dos expresiones se corresponden, pero la tercera comienza con un '+', que es un número legítimo pero que no se ajusta a la expresión regular. Por tanto, necesitamos una forma de decir "puede comenzar con un '+' o un '-'". En las expresiones regulares, los paréntesis tienen el efecto de agrupar una expresión, y la barra vertical '|' significa OR (disyunción). Por tanto,

`(-|\\+)?`

quiere decir que esta parte de la cadena de caracteres puede ser un '-' o un '+' o nada (debido al '?'). Puesto que el carácter '+' tiene un significado especial en las expresiones regulares, es necesario introducir la secuencia de escape '\W' para que aparezca como un carácter normal dentro de la expresión.

Una herramienta útil de expresiones regulares incorporada en `String` es `split()`, que significa "partir esta cadena de caracteres justo donde se produzcan las correspondencias con la expresión regular indicada".

```
//: strings/Splitting.java
import java.util.*;

public class Splitting {
    public static String knights =
        "Then, when you have found the shrubbery, you must " +
        "cut down the mightiest tree in the forest... " +
        "with... a herring!";
    public static void split(String regex) {
        System.out.println(
            Arrays.toString(knights.split(regex)));
    }
    public static void main(String[] args) {
        split(" "); // No tiene porqué contener caracteres regex
        split("\W+"); // Caracteres no pertenecientes a una palabra
        split("n\W+"); // 'n' seguida de caracteres no
                        // pertenecientes a una palabra
    }
} /* Output:
[Then,, when, you, have, found, the, shrubbery,, you, must, cut, down, the, mightiest,
tree, in, the, forest..., with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you, must, cut, down, the, mightiest,
tree, in, the, forest, with, a, herring]
[The, whe, you have found the shrubbery, you must cut dow, the mightiest tree i, the
forest... with... a herring!]
*///:-
```

En primer lugar, observe que puede utilizar caracteres normales como expresiones regulares, una expresión regular no tiene porqué contener caracteres especiales, como podemos ver en la primera llamada a `split()`, que simplemente efectúa la partición de acuerdo con los espacios en blanco.

La segunda y tercera llamadas a `split()` utilizan '\W', que representa caracteres que no pertenezcan a palabras (la versión en minúsculas, '\w', representa un carácter perteneciente a una palabra); podrá ver que los signos de puntuación han sido eliminados en el segundo caso. La tercera llamada a `split()` dice, "la letra n seguida de uno o más caracteres que no pertenezcan a palabras". Podrá ver que los patrones de división no aparecen en el resultado.

Una versión sobrecargada de `String.split()` nos permite limitar el número de divisiones que hayan de producirse.

La última de las herramientas de expresiones regulares incorporada en `String` es la de sustitución. Podemos sustituir la primera aparición o todas ellas:

```
//: strings/Replacing.java
import static net.mindview.util.Print.*;

public class Replacing {
    static String s = Splitting.knights;
    public static void main(String[] args) {
        print(s.replaceFirst("f\\w+", "located"));
    }
}
```

```

        print(s.replaceAll("shrubbery|tree|herring", "banana"));
    }
} /* Output:
Then, when you have located the shrubbery, you must cut down the mightiest tree in the
forest... with... a herring!
Then, when you have found the banana, you must cut down the mightiest banana in the
forest... with... a banana!
*///:-

```

La primera expresión se corresponde con la letra **f** seguida de uno o más caracteres de palabras (observe que el carácter **w** está en minúscula esta vez). Sólo sustituye la primera correspondencia que encuentra, por lo que la palabra “*found*” ha sido sustituida por la palabra “*located*”.

La segunda expresión se corresponde con cualquiera de las tres palabras separadas por las barras verticales que representan la operación OR, y sustituye todas las correspondencias que encuentra.

Más adelante veremos que las expresiones regulares que no son de tipo **String** disponen de herramientas de sustitución más potentes; por ejemplo, se pueden invocar métodos para llevar a cabo las sustituciones. Las expresiones regulares que no son de tipo **String** también son significativamente más eficientes cuando hace falta utilizar la expresión regular más de una vez.

Ejercicio 7: (5) Utilizando la documentación de **java.util.regex.Pattern** como referencia, escriba y pruebe una expresión regular de prueba que compruebe una frase para ver si comienza con una letra mayúscula y termina con un punto.

Ejercicio 8: (2) Divida la cadena **Splitting.knights** por las palabras “the” o “you”.

Ejercicio 9: (4) Utilizando la documentación de **java.util.regex.Pattern** como referencia, sustituya todas las vocales de **Splitting.knights** por guiones bajos.

Creación de expresiones regulares

Podemos comenzar a aprender expresiones regulares con un subconjunto de las estructuras posibles. En la documentación del JDK correspondiente a la clase **Pattern** de **java.util.regex** podrá encontrar la lista completa de las estructuras que pueden emplearse para construir las expresiones regulares.

| Caracteres | |
|---------------|---|
| B | El carácter específico B |
| \hh | Carácter con el valor hexadecimal 0xhh |
| \uhhhh | El carácter Unicode con la representación hexadecimal 0xuhhh |
| \t | Tabulador |
| \n | Nueva linea |
| \r | Retorno de carro |
| \f | Avance de página |
| \e | Escape |

La potencia de las expresiones regulares comienza a hacerse patente cuando se definen clases de caracteres. He aquí algunas formas típicas de crear clases de caracteres, junto con algunas clases predefinidas:

| Clases de caracteres | |
|----------------------|--|
| . | Cualquier carácter |
| [abc] | Cualquiera de los caracteres a, b o c (lo mismo que a b c) |
| [^abc] | Cualquier carácter excepto a, b y c (negación) |
| [a-zA-Z] | Cualquier carácter de la a a la z o A a la Z (rango) |
| [abc hij] | Cualquiera de a,b,c,h,i,j (lo mismo que a b c h i j) (unión) |
| [a-z&&[hij]] | Puede ser h, i o j (intersección) |
| \s | Un carácter de espaciado (espacio, tabulador, nueva linea, avance de página, retorno de carro) |
| \S | Un carácter que no sea de espaciado ([^\s]) |
| \d | Un dígito numérico [0-9] |
| \D | Un carácter que no sea un dígito [^0-9] |
| \w | Un carácter de palabra [a-zA-Z_0-9] |
| \W | Un carácter que no sea de palabra [^\w] |

Lo que se muestra aquí es sólo un ejemplo; consultando la página de documentación del JDK correspondiente a `java.util.regex.Pattern` podrá conocer todos los posibles patrones de expresiones regulares.

| Operadores lógicos | |
|--------------------|---|
| XY | X seguido de Y |
| X Y | X o Y |
| (X) | Un grupo de captura. Puede referirse posteriormente al <i>i</i> -ésimo grupo capturado en la expresión mediante \i. |

| Localizadores de contorno | |
|---------------------------|------------------------------------|
| ^ | Comienzo de linea |
| \$ | Fin de linea |
| \b | Frontera de palabra |
| \B | Frontera de no palabra |
| \G | Fin de la correspondencia anterior |

Por ejemplo, cada una de las siguientes expresiones permite localizar la secuencia de caracteres "Rudolph":

```
//: strings/Rudolph.java

public class Rudolph {
    public static void main(String[] args) {
        for(String pattern : new String[] { "Rudolph",
```

```

        "[rR]udolph", "[rR][aeiou][a-z]*", "R.*" })
System.out.println("Rudolph".matches(pattern));
}
/* Output:
true
true
true
true
*/

```

Por supuesto, el objetivo no debe ser crear la expresión regular más complicada sino la que sea más simple y baste para realizar la tarea que tengamos entre manos. Una vez que comience a escribir expresiones regulares, podrá ver cómo a menudo conviene referirse a los ejemplos de código escritos anteriormente, para facilitar la escritura de nuevas expresiones regulares.

Cuantificadores

Un *cuantificador* describe la forma en que un patrón absorbe el texto de entrada:

- *Avaricioso*: los cuantificadores son avariciosos a menos que se los modifique de alguna manera. Un expresión avariciosa trata de encontrar el máximo número posible de correspondencias para el patrón indicado. Una causa bastante común de problemas consiste en suponer que el patrón sólo se corresponderá con el primer grupo de caracteres, cuando lo cierto es que se trata de un patrón avaricioso y continuará procesando texto hasta que haya logrado establecer una correspondencia con la cadena de caracteres más larga posible.
- *Reluctante*: especificado con un signo de interrogación, este cuantificador hace que la correspondencia se establezca con el número mínimo de caracteres necesario para satisfacer el patrón. También se denomina *perezoso*, *de correspondencia mínima* o *no avaricioso*.
- *Posesivo*: en la actualidad, este tipo de cuantificador sólo está disponible en Java (no en otros lenguajes) y es más avanzado, por lo que es posible que no lo utilice al principio. A medida que se aplica una expresión regular a una cadena de caracteres, la expresión genera múltiples estados para poder retroceder si la correspondencia falla. Los cuantificadores posesivos no conservan dichos estados intermedios, evitando así el retroceso. Pueden utilizarse para impedir que una expresión regular quede fuera de control y también para hacer que se ejecute de manera más eficiente.

| Avaricioso | Reluctante | Posesivo | Correspondencia con |
|----------------|------------------|-----------------|--------------------------------------|
| X? | X?? | X?+ | X, una o ninguna |
| X* | X*? | X*+ | X, cero o más |
| X ⁺ | X ⁺ ? | X ⁺⁺ | X, una o más |
| X{n} | X{n}? | X{n}+ | X, exactamente n veces |
| X{n,} | X{n,}? | X{n,}+ | X, al menos n veces |
| X{n,m} | X{n,m}? | X{n,m}+ | X, al menos n pero no más de m veces |

Recuerde que la expresión 'X' necesitará a menudo encerrarse entre paréntesis para que funcione de la forma deseada. Por ejemplo:

abc+

podría parecer que se debería corresponder con la secuencia 'abc' una o más veces, y si la aplicamos a la cadena de entrada 'abcabcabc', obtendremos de hecho tres correspondencias. Sin embargo, lo que la expresión dice *en realidad* es: "localiza 'ab' seguido de una o más apariciones de 'c'". Para buscar correspondencias con la cadena completa 'abc' una o más veces, debemos decir:

```
(abc) +
```

Resulta bastante fácil equivocarse al utilizar expresiones regulares; se trata de un lenguaje completamente ortogonal a Java, que funciona sobre éste y que presenta diferencias con el lenguaje de programación.

CharSequence

La interfaz denominada **CharSequence** establece una definición generalizada de una secuencia de caracteres abstraída de las clase **CharBuffer**, **String**, **StringBuffer** o **StringBuilder**:

```
interface CharSequence {
    charAt(int i);
    length();
    subSequence(int start, int end);
    toString();
}
```

Dichas clases implementan esta interfaz. Muchas operaciones con expresiones regulares toman argumentos de tipo **CharSequence**.

Pattern y Matcher

En general, lo que se hace es compilar objetos de expresión regular en lugar de emplear las utilidades **String**, que son bastante limitadas. Para ello, importamos **java.util.regex**, y luego compilamos una expresión regular utilizando el método **static Pattern.compile()**. Esto genera un objeto **Pattern** basado en su argumento **String**. Para utilizar el objeto **Pattern**, lo que se hace es invocar el método **matcher()**, pasándole la cadena de caracteres que queremos buscar. El método **matcher()** genera un objeto **Matcher**, que tiene un conjunto de operaciones de entre las cuales podemos elegir (puede consultar todas las operaciones en la documentación del JDK correspondiente a **.util.regex.Matcher**). Por ejemplo, el método **replaceAll()** sustituye todas las correspondencias por el argumento que se proporcione.

Vamos a ver un primer ejemplo: la clase siguiente puede utilizarse para probar expresiones regulares con una cadena de entrada. El primer argumento de la línea de comandos es la cadena de entrada en la que hay que buscar las correspondencias, seguida de una o más expresiones regulares que haya que aplicar a la entrada. En Unix/Linux, las expresiones regulares deben estar entrecomilladas en la línea de comandos. Este programa puede resultar útil para probar expresiones regulares mientras las construimos con el fin de comprobar que esas expresiones establecen las correspondencias deseadas.

```
//: strings/TestRegularExpression.java
// Permite probar con facilidad expresiones regulares.
// {Args: abcabcabcdefabc "abc+" "(abc)+" "(abc){2,}"}
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            print("Usage:\njava TestRegularExpression " +
                  "characterSequence regularExpression+");
            System.exit(0);
        }
        print("Input: \"" + args[0] + "\"");
        for(String arg : args) {
            print("Regular expression: \"" + arg + "\"");
            Pattern p = Pattern.compile(arg);
            Matcher m = p.matcher(args[0]);
            while(m.find()) {
                print("Match \"" + m.group() + "\" at positions " +
                      m.start() + "-" + (m.end() - 1));
            }
        }
    }
}
```

```

} /* Output:
Input: "abcabcabcdefabc"
Regular expression: "abcabcdefabc"
Match "abcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
Regular expression: "(abc){2,}"
Match "abcabcabc" at positions 0-8
*/

```

Un objeto **Pattern** representa la versión compilada de una expresión regular. Como hemos visto en el ejemplo anterior, podemos utilizar el método **matcher()** y la cadena de entrada para generar un objeto **Matcher** a partir del objeto **Pattern** compilado. **Pattern** también tiene un método estático:

```
static boolean matches(String regex, CharSequence input)
```

para comprobar si **regex** se corresponde con el objeto **input** de tipo **CharSequence** utilizado como entrada, y un método **split()** que genera una matriz de tipo **String** después de descomponer la entrada según las correspondencias establecidas con la expresión regular **regex**.

Podemos generar un objeto **Matcher** invocando **Pattern.matcher()** con la cadena de entrada como argumento. Después el objeto **Matcher** se utiliza para acceder a los resultados, utilizando métodos para evaluar si se establecen o no diferentes tipos de correspondencias:

```

boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)

```

El método **matches()** tendrá éxito si el patrón se corresponde con la cadena de entrada completa, mientras que **lookingAt()** tendrá éxito si la cadena de entrada, comenzando por el principio, permite establecer una correspondencia con el patrón.

Ejercicio 10: (2) Para la frase “Java now has expresiones regulares” evalúe si las siguientes expresiones permitirán localizar la correspondencia:

```

^Java
\breg.*
n.w\s+h(a|i)s
s?
s*
s+
s{4}
s{1}.
s{0,3}

```

Ejercicio 11: (2) Aplique la expresión regular

```

(?i) ((^ [aeiou]) | (\s+ [aeiou])) \w+? [aeiou] \b
a
"Arline ate eight apples and one orange while Anita hadn't any"

```

find()

Matcher.find() puede utilizarse para descubrir múltiples correspondencias de patrón en el objeto **CharSequence** al cual se aplique. Por ejemplo:

```

//: strings/Finding.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Finding {
    public static void main(String[] args) {
        Matcher m = Pattern.compile("\\w+")
            .matcher("Evening is full of the linnet's wings");
        while(m.find())
            printnb(m.group() + " ");
        print();
        int i = 0;
        while(m.find(i)) {
            printnb(m.group() + " ");
            i++;
        }
    }
} /* Output:
Evening is full of the linnet's wings
Evening vening ening ning ing ng g is is s full full ull ll l of of
f the the he e linnet linnet innet nnet net et t s s wings wings ings ings gs s
*///:-
```

El patrón “\w+” divide la entrada en palabras. `find()` es como un iterador, que se desplaza hacia adelante a través de la cadena de caracteres de entrada. Sin embargo, la segunda versión de `find()` puede aceptar un argumento entero que le dice cuál es la posición del carácter en el que debe comenzar la búsqueda; esta versión reinicializa la posición de búsqueda con el valor de su argumento, como puede ver analizando la salida.

Grupos

Los grupos son expresiones regulares delimitadas por paréntesis y a las que luego se puede hacer referencia utilizando su número de grupo. El grupo 0 indica la expresión completa, el grupo 1 es el primer grupo entre paréntesis, etc. Por tanto, en

`A(B(C))D`

existen tres grupos: el grupo 0 es **ABCD**, el grupo 1 es **BC** y el grupo 2 es **C**.

El objeto **Matcher** dispone de métodos para proporcionarnos información acerca de los grupos:

public int groupCount() devuelve el número de grupos que hay en el patrón. El grupo 0 no se incluye dentro de este recuento.

public String group() devuelve el grupo 0 (la correspondencia completa) de la operación anterior de establecimiento de correspondencias (por ejemplo, `find()`).

public String group(int i) devuelve el número de grupo indicado dentro de la operación de establecimiento de correspondencias anterior. Si esa operación ha tenido éxito, pero el grupo especificado no se corresponde con ninguna parte de la cadena de entrada, devuelve el valor **null**.

public int start(int group) devuelve el índice de inicio del grupo encontrado en la operación anterior de establecimiento de correspondencias.

public int end(int group) devuelve el índice del último carácter, más uno, del grupo encontrado en la anterior operación de establecimiento de correspondencias.

He aquí un ejemplo:

```

//: strings/Groups.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Groups {
    static public final String POEM =

```

```

    "Twas brillig, and the slithy toves\n" +
    "Did gyre and gimble in the wabe.\n" +
    "All mimsy were the borogoves,\n" +
    "And the mome raths outgrabe.\n\n" +
    "Beware the Jabberwock, my son,\n" +
    "The jaws that bite, the claws that catch.\n" +
    "Beware the Jubjub bird, and shun!\n" +
    "The frumious Bandersnatch.";
public static void main(String[] args) {
    Matcher m =
        Pattern.compile("(?m) ((\\S+)\\s+((\\S+)\\s+(\\S+))$)")
            .matcher(POEM);
    while(m.find()) {
        for(int j = 0; j <= m.groupCount(); j++)
            printnb("[" + m.group(j) + "]");
        print();
    }
}
} /* Output:
[the slithy toves] [the] [slithy toves] [slithy] [toves]
[in the wabe.] [in] [the wabe.] [the] [wabe.]
[were the borogoves,] [were] [the borogoves,] [the] [borogoves,]
[mome raths outgrabe.] [mome] [raths outgrabe.] [raths] [outgrabe.]
[Jabberwock, my son,] [Jabberwock,] [my son,] [my] [son,]
[claws that catch.] [claws] [that catch.] [that] [catch.]
[bird, and shun] [bird,] [and shun] [and] [shun]
[The frumious Bandersnatch.] [The] [frumious Bandersnatch.] [frumious] [Bandersnatch.]
*///:-
```

Este poema es la primera parte de “Jabberwocky”, de Lewis Carroll extraído del libro *A través del espejo*. Puede ver que el patrón de expresión regular tiene una serie de grupos entre paréntesis, compuestos de cualquier número de caracteres que no sea de espaciado ('\S+') seguido de cualquier número de caracteres de espaciado ('\s+'). El objetivo es capturar las tres últimas palabras de cada línea, el final de una línea está delimitado por '\$'. Sin embargo, el comportamiento normal consiste en hacer corresponder '\$' con el final de la secuencia de entrada completa, por lo que es necesario decir explícitamente a la expresión regular que preste atención a los caracteres de nueva línea desde dentro de la entrada. Esto se consigue con el indicador de patrones '(?m)' al principio de la secuencia (los indicadores de patrones los veremos enseguida).

Ejercicio 12: (5) Modifique Groups.java para contar todas las palabras que no empiecen con una letra mayúscula.

start() y end()

Después de una operación de establecimiento de correspondencias que haya permitido encontrar al menos una correspondencia, **start()** devuelve el índice de inicio de la correspondencia anterior, mientras que **end()** devuelve el índice del último carácter de la correspondencia más uno. Al invocar **start()** o **end()** después de una operación de localización de correspondencias que no haya tenido éxito (o antes de intentar una operación de localización de correspondencias), se genera la excepción **IllegalStateException**. El siguiente programa también ilustra los métodos **matches()** y **lookingAt()**:³

```

//: strings/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class StartEnd {
    public static String input =
        "As long as there is injustice, whenever a\n" +
        "Targathian baby cries out, wherever a distress\n" +
        "signal sounds among the stars ... We'll be there.\n" +
        "This fine ship, and this fine crew ...\n" +
        "Never give up! Never surrender!";
}
```

³ El texto indicado es una cita de uno de los discursos del Comandante Taggart en *Galaxy Quest*.

```

private static class Display {
    private boolean regexPrinted = false;
    private String regex;
    Display(String regex) { this.regex = regex; }
    void display(String message) {
        if(!regexPrinted) {
            print(regex);
            regexPrinted = true;
        }
        print(message);
    }
}
static void examine(String s, String regex) {
    Display d = new Display(regex);
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while(m.find())
        d.display("find() '" + m.group() + "' start = " + m.start() + " end = " + m.end());
    if(mlookingAt()) // No reset() necessary
        d.display("lookingAt() start = " + m.start() + " end = " + m.end());
    if(m.matches()) // No reset() necessary
        d.display("matches() start = " + m.start() + " end = " + m.end());
}
public static void main(String[] args) {
    for(String in : input.split("\n")) {
        print("input : " + in);
        for(String regex : new String[]{"\\w*ere\\w*", "\\w*ever", "T\\w+", "Never,*?!"})
            examine(in, regex);
    }
}
/* Output:
input : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' start = 11 end = 16
\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' start = 27 end = 35
\w*ever
find() 'wherever' start = 27 end = 35
T\w+
find() 'Targathian' start = 0 end = 10
lookingAt() start = 0 end = 10
input : signal sounds among the stars ... We'll be there.
\w*ere\w*
find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+
find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20

```

```

lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*///:-

```

Observe que **find()** permite localizar la expresión regular en cualquier lugar de la entrada, mientras que **lookingAt()** y **matches()** sólo tienen éxito en la búsqueda si la expresión regular se corresponde desde el principio de la entrada. Mientras que **matches()** sólo tiene éxito en la búsqueda si *toda* la entrada se corresponde con expresión regular, **lookingAt()**⁴ tiene éxito en la búsqueda aunque sólo se corresponda la expresión regular con la primera parte de la entrada.

Ejercicio 13: (2) Modifique **StartEnd.java** para que utilice **Groups.POEM** como entrada, pero siga produciendo resultados positivos para **find()**, **lookingAt()** y **matches()**.

Indicadores de Pattern

Hay un método **compile()** alternativo que acepta indicadores que afectan al comportamiento de búsqueda de correspondencias:

```
Pattern pattern.compile(String regex, int flag)
```

donde **flag** puede ser una de las siguientes constantes de la clase **Pattern**:

| Indicador de compilación | Efecto |
|--|---|
| Pattern.CANON_EQ (? <i>i</i>) | Se considera que dos caracteres se corresponden si y sólo si sus descomposiciones canónicas completas lo hacen. Por ejemplo, la expresión '\u003F' se corresponderá con la cadena '?' cuando se especifique este indicador. De manera predeterminada, la búsqueda de correspondencias no tiene en cuenta la equivalencia canónica. |
| Pattern.CASE_INSENSITIVE (? <i>i</i>) | Por omisión, la búsqueda de correspondencias sin distinción de mayúsculas y minúsculas presupone que sólo se están utilizando caracteres del conjunto de caracteres US-ASCII. Este indicador permite establecer una correspondencia con el patrón sin tener en cuenta mayúsculas o minúsculas. Puede habilitarse la búsqueda de correspondencias Unicode sin distinción de mayúsculas y minúsculas especificando el indicador UNICODE_CASE en conjunción con este indicador. |
| Pattern.COMMENTS (? <i>x</i>) | En este modo, se ignoran los caracteres de espacioado, y también se ignoran los comentarios incrustados que comienzan con # hasta el final de la linea. También puede habilitarse el modo de líneas Unix mediante la expresión de indicador incrustado. |
| Pattern.DOTALL (? <i>s</i>) | En este modo, la expresión '.' se corresponde con cualquier carácter, incluyendo el terminador de línea. Por omisión, la expresión '.' no se corresponde con los terminadores de línea. |
| Pattern.MULTILINE (? <i>m</i>) | En el modo multilinea, las expresiones '^' y '\$' se corresponden con el principio y el final de una linea, respectivamente. '^' también se corresponde con el principio de la cadena de entrada y '\$' lo hace con el final de la cadena de entrada. De manera predeterminada, estas expresiones sólo se corresponden con el principio y el final de la cadena de entrada completa. |
| Pattern.UNICODE_CASE (? <i>u</i>) | La correspondencia sin distinción de mayúsculas y minúsculas, si está habilitada por el indicador CASE_INSENSITIVE , se realiza de manera coherente con el estándar Unicode. De manera predeterminada, la correspondencia sin distinción de mayúsculas y minúsculas presupone que sólo se están buscando correspondencias con caracteres del conjunto de caracteres US-ASCII. |
| Pattern.UNIX_LINES (? <i>d</i>) | En este modo, sólo se reconoce el terminador de linea '\n' en el comportamiento de '.', '^' y '\$'. |

⁴ No sé por qué dieron este nombre a dicho método ni a qué refiere ese nombre. Pero resulta reconfortante saber que quienquiera que sea que inventa esos nombres de métodos tan poco intuitivos continúa empleado en Sun y que su aparente política de no revisar los diseños de código sigue estando vigente. Perdón por el sarcasmo, pero es que este tipo de cosas empiezan a cansar después de unos cuantos años.

De especial utilidad entre todos estos indicadores son **Pattern.CASE_INSENSITIVE**, **Pattern.MULTILINE** y **Pattern.COMMENTS** (que resulta útil para mejorar la claridad y/o con propósitos de documentación). Observe que el comportamiento de la mayor parte de los indicadores puede obtenerse también insertando en la expresión regular los caracteres entre paréntesis que se muestran en los indicadores de la tabla, justo antes del lugar donde se quiera que ese modo tenga efecto.

También podemos combinar el efecto de estos y otros indicadores mediante una operación "OR" ("|"):

```
//: strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("^java",
            Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(
            "java has regex\nJava has regex\n" +
            "JAVA has pretty good regular expressions\n" +
            "Regular expressions are in Java");
        while(m.find())
            System.out.println(m.group());
    }
} /* Output:
java
Java
JAVA
*///:-
```

Esto crea un patrón que se corresponderá con las líneas que comiencen por "java," "Java," "JAVA," etc., y que intentará buscar una correspondencia con cada línea que forme parte de un conjunto multilinea (correspondencias que comienzan al principio de la secuencia de caracteres y a continuación de cada terminador de línea contenido dentro de la secuencia de caracteres). Observe que el método **group()** sólo devuelve la porción con la que se ha establecido la correspondencia.

split()

split() divide una cadena de caracteres de entrada en una matriz de objetos **String**, utilizando como delimitador la expresión regular.

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

Ésta es una forma cómoda de descomponer el texto de entrada utilizando una frontera divisoria común:

```
//: strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SplitDemo {
    public static void main(String[] args) {
        String input =
            "This!!unusual use!!of exclamation!!points";
        print(Arrays.toString(
            Pattern.compile("!!").split(input)));
        // Hacer sólo las tres primeras:
        print(Arrays.toString(
            Pattern.compile("!!").split(input, 3)));
    }
} /* Output:
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*///:-
```

La segunda forma de **split()** limita el número de divisiones que pueden tener lugar.

Ejercicio 14: (1) Escriba de nuevo **SplitDemo** utilizando **String.split()**.

Operaciones de sustitución

Las expresiones regulares resultan especialmente útiles para sustituir texto. He aquí los métodos disponibles:

replaceFirst(String replacement) sustituye por **replacement** la primera parte que se corresponde de la cadena de caracteres.

replaceAll(String replacement) sustituye por **replacement** todas aquellas partes que se correspondan en la cadena de caracteres de entrada.

appendReplacement(StringBuffer sbuf, String replacement) realiza sustituciones paso a paso en **sbuff**, en lugar de sustituir sólo la primera o todas ellas, como sucede con **replaceFirst()** y **replaceAll()**, respectivamente. Éste es un método *muy* importante, porque permite invocar métodos y realizar otros tipos de procesamiento para generar la cadena de sustitución (**replacement** (**replaceFirst()** y **replaceAll()** sólo pueden utilizar cadenas de caracteres fijas para la sustitución). Con este método, podemos separar los grupos mediante programa y crear potentes rutinas de sustitución.

appendTail(StringBuffer sbuff, String replacement) se invoca después de una o más invocaciones del método **appendReplacement()** para copiar el resto de la cadena de caracteres de entrada.

He aquí un ejemplo que muestra el uso de todas las operaciones de sustitución. El bloque de texto comentado al principio del programa es extraído y procesado con expresiones regulares para usarlo como entrada en el resto del ejemplo:

```
//: strings/TheReplacements.java
import java.util.regex.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

/*! Here's a block of text to use as input to
the regular expression matcher. Note that we'll
first extract the block of text by looking for
the special delimiters, then process the
extracted block. !*/

public class TheReplacements {
    public static void main(String[] args) throws Exception {
        String s = TextFile.read("TheReplacements.java");
        // Establecer correspondencia con el bloque de texto con
        // comentarios especiales mostrado anteriormente:
        Matcher mInput =
            Pattern.compile("//\\*!(.*!)\\*/", Pattern.DOTALL)
                .matcher(s);
        if(mInput.find())
            s = mInput.group(1); // Capturado por paréntesis
        // Sustituir dos o más espacios por un único espacio:
        s = s.replaceAll(" {2,}", " ");
        // Eliminar dos o más espacios al principio de cada
        // línea. Hay que habilitar el modo MULTILÍNEA:
        s = s.replaceAll("(?m)^ +", "");
        print(s);
        s = s.replaceFirst("[aeiou]", "(VOWEL1)");
        StringBuffer sbuff = new StringBuffer();
        Pattern p = Pattern.compile("[aeiou]");
        Matcher m = p.matcher(s);
        // Procesar la información de localización a medida
        // que se realizan las sustituciones:
        while(m.find())
```

```

        m.appendReplacement(sbuf, m.group().toUpperCase());
        // Insertar el resto del texto:
        m.appendTail(sbuf);
        print(sbuf);
    }
} /* Output:
Here's a block of text to use as input to
the regular expression matcher. Note that we'll
first extract the block of text by looking for
the special delimiters, then process the
extracted block.
H(VOWEL1)rE's A bLoCk Of tExt tO UsE As InpUt tO
thE rEgUlAr EPrEssIOn mAtchEr. NOtE thAt wE'll
fIrSt ExtrAct tHe bLoCk Of tExt by lOOkInG fOr
thE spEcIAL dElImItErS, thEn prOcess thE
ExtrActEd bLoCk.
*///:-
```

El archivo se abre y se lee utilizando la clase **TextFile** de la biblioteca **net.mindview.util** (el código correspondiente se mostrará en el Capítulo 18, *E/S*). El método estático **read()** lee el archivo completo y lo devuelve como objeto **String**. **mInput** se crea para corresponderse con todo el texto (observe los paréntesis de agrupamiento) comprendido entre `'/*!'` y `'!*/'`. Después, los conjuntos de más de dos espacios seguidos se reducen a un único espacio y se eliminan todos los espacios situados al principio de cada línea (para hacer esto en todas las líneas y no sólo al principio de la cadena de entrada, es necesario habilitar el modo multilínea). Estas dos sustituciones se realizan con el método equivalente (pero más cómodo, en este caso) **replaceAll()** que forma parte de **String**. Observe que, como cada sustitución sólo se emplea una vez en el programa, no se produce ningún coste adicional por hacerlo así en lugar de precompilar la operación en forma de un objeto **Pattern**.

replaceFirst() sólo sustituye la primera correspondencia que encuentre. Además, las cadenas de sustitución en **replaceFirst()** y **replaceAll()** son simplemente literales, por lo que si queremos realizar algún procesamiento en cada sustitución, no nos sirven de ninguna ayuda. En dicho caso, tendremos que utilizar **appendReplacement()**, que nos permite escribir cualquier código que queramos para realizar la sustitución. En el ejemplo anterior, se selecciona y procesa un grupo (en este caso, poniendo en mayúscula la vocal encontrada por la expresión regular) a medida que se construye el objeto **sbuf** resultante. Normalmente, lo que haremos será recorrer toda la entrada y hacer todas las sustituciones y luego invocar a **appendTail()**, pero si queremos simular **replaceFirst()** (o una operación de “sustitución de n apariciones”), basta con hacer la sustitución una vez y luego invocar **appendTail()** para insertar el resto de la información en **sbuf**.

appendReplacement() también nos permite hacer referencia directamente en la cadena de sustitución a los grupos capturados mediante la notación `"$g"`, donde `'g'` es el número de grupo. Sin embargo, este método sólo sirve para tareas de procesamiento simples y no nos varía los resultados deseados en el programa anterior.

reset()

Podemos aplicar un objeto **Matcher** existente a una nueva secuencia de caracteres utilizando los métodos **reset()**:

```

//: strings/Resetting.java
import java.util.regex.*;

public class Resetting {
    public static void main(String[] args) throws Exception {
        Matcher m = Pattern.compile("[frb] [aiu] [gx]")
            .matcher("fix the rug with bags");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.print(m.group() + " ");
    }
} /* Output:
```

```
fix rug bag
fix rig rag
*///:-
```

`reset()` sin ningún argumento hace que `Matcher` se sitúe al principio de la secuencia actual.

Expresiones regulares y E/S en Java

La mayoría de los ejemplos vistos hasta ahora mostraban la aplicación de las expresiones regulares a cadenas de caracteres estáticas. El siguiente ejemplo muestra una forma de aplicar expresiones regulares a la búsqueda de correspondencias en un archivo. Inspirada en la utilidad *grep* de Unix, **JGrep.java** toma dos argumentos: un nombre de archivo y la expresión regular con la que se quiere buscar correspondencias. La salida muestra cada línea en la que se ha detectado una correspondencia y la posición o posiciones de las correspondencias dentro de la línea:

```
//: strings/JGrep.java
// Una versión muy simple del programa "grep".
// {Args: JGrep.java "\b[Ssct]\w+"}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("Usage: java JGrep file regex");
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Iterar a través de las líneas del archivo de entrada:
        int index = 0;
        Matcher m = p.matcher("");
        for(String line : new TextFile(args[0])) {
            m.reset(line);
            while(m.find())
                System.out.println(index++ + ": " +
                    m.group() + ":" + m.start());
        }
    }
} /* Output: (Sample)
0: strings: 4
1: simple: 10
2: the: 28
3: Ssct: 26
4: class: 7
5: static: 9
6: String: 26
7: throws: 41
8: System: 6
9: System: 6
10: compile: 24
11: through: 15
12: the: 23
13: the: 36
14: String: 8
15: System: 8
16: start: 31
*///:-
```

El archivo se abre como un objeto `net.mindview.util.TextFile` (del que hablaremos en el Capítulo 18, *E/S*), que lee las líneas del archivo en un contenedor tipo `ArrayList`. Esto significa que podemos utilizar la sintaxis `foreach` para iterar a través de las líneas almacenadas en el objeto `TextFile`.

Aunque es posible crear un nuevo objeto **Matcher** dentro del bucle **for**, resulta ligeramente más óptimo crear un objeto vacío **Matcher** fuera del bucle y utilizar el método **reset()** para asignar cada línea de la entrada al objeto **Matcher**. El resultado se analiza con **find()**.

Los argumentos de prueba abren el archivo **JGrep.java** para leerlo como entrada y buscan las palabras que comiencen por **[Ssct]**.

Puede aprender mucho más acerca de las expresiones regulares en *Mastering Regular Expressions, 2^a Edición*, por Jeffrey E. F. Friedl (O'Reilly, 2002). Hay también numerosas introducciones a las expresiones regulares en Internet, y también se puede encontrar a menudo información útil en la documentación de lenguajes tales como Perl y Python.

Ejercicio 15: (5) Modifique **JGrep.java** para aceptar indicadores como argumentos (por ejemplo, **Pattern.CASE_INSENSITIVE**, **Pattern.MULTILINE**).

Ejercicio 16: (5) Modifique **JGrep.java** para aceptar un nombre de directorio o un nombre de archivo como argumento (si se proporciona un directorio, la búsqueda debe extenderse a todos los archivos de directorio). Consejo: puede generar una lista de nombres de archivo con:

```
File[] files = new File(".").listFiles();
```

Ejercicio 17: (8) Escriba un programa que lea un archivo de código fuente Java (tendrá que proporcionar el nombre del archivo en la línea de comandos) y muestre todos los comentarios.

Ejercicio 18: (8) Escriba un programa que lea un archivo de código fuente Java (tendrá que proporcionar el nombre del archivo en la línea de comandos) y muestre todos los literales de cadena presentes en el código.

Ejercicio 19: (8) Utilizando los resultados de los dos ejercicios anteriores, escriba un programa que examine el código fuente Java y genere todos los nombres de clases utilizados en un programa concreto.

Análisis de la entrada

Hasta ahora, resultaba relativamente complicado leer datos de un archivo de texto legible o desde la entrada estándar. La solución usual consiste en leer una línea de texto, extraer los elementos y luego utilizar los diversos métodos de análisis sintáctico de **Integer**, **Double**, etc., para analizar los datos:

```
//: strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
    public static BufferedReader input = new BufferedReader(
        new StringReader("Sir Robin of Camelot\n22 1.61803"));
    public static void main(String[] args) {
        try {
            System.out.println("What is your name?");
            String name = input.readLine();
            System.out.println(name);
            System.out.println(
                "How old are you? What is your favorite double?");
            System.out.println("(input: <ages> <double>)");
            String numbers = input.readLine();
            System.out.println(numbers);
            String[] numArray = numbers.split(" ");
            int age = Integer.parseInt(numArray[0]);
            double favorite = Double.parseDouble(numArray[1]);
            System.out.format("Hi %s.\n", name);
            System.out.format("In 5 years you will be %d.\n",
                age + 5);
            System.out.format("My favorite double is %f.\n",
                favorite / 2);
        } catch(IOException e) {
```

```

        System.err.println("I/O exception");
    }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:-
```

El campo **input** utiliza clases de **java.io**, que no vamos a presentar oficialmente hasta el Capítulo 18, E/S. Un objeto **StringReader** transforma un dato de tipo **String** en un flujo de datos legible y este objeto se emplea para crear un objeto **BufferedReader** porque **BufferedReader** tiene un método **readLine()**. El resultado es que el objeto **input** puede leerse de linea en linea, como si fuera la entrada estándar procedente de la consola.

readLine() se utiliza para obtener la cadena de caracteres correspondiente a cada linea de entrada. Resulta bastante cómodo de utilizar cuando queremos obtener un dato de entrada por cada linea de datos, pero si hay dos valores de entrada que se encuentran en una misma linea, las cosas empiezan a complicarse: la linea debe dividirse para poder analizar cada dato de entrada por separado. Aquí, la división tiene lugar al crear **numArray**, pero observe que el método **split()** fue introducido en J2SE1.4, por lo que en las versiones anteriores era necesario hacer alguna otra cosa.

La clase **Scanner**, añadida en Java SE5, elimina buena parte de la complejidad relacionada con el análisis de la entrada:

```

//: strings/BetterRead.java
import java.util.*;

public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(SimpleRead.input);
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println(
            "How old are you? What is your favorite double?");
        System.out.println("(input: <age> <double>)");
        int age = stdin.nextInt();
        double favorite = stdin.nextDouble();
        System.out.println(age);
        System.out.println(favorite);
        System.out.format("Hi %s.\n", name);
        System.out.format("In 5 years you will be %d.\n",
            age + 5);
        System.out.format("My favorite double is %f.",
            favorite / 2);
    }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:-
```

El constructor de **Scanner** admite casi cualquier tipo de objeto de entrada, incluido el objeto **File** (que también veremos en el Capítulo 18, *E/S*), un objeto **InputStream**, un objeto **String** o, como en este caso un objeto **Readable**, que es una interfaz introducida en Java SE5 para describir “algo que dispone de un método `read()`”. El objeto **BufferedReader** del ejemplo anterior cae dentro de esta categoría.

Con **Scanner**, los pasos de entrada, extracción de elementos y análisis sintáctico están implementados mediante diferentes tipos de métodos de “continuación”. Un método `next()` devuelve el siguiente elemento **String** y existen elementos similares para todos los tipos primitivos (excepto **char**), así como para **BigDecimal** y **BigInteger**. Todos estos métodos se *bloquean*, lo que significa que sólo terminan después de que haya un elemento de datos completo disponible como entrada. También existen los métodos “`hasNext`” correspondientes que devuelven **true** si el siguiente elemento de entrada es del tipo correcto.

Una interesante diferencia entre los dos ejemplos anteriores es la falta de un bloque `try` para las excepciones **IOException** en **BetterRead.java**. Una de las suposiciones hechas por el objeto **Scanner** es que **IOException** marca el final de la entrada, por lo que estas excepciones son capturadas y hechas desaparecer por el objeto **Scanner**. Sin embargo, la excepción más reciente está disponible a través del método `IOException()`, por lo que podemos examinarla en caso necesario.

Ejercicio 20: (2) Cree una clase que contenga campos **int**, **long**, **float**, **double** y **String**. Cree un constructor para esta clase que tenga un único argumento **String**, y analice dicha cadena de caracteres para llenar los diferentes campos. Añada un método `toString()` y demuestre que la clase funciona correctamente.

Delimitadores de Scanner

De manera predeterminada, un objeto **Scanner** divide los elementos de entrada según los caracteres de espaciado, pero también podemos especificar nuestro patrón delimitador en forma de una expresión regular:

```
//: strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\\\\s*,\\\\s*");
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
} /* Output:
12
42
78
99
42
*///:-
```

Este ejemplo utiliza comas (rodeadas por cantidades arbitrarias de espacios) como los delimitadores a la hora de leer la cadena de caracteres dada. Esta misma técnica puede utilizarse para leer desde archivos delimitados por comas. Además del método `useDelimiter()`, que sirve para fijar el patrón de delimitación, existe también otro método denominado `delimiter()`, que devuelve el objeto **Pattern** que esté siendo actualmente utilizado como delimitador.

Análisis con expresiones regulares

Además de analizar en busca de tipos predefinidos, también podemos realizar el análisis empleando nuestros propios patrones definidos por el usuario, lo cual resulta muy útil a la hora de analizar datos más complejos. El siguiente ejemplo analiza un archivo de registro generado por un cortafuegos en busca de datos que revelen potenciales amenazas.

```
//: strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;
```

```

public class ThreatAnalyzer {
    static String threatData =
        "58.27.82.161@02/10/2005\n" +
        "204.45.234.40@02/11/2005\n" +
        "58.27.82.161@02/11/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "[Next log section with different data format]";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(threatData);
        String pattern = "(\\d+\\.\\d+\\.\\d+\\.\\d+)@(" +
            "(\\d{2})/(\\d{2})/(\\d{4})";
        while(scanner.hasNext(pattern)) {
            scanner.next(pattern);
            MatchResult match = scanner.match();
            String ip = match.group(1);
            String date = match.group(2);
            System.out.format("Threat on %s from %s\n", date, ip);
        }
    }
} /* Output:
Threat on 02/10/2005 from 58.27.82.161
Threat on 02/11/2005 from 204.45.234.40
Threat on 02/11/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
*/

```

Cuando utilizamos `next()` con un patrón específico, se trata de hacer corresponder dicho patrón con el siguiente elemento de entrada. El resultado es devuelto por el método `match()` y, como podemos ver en el ejemplo, el mecanismo funciona exactamente igual que las búsquedas con expresiones regulares que hemos visto anteriormente.

Existe un problema a la hora de analizar con expresiones regulares. El patrón se hace corresponder únicamente con el siguiente elemento de entrada, por lo que si el patrón contiene un delimitador nunca se encontrará una correspondencia.

StringTokenizer

Antes de que hubiera disponibles expresiones regulares (en J2SE1.4) o la clase `Scanner` (en Java SE5), la forma de dividir una cadena en sus partes componentes era extraer los elementos con `StringTokenizer`. Pero ahora resulta mucho más fácil y más sencillo hacer lo mismo con expresiones regulares o la clase `Scanner`. He aquí una comparación simple de `StringTokenizer` con las otras dos técnicas:

```

//: strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input = "But I'm not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
        System.out.println(Arrays.toString(input.split(" ")));
        Scanner scanner = new Scanner(input);
        while(scanner.hasNext())
            System.out.print(scanner.next() + " ");
    }
} /* Output:
But I'm not dead yet! I feel happy!
[Ljava.lang.String;@1c33333
But I'm not dead yet! I feel happy!
*/

```

```
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*///:-
```

Con las expresiones regulares o con los objetos **Scanner**, también puede dividirse una cadena en sus partes componentes utilizando patrones más complejos, cosa que resulta difícil de realizar con **StringTokenizer**. Podemos decir, sin mucho temor a equivocarnos que **StringTokenizer** está obsoleto.

Resumen

En el pasado, el soporte que Java tenía para la manipulación de cadenas de caracteres era rudimentario, pero en las ediciones recientes del lenguaje hemos podido ver cómo se han ido adoptando funcionalidades más sofisticadas, copiadas de otros lenguajes. Actualmente, el soporte para cadenas de caracteres es razonablemente completo, aunque en ocasiones es necesario prestar algo de atención a los detalles de eficiencia, como por ejemplo haciendo un uso apropiado de **StringBuilder**.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Información de tipos

14

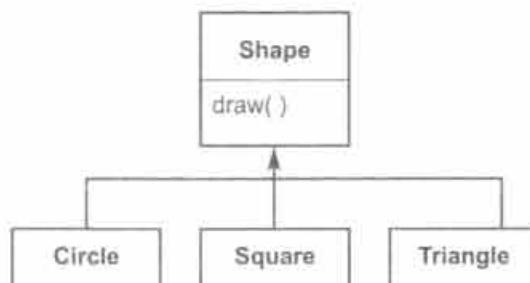
El mecanismo RTTI (*Runtime type information*, información de tipos en tiempo de ejecución) nos permite averiguar y utilizar la información acerca de los tipos de los datos mientras un programa se está ejecutando.

Este mecanismo nos libera de la restricción de efectuar operaciones orientadas a tipos únicamente en tiempo de compilación, y permite construir algunos programas de gran potencia. La necesidad de RTTI permite descubrir una gran cantidad de cuestiones del diseño orientado a objetos de gran interés (y a menudo bastante intrigantes) y hace que surjan cuestiones fundamentales acerca de cuál tiene que ser la manera de estructurar los programas.

En este capítulo se examinan las formas en que Java permite averiguar la información acerca de los objetos y las clases en tiempo de ejecución. Estos mecanismos adoptan dos formas diferentes: RTTI “tradicional”, que supone que tenemos todos los tipos disponibles en tiempo de compilación y el mecanismos de *reflexión* que permite descubrir y utilizar la información sobre clases exclusivamente en tiempo de ejecución.

La necesidad de RTTI

Considere el ya familiar ejemplo de una jerarquía de clases que utiliza los mecanismos de polimorfismo. El tipo genérico es la clase base **Shape** y los tipos derivados específicos son **Circle**, **Square** y **Triangle**:



Se trata de un diagrama de jerarquía de clases típico, con la clase base en la parte superior y las clases derivadas distribuyéndose en sentido descendente. El objetivo normal de la programación orientada a objetos es que el código manipule referencias a tipo base (en este caso, **Shape**), de modo que si decidimos ampliar el programa añadiendo una nueva clase (como por ejemplo **Rhomboide**, derivada de **Shape**), el grueso del código no se vea afectado. En este ejemplo, el método de acoplamiento dinámico de la interfaz **Shape** es **draw()**, por lo que la intención es que el programador de clientes invoque a **draw()** a través de una referencia genérica a **Shape**. En todas las clases derivadas, el método **draw()** se sustituye y, puesto que es un método con acoplamiento dinámico, obtendremos el comportamiento adecuado aún cuando se invoque al método sustituto a través de una referencia genérica a **Shape**. Esto es, ni más ni menos, polimorfismo.

Por tanto, lo que hacemos en general es crear un objeto específico (**Circle**, **Square** o **Triangle**), generalizarlo a **Shape** (olvidando el tipo específico de objeto) y utilizar esa referencia anónima a **Shape** en el resto del programa.

Podemos codificar la jerarquía **Shape** de la siguiente manera:

```

//: typeinfo/Shapes.java
import java.util.*;

abstract class Shape {
    void draw() { System.out.println(this + ".draw()"); }
    abstract public String toString();
}

class Circle extends Shape {
    public String toString() { return "Circle"; }
}

class Square extends Shape {
    public String toString() { return "Square"; }
}

class Triangle extends Shape {
    public String toString() { return "Triangle"; }
}

public class Shapes {
    public static void main(String[] args) {
        List<Shape> shapeList = Arrays.asList(
            new Circle(), new Square(), new Triangle()
        );
        for(Shape shape : shapeList)
            shape.draw();
    }
} /* Output:
Circle.draw()
Square.draw()
Triangle.draw()
*/

```

La clase base contiene un método `draw()` que utiliza indirectamente `toString()` para imprimir un identificador de la clase, pasando `this` a `System.out.println()` (observe que `toString()` se declara como abstracto para obligar a las clases herederas a sustituirlo, y para impedir la instantación de un objeto `Shape` simple). Si un objeto aparece en una expresión de concatenación de cadenas (donde están involucrados '+' y objetos `String`), se invoca automáticamente el método `toString()` para generar una representación de tipo `String` de dicho objeto. Cada una de las clases derivadas sustituye el método `toString()` (de `Object`) de modo que `draw()` termine (polimórficamente) imprimiendo algo distinto en cada caso.

En este ejemplo, la generalización tiene lugar cuando se coloca la forma geométrica en el contenedor `List<Shape>`. Durante la generalización a `Shape`, el hecho de que los objetos sean *tipos específicos* de `Shape` se pierde. Para la matriz se trata simplemente de objetos `Shape`.

En el momento en que se extrae un elemento de la matriz, el contenedor (que en la práctica almacena todos los elementos como si fueran de tipo `Object`) proyecta automáticamente el resultado sobre un objeto `Shape`. Éste es el tipo más básico del mecanismo RTTI, porque todas las proyecciones de tipos se comprueban en tiempo de ejecución para comprobar su corrección. Eso es lo que RTTI significa: el tipo de los objetos se identifica en tiempo de ejecución.

En este caso, la proyección RTTI sólo es parcial: el objeto `Object` se proyecta sobre `Shape`, y no sobre `Circle`, `Square` o `Triangle`. Eso es debido a que lo único que sabemos en este punto es que el contenedor `List<Shape>` está lleno de objetos `Shape`. En tiempo de compilación, esto se impone mediante el contenedor y el sistema genérico de Java, pero en tiempo de ejecución es la proyección la que garantiza que esto sea así.

Ahora es cuando entra en acción el polimorfismo y se determina el código exacto que ejecutará el objeto `Shape` viendo si la referencia corresponde a un objeto `Circle`, `Square` o `Triangle`. Y, en general, así es como deben ser las cosas. Lo que queremos es que la mayor parte de nuestro código sepa lo menos posible acerca de los tipos *específicos* de los objetos, debiendo limitarse a tratar con la representación general de una familia de objetos (en este caso, `Shape`). Como resultado, el código será más fácil de escribir, de leer y de mantener, y los diseños serán más sencillos de implementar, comprender y

modificar. Por ello, el polimorfismo es uno de los objetivos generales que se persiguen con la programación orientada a objetos.

¿Pero qué sucede si tenemos un problema especial de programación que resulta más fácil de resolver si conocemos el tipo exacto de una referencia genérica? Por ejemplo, suponga que queremos permitir a nuestros usuarios que resalten todas las formas geométricas de un cierto tipo concreto, asignándolas un color especial, de esta forma, pueden localizar todos los triángulos de la pantalla resaltándolos. O, por ejemplo, imagine que nuestro método necesita "rotar" una lista de formas geométricas, pero que no tiene sentido rotar un círculo, por lo que preferimos saltarnos los círculos al implementar la rotación de todas las formas. Con RTTI, podemos preguntar a una referencia de tipo **Shape** cuál es el tipo exacto al que está apuntando, lo que nos permite seleccionar y aislar los casos especiales.

El objeto Class

Para comprender cómo funciona el mecanismo RTTI en Java, primero tenemos que saber cómo se representa la información de tipos en tiempo de ejecución. Esto se lleva a cabo mediante un tipo de objeto especial denominado *objeto Class*, que contiene información acerca de la clase. De hecho, el objeto **Class** se utiliza para crear todos los objetos "normales" de una clase. Java implementa el mecanismo RTTI utilizando el objeto **Class**, incluso si lo que estamos haciendo es algo como una proyección de tipos. La clase **Class** también permite otra serie de formas de utilización de RTTI.

Existe un objeto **Class** para cada clase que forme parte del programa. En otras palabras, cada vez que escribimos y compilamos una nueva clase, también se crea un determinado objeto **Class** (y ese objeto se almacena en un archivo .class de nombre idéntico). Para crear un objeto de esa clase, la máquina virtual Java (JVM) que esté ejecutando el programa utiliza un subsistema denominado *cargador de clases*.

El subsistema cargador de clases puede comprender, en la práctica, una cadena de cargadores de clases, pero sólo existe un *cargador de clases primordial*, que forma parte de la implementación de la JVM. El cargador de clases primordial carga las que se denominan *clases de confianza*, que incluyen las clases de las interfaces API de Java, y esa carga se realiza normalmente desde el disco local. Usualmente no es necesario tener cargadores de clases adicionales en la cadena, pero si tenemos necesidades especiales (como por ejemplo, cargar clases de alguna manera especial para dar soporte a aplicaciones de servidor web, o descargar clases a través de una red), entonces disponemos de una manera de enlazar cargadores de clases adicionales.

Todas las clases se cargan en la JVM dinámicamente, cuando se utiliza la clase por primera vez. Esto sucede cuando el programa hace referencia a un miembro estático de dicha clase. Resulta que el constructor también es un método estático de una clase, aún cuando no se utilice la palabra clave **static** para el constructor. Por tanto, el crear un nuevo objeto de dicha clase utilizando el operador **new** también cuenta como una referencia a un miembro estático de la clase.

Por tanto, los programas Java no se cargan por completo antes de comenzar la ejecución, sino que se van cargando los distintos fragmentos del programa a medida que son necesarios. Esto difiere de muchos lenguajes tradicionales. El mecanismo de carga permite conseguir un tipo de comportamiento que resulta muy difícil, o incluso imposible, de obtener con un lenguaje estático de carga como pueda ser C++.

El cargador de clases comprueba primero si el objeto **Class** de dicho tipo está cargado. Si no lo está, el cargador de clases predeterminado localiza el archivo .class con dicho nombre (un cargador de clases adicional podría, por ejemplo, extraer el código intermedio de una base de datos en lugar de hacerlo de un archivo). A medida que se carga el código intermedio correspondiente a la clase, dicho código se verifica para garantizar que no esté corrompido y que no incluya código Java mal formado (ésta es una de las líneas de defensa de los mecanismos de seguridad de Java).

Una vez que el objeto **Class** de dicho tipo se encuentra en memoria se le utiliza para crear todos los objetos de dicho tipo. He aquí un programa que ilustra esta forma de actuar:

```
//: typeinfo/SweetShop.java
// Examen de la forma en que funciona el cargador de clases.
import static net.mindview.util.Print.*;

class Candy {
    static { print("Loading Candy"); }
}
```

```

class Gum {
    static { print("Loading Gum"); }
}

class Cookie {
    static { print("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        print("inside main");
        new Candy();
        print("After creating Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            print("Couldn't find Gum");
        }
        print("After Class.forName(\"Gum\")");
        new Cookie();
        print("After creating Cookie");
    }
} /* Output:
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
*///:-*

```

Cada una de las clases de **Candy**, **Gum** y **Cookie** tiene una cláusula **static** que se ejecuta cuando se carga la clase por primera vez. Se imprimirá la información para decírnos cuándo tiene lugar la carga de esa clase. En **main()**, las creaciones de objetos están mezcladas con instrucciones de impresión, como ayuda para determinar el instante de la carga.

Podemos ver, a partir de la salida, que cada objeto **Class** sólo se carga cuando es necesario, y que la inicialización de tipo **static** se realiza durante la carga de la clase. Una línea particularmente interesante es:

```
Class.forName("Gum");
```

Todos los objetos **Class** pertenecen a la clase **Class**. Un objeto **Class** es como cualquier otro objeto, por lo que se puede obtener y manipular una referencia a él (esto es lo que hace el cargador). Una de las formas de obtener una referencia al objeto **Class** es el método estático **forName()**, que toma un argumento de tipo **String** que contiene el nombre textual (¡tenga cuidado con la ortografía y el uso de mayúsculas!) de la clase concreta de la cual se quiera obtener una referencia. El método devuelve una referencia de tipo **Class**, que en este ejemplo se ignora; la llamada a **forName()** se realiza debido a su efecto secundario que consiste en cargar la clase **Gum** si no está ya cargada. Durante el proceso de carga se ejecuta la cláusula **static** de **Gum**.

En el ejemplo anterior, si **Class.forName()** falla porque no puede encontrar la clase que estemos intentando cargar, generará una excepción **ClassNotFoundException**. Aquí, simplemente nos limitamos a informar del problema y a continuar, pero en otros programas más sofisticados podríamos intentar resolver el problema dentro de la rutina de tratamiento de excepciones.

Siempre que queramos utilizar la información de tipos en tiempo de ejecución, debemos primero obtener una referencia al objeto **Class** apropiado. **Class.forName()** es una forma cómoda de hacer esto, porque no necesitamos un objeto de dicho tipo para obtener la referencia **Class**. Sin embargo, si ya disponemos de un objeto del tipo que nos interesa, podemos extraer la referencia **Class** invocando un método que forma parte de la clase raíz **Object**: **getClass()**. Este mecanismo devuelve la referencia **Class** que representa el tipo concreto de objeto. **Class** dispone de muchos métodos interesantes; el siguiente es un ejemplo que ilustra algunos de ellos:

```

//; typeinfo/toys/ToyTest.java
// Prueba de la clase Class.
package typeinfo.toys;
import static net.mindview.util.Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
    // Desactive con un comentario el constructor predeterminado
    // siguiente para ver NoSuchMethodError de (*1*)
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
    FancyToy() { super(1); }
}

public class ToyTest {
    static void printInfo(Class cc) {
        print("Class name: " + cc.getName() +
            " is interface? [" + cc.isInterface() + "]");
        print("Simple name: " + cc.getSimpleName());
        print("Canonical name : " + cc.getCanonicalName());
    }
    public static void main(String[] args) {
        Class c = null;
        try {
            c = Class.forName("typeinfo.toys.FancyToy");
        } catch(ClassNotFoundException e) {
            print("Can't find FancyToy");
            System.exit(1);
        }
        printInfo(c);
        for(Class face : c.getInterfaces())
            printInfo(face);
        Class up = c.getSuperclass();
        Object obj = null;
        try {
            // Requiere un constructor predeterminado:
            obj = up.newInstance();
        } catchInstantiationException e) {
            print("Cannot instantiate");
            System.exit(1);
        } catch(IllegalAccessException e) {
            print("Cannot access");
            System.exit(1);
        }
        printInfo(obj.getClass());
    }
} /* Output:
Class name: typeinfo.toys.FancyToy is interface? [false]
Simple name: FancyToy
Canonical name : typeinfo.toys.FancyToy
Class name: typeinfo.tcs.HasBatteries is interface? [true]
Simple name: HasBatteries
*/

```

```

Canonical name : typeinfo.toys.HasBatteries
Class name: typeinfo.toys.Waterproof is interface? [true]
Simple name: Waterproof
Canonical name : typeinfo.toys.Waterproof
Class name: typeinfo.toys.Shoots is interface? [true]
Simple name: Shoots
Canonical name : typeinfo.toys.Shoots
Class name: typeinfo.toys.Toy is interface? [false]
Simple name: Toy
Canonical name : typeinfo.toys.Toy
*///:-

```

FancyToy hereda de **Toy** e implementa las interfaces **HasBatteries**, **Waterproof** y **Shoots**. En **main()**, se crea una referencia **Class** y se la inicializa para que apunte al objeto **Class FancyToy Class** utilizando **forName()** dentro de un bloque **try** apropiado. Observe que hay que utilizar el nombre completamente cualificado (incluyendo el nombre del paquete) en la cadena de caracteres que se pasa a **forName()**.

printInfo() utiliza **getName()** para generar el nombre de clase completamente cualificado, y **getSimpleName()** y **getCanonicalName()** (introducidos en Java SE5) para generar el nombre sin el paquete y el nombre completamente cualificado, respectivamente. Como su propio nombre indica, **isInterface()** nos dice si este objeto **Class** representa un interfaz. Por tanto, con el objeto **Class** podemos averiguar casi todo lo que necesitemos saber acerca de un determinado tipo.

El método **Class.getInterfaces()** invocado en **main()** devuelve una matriz de objetos **Class** que representa las interfaces contenidas en el objeto **Class** de interés.

Si tenemos un objeto **Class**, también podemos preguntarle cuál es su clase base directa utilizando **getSuperclass()**. Este método devuelve una referencia **Class** que podemos, a su vez, consultar. Por tanto, podemos determinar la jerarquía de clases completa de un objeto en tiempo de ejecución.

El método **newInstance()** de **Class** constituye una forma de implementar un “constructor virtual” que nos permite decir: “No sé exactamente de qué tipo eres, pero crea una instancia de tí mismo de todas formas”. En el ejemplo anterior, **up** es simplemente una referencia **Class** de la cual no se conoce ninguna información de tipos adicional en tiempo de compilación. Y cuando creamos una nueva instancia, obtenemos como resultado una referencia **Object**. Pero dicha referencia apunta a un objeto **Toy**. Por supuesto, antes de poder enviar a ese objeto ningún mensaje diferente de los que admite **Object**, es necesario investigar un poco acerca del objeto y efectuar algunas proyecciones de tipos. Además, la clase que se crea con **newInstance()** debe disponer de un constructor predeterminado. Posteriormente en este capítulo, veremos cómo crear objetos dinámicamente de una cierta clase utilizando cualquier constructor, empleando para ello la API de *reflexión* de Java.

Ejercicio 1: (1) En **ToyTest.java**, desactive mediante un comentario el constructor predeterminado de **Toy** y explique lo que sucede.

Ejercicio 2: (2) Incorpore un nuevo tipo de interfaz en **ToyTest.java** y verifique que dicha interfaz se detecta y se muestra adecuadamente.

Ejercicio 3: (2) Añada **Rhomboid** a **Shapes.java**. Cree un objeto **Rhomboid** y generalicelo a **Shape**, y vuelva a especializarlo a **Rhomboid**. Trate de especializarlo a un objeto **Circle** y vea lo que sucede.

Ejercicio 4: (2) Modifique el ejercicio anterior para que utilice **instanceof** con el fin de comprobar el tipo, antes de efectuar la especialización.

Ejercicio 5: (3) Implemente un método **rotate(Shape)** en **Shapes.java**, que compruebe si está girando un círculo (y, en caso afirmativo, no realice la operación).

Ejercicio 6: (4) Modifique **Shapes.java** para que permita “resaltar” (activando un indicador) todas las formas de un tipo concreto. El método **toString()** para cada objeto derivado de **Shape** debe indicar si dicho objeto **Shape** está “resaltado”.

Ejercicio 7: (3) Modifique **SweetShop.java** para que la creación de cada tipo de objeto esté controlada por un argumento de la línea de comandos. En otras palabras, si la línea de comandos es “**java SweetShop Candy**”, entonces sólo se creará el objeto **Candy**. Observe cómo se pueden controlar los objetos **Class** que se cargan, utilizando argumentos de la línea de comandos.

- Ejercicio 8:** (5) Escriba un método que tome un objeto e imprima de manera recursiva todas las clases presentes en la jerarquía de ese objeto.
- Ejercicio 9:** (5) Modifique el ejercicio anterior de modo que utilice `Class.getDeclaredFields()` con el fin de mostrar también información acerca de los campos contenidos en cada clase.
- Ejercicio 10:** (3) Escriba un programa para determinar si una matriz de `char` es un tipo primitivo o un verdadero objeto.

Literales de clase

Java proporciona una segunda forma de generar la referencia al objeto `Class`: el *literal de clase*. En el programa anterior, dicho literal de clase tendría el aspecto:

```
FancyToy.class;
```

lo que no sólo es más simple, sino también más seguro ya que se comprueba en tiempo de compilación (y no necesita, por tanto, colocarse dentro de un bloque `try`). Asimismo, puesto que elimina la llamada al método `forName()`, es también más eficiente.

Los literales de clase funcionan tanto con las clases normales como con las interfaces, matrices y tipos primitivos. Además, existe un campo estándar denominado `TYPE` en cada una de las clases envoltorio de los tipos primitivos. El campo `TYPE` produce una referencia al objeto `Class` correspondiente al tipo primitivo asociado, de modo que:

| <i>... es equivalente a ...</i> | |
|---------------------------------|-----------------------------|
| <code>boolean.class</code> | <code>Boolean.TYPE</code> |
| <code>char.class</code> | <code>Character.TYPE</code> |
| <code>byte.class</code> | <code>Byte.TYPE</code> |
| <code>short.class</code> | <code>Short.TYPE</code> |
| <code>int.class</code> | <code>Integer.TYPE</code> |
| <code>long.class</code> | <code>Long.TYPE</code> |
| <code>float.class</code> | <code>Float.TYPE</code> |
| <code>double.class</code> | <code>Double.TYPE</code> |
| <code>void.class</code> | <code>Void.TYPE</code> |

En mi opinión, es mejor utilizar las versiones “`.class`” siempre que se pueda, ya que son más coherentes con las clases normales.

Es interesante observar que al crear una referencia a un objeto `Class` utilizando “`.class`” no se inicializa automáticamente el objeto `Class`. La preparación de una clase para su uso consta, en realidad, de tres pasos diferentes:

1. *Carga*, que es realizada por el cargador de clases. Este proceso localiza el código intermedio (que usualmente se encuentra en el disco, dentro de la ruta de clases, aunque no tiene porqué ser necesariamente así) y crea un objeto `Class` a partir de dicho código intermedio.
2. *Montaje*. La fase de montaje verifica el código intermedio de la clase, asigna el campo de almacenamiento para los campos estáticos y, en caso necesario, resuelve todas las referencias que esta clase haga a otras clases.
3. *Inicialización*. Si hay una superclase, es preciso inicializarla, ejecutando los inicializadores de tipo `static` y los bloques de inicialización de tipo `static`.

La inicialización se retarda hasta que produce la primera referencia a un método estático (el constructor es implicitamente de tipo `static`) o a un campo estático no constante:

```

//: typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
    static final int staticFinal = 47;
    static final int staticFinal2 =
        ClassInitialization.rand.nextInt(1000);
    static {
        System.out.println("Initializing Initable");
    }
}

class Initable2 {
    static int staticNonFinal = 147;
    static {
        System.out.println("Initializing Initable2");
    }
}

class Initable3 {
    static int staticNonFinal = 74;
    static {
        System.out.println("Initializing Initable3");
    }
}

public class ClassInitialization {
    public static Random rand = new Random(47);
    public static void main(String[] args) throws Exception {
        Class initable = Initable.class;
        System.out.println("After creating Initable ref");
        // No provoca la inicialización:
        System.out.println(Initable.staticFinal);
        // Provoca la inicialización:
        System.out.println(Initable.staticFinal2);
        // Provoca la inicialización:
        System.out.println(Initable2.staticNonFinal);
        Class initable3 = Class.forName("Initable3");
        System.out.println("After creating Initable3 ref");
        System.out.println(Initable3.staticNonFinal);
    }
} /* Output:
After creating Initable ref
47
Initializing Initable
258
Initializing Initable2
147
Initializing Initable3
After creating Initable3 ref
74
*///:-
```

En la práctica, la inicialización es lo más tardía posible. Analizando la creación de la referencia **initable**, podemos ver que usar simplemente la sintaxis `.class` para obtener una referencia a la clase no provoca la inicialización. Sin embargo, `Class.forName()` inicializa la clase inmediatamente para generar la referencia `Class`, como puede ver analizando la creación de **initable3**.

Si un valor final estático es una “constante de tiempo de compilación”, tal como `Initable.staticFinal`, dicho valor puede leerse sin que ello haga que la clase **Initable** se inicialice. Sin embargo, definir un campo como estático y final no garanti-

za este comportamiento; al acceder a **Initable.staticFinal2** se fuerza a la inicialización de clase, porque dicho campo no puede ser una constante de tiempo de compilación.

Si un campo estático no es de tipo **final**, acceder al mismo requiere siempre que se ejecute la fase de montaje (para asignar el espacio de almacenamiento para el campo) y también la de inicialización (para inicializar dicho espacio de almacenamiento) antes de que el valor pueda ser leído, como puede ver analizando el acceso a **Initable2.staticNonFinal**.

Referencias de clase genéricas

Una referencia **Class** apunta a un objeto **Class**, que genera instancias de las clases y contiene todo el código de los métodos correspondientes a dichas instancias. También contiene los valores estáticos de dicha clase. Por tanto, una referencia **Class** realmente indica el tipo exacto de aquello a lo que está apuntando: un objeto de la clase **Class**.

Sin embargo, los diseñadores de Java SE5 vieron la oportunidad de hacer esto un poco más específico, permitiendo restringir el tipo de objeto **Class** al que la referencia **Class** apunta, utilizando para ello la sintaxis genérica. En el siguiente ejemplo, ambos tipos de sintaxis son correctos:

```
//: typeinfo/GenericClassReferences.java

public class GenericClassReferences {
    public static void main(String[] args) {
        Class intClass = int.class;
        Class<Integer> genericIntClass = int.class;
        genericIntClass = Integer.class; // Lo mismo
        intClass = double.class;
        // genericIntClass = double.class; // Illegal
    }
} ///:-
```

La referencia de clase normal no genera ninguna advertencia de compilación. Sin embargo, puede ver que la referencia de clase normal puede reasignarse a cualquier otro objeto **Class**, mientras que la referencia de clase genérica sólo puede asignarse a su tipo declarado. Utilizando la sintaxis genérica, permitimos que el compilador imponga comprobaciones adicionales de los tipos.

¿Qué sucede si queremos relajar un poco las restricciones? Inicialmente, parece que deberíamos ser capaces de hacer algo como lo siguiente:

```
Class<Number> genericNumberClass = int.class;
```

Esto parece tener sentido, porque **Integer** hereda de **Number**. Sin embargo, este método no funciona, porque el objeto **Class Integer** no es una subclase del objeto **Class Number** (puede parecer que esta distinción resulta demasiado sutil; la analizaremos con más detalle en el Capítulo 15, *Genéricos*).

Para relajar las restricciones al utilizar referencias **Class** genéricas, yo personalmente empleo el *comodín*, que forma parte de los genéricos de Java. El símbolo del comodín es "?", e indica "cualquier cosa". Por tanto, podemos añadir comodines a la referencia **Class** del ejemplo anterior y generar los mismos resultados:

```
//: typeinfo/WildcardClassReferences.java

public class WildcardClassReferences {
    public static void main(String[] args) {
        Class<?> intClass = int.class;
        intClass = double.class;
    }
} ///:-
```

En Java SE5, se prefiere utilizar **Class<?>** en lugar de **Class**, aún cuando ambos son equivalentes y la referencia **Class** normal, como hemos visto, no genera ninguna advertencia del compilador. La ventaja de **Class<?>** es que indica que no estamos pasando una referencia de clase no específica simplemente por accidente o por ignorancia, sino que hemos *elegido* la versión no específica.

Para crear una referencia **Class** que esté restringida a un determinado tipo o a *cualquiera de sus subtipos*, podemos combinar un comodín con la palabra clave **extends** para crear un *límite*. Por tanto, en lugar de decir simplemente **Class<Number>**, lo que diríamos sería:

```
//: typeinfo/BoundedClassReferences.java

public class BoundedClassReferences {
    public static void main(String[] args) {
        Class<? extends Number> bounded = int.class;
        bounded = double.class;
        bounded = Number.class;
        // O cualquier otra cosa derivada de Number.
    }
} /*:-
```

La razón de añadir la sintaxis genérica a las referencias **Class** esriba únicamente, en realizar una comprobación de los tipos en tiempo de compilación, de modo que si hacemos algo incorrecto lo detectaremos un poco antes. No es posible realizar nada realmente destructivo con las referencias **Class** normales, pero si cometemos un error no podremos detectarlo hasta el tiempo de ejecución, lo que puede resultar incómodo.

He aquí un ejemplo donde se utiliza la sintaxis de clases genéricas. El ejemplo almacena una referencia de clase y luego genera un contenedor **List** lleno con objetos generados mediante **newInstance()**:

```
//: typeinfo/FilledList.java
import java.util.*;

class CountedInteger {
    private static long counter;
    private final long id = counter++;
    public String toString() { return Long.toString(id); }
}

public class FilledList<T> {
    private Class<T> type;
    public FilledList(Class<T> type) { this.type = type; }
    public List<T> create(int nElements) {
        List<T> result = new ArrayList<T>();
        try {
            for(int i = 0; i < nElements; i++)
                result.add(type.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        return result;
    }
    public static void main(String[] args) {
        FilledList<CountedInteger> fl =
            new FilledList<CountedInteger>(CountedInteger.class);
        System.out.println(fl.create(15));
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*:-
```

Observe que esta clase debe asumir que cualquier tipo con el que trabaje dispondrá de un constructor predeterminado (uno que no tenga argumentos), obteniéndose una excepción si no es éste el caso. El compilador no genera ningún tipo de advertencia para este programa.

Cuando utilizamos la sintaxis genérica para los objetos **Class** sucede algo interesante: **newInstance()** devolverá el tipo exacto del objeto, en lugar de simplemente un objeto básico **Object** como vimos en **ToyTest.java**. Esto resulta un tanto limitado:

```
//: typeinfo/toys/GenericToyTest.java
// Prueba de la clase Class.
package typeinfo.toys;

public class GenericToyTest {
    public static void main(String[] args) throws Exception {
        Class<FancyToy> ftClass = FancyToy.class;
        // Produce el tipo exacto:
        FancyToy fancyToy = ftClass.newInstance();
        Class<? super FancyToy> up = ftClass.getSuperclass();
        // Esto no se compilará:
        // Class<Toy> up2 = ftClass.getSuperclass();
        // Sólo produce Object:
        Object obj = up.newInstance();
    }
} ///:-
```

Si obtenemos la superclase, el compilador sólo nos permitirá decir que la referencia a la superclase es “alguna clase que es superclase de **FancyToy**”, como podemos ver en la expresión `Class<? super FancyToy>`. No aceptará una declaración de `Class<Toy>`. Esto parece un poco extraño, porque `getSuperclass()` devuelve la *clase* base (no una interfaz) y el compilador conoce en tiempo de compilación lo que esa clase es; en este caso, `Toy.class`, no simplemente “alguna superclase de **FancyToy**”. En cualquier caso, debido a la vaguedad, el valor de retorno de `up.newInstance()` no es de un tipo preciso, sino sólo de tipo **Object**.

Nueva sintaxis de proyección

Java SE5 también ha añadido una sintaxis de proyección para utilizarla con las referencias **Class**, nos referimos al método `cast()`:

```
//: typeinfo/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
    public static void main(String[] args) {
        Building b = new House();
        Class<House> houseType = House.class;
        House h = houseType.cast(b);
        h = (House)b; // ... o haga simplemente esto.
    }
} ///:-
```

El método `cast()` toma el objeto proporcionado como argumento y lo proyecta sobre el tipo de la referencia **Class**. Por supuesto, si examinamos el código anterior parece que es demasiado trabajo adicional, si lo comparamos con la última línea de `main()`, que hace exactamente lo mismo.

La nueva sintaxis de proyección resulta útil en aquellas situaciones en las que *no podemos* utilizar una proyección ordinaria. Esto sucede, usualmente, cuando estamos escribiendo código genérico (de lo que hablaremos en el Capítulo 15, *Genéricos*), y hemos almacenado una referencia **Class** que queremos utilizar en algún momento posterior para efectuar la proyección. Este caso no resulta muy frecuente; de hecho, sólo he podido encontrar una única ocasión en la que `cast()` se use dentro de la biblioteca de Java SE5 (concretamente en `com.sun.mirror.util.DeclarationFilter`).

Hay otra nueva funcionalidad que *no se utiliza* en absoluto en la biblioteca Java SE5: `Class.asSubclass()`. Este método permite proyectar el objeto de clase sobre un tipo más específico.

Comprobación antes de una proyección

Hasta ahora, hemos visto varias formas de RTTI, incluyendo:

1. La proyección clásica, por ejemplo, “(Shape),” que utiliza RTTI para asegurarse de que la proyección es correcta. Esto generará **ClassCastException** si se ha realizado una proyección incorrecta.
2. El objeto **Class** representativo del objeto. Podemos consultar el objeto **Class** para obtener información útil en tiempo de ejecución.

En C++, la proyección clásica “(Shape)” *no utiliza* mecanismos RTTI. Simplemente le dice al compilador que trate el objeto como si fuera del tipo indicado. En Java, si realiza la comprobación de tipos, esta proyección se denomina a menudo “especialización segura en lo que respecta a tipos”. La razón de utilizar el término “especialización” se basa en la disposición históricamente utilizada en los diagramas de jerarquías de clases. Si la proyección de **Circle** sobre **Shape** es una generalización, entonces la proyección de **Shape** sobre **Circle** es una especialización. Sin embargo, puesto que el compilador sabe que un objeto **Circle** es también de tipo **Shape**, permite que se realicen libremente asignaciones de generalización, sin que sea obligatorio incluir una sintaxis de proyección específica. El compilador *no puede saber*, dado un objeto **Shape**, de qué tipo concreto es ese objeto; podría ser exactamente de **Shape**, o podría ser un subtipo de **Shape**, como **Circle**, **Square**, **Triangle** o algún otro tipo. En tiempo de compilación, el compilador sólo ve un objeto **Shape**. Por tanto, no nos permitirá que realicemos una asignación de especialización sin utilizar una proyección específica, con la que le decimos al compilador que disponemos de información adicional que nos permite saber que se trata de un tipo concreto (el compilador *comprobará* si dicha especialización es razonable, por lo que no nos permitirá efectuar especializaciones sobre un tipo que no sea realmente una subclase del anterior).

Existe un tercer mecanismo de RTTI en Java. Se trata de la palabra clave **instanceof**, que nos dice si un objeto es una instancia de un tipo concreto. Devuelve un valor de tipo **boolean**, así que esta palabra clave se utiliza en forma de pregunta, como en el fragmento siguiente:

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

La instrucción **If** comprueba si el objeto **x** pertenece a la clase **Dog** *antes* de proyectar **x** sobre **Dog**. Es importante utilizar **instanceof** antes de una especialización cuando no dispongamos de otra información que nos indique el tipo del objeto; en caso contrario, obtendremos una excepción **ClassCastException**.

Normalmente, lo que estaremos tratando de localizar es un determinado tipo (por ejemplo, para pintar de púrpura todos los triángulos), pero podemos fácilmente seleccionar *todos* los objetos utilizando **instanceof**. Por ejemplo, suponga que disponemos de una familia de clases para describir mascotas, **Pet**, (y sus propietarios, una característica que nos será útil en un ejemplo posterior). Cada individuo (**Individual**) de la jerarquía tiene un identificador **id** y un nombre opcional. Aunque las clases que siguen heredan de **Individual**, existen ciertas complejidades en la clase **Individual**, por lo que mostraremos y explicaremos dicho código en el Capítulo 17, *Análisis detallado de los contenedores*. En realidad, no es imprescindible analizar el código de **Individual** en este momento; lo único que necesitamos saber es que podemos crear un individuo con o sin nombre, y que cada objeto **Individual** tiene un método **id()** que devuelve un identificador único (creado mediante un simple recuento de los objetos). También hay un método **toString()**; si no se proporciona un nombre para un objeto **Individual**, **toString()** sólo genera el nombre simple del tipo.

He aquí la jerarquía de clases que hereda de **Individual**:

```
//: typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
    public Person(String name) { super(name); }
} //:-

//: typeinfo/pets/Pet.java
package typeinfo.pets;

public class Pet extends Individual {
    public Pet(String name) { super(name); }
    public Pet() { super(); }
} //:-

//: typeinfo/pets/Dog.java
```

```

package typeinfo.pets;

public class Dog extends Pet {
    public Dog(String name) { super(name); }
    public Dog() { super(); }
} //:-

//: typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
    public Mutt(String name) { super(name); }
    public Mutt() { super(); }
} //:-

//: typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
    public Pug(String name) { super(name); }
    public Pug() { super(); }
} //:-

//: typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
    public Cat(String name) { super(name); }
    public Cat() { super(); }
} //:-

//: typeinfo/pets/EgyptianMau.java
package typeinfo.pets;

public class EgyptianMau extends Cat {
    public EgyptianMau(String name) { super(name); }
    public EgyptianMau() { super(); }
} //:-

//: typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
    public Manx(String name) { super(name); }
    public Manx() { super(); }
} //:-

//: typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
    public Cymric(String name) { super(name); }
    public Cymric() { super(); }
} //:-

//: typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
}

```

```

public Rodent(String name) { super(name); }
public Rodent() { super(); }
} //:-

//: typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
    public Rat(String name) { super(name); }
    public Rat() { super(); }
} //:-

//: typeinfo/pets/Mouse.java
package typeinfo.pets;

public class Mouse extends Rodent {
    public Mouse(String name) { super(name); }
    public Mouse() { super(); }
} //:-

//: typeinfo/pets/Hamster.java
package typeinfo.pets;

public class Hamster extends Rodent {
    public Hamster(String name) { super(name); }
    public Hamster() { super(); }
} //:-
```

A continuación, necesitamos una forma de crear aleatoriamente diferentes tipos de mascotas, y por comodidad, vamos a crear matrices y listas de mascotas. Para permitir que esta herramienta evolucione a través de varias implementaciones diferentes, vamos a definir dicha herramienta como una clase abstracta:

```

//: typeinfo/pets/PetCreator.java
// Crea secuencias aleatorias de objetos Pet.
package typeinfo.pets;
import java.util.*;

public abstract class PetCreator {
    private Random rand = new Random(47);
    // La lista de los diferentes tipos de Pet que hay que crear:
    public abstract List<Class<? extends Pet>> types();
    public Pet randomPet() { // Crear un objeto Pet aleatorio
        int n = rand.nextInt(types().size());
        try {
            return types().get(n).newInstance();
        } catch(InstantiationException e) {
            throw new RuntimeException(e);
        } catch(IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
    public Pet[] createArray(int size) {
        Pet[] result = new Pet[size];
        for(int i = 0; i < size; i++)
            result[i] = randomPet();
        return result;
    }
    public ArrayList<Pet> arrayList(int size) {
        ArrayList<Pet> result = new ArrayList<Pet>();
        Collections.addAll(result, createArray(size));
    }
}
```

```

        return result;
    }
} //:-
```

El método abstracto `getTypes()` deja para las clases derivadas la tarea de obtener la lista de objetos `Class` (esto es una variante del patrón de diseño basado en el método de las plantillas). Observe que el tipo de clase se especifica como “cualquier cosa derivada de `Pet`”, por lo que `newInstance()` produce un objeto `Pet` sin requerir ninguna proyección. `randomPet()` realiza una indexación aleatoria en el contenedor de tipo `List` y utiliza el objeto `Class` seleccionado para generar una nueva instancia de dicha clase con `Class.newInstance()`. El método `createArray()` utiliza `randomPet()` para llenar una matriz y `arrayList()` emplea a su vez `createArray()`.

Podemos obtener dos tipos de excepciones al llamar a `newInstance()`. Analizando el ejemplo, podrá ver que estas excepciones se tratan en las cláusulas `catch` que siguen al bloque `try`. De nuevo, los nombres de las excepciones son casi autoexplicativos e indican cuál es el problema (`IllegalAccessException` está relacionado con una violación del mecanismo de seguridad de Java, en este caso si el constructor predeterminado es de tipo `private`).

Cuando derivamos una subclase de `PetCreator`, lo único que necesitamos suministrar es el contenedor `List` de todos los tipos de mascotas que queremos crear mediante `randomPet()` y los otros métodos. El método `getTypes()` normalmente devolverá, simplemente, una referencia a un lista estática. He aquí una implementación utilizando `forName()`:

```

//: typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
    private static List<Class<? extends Pet>> types =
        new ArrayList<Class<? extends Pet>>();
    // Tipos que queremos crear aleatoriamente:
    private static String[] typeNames = {
        "typeinfo.pets.Mutt",
        "typeinfo.pets.Pug",
        "typeinfo.pets.EgyptianMau",
        "typeinfo.pets.Manx",
        "typeinfo.pets.Cymric",
        "typeinfo.pets.Rat",
        "typeinfo.pets.Mouse",
        "typeinfo.pets.Hamster"
    };
    @SuppressWarnings("unchecked")
    private static void loader() {
        try {
            for(String name : typeNames)
                types.add(
                    (Class<? extends Pet>)Class.forName(name));
        } catch(ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
    static { loader(); }
    public List<Class<? extends Pet>> types() { return types; }
} //:-
```

El método `loader()` crea la lista de objetos `Class` utilizando `Class.forName()`. Esto puede generar la excepción `ClassNotFoundException`, lo cual tiene bastante sentido, ya que estamos pasándole un objeto `String` que no puede validarse en tiempo de compilación. Puesto que los objetos `Pet` se encuentran en el paquete `typeinfo`, es necesario utilizar el nombre del paquete al hacer referencia a las clases.

Para generar una lista con tipos de objetos `Class`, se necesita una proyección, lo cual produce una advertencia en tiempo de compilación. El método `loader()` se define por separado y luego se incluye dentro de una cláusula de inicialización estática, porque la anotación `@SuppressWarnings` no puede insertarse directamente en la cláusula de inicialización estática.

Para recomptar el número de mascotas, necesitamos una herramienta que vaya controlando el número de los diferentes tipos de mascotas. Un contenedor **Map** resulta perfecto para esta tarea; las claves con los nombres de los tipos de objetos **Pet** y los valores son enteros que almacenan el número de objetos **Pet** de cada tipo. De esta forma, podemos preguntar cosas como “¿Cuántos objetos **Hamster** hay?”. Podemos utilizar **instanceof** para recomptar las mascotas:

```
//: typeinfo/PetCount.java
// Utilización de instanceof.
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetCount {
    static class PetCounter extends HashMap<String, Integer> {
        public void count(String type) {
            Integer quantity = get(type);
            if(quantity == null)
                put(type, 1);
            else
                put(type, quantity + 1);
        }
    }
    public static void countPets(PetCreator creator) {
        PetCounter counter= new PetCounter();
        for(Pet pet : creator.createArray(20)) {
            // Enumerar las mascotas individuales:
            printnb(pet.getClass().getSimpleName() + " ");
            if(pet instanceof Pet)
                counter.count("Pet");
            if(pet instanceof Dog)
                counter.count("Dog");
            if(pet instanceof Mutt)
                counter.count("Mutt");
            if(pet instanceof Pug)
                counter.count("Pug");
            if(pet instanceof Cat)
                counter.count("Cat");
            if(pet instanceof Manx)
                counter.count("EgyptianMau");
            if(pet instanceof Manx)
                counter.count("Manx");
            if(pet instanceof Manx)
                counter.count("Cymric");
            if(pet instanceof Rodent)
                counter.count("Rodent");
            if(pet instanceof Rat)
                counter.count("Rat");
            if(pet instanceof Mouse)
                counter.count("Mouse");
            if(pet instanceof Hamster)
                counter.count("Hamster");
        }
        // Mostrar las cantidades:
        print();
        print(counter);
    }
    public static void main(String[] args) {
        countPets(new ForNameCreator());
    }
}
```

```

} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau Hamster
EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
{Pug=3, Cat=9, Hamster=1, Cymric=7, Mouse=2, Mutt=3, Rodent=5,
Pet=20, Manx=7, EgyptianMau=7, Dog=6, Rat=2}
*///:-
```

En `countPets()`, se rellena aleatoriamente una matriz con objetos `Pet` utilizando un objeto `PetCreator`. Después, cada objeto `Pet` de la matriz se comprueba y se recuenta utilizando `instanceof`.

Existe una pequeña restricción en la utilización de `instanceof`: podemos comparar únicamente con un tipo nominado, y no con un objeto `Class`. En el ejemplo anterior, podría parecer que resulta tedioso escribir todas esas expresiones `instanceof`, y efectivamente lo es. Pero no hay ninguna forma inteligente de automatizar `instanceof` creando una matriz de objetos `Class` y realizando la comparación de dichos objetos (aunque, si sigue leyendo, verá que existe una alternativa). Sin embargo, esta restricción no es tan grave como pudiera parecer, porque más adelante veremos que si un diseño nos exige escribir una gran cantidad de expresiones `instanceof` probablemente eso signifique que el diseño no está bien hecho.

Utilización de literales de clase

Si reimplementamos la clase `PetCreator` usando literales de clase, el resultado es mucho más limpio en muchos aspectos:

```

//: typeinfo/pets/LiteralPetCreator.java
// Utilización de literales de clase.
package typeinfo.pets;
import java.util.*;

public class LiteralPetCreator extends PetCreator {
    // No hace falta bloque try.
    @SuppressWarnings("unchecked")
    public static final List<Class<? extends Pet>> allTypes =
        Collections.unmodifiableList(Arrays.asList(
            Pet.class, Dog.class, Cat.class, Rodent.class,
            Mutt.class, Pug.class, EgyptianMau.class, Manx.class,
            Cymric.class, Rat.class, Mouse.class, Hamster.class));
    // Tipos para la creación aleatoria:
    private static final List<Class<? extends Pet>> types =
        allTypes.subList(allTypes.indexOf(Mutt.class),
            allTypes.size());
    public List<Class<? extends Pet>> types() {
        return types;
    }
    public static void main(String[] args) {
        System.out.println(types());
    }
} /* Output:
[class typeinfo.pets.Mutt, class typeinfo.pets.Pug, class typeinfo.pets.EgyptianMau, class
typeinfo.pets.Manx, class typeinfo.pets.Cymric, class typeinfo.pets.Rat, class
typeinfo.pets.Mouse, class typeinfo.pets.Hamster]
*///:-
```

En el ejemplo `PetCount3.java`, que veremos más adelante, necesitamos precargar un contenedor `Map` con todos los tipos de `Pet` (no sólo con aquellos que hayan de ser generados aleatoriamente), por lo que es necesaria la lista `allTypes List`, que incluye todos los tipos. La lista `types` es la parte de `allTypes` (creada mediante `List.subList()`) que incluye los tipos exactos de mascota, por lo que es la que se utiliza para la generación aleatoria de objetos `Pet`.

Esta vez, la creación de `types` no necesita estar rodeada por un bloque `try`, puesto que se evalúa en tiempo de compilación y no generará ninguna excepción a diferencia de `Class.forName()`.

Ahora tenemos dos implementaciones de `PetCreator` en la biblioteca `typeinfo.pets`. Para definir la segunda como implementación predeterminada, podemos crear un envoltorio que utilice `LiteralPetCreator`:

```
//: typeinfo/pets/Pets.java
// Envoltorio para generar un PetCreator predeterminado.
package typeinfo.pets;
import java.util.*;

public class Pets {
    public static final PetCreator creator =
        new LiteralPetCreator();
    public static Pet randomPet() {
        return creator.randomPet();
    }
    public static Pet[] createArray(int size) {
        return creator.createArray(size);
    }
    public static ArrayList<Pet> arrayList(int size) {
        return creator.arrayList(size);
    }
} //:~
```

Esto proporciona también una indirección para acceder a `randomPet()`, `createArray()` y `arrayList()`.

Puesto que `PetCount.countPets()` toma como argumento `PetCreator`, podemos probar fácilmente la clase de `LiteralPetCreator` (mediante el envoltorio anteriormente definido):

```
//: typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
    public static void main(String[] args) {
        PetCount.countPets(Pets.creator);
    }
} /* (Execute to see output) */:-
```

La salida es igual que la de `PetCount.java`.

Instanceof dinámico

El método `Class.isInstance()` proporciona una forma para probar dinámicamente el tipo de un objeto. Por tanto, podemos eliminar todas esas tediosas instrucciones `instanceof` de `PetCount.java`:

```
//: typeinfo/PetCount3.java
// Utilización de isInstance()
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount3 {
    static class PetCounter
        extends LinkedHashMap<Class<? extends Pet>, Integer> {
        public PetCounter() {
            super(MapData.map(LiteralPetCreator.allTypes, 0));
        }
        public void count(Pet pet) {
            // Class.isInstance() elimina instrucciones instanceof:
            for(Map.Entry<Class<? extends Pet>, Integer> pair
                : entrySet())
                if(pair.getKey().isInstance(pet))
                    put(pair.getKey(), pair.getValue() + 1);
        }
        public String toString() {
```

```

StringBuilder result = new StringBuilder("{}");
for(Map.Entry<Class<? extends Pet>,Integer> pair
    : entrySet()) {
    result.append(pair.getKey().getSimpleName());
    result.append("=");
    result.append(pair.getValue());
    result.append(", ");
}
result.delete(result.length()-2, result.length());
result.append("}");
return result.toString();
}
}

public static void main(String[] args) {
    PetCounter petCount = new PetCounter();
    for(Pet pet : Pets.createArray(20)) {
        printnb(pet.getClass().getSimpleName() + " ");
        petCount.count(pet);
    }
    print();
    print(petCount);
}
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
{Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3, EgyptianMau=2,
Manx=7, Cymric=5, Rat=2, Mouse=2, Hamster=1}
*///:-

```

Para contar todos los tipos diferentes de objetos **Pet**, se precarga el mapa **PetCounter Map** con los tipos de **LiteralPetCreator.allTypes**. Esto utiliza la clase **net.mindview.util.MapData**, que toma un objeto **Iterable** (la lista **allTypes**) y un valor constante (cero, en este caso) y rellena el mapa con claves tomadas de **allTypes** y valores iguales a cero). Sin precargar el contenedor de tipo **Map**, lo que haríamos sería contar los tipos que se generan aleatoriamente y no los tipos base como **Pet** y **Cat**.

Como puede ver, el método **isInstance()** ha eliminado la necesidad de utilizar expresiones **instanceof**. Además, esto significa que podemos añadir nuevos tipos de **Pet** simplemente cambiando la matriz **LiteralPetCreator.types**; el resto del programa no necesita modificación (al revés de lo que sucedía al utilizar expresiones **instanceof**).

El método **toString()** ha sido sobrecargado para obtener una salida más legible que siga correspondiendo con la salida típica que podemos ver a la hora de imprimir un contenedor de tipo **Map**.

Recuento recursivo

El mapa en **PetCount3.PetCounter** estaba precargado con todas las diferentes clases de objetos **Pet**. En lugar de sobrecargar el mapa, podemos utilizar **Class.isAssignableFrom()** y crear una herramienta de propósito general que no esté limitada a recomptar objetos **Pet**:

```

//: net/mindview/util/TypeCounter.java
// Recuenta instancias de una familia de tipos.
package net.mindview.util;
import java.util.*;

public class TypeCounter extends HashMap<Class<?>, Integer>{
    private Class<?> basePath;
    public TypeCounter(Class<?> basePath) {
        this.basePath = basePath;
    }
    public void count(Object obj) {
        Class<?> type = obj.getClass();

```

```

        if (!baseType.isAssignableFrom(type))
            throw new RuntimeException(obj + " incorrect type: "
                + type + ", should be type or subtype of "
                + baseType);
        countClass(type);
    }
    private void countClass(Class<?> type) {
        Integer quantity = get(type);
        put(type, quantity == null ? 1 : quantity + 1);
        Class<?> superClass = type.getSuperclass();
        if (superClass != null &&
            baseType.isAssignableFrom(superClass))
            countClass(superClass);
    }
    public String toString() {
        StringBuilder result = new StringBuilder("(");
        for (Map.Entry<Class<?>, Integer> pair : entrySet()) {
            result.append(pair.getKey().getSimpleName());
            result.append("=");
            result.append(pair.getValue());
            result.append(", ");
        }
        result.delete(result.length() - 2, result.length());
        result.append("}");
        return result.toString();
    }
} //:-/

```

El método **count()** obtiene el objeto **Class** de su argumento y utiliza **isAssignableFrom()** para realizar una comprobación en tiempo de ejecución con el fin de verificar que el objeto que se le haya pasado pertenece verdaderamente a la jerarquía de clases que nos interesa. **countClass()** incrementa primero el contador correspondiente al tipo exacto de la clase. Después, si **baseType** es assignable desde la superclase, se invoca a **countClass()** recursivamente en la superclase.

```

//: typeinfo/PetCount4.java
import typeinfo.pets.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount4 {
    public static void main(String[] args) {
        TypeCounter counter = new TypeCounter(Pet.class);
        for (Pet pet : Pets.createArray(20)) {
            printnb(pet.getClass().getSimpleName() + " ");
            counter.count(pet);
        }
        print();
        print(counter);
    }
} /* Output: (Sample)
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
(Mouse=2, Dog=6, Manx=7, EgyptianMau=2, Rodent=5, Pug=3,
Mutt=3, Cymric=5, Cat=9, Hamster=1, Pet=20, Rat=2)
*///:-/

```

Como puede ver analizando la salida, se cuentan ambos tipos base así como los tipos exactos.

Ejercicio 11: (2) Añada **Gerbil** a la biblioteca **typeinfo.pets** y modifique todos los ejemplos del capítulo para adaptarlos a esta nueva clase.

Ejercicio 12: (3) Utilice **TypeCounter** con la clase **CoffeeGenerator.java** del Capítulo 15, *Genéricos*.

Ejercicio 13: (3) Utilice TypeCounter con el ejemplo RegisteredFactories.java de este capítulo.

Factorías registradas

Uno de los problemas a la hora de crear objetos de la jerarquía **Pet** es el hecho de que cada vez que añadimos un nuevo tipo de objeto **Pet** a la jerarquía tenemos que acordarnos de añadirlo a las entradas de **LiteralPetCreator.java**. En aquellos sistemas donde tengamos que añadir un gran número de clases de forma habitual, esto puede llegar a ser problemático.

Podríamos pensar en añadir un inicializador estático a cada subclase, de modo que el inicializador añadiera su clase a una lista que se conservara en algún lugar. Desafortunadamente, los inicializadores estáticos sólo se invocan cuando se carga por primera vez la clase, así que tenemos el típico problema de la gallina y el huevo: el generador no tiene la clase en su lista, por lo que nunca puede crear un objeto de esa clase, así que la clase no se cargará y no podrá ser incluida en la lista.

Básicamente, podemos obligarnos a crear la lista nosotros mismos de manera manual (a menos que queramos escribir una herramienta que analice el código fuente y luego genere y compile la lista). Por tanto, lo mejor que podemos hacer, probablemente, es colocar la lista en algún lugar central lo suficientemente obvio. Seguramente, el mejor lugar será la clase base de la jerarquía de clases que nos interese.

El otro cambio que vamos a hacer aquí es diferir la creación del objeto, dejándoselo a la propia clase, utilizando el patrón de diseño denominado *método de factoría*. Un método de factoría puede invocarse polimórficamente y se encarga de crear por nosotros un objeto del tipo apropiado. En esta versión muy simple, el método factoría es el método **create()** de la interfaz **Factory**:

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } //:-
```

El parámetro genérico **T** permite a **create()** devolver un tipo diferente por cada implementación de **Factory**. Esto hace uso también de los tipos de retorno covariantes.

En este ejemplo, la clase base **Part** contiene un contenedor **List** de objetos factoría. Las factorías correspondientes a los tipos que deben generarse mediante el método **createRandom()** se “registran” ante la clase base añadiéndolos a la lista **partFactories**:

```
//: typeinfo/RegisteredFactories.java
// Registro de factorías de clases en la clase base.
import typeinfo.factory.*;
import java.util.*;

class Part {
    public String toString() {
        return getClass().getSimpleName();
    }
    static List<Factory<? extends Part>> partFactories =
        new ArrayList<Factory<? extends Part>>();
    static {
        // Collections.addAll() genera una advertencia "unchecked generic
        // array creation ... for varargs parameter".
        partFactories.add(new FuelFilter.Factory());
        partFactories.add(new AirFilter.Factory());
        partFactories.add(new CabinAirFilter.Factory());
        partFactories.add(new OilFilter.Factory());
        partFactories.add(new FanBelt.Factory());
        partFactories.add(new PowerSteeringBelt.Factory());
        partFactories.add(new GeneratorBelt.Factory());
    }
    private static Random rand = new Random(47);
    public static Part createRandom() {
        int n = rand.nextInt(partFactories.size());
        return partFactories.get(n).create();
    }
}
```

```

    }

}

class Filter extends Part {}

class FuelFilter extends Filter {
    // Crear una factoría de clases para cada tipo específico:
    public static class Factory
        implements typeinfo.factory.Factory<FuelFilter> {
            public FuelFilter create() { return new FuelFilter(); }
        }
    }

class AirFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<AirFilter> {
            public AirFilter create() { return new AirFilter(); }
        }
    }

class CabinAirFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<CabinAirFilter> {
            public CabinAirFilter create() {
                return new CabinAirFilter();
            }
        }
    }

class OilFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<OilFilter> {
            public OilFilter create() { return new OilFilter(); }
        }
    }

class Belt extends Part {}

class FanBelt extends Belt {
    public static class Factory
        implements typeinfo.factory.Factory<FanBelt> {
            public FanBelt create() { return new FanBelt(); }
        }
    }

class GeneratorBelt extends Belt {
    public static class Factory
        implements typeinfo.factory.Factory<GeneratorBelt> {
            public GeneratorBelt create() {
                return new GeneratorBelt();
            }
        }
    }

class PowerSteeringBelt extends Belt {
    public static class Factory
        implements typeinfo.factory.Factory<PowerSteeringBelt> {
            public PowerSteeringBelt create() {
                return new PowerSteeringBelt();
            }
        }
    }
}

```

```

    }
}

public class RegisteredFactories {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            System.out.println(Part.createRandom());
    }
} /* Output:
GeneratorBelt
CabinAirFilter
GeneratorBelt
AirFilter
PowerSteeringBelt
CabinAirFilter
FuelFilter
PowerSteeringBelt
PowerSteeringBelt
FuelFilter
*///:-
```

No todas las clases de la jerarquía deben instanciarse; en este caso, **Filter** y **Belt** son simplemente clasificadores, por lo que no se crea ninguna instancia de ninguno de ellos, sino sólo de sus subclases. Si una clase *debe* ser creada por **createRandom()**, contendrá una clase **Factory** interna. La única forma de reutilizar el nombre **Factory**, como hemos visto antes, es mediante la cualificación **typeinfo.factory.Factory**.

Aunque podemos utilizar **Collections.addAll()** para añadir las factorías a la lista, el compilador se quejará, generando una advertencia relativa a la “creación de una matriz genérica” (lo que se supone que es imposible, como veremos en el Capítulo 15, *Genéricos*), por lo que hemos preferido invocar **add()**. El método **createRandom()** selecciona aleatoriamente un objeto factoría de **partFactories** e invoca su método **create()** para generar un nuevo objeto **Part**.

Ejercicio 14: (4) Un constructor es un tipo de método de factoría. Modifique **RegisteredFactories.java** para que en lugar de utilizar una factoría explícita, el objeto clase se almacene en el contenedor **List**, utilizándose **newInstance()** para crear cada objeto.

Ejercicio 15: (4) Implemente un nuevo **PetCreator** utilizando factorías registradas y modifique el método envoltorio de la sección “Utilización de literales de clase” para que emplee este nuevo objeto en lugar de los otros dos. Haga los cambios necesarios para que el resto de los ejemplos que utilicen **Pets.java** sigan funcionando correctamente.

Ejercicio 16: (4) Modifique la jerarquía **Coffee** del Capítulo 15, *Genéricos*, para utilizar jerarquías registradas.

instanceof y equivalencia de clases

Cuando tratamos de extraer información sobre los tipos, existe una diferencia importante entre ambas formas de **instanceof** (es decir, **instanceof** o **isInstance()**, que produce resultados equivalentes) y la comparación directa de los objetos **Class**. He aquí un ejemplo que ilustra la diferencia:

```

//: typeinfo/FamilyVsExactType.java
// La diferencia entre instanceof y los objetos clase
package typeinfo;
import static net.mindview.util.Print.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
    static void test(Object x) {
        print("Testing x of type " + x.getClass());
        print("x instanceof Base " + (x instanceof Base));
```

```

print("x instanceof Derived " + (x instanceof Derived));
print("Base.isInstance(x) " + Base.class.isInstance(x));
print("Derived.isInstance(x) " +
    Derived.class.isInstance(x));
print("x.getClass() == Base.class " +
    (x.getClass() == Base.class));
print("x.getClass() == Derived.class " +
    (x.getClass() == Derived.class));
print("x.getClass().equals(Base.class) " +
    (x.getClass().equals(Base.class)));
print("x.getClass().equals(Derived.class) " +
    (x.getClass().equals(Derived.class)));
}
public static void main(String[] args) {
    test(new Base());
    test(new Derived());
}
/* Output:
Testing x of type class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class) true
x.getClass().equals(Derived.class) false
Testing x of type class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class) false
x.getClass().equals(Derived.class) true
*/

```

El método `test()` realiza una comprobación de tipos con su argumento, utilizando ambas formas de `instanceof`. Después, obtiene la referencia al objeto `Class` y emplea `==` y `equals()` para comprobar la igualdad de los objetos `Class`. Como cabría esperar, `instanceof` e `isInstance()` producen exactamente los mismos resultados, al igual que `equals()` y `==`. Pero las pruebas muestran que se obtienen diferentes conclusiones. Basándose en el concepto de tipos, `instanceof` dice: “¿Perteneces a esta clase o a una clase derivada de ésta?”. Sin embargo, si comparamos los objetos `Class` utilizando `==`, no entran en juego los conceptos de herencia; o son tipos exactamente iguales o no lo son.

Reflexión: información de clases en tiempo de ejecución

Si no conocemos el tipo concreto de un objeto, el mecanismo RTTI nos los dirá. Sin embargo, existe una limitación: el tipo debe ser conocido en tiempo de compilación, para poder detectarlo utilizando RTTI y para poder hacer algo útil con la información. Dicho de otro modo, el compilador debe conocer todas las clases con las que estemos trabajando.

A primera vista, esto no parece que sea una limitación importante, pero suponga que nos entregan una referencia a un objeto que no se encuentra en nuestro espacio de programa. De hecho, suponga que la clase del objeto no está ni siquiera disponible para nuestro programa en tiempo de compilación. Por ejemplo, suponga que extraemos una serie de bytes de un archivo de disco o de una conexión de red, y nos dicen que esos bytes representan una clase. Dado que esta clase aparece después de que el compilador haya generado el código de nuestro programa, ¿cómo podríamos utilizar esta clase?

En un entorno de programación tradicional, este escenario parece un poco futurista. Sin embargo, a medida que nos desplazamos hacia un mundo de programación más amplio, aparecen casos de gran importancia en los que lo que sucede es pre-

cisamente esto. El primero de esos casos es la programación basada en componentes, en la que construimos los proyectos utilizando herramientas RAD (*Rapid Application Development*, desarrollo rápido de aplicaciones) dentro de un *entorno IDE* (*Integrated Development Environment*, entorno integrado de desarrollo), que forma parte de una herramienta de generación de aplicaciones. Se trata de un enfoque visual para la creación de programas, mediante el que se desplazan hasta un formulario una serie de iconos que representan componentes. Estos componentes se configuran entonces estableciendo algunos de sus valores durante el desarrollo. Esta configuración en tiempo de diseño requiere que todos los componentes sean instantiables, que expongan hacia el exterior partes de sí mismos y que permitan que sus propiedades se lean y se modifiquen. Además, los componentes que gestionan sucesos GUI (*Graphical User Interface*) deben exponer la información acerca de los métodos apropiados, de modo que el entorno IDE pueda ayudar al programador a la hora de sustituir dichos métodos de tratamiento de sucesos. La reflexión proporciona el mecanismo para detectar los métodos disponibles y generar los nombres de los métodos. Java proporciona una estructura para la programación basada en componentes mediante JavaBeans (este tema se describe en el Capítulo 22, *Interfaces gráficas de usuario*).

Otra razón importante para descubrir la información de clases en tiempo de ejecución es tener la posibilidad de crear y ejecutar objetos en plataformas remotas, a través de una red. Esto se denomina *invocación remota de métodos* (RMI, *Remote Method Invocation*), y permite a un programa Java tener objetos distribuidos entre muchas máquinas. Esta distribución puede tener lugar por diversas razones. Por ejemplo, quizás estemos realizando una tarea que requiera cálculos intensivos y, para acelerar las cosas, podemos intentar descomponerla y asignar partes del trabajo a las máquinas que estén inactivas. En otras situaciones, puede que queramos colocar el código que gestiona tipos concretos de tareas (por ejemplo, "reglas de negocio" en una arquitectura cliente/servidor multinivel) en una máquina concreta, de modo que la máquina se convierta en un repositorio común que describa dichas acciones y que pueda ser fácilmente modificado para que los cambios afecten a todo el sistema (se trata de un concepto bastante interesante, ya que la máquina existe exclusivamente para facilitar la modificación del software). En la misma línea, la informática distribuida también soporta la utilización de hardware especializado que puede resultar adecuado para una tarea concreta, por ejemplo, en inversiones de matrices, pero inapropiado o demasiado caro para la programación de propósito general.

La clase **Class** soporta el concepto de *reflexión*, junto con la biblioteca `java.lang.reflect` que contiene las clases **Field**, **Method** y **Constructor** (cada una de las cuales implementa la interfaz **Member**). Los objetos de estos tipos son creados por la máquina JVM en tiempo de ejecución para representar el miembro correspondiente de la clase desconocida. Entonces, podemos utilizar los objetos **Constructor** (constructores) para crear nuevos objetos, los métodos `get()` y `set()` para leer y modificar los campos asociados con los objetos **Field** y el método `invoke()` para invocar un método asociado con un objeto **Method**. Además, podemos invocar los métodos de utilidad `getFields()`, `getMethods()`, `getConstructors()`, etc., con el fin de obtener como resultado matrices de objetos que representen los campos, métodos y constructores (puede averiguar más detalles examinando la clase **Class** en la documentación del JDK). Así, la información de clase para objetos anónimos puede determinarse completamente en tiempo de ejecución y no es necesario tener ninguna información en tiempo de compilación.

Es importante comprender que no hay ninguna especie de mecanismo mágico en la reflexión. Cuando se utiliza la reflexión para interactuar con un objeto de un tipo desconocido, la máquina JVM simplemente examinará el objeto y comprobará que pertenece a una clase concreta (al igual que con el mecanismo RTTI normal). Antes de poder hacer nada con él, es necesario cargar el objeto **Class**. Por tanto, el archivo `.class` para ese tipo concreto deberá seguir estando disponible para la JVM, bien en la máquina local o a través de la red. Por tanto, la verdadera diferencia entre RTTI y la reflexión es que, con la RTTI, el compilador abre y examina el archivo `.class` en tiempo de compilación. Dicho de otra forma, podemos invocar todos los métodos de un objeto de la forma "normal". Con el mecanismo de reflexión, el archivo `.class` no está disponible en tiempo de compilación, sino que el que lo abre y examina es el entorno de tiempo de ejecución.

Un extractor de métodos de clases

Normalmente, no vamos a necesitar utilizar las herramientas de reflexión directamente, pero si que pueden resultar útiles cuando necesitemos crear código más dinámico. La reflexión se ha incluido en el lenguaje para soportar otras características de Java, como la serialización de objetos y JavaBeans (ambos temas se tratan posteriormente en el libro). Sin embargo, hay ocasiones en las que resulta muy útil extraer dinámicamente la información acerca de una clase.

Consideremos el caso de un extractor de métodos de clases. Examinando el código fuente de la definición de una clase o la documentación del JDK, sólo podemos conocer los métodos definidos o sustituidos *dentro de dicha definición de clase*. Pero puede haber otra docena de métodos disponibles que procedan de las clases base. Localizar estos métodos es muy tedioso

y requiere mucho tiempo¹. Afortunadamente, el mecanismo de reflexión proporciona una forma de escribir una herramienta simple que nos muestre automáticamente la interfaz completa. He aquí la forma en que funciona:

```

//: typeinfo>ShowMethods.java
// Utilización de la reflexión para mostrar todos los métodos de una clase,
// incluso aunque los métodos estén definidos en la clase base.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class ShowMethods {
    private static String usage =
        "usage:\n" +
        "ShowMethods qualified.class.name\n" +
        "To show all methods in class or:\n" +
        "ShowMethods qualified.class.name word\n" +
        "To search for methods involving 'word'";
    private static Pattern p = Pattern.compile("\\w+\\w+");
    public static void main(String[] args) {
        if(args.length < 1) {
            print(usage);
            System.exit(0);
        }
        int lines = 0;
        try {
            Class<?> c = Class.forName(args[0]);
            Method[] methods = c.getMethods();
            Constructor[] ctors = c.getConstructors();
            if(args.length == 1) {
                for(Method method : methods)
                    print(
                        p.matcher(method.toString()).replaceAll(""));
                for(Constructor ctor : ctors)
                    print(p.matcher(ctor.toString()).replaceAll(""));
                lines = methods.length + ctors.length;
            } else {
                for(Method method : methods)
                    if(method.toString().indexOf(args[1]) != -1) {
                        print(
                            p.matcher(method.toString()).replaceAll(""));
                        lines++;
                    }
                for(Constructor ctor : ctors)
                    if(ctor.toString().indexOf(args[1]) != -1) {
                        print(p.matcher(
                            ctor.toString()).replaceAll ""));
                        lines++;
                    }
            }
        } catch(ClassNotFoundException e) {
            print("No such class: " + e);
        }
    }
}
```

¹ Especialmente en el pasado. Sin embargo, Sun ha mejorado enormemente su documentación HTML sobre Java, por lo que ahora es más fácil consultar los métodos de las clases base.

```

} /* Output:
public static void main(String[])
public native int hashCode()
public final native Class getClass()
public final void wait(long,int) throws InterruptedException
public final void wait() throws InterruptedException
public final native void wait(long) throws InterruptedException
public boolean equals(Object)
public String toString()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
*///:~

```

Los métodos `getMethods()` y `getConstructors()` de `Class` devuelven una matriz de tipo `Method` y una matriz de tipo `Constructor`, respectivamente. Cada una de estas clases tiene métodos adicionales para disecionar los nombres, argumentos y valores de retorno de los métodos que representan. Pero también podemos utilizar `toString()`, como se hace en el ejemplo, para producir una cadena de caracteres con toda la firma del método. El resto del código extrae la información de la línea de comandos, determina si una firma concreta se corresponde con la cadena buscada (utilizando `indexOf()`) y elimina los cualificadores de los nombres utilizando expresiones regulares (presentadas en el Capítulo 13, *Cadenas de caracteres*).

El resultado producido por `Class.forName()` no puede ser conocido en tiempo de compilación, y por tanto toda la información de firmas de métodos se está extrayendo en tiempo de ejecución. Si analiza la documentación del JDK sobre el mecanismo de reflexión, verá que existe el suficiente soporte como para poder realizar una invocación de un método sobre un objeto que sea totalmente desconocido en tiempo de compilación (más adelante en el libro se proporcionan ejemplos de esto). Aunque inicialmente pueda parecer que no vamos a llegar nunca a necesitar esta funcionalidad, el valor de los mecanismos de reflexión puede resultar ciertamente sorprendente.

La salida anterior se genera mediante la línea de comandos:

```
java ShowMethods ShowMethods
```

Puede ver que la salida incluye un constructor predeterminado público, aún cuando no se haya definido ningún constructor. El constructor que vemos es el que el compilador sintetiza de forma automática. Si luego ejecutamos `ShowMethods` con una clase no pública (es decir, acceso de paquete), el constructor predeterminado sintetizado no aparecerá en la salida. El constructor predeterminado sintetizado recibe automáticamente el mismo acceso que la clase.

Otro experimento interesante consiste en invocar `java ShowMethods java.lang.String` con un argumento adicional de tipo `char`, `int`, `String`, etc.

Esta herramienta puede ahorrar mucho tiempo mientras programamos, en aquellos casos en los que no recordemos si una clase dispone de un método concreto y no tengamos ganas de examinar el índice o la jerarquía de clases en la documentación del JDK, o bien si no sabemos, por ejemplo, si dicha clase puede hacer algo con, por ejemplo, objetos de tipo `Color`.

El Capítulo 22, *Interfaces gráficas de usuario*, contiene una versión GUI de este programa (personalizada para extraer información para componentes Swing), por lo que puede dejar ese programa ejecutándose mientras esté escribiendo código para poder realizar búsquedas rápidas.

Ejercicio 17: (2) Modifique la expresión regular de `ShowMethods.java` para eliminar también las palabras clave `native` y `final` (consejo: utilice el operador OR '|').

Ejercicio 18: (1) Defina `ShowMethods` como una clase no pública y verifique que el constructor predeterminado sintetizado no aparece a la salida.

Ejercicio 19: (4) En `ToyTest.java`, utilice la reflexión para crear un objeto `Toy` utilizando el constructor no predeterminado.

Ejercicio 20: (5) Examine la interfaz de `java.lang.Class` en la documentación del JDK que podrá encontrar en <http://java.sun.com>. Escriba un programa que tome el nombre de una clase como un argumento de la línea de comandos, y luego utilice los métodos `Class` para volcar toda la información disponible para esa clase. Compruebe el programa con una clase de la biblioteca estándar y con una clase que usted mismo defina.

Proxies dinámicos

El patrón de diseño *Proxy* es uno de los patrones de diseño básicos. Se trata de un objeto que insertamos en lugar del objeto "real" para proporcionar operaciones adicionales o diferentes, estos objetos normalmente se comunican con un objeto "real", de manera que un *proxy* actúa típicamente como un intermediario. He aquí un ejemplo trivial para mostrar la estructura de un *proxy*:

```
//: typeinfo/SimpleProxyDemo.java
import static net.mindview.util.Print.*;

interface Interface {
    void doSomething();
    void somethingElse(String arg);
}

class RealObject implements Interface {
    public void doSomething() { print("doSomething"); }
    public void somethingElse(String arg) {
        print("somethingElse " + arg);
    }
}

class SimpleProxy implements Interface {
    private Interface proxied;
    public SimpleProxy(Interface proxied) {
        this.proxied = proxied;
    }
    public void doSomething() {
        print("SimpleProxy doSomething");
        proxied.doSomething();
    }
    public void somethingElse(String arg) {
        print("SimpleProxy somethingElse " + arg);
        proxied.somethingElse(arg);
    }
}

class SimpleProxyDemo {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        consumer(new RealObject());
        consumer(new SimpleProxy(new RealObject()));
    }
} /* Output:
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo.
*///:-
```

Puesto que `consumer()` acepta una **Interface**, no puede saber si está conteniendo un objeto real **RealObject** o un **Proxy**, porque ambos implementan **Interface**. Pero el **Proxy**, que se ha insertado entre el cliente y el objeto **RealObject**, realiza operaciones y luego invoca el método idéntico de **RealObject**.

Un proxy puede ser útil siempre que queramos incluir operaciones adicionales en un lugar distinto que el propio "objeto real", y especialmente cuando queramos poder cambiar fácilmente entre una situación en la que se usen esas operaciones adicionales y otra en la que no se empleen, y viceversa (el objeto de utilizar patrones de diseño consiste en encapsular los cambios, así que sólo si se tienen que efectuar modificaciones para justificar el uso de un patrón). Por ejemplo, ¿qué sucede si quisieramos controlar las llamadas a los métodos del objeto **RealObject**, o medir la carga de procesamiento asociada a dichas llamadas? Este tipo de código no conviene incorporarlo en la aplicación, por lo que un *proxy* nos permite añadirlo y eliminarlo fácilmente.

El concepto *proxy dinámico* de Java lleva el concepto de *proxy* un paso más allá, tanto porque crea el objeto dinámicamente como porque gestiona dinámicamente las llamadas a los métodos para los cuales hemos insertado un *proxy*. Todas las llamadas realizadas a un *proxy* dinámico se redirigen a un único *gestor de invocaciones*, cuya tarea consiste en descubrir qué es cada llamada y en decidir qué hacer con ella. He aquí el programa **SimpleProxyDemo.java** reescrito para utilizar un *proxy* dinámico:

```
//: typeinfo/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
    private Object proxied;
    public DynamicProxyHandler(Object proxied) {
        this.proxied = proxied;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        System.out.println("**** proxy: " + proxy.getClass() +
            ", method: " + method + ", args: " + args);
        if(args != null)
            for(Object arg : args)
                System.out.println(" " + arg);
        return method.invoke(proxied, args);
    }
}

class SimpleDynamicProxy {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        RealObject real = new RealObject();
        consumer(real);
        // Insertar un proxy y llamar de nuevo;
        Interface proxy = (Interface)Proxy.newProxyInstance(
            Interface.class.getClassLoader(),
            new Class[]{ Interface.class },
            new DynamicProxyHandler(real));
        consumer(proxy);
    }
} /* Output: (95% match)
doSomething
somethingElse bonobo
**** proxy: class $Proxy0, method: public abstract void
Interface.doSomething(), args: null
doSomething
**** proxy: class $Proxy0, method: public abstract void
Interface.somethingElse(java.lang.String), args:
[Ljava.lang.Object;@42e816
bonobo
```

```
somethingElse bonobo
*///:-
```

Para crear un *proxy* dinámico se invoca el método estático **Proxy.newProxyInstance()**, que requiere un cargador de clases (generalmente, podemos pasarle un cargador de clases de un objeto que ya haya sido cargado), una lista de interfaces (no clases ni clases abstractas) que queramos que el *proxy* implemente y una implementación de la interfaz **InvocationHandler** (gestor de invocaciones). El *proxy* dinámico redirigirá todas las llamadas al gestor de invocaciones, de modo que al constructor para el gestor de invocaciones usualmente se le entrega la referencia al objeto "real" para que pueda redirigirle las solicitudes una vez que haya terminado de llevar a cabo su tarea intermediaria.

Al método **invoke()** se le pasa el objeto *proxy*, en caso de que necesitemos distinguir de dónde viene la solicitud, (aunque en muchos casos esto no nos preocupará). Sin embargo, tenga cuidado cuando invoque métodos del *proxy* dentro de **invoke()**, porque las llamadas a través de la interfaz se redirigen a través del *proxy*.

En general, lo que haremos será realizar la operación intermediario y luego usar **Method.invoke()** para redirigir la solicitud hacia el objeto real pasándole los argumentos necesarios. Puede que esto parezca a primera vista algo limitado, como si sólo se pudieran realizar operaciones genéricas. Sin embargo, podemos filtrar ciertas llamadas a métodos, dejando pasar las otras directamente:

```
//: typeinfo>SelectingMethods.java
// Búsqueda de métodos concretos en un proxy dinámico.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

class MethodSelector implements InvocationHandler {
    private Object proxied;
    public MethodSelector(Object proxied) {
        this.proxied = proxied;
    }
    public Object
    invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
        if(method.getName().equals("interesting"))
            print("Proxy detected the interesting method");
        return method.invoke(proxied, args);
    }
}

interface SomeMethods {
    void boring1();
    void boring2();
    void interesting(String arg);
    void boring3();
}

class Implementation implements SomeMethods {
    public void boring1() { print("boring1"); }
    public void boring2() { print("boring2"); }
    public void interesting(String arg) {
        print("interesting " + arg);
    }
    public void boring3() { print("boring3"); }
}

class SelectingMethods {
    public static void main(String[] args) {
        SomeMethods proxy= (SomeMethods) Proxy.newProxyInstance(
            SomeMethods.class.getClassLoader(),
            new Class[]{ SomeMethods.class },
            new MethodSelector(new Implementation()));
    }
}
```

```

proxy.boring1();
proxy.boring2();
proxy.interesting("bonobo");
proxy.boring3();
}
} /* Output:
boring1
boring2
Proxy detected the interesting method
interesting bonobo
boring3
*///:-

```

Aquí, simplemente examinamos los nombres de los métodos, pero también podríamos examinar los aspectos de la firma del método, incluso podríamos buscar valores concretos de los argumentos.

El proxy dinámico no es una herramienta para utilizarla todos los días, pero permite resolver ciertos tipos de problemas muy elegantemente. Puede obtener más detalles acerca del patrón de diseño *Proxy* y de otros patrones de diseño en *Thinking in Patterns* (véase www.MindView.net) y *Design Patterns*, de Erich Gamma *et al.* (Addison-Wesley, 1995).

Ejercicio 21: (3) Modifique **SimpleProxyDemo.java** para que mida los tiempos de llamada a los métodos.

Ejercicio 22: (3) Modifique **SimpleDynamicProxy.java** para que mida los tiempos de llamada a los métodos.

Ejercicio 23: (3) Dentro de **invoke()** en **SimpleDynamicProxy.java**, trate de imprimir el argumento **proxy** y explique lo que sucede.

Proyecto:² Escriba un sistema utilizando *proxies* dinámicos para implementar *transacciones*, donde el *proxy* se encargue de *confirmar la transacción* si la llamada realizada al objeto real tiene éxito (no genera ninguna excepción), debiendo *anular la transacción* si la llamada falla. La confirmación y anulación deben funcionar como un archivo de texto externo, que se encuentra fuera del control de las excepciones Java. Tendrá que prestar atención a la *atomicidad* de las operaciones.

Objetos nulos

Cuando se utiliza el valor predeterminado **null** para indicar la ausencia de un objeto, es preciso comprobar si las referencias son iguales a **null** cada vez que se utiliza. Esta labor puede llegar a ser muy tediosa y el código resultante es muy complejo. El problema es que **null** no tiene ningún comportamiento propio, salvo generar una excepción **NullPointerException** si se intenta hacer algo con el valor. Algunas veces, resulta útil introducir la idea de un *objeto nulo*³, que aceptará mensajes en lugar del objeto al cual "representa", pero que devolverá valores indicando que no existe ahí ningún objeto "real". De esta forma, podemos asumir que todos los objetos son válidos y no tenemos por qué desperdiciar tiempo de programación comprobando la igualdad con **null** (y leyendo el código resultante).

Aunque resulta divertido imaginarse un lenguaje de programación que cree automáticamente objetos nulos por nosotros, en la práctica no tiene sentido usarlos en todas partes; en ocasiones, será adecuado realizar las comprobaciones de valor **null**, en otros casos podremos asumir razonablemente que no vamos a encontrarnos con el valor **null**, e incluso, en otras ocasiones, será perfectamente aceptable detectar las aberraciones a través de **NullPointerException**. El lugar donde los objetos nulos parecen ser más útiles es en "el lugar más próximo a los datos", con objetos que representen entidades en el espacio del problema. Como ejemplo simple, muchos sistemas dispondrán de una clase **Person**, y hay situaciones en el código en las que no disponemos de una persona real (o sí disponemos de ella, pero no tenemos todavía toda la información acerca de dicha persona), por lo que tradicionalmente utilizaremos una referencia **null** y comprobaremos si las referencias son nulas. En lugar de ello, podemos crear un objeto nulo, pero aún cuando el objeto nulo responderá a todos los mensajes.

² Los proyectos son sugerencias que pueden utilizarse, por ejemplo, como trabajos de clase. Las soluciones a los proyectos no se incluyen en la guía de soluciones.

³ Descubierto por Bobby Woolf y Bruce Anderson. Puede verse como un caso especial del patrón de diseño basado en *estrategia*. Una variante del *Objeto Nulo* es el patrón de diseño de *Iterador Nulo*, que hace que la interacción a través de los nodos en una jerarquía compuesta sea transparente para el cliente (el cliente puede entonces utilizar la misma lógica para iterar a través de la jerarquía compuesta y a través de los nodos hoja).

a los que el objeto “real” respondería, sigue siendo necesario disponer de una manera de comprobar si hay valores nulos. La forma más simple de hacer esto consiste en crear una interfaz de marcado:

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} //:-
```

```
//: typeinfo/Person.java
// Una clase con un objeto Null.
import net.mindview.util.*;

class Person {
    public final String first;
    public final String last;
    public final String address;
    // etc.
    public Person(String first, String last, String address) {
        this.first = first;
        this.last = last;
        this.address = address;
    }
    public String toString() {
        return "Person: " + first + " " + last + " " + address;
    }
    public static class NullPerson
    extends Person implements Null {
        private NullPerson() { super("None", "None", "None"); }
        public String toString() { return "NullPerson"; }
    }
    public static final Person NULL = new NullPerson();
} //:-
```

En general, el objeto nulo será un objeto simple (no una serie de objetos agrupados en contenedores), por lo que aquí se crea como una instancia estática final. Esto funciona porque **Person** es *immutable*: sólo podemos fijar los valores en el constructor, y luego leer dichos valores, pero no modificarlos (porque los propios **String** son inherentemente inmutables). Si deseas modificar un objeto **NullPerson**, sólo puedes sustituirle con un nuevo objeto **Person**. Observe que disponemos de la opción de detectar el genérico **Null** o el más específico **NullPerson** utilizando **instanceof**, pero como se trata de un valor simple también podemos utilizar **equals()** o incluso **==** para comparar con **Person.NULL**.

Ahora suponga que estuviéramos en la época dorada de las empresas de Internet y que alguien hubiera invertido una gran cantidad de dinero en una maravillosa idea que hubiéramos tenido. Imagine que estamos listos para reclutar personal pero que, mientras esperamos a que las vacantes sean cubiertas, podemos utilizar objetos nulos **Person** para asignarlos a cada puesto de trabajo (**Position**):

```
//: typeinfo/Position.java

class Position {
    private String title;
    private Person person;
    public Position(String jobTitle, Person employee) {
        title = jobTitle;
        person = employee;
        if(person == null)
            person = Person.NULL;
    }
    public Position(String jobTitle) {
        title = jobTitle;
```

```

    person = Person.NULL;
}
public String getTitle() { return title; }
public void setTitle(String newTitle) {
    title = newTitle;
}
public Person getPerson() { return person; }
public void setPerson(Person newPerson) {
    person = newPerson;
    if(person == null)
        person = Person.NULL;
}
public String toString() {
    return "Position: " + title + " " + person;
}
} //:-

```

Con **Position**, no tenemos necesidad de crear un objeto nulo, porque la existencia de **Person.NULL** implica un objeto **Position** nulo (es posible que más adelante descubramos que si se necesita añadir un objeto nulo explícito para **Position**, pero hay una regla que dice que siempre debemos implementar la solución más simple que funcione en nuestro primer diseño, y esperar a que algún aspecto del programa requiera que añadamos la característica adicional, en lugar de asumir desde el principio que esa característica es necesaria).⁴

La clase **Staff** puede ahora buscar los objetos nulos a la hora de llenar las vacantes:

```

//: typeinfo/Staff.java
import java.util.*;

public class Staff extends ArrayList<Position> {
    public void add(String title, Person person) {
        add(new Position(title, person));
    }
    public void add(String... titles) {
        for(String title : titles)
            add(new Position(title));
    }
    public Staff(String... titles) { add(titles); }
    public boolean positionAvailable(String title) {
        for(Position position : this)
            if(position.getTitle().equals(title) &&
                position.getPerson() == Person.NULL)
                return true;
        return false;
    }
    public void fillPosition(String title, Person hire) {
        for(Position position : this)
            if(position.getTitle().equals(title) &&
                position.getPerson() == Person.NULL) {
                position.setPerson(hire);
                return;
            }
        throw new RuntimeException(
            "Position " + title + " not available");
    }
    public static void main(String[] args) {
        Staff staff = new Staff("President", "CTO",
            "Marketing Manager", "Product Manager",
            "Project Lead", "Software Engineer",

```

⁴ Esta tendencia a implementar la solución más simple posible es una de las recomendaciones de *Extreme Programming* (XP).

```

        "Software Engineer", "Software Engineer",
        "Software Engineer", "Test Engineer",
        "Technical Writer");
staff.fillPosition("President",
    new Person("Me", "Last", "The Top, Lonely At"));
staff.fillPosition("Project Lead",
    new Person("Janet", "Planner", "The Burbs"));
if(staff.positionAvailable("Software Engineer"))
    staff.fillPosition("Software Engineer",
        new Person("Bob", "Coder", "Bright Light City"));
System.out.println(staff);
}
} /* Output:
[Position: President Person: Me Last The Top, Lonely At, Position: CTO NullPerson,
Position: Marketing Manager NullPerson, Position: Product Manager NullPerson, Position:
Project Lead Person: Janet Planner The Burbs, Position: Software Engineer Person: Bob
Coder Bright Light City, Position: Software Engineer NullPerson, Position: Software
Engineer NullPerson, Position: Software Engineer NullPerson, Position: Test Engineer
NullPerson, Position: Technical Writer NullPerson]
*///:-
```

Observe que sigue siendo necesario comprobar la existencia de objetos nulos en algunos lugares, lo cual no difiere mucho de comprobar la igualdad con el valor **null**, pero en otros lugares (como en las conversiones **toString()**, en este caso), no es necesario realizar comprobaciones adicionales; podemos limitarnos a asumir que todas las referencias a objetos son válidas.

Si estamos trabajando con interfaces en lugar de con clases concretas, es posible utilizar un objeto **DynamicProxy** para crear automáticamente los objetos nulos. Suponga que tenemos una interfaz **Robot** que define un nombre, un modelo y una lista **List<Operation>** que describe lo que el **Robot** es capaz de hacer. **Operation** contiene una descripción y un comando (es un tipo del patrón de diseño basado en comandos):

```

//: typeinfo/Operation.java

public interface Operation {
    String description();
    void command();
} /*:-
```

Podemos acceder a los servicios de un objeto **Robot** invocando **operations()**:

```

//: typeinfo/Robot.java
import java.util.*;
import net.mindview.util.*;

public interface Robot {
    String name();
    String model();
    List<Operation> operations();
    class Test {
        public static void test(Robot r) {
            if(r instanceof Null)
                System.out.println("[Null Robot]");
            System.out.println("Robot name: " + r.name());
            System.out.println("Robot model: " + r.model());
            for(Operation operation : r.operations()) {
                System.out.println(operation.description());
                operation.command();
            }
        }
    }
} /*:-
```

Esto incorpora también una clase anidada para realizar las pruebas.

Ahora podemos crear un objeto **Robot** especializado:

```
//: typeinfo/SnowRemovalRobot.java
import java.util.*;

public class SnowRemovalRobot implements Robot {
    private String name;
    public SnowRemovalRobot(String name) { this.name = name; }
    public String name() { return name; }
    public String model() { return "SnowBot Series 11"; }
    public List<Operation> operations() {
        return Arrays.asList(
            new Operation() {
                public String description() {
                    return name + " can shovel snow";
                }
                public void command() {
                    System.out.println(name + " shoveling snow");
                }
            },
            new Operation() {
                public String description() {
                    return name + " can chip ice";
                }
                public void command() {
                    System.out.println(name + " chipping ice");
                }
            },
            new Operation() {
                public String description() {
                    return name + " can clear the roof";
                }
                public void command() {
                    System.out.println(name + " clearing roof");
                }
            }
        );
    }
    public static void main(String[] args) {
        Robot.Test.test(new SnowRemovalRobot("Slusher"));
    }
} /* Output:
Robot name: Slusher
Robot model: SnowBot Series 11
Slusher can shovel snow
Slusher shoveling snow
Slusher can chip ice
Slusher chipping ice
Slusher can clear the roof
Slusher clearing roof
*///:-
```

Existirán presumiblemente muchos tipos diferentes de objetos **Robot**, y lo que nos gustaría es que cada objeto nulo hiciera algo especial para cada tipo de **Robot**; en este caso, incorporar información acerca del tipo exacto de **Robot** que el objeto nulo representa. Esta información será capturada por el proxy dinámico:

```
//: typeinfo/NullRobot.java
// Utilización de un proxy dinámico para crear un objeto nulo,
```

```

import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class NullRobotProxyHandler implements InvocationHandler {
    private String nullName;
    private Robot proxied = new NRobot();
    NullRobotProxyHandler(Class<? extends Robot> type) {
        nullName = type.getSimpleName() + " NullRobot";
    }
    private class NRobot implements Null, Robot {
        public String name() { return nullName; }
        public String model() { return nullName; }
        public List<Operation> operations() {
            return Collections.emptyList();
        }
    }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        return method.invoke(proxied, args);
    }
}

public class NullRobot {
    public static Robot
        newNullRobot(Class<? extends Robot> type) {
        return (Robot) Proxy.newProxyInstance(
            NullRobot.class.getClassLoader(),
            new Class[]{ Null.class, Robot.class },
            new NullRobotProxyHandler(type));
    }
    public static void main(String[] args) {
        Robot[] bots = {
            new SnowRemovalRobot("SnowBee"),
            newNullRobot(SnowRemovalRobot.class)
        };
        for(Robot bot : bots)
            Robot.Test.test(bot);
    }
} /* Output:
Robot name: SnowBee
Robot model: SnowBot Series II
SnowBee can shovel snow
SnowBee shoveling snow
SnowBee can chip ice
SnowBee chipping ice
SnowBee can clear the roof
SnowBee clearing roof
[Null Robot]
Robot name: SnowRemovalRobot NullRobot
Robot model: SnowRemovalRobot NullRobot
*///:-
```

Cuando se necesita un objeto **Robot** nulo, simplemente se invoca **newNullRobot()**, pasándole al método el tipo de **Robot** para el que queremos que actúe como *proxy*. El *proxy* satisface los requisitos de las interfaces **Robot** y **Null**, y proporciona el nombre específico para el que actúa como *proxy*.

Objetos maqueta y *stubs*

Hay dos tipos variantes del objeto nulo: el *objeto maqueta* y el *stub*. Al igual que el objeto nulo, ambos tipos de objetos se utilizan en lugar del objeto "real" que empleará el programa terminado. Sin embargo, tanto el objeto maqueta como el *stub* pretenden ser objetos vivos que entregan información real en lugar de ser un sustituto un poco más inteligente de `null`, como es el caso del objeto nulo.

La diferencia entre el objeto maqueta y un *stub* es bastante sutil. Los objetos maqueta tienden a ser ligeros (poco complejos) y tienen capacidad de auto-comprobación, y usualmente se crean muchos de ellos para gestionar distintas situaciones de prueba. Los *stubs* son típicamente más pesados y a menudo se reutilizan entre una prueba y otra. Los *stubs* pueden configurarse para cambiar de comportamiento, dependiendo de cómo se los invoque. Por tanto, un *stub* es un objeto sofisticado que lleva a cabo una de esas tareas; mientras que para hacer esas mismas tareas con objetos maqueta lo que normalmente haríamos es crear muchos objetos maqueta pequeños y simples.

Ejercicio 24: (4) Añada objetos nulos a `RegisteredFactories.java`.

Interfaces e información de tipos

Un objetivo clave de la palabra clave **interface** es permitir al programador aislar componentes, reduciendo así el acoplamiento. Si escribimos el código basándonos en interfaces, conseguimos este objetivo, pero con la información de tipos es posible saltarse los controles; las interfaces no son una garantía de desacoplamiento. He aquí un ejemplo comenzando con una interfaz:

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;

public interface A {
    void f();
} /*:-
```

Esta interfaz se implementa a continuación y podemos ver fácilmente cómo saltarnos los controles para obtener el tipo real de implementación:

```
//: typeinfo/InterfaceViolation.java
// Sorteando una interfaz.
import typeinfo.interfacea.*;

class B implements A {
    public void f() {}
    public void g() {}
}

public class InterfaceViolation {
    public static void main(String[] args) {
        A a = new B();
        a.f();
        // a.g(); // Error de compilación
        System.out.println(a.getClass().getName());
        if(a instanceof B) {
            B b = (B)a;
            b.g();
        }
    }
} /* Output:
B
*///:-
```

Utilizando RTTI, descubrimos que `a` ha sido implementado como `B`. Proyectando sobre `B`, podemos invocar un método que no se encuentre en `A`.

Esto es perfectamente legal y aceptable, pero puede que no queramos que los programadores de clientes hagan esto, ya que esto les da la oportunidad para acoplarse más estrechamente con nuestro código de lo que queríamos. En otras palabras, podríamos pensar que la palabra clave **interface** nos está protegiendo, pero en realidad no es así, y el hecho de que utilicemos **B** para implementar **A** en este caso es algo de dominio público.⁵

Una solución consiste simplemente en decir que los programadores serán los responsables si deciden utilizar la clase real en lugar de la interfaz. Esto es probablemente razonable en muchos casos, pero si ese “probablemente” no es suficiente, conviene aplicar otros controles más estrictos.

La técnica más sencilla consiste en utilizar acceso de paquete para la implementación, de modo que los clientes situados fuera del paquete no puedan verla:

```
//: typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class C implements A {
    public void f() { print("public C.f()"); }
    public void g() { print("public C.g()"); }
    void u() { print("package C.u()"); }
    protected void v() { print("protected C.v()"); }
    private void w() { print("private C.w()"); }
}

public class HiddenC {
    public static A makeA() { return new C(); }
} //:-
```

La única parte pública de este paquete, **HiddenC**, produce una interfaz **A** cuando se la invoca. Lo que es interesante acerca de este ejemplo es que incluso si devolviéramos un objeto **C** desde **makeA()**, seguiríamos sin poder utilizar ninguna otra cosa distinta de **A** desde fuera del paquete, ya que no podemos nombrar **C** fuera del paquete.

Ahora, si tratamos de efectuar una especialización sobre **C**, no podemos hacerlo, porque no hay ningún tipo ‘**C**’ disponible fuera del paquete:

```
//: typeinfo/HiddenImplementation.java
// Sorteando el acceso de paquete.
import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
    public static void main(String[] args) throws Exception {
        A a = HiddenC.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Error de compilación: no se puede encontrar el símbolo 'C':
        /* if(a instanceof C) {
            C c = (C)a;
            c.g();
        }*/
        // ¡Caramba! La reflexión nos permite invocar g():
        callHiddenMethod(a, "g");
        // ¡E incluso métodos que son menos accesibles!
```

⁵ El caso más famoso es el sistema operativo Windows, que tenía una API pública con la que se suponía que había que desarrollar programas y un conjunto no publicado pero visible de funciones que podíamos descubrir e invocar. Para resolver los problemas, los programadores utilizaban las funciones ocultas de la API, lo que forzó a Microsoft a mantenerlas como si fueran parte de la API pública. Esto se convirtió en una fuente de grandes costes y de enorme trabajo para la empresa.

```

callHiddenMethod(a, "u");
callHiddenMethod(a, "v");
callHiddenMethod(a, "w");
}
static void callHiddenMethod(Object a, String methodName)
throws Exception {
    Method g = a.getClass().getDeclaredMethod(methodName);
    g.setAccessible(true);
    g.invoke(a);
}
/* Output:
public C.f()
typeinfo.packageaccess.C
public C.g()
package C.u()
protected C.v()
private C.w()
*///:-

```

Como puede ver, sigue siendo posible meterse en las entrañas e invocar *todos* los métodos utilizando el mecanismo de reflexión. ¡Incluso los métodos privados! Si se conoce el nombre del método, se puede invocar **setAccessible(true)** sobre el objeto **Method** para hacerlo invocable, como podemos ver en **callHiddenMethod()**.

Podriamos pensar que es posible impedir esto distribuyendo sólo el código compilado, pero no es una solución. Basta con ejecutar **javap**, que es el descompilador incluido en el JDK. He aquí la línea de comandos necesaria:

```
javap -private C
```

El indicador **-private** especifica que deben mostrarse todos los miembros, incluso los privados. He aquí la salida que se obtiene:

```

class typeinfo.packageaccess.C extends
java.lang.Object implements typeinfo.interfacea.A {
    typeinfo.packageaccess.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}

```

Por tanto, cualquiera puede obtener los nombres y las firmas de los métodos más privados e invocarlos.

¿Qué sucede si implementamos la interfaz con una clase interna privada? He aquí un ejemplo:

```

//: typeinfo/InnerImplementation.java
// Las clases internas privadas no pueden ocultarse del mecanismo de
// reflexión.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class InnerA {
    private static class C implements A {
        public void f() { print("public C.f()"); }
        public void g() { print("public C.g()"); }
        void u() { print("package C.u()"); }
        protected void v() { print("protected C.v()"); }
        private void w() { print("private C.w()"); }
    }
    public static A makeA() { return new C(); }
}

```

```

public class InnerImplementation {
    public static void main(String[] args) throws Exception {
        A a = InnerA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // La reflexión sigue permitiendo entrar en la clase privada:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
} /* Output:
public C.f()
InnerA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*///:-
```

Esta solución no nos ha permitido ocultar nada a ojos del mecanismo de reflexión. ¿Qué sucedería con una clase anónima?

```

//: typeinfo/AnonymousImplementation.java
// Las clases internas anónimas no pueden ocultarse del mecanismo de
// reflexión.
import typeinfo.interfaces.*;
import static net.mindview.util.Print.*;

class AnonymousA {
    public static A makeA() {
        return new A() {
            public void f() { print("public C.f()"); }
            public void g() { print("public C.g()"); }
            void u() { print("package C.u()"); }
            protected void v() { print("protected C.v()"); }
            private void w() { print("private C.w()"); }
        };
    }
}

public class AnonymousImplementation {
    public static void main(String[] args) throws Exception {
        A a = AnonymousA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // La reflexión sigue pudiendo entrar en la clase anónima:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
} /* Output:
public C.f()
AnonymousA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*///:-
```

Parece que no existe ninguna forma de impedir que el mecanismo de reflexión entre e invoque los métodos que no tienen acceso público. Esto también se cumple para los campos, incluso para los campos privados:

```
//: typeinfo/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
    private int i = 1;
    private final String s = "I'm totally safe";
    private String s2 = "Am I safe?";
    public String toString() {
        return "i = " + i + ", " + s + ", " + s2;
    }
}

public class ModifyingPrivateFields {
    public static void main(String[] args) throws Exception {
        WithPrivateFinalField pf = new WithPrivateFinalField();
        System.out.println(pf);
        Field f = pf.getClass().getDeclaredField("i");
        f.setAccessible(true);
        System.out.println("f.getInt(pf): " + f.getInt(pf));
        f.setInt(pf, 47);
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s2");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
    }
} /* Output:
i = 1, I'm totally safe, Am I safe?
f.getInt(pf): 1
i = 47, I'm totally safe, Am I safe?
f.get(pf): I'm totally safe
i = 47, I'm totally safe, Am I safe?
f.get(pf): Am I safe?
i = 47, I'm totally safe, No, you're not!
*///:~
```

Sin embargo, los campos de tipo **final** si que están protegidos frente a los cambios. El sistema de tiempo de ejecución acepta los intentos de cambio sin quejarse, pero no se produce cambio alguno.

En general, todas estas violaciones de acceso no constituyen un problema grave. Si alguien utiliza una de estas técnicas para invocar métodos que se han marcado como privados o con acceso de paquete (lo cual indica claramente que no deberían invocarse), entonces es difícil que esas personas puedan quejarse si decidimos cambiar posteriormente algunos aspectos de esos métodos. Por otro lado, el hecho de que siempre exista una puerta trasera para entrar en una clase nos permite resolver ciertos tipos de problemas que en otro caso serían difíciles o imposibles, y los beneficios del mecanismo de reflexión son, por regla general, incuestionables.

Ejercicio 25: (2) Defina una clase que contenga métodos privados, protegidos y con acceso de paquete. Escriba código para acceder a dichos métodos desde fuera del paquete de la clase.

Resumen

RTTI nos permite descubrir la información de tipos a partir de una referencia anónima a una clase base. Es por ello que se presta a una inadecuada utilización por los usuarios menos expertos, ya que resulta más fácil de comprender que las llamadas polimórficas a métodos. Para las personas que tienen experiencia previa en lenguajes procedimentales, resulta difícil organizar los programas en conjuntos de instrucciones `switch`. Este tipo de estructura puede implementarse fácilmente con RTTI perdiéndose así el importante valor que el polimorfismo añade al desarrollo y el mantenimiento del código. La intención de la programación orientada a objetos es utilizar llamadas polimórficas a métodos siempre que se pueda y RTTI sólo cuando no haya más remedio.

Sin embargo, las llamadas polimórficas a métodos, tal como está prevista en el lenguaje, requiere que tengamos control de la definición de la clase base, porque en algún punto dentro del proceso de extensión del programa podemos llegar a descubrir que la clase base no incluye el método que necesitamos. Si la clase base proviene de una biblioteca desarrollada por algún otro programador, una solución es RTTI: podemos heredar un nuevo tipo y añadir el método adicional que necesitamos. En el resto del código podemos entonces detectar ese tipo concreto que hemos añadido y llamar a ese método especial. Esto no destruye el polimorfismo y la extensibilidad del programa, porque el añadir un nuevo tipo no requiere que andemos a la caza de instrucciones `switch` en nuestro programa. Sin embargo, cuando añadimos código que dependa de la nueva funcionalidad añadida, nos veremos obligados a utilizar RTTI para detectar el tipo concreto que hayamos definido.

Añadir una funcionalidad en una clase base puede implicar que, a cambio de obtener exclusivamente un beneficio en esa clase concreta, todas las demás clases derivadas de la misma deberán cargar con un esqueleto de método completamente carente de significado. Esto hace que la interfaz sea menos clara y resulta bastante molesto para aquellos que se ven obligados a sustituir métodos abstractos cuando derivan otra clase a partir de esa clase base. Por ejemplo, considere una jerarquía de clases que representa instrumentos musicales. Suponga que desea limpiar las válvulas de las boquillas de todos los instrumentos apropiados de su orquesta. Una opción es incluir un método `limpiarValvula()` en la clase base `Instrumento`, pero esto resulta confuso, porque implicaría que los instrumentos de `Percusión`, `Cuerda` y `Electrónicos` también tienen boquillas y válvulas. RTTI proporciona una solución mucho más razonable, porque nos permite colocar el método en la clase específica donde resulta apropiado (`Viento`, en este caso). Al mismo tiempo, podemos descubrir que existe una solución más lógica, que en este caso consistiría en incluir un método `prepararInstrumento()`. Sin embargo, puede que no veamos esa solución cuando estemos tratando por primera vez de resolver el problema y, como consecuencia, podríamos asumir erróneamente que es necesario utilizar RTTI.

Finalmente, RTTI permite en ocasiones resolver problemas de eficiencia. Suponga que nuestro código utiliza apropiadamente el polimorfismo, pero resulta que uno de los objetos reacciona a este código de propósito general de una manera terriblemente poco eficiente. Podemos detectar ese tipo concreto de objeto utilizando RTTI y escribir código específico para mejorar la eficiencia. No caiga en la tentación, sin embargo, de estructurar sus programas demasiado pronto pensando en la eficiencia. Se trata de una trampa bastante tentadora. Lo mejor es conseguir *primero* que el programa funcione y luego decidir si está funcionando lo suficientemente rápido. Sólo entonces deberemos abordar los problemas de eficiencia con una herramienta de perfilado (consulte el suplemento en <http://MindView.net/Books/BetterJava>).

También hemos visto que el mecanismo de reflexión abre un nuevo mundo de posibilidades de programación, permitiendo un estilo de programación mucho más dinámico. Existen programadores para los que la naturaleza dinámica del mecanismo de reflexión resulta bastante perturbadora. El hecho de que podamos hacer cosas que sólo pueden comprobarse en tiempo de ejecución y de las que sólo se puede informar mediante el mecanismo de excepciones, parece, para las mentes cómodamente acostumbradas a la seguridad de las comprobaciones estáticas de tipos, algo bastante pernicioso. Algunas personas sostienen incluso, que el introducir la posibilidad de una excepción en tiempo de ejecución es una indicación clara de que dicho tipo de código debe evitarse. En mi opinión, esta sensación de seguridad no es más que una ilusión, siempre hay cosas que pueden suceder en tiempo de ejecución y que pueden generar excepciones, incluso en un programa que no contenga ningún bloque `try` ni ninguna especificación de excepción. En lugar de ello, en mi opinión, la existencia de un modelo coherente de información de errores *nos permite* escribir código dinámico utilizando los mecanismos de reflexión. Por supuesto, merece la pena tratar de escribir código que pueda comprobarse estáticamente... siempre que se pueda. Pero creo que el código dinámico es una de las características más importantes que diferencia a Java de otros lenguajes como C++.

Ejercicio 26: (3) Implemente un método `limpiarValvula()` como el descrito en este resumen.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Los métodos y clases ordinarios funcionan con tipos específicos: con tipos primitivos o con clases. Si lo que queremos es escribir código que pueda utilizarse con un tipo más amplio de tipos, esta rigidez puede resultar demasiado restrictiva.¹

Una de las formas en que los lenguajes orientados a objetos permiten la generalización es a través del polimorfismo. Por ejemplo, podemos escribir un método que tome un objeto de una clase base como argumento, y luego utilice dicho método con cualquier clase derivada de dicha clase base. Con ello, el método será algo más general y podrá ser utilizado en más lugares. Lo mismo cabe decir dentro de las clases: en cualquier lugar donde utilicemos un tipo específico, un tipo base proporcionará mayor flexibilidad. Por supuesto, podemos extender todas las clases salvo aquellas que hayan sido definidas como finales², por lo que esta flexibilidad se obtiene de manera automática la mayor parte de las veces.

En ocasiones, limitarse a una única jerarquía puede resultar demasiado restrictivo. Si el argumento de un método es una interfaz en lugar de una clase, las limitaciones se relajan de modo que ahora se incluirán todas aquellas clases que implementen la interfaz, incluyendo clases que todavía no hayan sido desarrolladas. Esto proporciona al programador de clientes la opción de implementar una interfaz para adaptarse a nuestra clase o método. Con esto, las interfaces nos permiten establecer un vínculo entre jerarquías de clases, siempre y cuando tengamos la opción de crear una nueva clase para implementar ese vínculo.

Algunas veces, incluso una interfaz resulta demasiado restrictiva. Las interfaces siguen requiriendo que nuestro código funcione con esa interfaz concreta. Podriamos escribir código todavía más general si el lenguaje nos permitiera decir que ese código funciona con “algún tipo no especificado”, en lugar de con una interfaz o clase específicas.

En esto se basa el concepto de genéricos, un cambio de los más significativos en Java SE5. Los genéricos implementan el concepto de *tipos parametrizados*, que permiten crear componentes (especialmente contenedores) que resultan fáciles de utilizar con múltiples tipos. El término “genérico” significa “perteneciente o apropiado para grandes grupos de clases”. La intención original de los genéricos en los lenguajes de programación era dotar al programador de la mayor capacidad expresiva posible a la hora de escribir clases o métodos, relajando las restricciones que afectan a los tipos con los que esas clases o métodos pueden funcionar. Como veremos en este capítulo, la implementación de los genéricos en Java no tiene un alcance tan grande; de hecho, podriamos cuestionarnos si el término “genérico” resulta siquiera apropiado para esta funcionalidad de Java.

Si no ha visto antes ningún mecanismo de tipos parametrizados, los genéricos de Java le parecerán, probablemente, una mejora sustancial del lenguaje. Cuando se crea una instancia de un tipo parametrizado, el lenguaje se encarga de realizar las proyecciones de los tipos por nosotros y la corrección de los tipos se garantiza en tiempo de compilación. Evidentemente, parece que este mecanismo es toda una mejora.

Sin embargo, si el lector ya tiene experiencia con algún mecanismo de tipos parametrizados, como por ejemplo en C++, encontrará que no se pueden hacer con los genéricos de Java todas las cosas que cabría esperar. Mientras que utilizar un tipo genérico desarrollado por alguna otra persona resulta bastante sencillo, a la hora de crear nuestros propios genéricos nos

¹ Quiero dar las gracias a Angelika Langer por su lista de preguntas frecuentes *Java Generics FAQ* (véase www.langer.camelot.de), así como por sus otros escritos (hechos en colaboración con Klaus Kreft). Esos trabajos han resultado enormemente valiosos de cara a la preparación de este capítulo.

² O clases que dispongan de un constructor privado.

encontraremos con diversas sorpresas. Uno de los aspectos que trataremos de explicar en este capítulo son los motivos por los que la funcionalidad se ha implementado en Java en la manera en que se ha hecho.

No queremos decir que los genéricos de Java sean inútiles. En muchos casos, consiguen que el código sea más directo e incluso más elegante. Pero, si el lector ha utilizado anteriormente algún lenguaje donde esté implementada una versión más pura de los genéricos, puede que la solución de Java le desilusione. En este capítulo, vamos a examinar tanto las fortalezas como las debilidades de los genéricos de Java, con el fin de que el lector pueda utilizar esta nueva funcionalidad de manera más efectiva.

Comparación con C++

Los diseñadores de Java han dejado claro que buena parte de la inspiración del lenguaje proviene de C++. A pesar de ello, resulta perfectamente posible enseñar a programar en Java sin hacer apenas referencia a C++, y en este libro hemos intentado hacerlo así, salvo en aquellos casos en los que la comparación puede facilitar entender mejor el lenguaje.

Los genéricos requieren que realicemos una comparación más detallada con C++ por dos razones. En primer lugar, comprender ciertos aspectos de las *plantillas C++* (la principal inspiración de los genéricos, incluyendo su sintaxis básica) nos permitirá entender los fundamentos del concepto, así como (y esto es particularmente importante) las limitaciones que afectan a lo que se puede hacer con los genéricos de Java, y los motivos subyacentes de la existencia de esas limitaciones. El objetivo último es que el lector comprenda claramente dónde están los límites, porque entendiendo esos límites se puede llegar a ser un programador más eficiente. Sabiendo lo que no puede hacerse, podemos emplear mejor aquellas cosas que sí podemos hacer (en parte porque no nos vemos obligados a perder tiempo rompiéndonos la cabeza contra una pared).

La segunda razón es que existen muchas concepciones erróneas en la comunidad Java acerca de las plantillas C++, y estos conceptos erróneos pueden aumentar nuestra confusión acerca del objetivo de los genéricos.

Por tanto, vamos a introducir unos cuantos ejemplos de plantillas C++ en este capítulo, aunque tratando siempre de limitar al máximo las explicaciones acerca del lenguaje C++.

Genéricos simples

Una de las razones iniciales más fuertes para introducir los genéricos era crear *clases de contenedores*, de las que ya hemos hablado en el Capítulo 11, *Almacenamiento de objetos* (hablaremos más acerca de estas clases en el Capítulo 17, *Análisis detallado de los contenedores*). Un contenedor es un lugar en el que almacenar objetos mientras trabajamos con ellos. Aunque esto también es cierto para las matrices, los contenedores tienden a ser más flexibles y sus características son distintas a las de las matrices simples. Casi todos los programas requieren que almacenemos un grupo de objetos mientras los utilizamos, por lo que los contenedores son una de las bibliotecas de clases más inherentemente reutilizables.

Examinemos una clase que almacena un único objeto. Por supuesto, la clase podría especificar el tipo exacto del objeto de la forma siguiente:

```
//: generics/Holder1.java

class Automobile {}

public class Holder1 {
    private Automobile a;
    public Holder1(Automobile a) { this.a = a; }
    Automobile get() { return a; }
} //:-
```

Pero esta herramienta no es muy reutilizable, ya que no puede emplearse para almacenar ninguna otra cosa. Preferiríamos no tener que escribir una nueva clase de este estilo para cada tipo con el que nos encontramos.

Antes de Java SE5, lo que haríamos simplemente es hacer que la clase almacenara un objeto de tipo **Object**:

```
//: generics/Holder2.java

public class Holder2 {
```

```

private Object a;
public Holder2(Object a) { this.a = a; }
public void set(Object a) { this.a = a; }
public Object get() { return a; }
public static void main(String[] args) {
    Holder2 h2 = new Holder2(new Automobile());
    Automobile a = (Automobile)h2.get();
    h2.set("Not an Automobile");
    String s = (String)h2.get();
    h2.set(1); // Se transforma automáticamente en Integer
    Integer x = (Integer)h2.get();
}
} //:-

```

Ahora, la clase **Holder2** puede almacenar cualquier cosa y, en este ejemplo, un único objeto **Holder2** almacena tres tipos distintos de objetos.

Hay algunos casos en los que queremos que un contenedor almacene múltiples tipos de objetos, pero lo más normal es que sólo coloquemos un tipo de objeto en cada contenedor. Una de las principales motivaciones de los genéricos consiste en especificar el tipo de objeto que un contenedor almacena, y hacer que dicha especificación quede respaldada por el compilador.

Por tanto, en lugar de emplear **Object**, lo que queríamos es poder utilizar un tipo no especificado, lo que podremos decidir en algún momento posterior. Para hacer esto, incluimos un *parámetro de tipo* entre corchetes angulares después del nombre de la clase y luego, al utilizar la clase, sustituimos ese parámetro por un tipo real. Para nuestra clase contenedora anterior, la técnica consistiría en lo siguiente, donde **T** es el parámetro de tipo:

```

//: generics/Holder3.java

public class Holder3<T> {
    private T a;
    public Holder3(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }
    public static void main(String[] args) {
        Holder3<Automobile> h3 =
            new Holder3<Automobile>(new Automobile());
        Automobile a = h3.get(); // No hace falta proyección
        // h3.set("Not an Automobile"); // Error
        // h3.set(1); // Error
    }
} //:-

```

Ahora, cuando creamos un objeto **Holder3**, deberemos especificar el tipo que queramos almacenar en el mismo, utilizando la sintaxis de corchetes angulares, como puede verse en **main()**. Sólo podemos introducir en el contenedor objetos de dicho tipo (o de alguno de sus subtipos, ya que el principio de sustitución sigue funcionando con los genéricos). Y al extraer del contenedor un valor, dicho valor tendrá automáticamente el tipo correcto.

Ésta es la idea fundamental de los genéricos de Java: le decimos al compilador qué tipo queremos usar y el compilador se encarga de los detalles.

En general, podemos tratar los genéricos como si fueran otro tipo más que en lo único que se diferencia de los tipos normales es en que tiene parámetros de tipo. Pero, como veremos en breve, podemos utilizar los genéricos simplemente nombrándolos junto con su lista de argumentos de tipo.

Ejercicio 1: (1) Utilice **Holder3** con la biblioteca **typeinfo.pets** para demostrar que un objeto **Holder3** que se haya especificado para almacenar un tipo base, también puede almacenar un tipo derivado.

Ejercicio 2: (1) Cree una clase contenedora que almacene tres objetos del mismo tipo, junto con los métodos para almacenar y extraer dichos objetos y un constructor para inicializar los tres.

Una biblioteca de tuplas

Una de las cosas que a menudo hace falta hacer es devolver múltiples objetos de una llamada a método. La instrucción `return` sólo permite especificar un único objeto, por lo que la respuesta consiste en crear otro objeto que almacene los múltiples objetos que queramos devolver. Por supuesto, podemos escribir una clase especial cada vez que nos encontramos con esta situación, pero con los genéricos es posible resolver el problema de una vez y ahorrarnos un montón de esfuerzo en el futuro. Al mismo tiempo estaremos garantizando la seguridad de los tipos en tiempo de compilación.

Este concepto se denomina *tupla*, y consiste simplemente en un grupo de objetos que se envuelven juntos dentro de otro objeto único. El receptor del objeto estará autorizado a leer los elementos, pero no a introducir otros nuevos (este concepto también se conoce como *Objeto de transferencia de datos* o *Mensajero*).

Las tuplas pueden tener, normalmente, cualquier longitud, y cada objeto de la tupla puede tener un tipo distinto. Sin embargo, lo que nos interesa es especificar el tipo de cada objeto y garantizar que, cuando el receptor lea los valores, obtenga el tipo correcto. Para tratar con el problema de las múltiples longitudes, podemos crear múltiples tuplas diferentes. He aquí una que almacena dos objetos:

```
//: net/mindview/util/TwoTuple.java
package net.mindview.util;

public class TwoTuple<A,B> {
    public final A first;
    public final B second;
    public TwoTuple(A a, B b) { first = a; second = b; }
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
} //:-
```

El constructor captura el objeto que hay que almacenar y `toString()` es una función de utilidad que permite mostrar los valores de una lista. Observe que una tupla conserva implicitamente sus elementos en orden.

Al examinar el ejemplo por primera vez, podría pensarse que viola los principios comunes de seguridad en la programación Java. ¿No deberían ser `first` y `second` privados, y no debería accederse a ellos únicamente con los métodos denominados `getFirst()` y `getSecond()`? Consideré la seguridad que se obtendría en dicho caso: los clientes podrían seguir leyendo los objetos y hacer lo que quisieran con los mismos, pero no podrían asignar `first` o `second` a ninguna otra cosa. La declaración `final` nos proporciona esa misma seguridad, pero la forma empleada en el ejemplo es más corta y más simple.

Otra observación de diseño importante es que puede que *queramos* permitir a un programador de clientes que haga apuntar a `first` o `second` a algún otro objeto. Sin embargo, es más seguro dejar el ejemplo tal cual está, y limitarse a obligar al usuario a crear un nuevo objeto `TwoTuple` si desea disponer de uno que tenga diferentes elementos.

Las tuplas de mayor longitud puede crearse mediante herencia. Como podemos ver, añadir más parámetros de tipo resulta bastante simple:

```
//: net/mindview/util/ThreeTuple.java
package net.mindview.util;

public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
    public final C third;
    public ThreeTuple(A a, B b, C c) {
        super(a, b);
        third = c;
    }
    public String toString() {
        return "(" + first + ", " + second + ", " + third + ")";
    }
} //:-

//: net/mindview/util/FourTuple.java
package net.mindview.util;
```

```

public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
    public final D fourth;
    public FourTuple(A a, B b, C c, D d) {
        super(a, b, c);
        fourth = d;
    }
    public String toString() {
        return "(" + first + ", " + second + ", " +
               third + ", " + fourth + ")";
    }
} //:-

//: net/mindview/util/FiveTuple.java
package net.mindview.util;

public class FiveTuple<A,B,C,D,E>
extends FourTuple<A,B,C,D> {
    public final E fifth;
    public FiveTuple(A a, B b, C c, D d, E e) {
        super(a, b, c, d);
        fifth = e;
    }
    public String toString() {
        return "(" + first + ", " + second + ", " +
               third + ", " + fourth + ", " + fifth + ")";
    }
} //:-
```

Para utilizar una tupla, simplemente definimos la tupla de la longitud adecuada como valor de retorno para nuestra función y luego la creamos y la devolvemos en la instrucción **return**:

```

//: generics/TupleTest.java
import net.mindview.util.*;

class Amphibian {}
class Vehicle {}

public class TupleTest {
    static TwoTuple<String, Integer> f() {
        // El mecanismo de autoboxing convierte int en Integer:
        return new TwoTuple<String, Integer>("hi", 47);
    }
    static ThreeTuple<Amphibian, String, Integer> g() {
        return new ThreeTuple<Amphibian, String, Integer>(
            new Amphibian(), "hi", 47);
    }
    static
    FourTuple<Vehicle, Amphibian, String, Integer> h() {
        return
            new FourTuple<Vehicle, Amphibian, String, Integer>(
                new Vehicle(), new Amphibian(), "hi", 47);
    }
    static
    FiveTuple<Vehicle, Amphibian, String, Integer, Double> k() {
        return new
            FiveTuple<Vehicle, Amphibian, String, Integer, Double>(
                new Vehicle(), new Amphibian(), "hi", 47, 11.1);
    }
    public static void main(String[] args) {
        TwoTuple<String, Integer> ttsi = f();
```

```

        System.out.println(ttsi);
        // ttsi.first = "there"; // Error de compilación: final
        System.out.println(g());
        System.out.println(h());
        System.out.println(k());
    }
} /* Output: (80% match)
(hi, 47)
(Amphibian@1f6a7b9, hi, 47)
(Vehicle@35ce36, Amphibian@757aef, hi, 47)
(Vehicle@9cab16, Amphibian@1a46e30, hi, 47, 11.1)
*///:-
```

Gracias a los genéricos, podemos crear fácilmente cualquier tupla para devolver cualquier grupo de tipos, simplemente escribiendo la expresión correspondiente.

Podemos ver cómo la especificación **final** en los campos públicos impide que sean reasignados después de la construcción, puede observarlo viendo cómo falla la instrucción **ttsi.first = “there”**.

Las expresiones **new** son demasiado complejas. Posteriormente en el capítulo veremos cómo simplificarlas utilizando *métodos genéricos*.

Ejercicio 3: (1) Cree y pruebe un genérico **SixTuple** con seis elementos.

Ejercicio 4: (3) Reescriba **innerclasses/Sequence.java** utilizando genéricos.

Una clase que implementa una pila

Examinemos algo ligeramente más complicado: la típica estructura de pila. En el Capítulo 11, *Almacenamiento de objetos*, vimos cómo implementar una pila utilizando un contenedor **LinkedList** en la clase **net.mindview.util.Stack**. En dicho ejemplo, podemos ver que **LinkedList** ya dispone de los métodos necesarios para crear una pila. La clase **Stack** que implementaba la pila se construyó componiendo una clase genérica (**Stack<T>**) con otra clase genérica (**LinkedList<T>**). En dicho ejemplo, observe que (con unas cuantas excepciones que posteriormente realizaremos) un tipo genérico no es otra cosa que un tipo normal.

En lugar de utilizar **LinkedList**, podemos implementar nuestro propio mecanismo interno de almacenamiento enlazado.

```

//: generics/LinkedStack.java
// Una pila implementada con una estructura enlazada interna.

public class LinkedStack<T> {
    private static class Node<U> {
        U item;
        Node<U> next;
        Node() { item = null; next = null; }
        Node(U item, Node<U> next) {
            this.item = item;
            this.next = next;
        }
        boolean end() { return item == null && next == null; }
    }
    private Node<T> top = new Node<T>(); // Indicador de fin
    public void push(T item) {
        top = new Node<T>(item, top);
    }
    public T pop() {
        T result = top.item;
        if(!top.end())
            top = top.next;
        return result;
    }
}
```

```

public static void main(String[] args) {
    LinkedStack<String> lss = new LinkedStack<String>();
    for(String s : "Phasers on stun!".split(" "))
        lss.push(s);
    String s;
    while((s = lss.pop()) != null)
        System.out.println(s);
}
} /* Output:
stun!
on
Phasers
*///:-

```

La clase interna **Node** también es un genérico y tiene su propio parámetro de tipo.

Este ejemplo hace uso de un *índicador de fin* para determinar cuándo la pila está vacía. El indicador de fin se crea en el momento de construir el contenedor **LinkedStack**, y cada vez que se invoca **push()** se crea un nuevo objeto **Node<T>** y se enlaza con el objeto **Node<T>** anterior. Cuando se invoca a **pop()**, siempre se devuelve **top.item**, y luego se descarta el objeto **Node<T>** actual y nos desplazamos al siguiente, excepto cuando encontramos el indicador de fin, en cuyo caso no nos desplazamos. De esa forma, si el cliente continúa invocando **pop()**, obtendrá como respuesta valores **null** para indicar que la pila está vacía.

Ejercicio 5: (2) Elimine el parámetro de tipo en la clase **Node** y modifique el resto del código en **LinkedStack.java** para demostrar que una clase interna tiene acceso a los parámetros de tipo genérico de su clase externa.

RandomList

Como ejemplo adicional de contenedor, suponga que queremos disponer de un tipo especial de lista que seleccione aleatoriamente uno de sus elementos cada vez que invoquemos **select()**. Al hacer esto, podemos construir una herramienta que funcione para todos los objetos, así que utilizamos genéricos:

```

//: generics/RandomList.java
import java.util.*;

public class RandomList<T> {
    private ArrayList<T> storage = new ArrayList<T>();
    private Random rand = new Random(47);
    public void add(T item) { storage.add(item); }
    public T select() {
        return storage.get(rand.nextInt(storage.size()));
    }
    public static void main(String[] args) {
        RandomList<String> rs = new RandomList<String>();
        for(String s: ("The quick brown fox jumped over " +
                      "the lazy brown dog").split(" "))
            rs.add(s);
        for(int i = 0; i < 11; i++)
            System.out.print(rs.select() + " ");
    }
} /* Output:
brown over fox quick quick dog brown The brown lazy brown
*///:-

```

Ejercicio 6: (1) Utilice **RandomList** con dos tipos adicionales además del que se muestra en **main()**.

Interfaces genéricas

Los genéricos también funcionan con las interfaces. Por ejemplo, un *generador* es una clase que crea objetos. En la práctica, una especialización del patrón de diseño basado en el *método de factoría*, pero cuando pedimos a un generador que cree

un nuevo objeto no le pasamos ningún argumento, al contrario de lo que sucede con un método de factoría. El generador sabe cómo crear nuevos objetos sin ninguna información adicional.

Típicamente, un generador simplemente define un método, el método que produce nuevos objetos. Aquí, lo denominaremos **next()** y lo incluiremos en las utilidades estándar:

```
//: net/mindview/util/Generator.java
// Una interfaz genérica.
package net.mindview.util;
public interface Generator<T> { T next(); } //:-
```

El tipo de retorno de **next()** se parametriza como **T**. Como puede ver, la utilización de genéricos con interfaces no es diferente de la utilización de genéricos con clases.

Para ilustrar la implementación de un objeto **Generator**, necesitaremos algunas clases. He aquí una jerarquía de ejemplo:

```
//: generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getSimpleName() + " " + id;
    }
} //:-

//: generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {} //:-

//: generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {} //:-

//: generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {} //:-

//: generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {} //:-

//: generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {} //:-
```

Ahora, podemos implementar un objeto **Generator<Coffee>** que genera aleatoriamente diferentes tipos de objetos **Coffee**:

```
//: generics/coffee/CoffeeGenerator.java
// Generar diferentes tipos de objetos Coffee:
package generics.coffee;
import java.util.*;
import net.mindview.util.*;

public class CoffeeGenerator
    implements Generator<Coffee>, Iterable<Coffee> {
    private Class[] types = { Latte.class, Mocha.class,
        Cappuccino.class, Americano.class, Breve.class, };
    private static Random rand = new Random(47);
    public CoffeeGenerator() {}
    // Para iteración:
```

```

private int size = 0;
public CoffeeGenerator(int sz) { size = sz; }
public Coffee next() {
    try {
        return (Coffee)
            types[rand.nextInt(types.length)].newInstance();
        // Informar de errores del programador en tiempo de ejecución:
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
}
class CoffeeIterator implements Iterator<Coffee> {
    int count = size;
    public boolean hasNext() { return count > 0; }
    public Coffee next() {
        count--;
        return CoffeeGenerator.this.next();
    }
    public void remove() { // No implementado
        throw new UnsupportedOperationException();
    }
};
public Iterator<Coffee> iterator() {
    return new CoffeeIterator();
}
public static void main(String[] args) {
    CoffeeGenerator gen = new CoffeeGenerator();
    for(int i = 0; i < 5; i++)
        System.out.println(gen.next());
    for(Coffee c : new CoffeeGenerator(5))
        System.out.println(c);
}
} /* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*///:-

```

La interfaz **Generator** parametrizada garantiza que **next()** devuelva el tipo definido en el parámetro. **CoffeeGenerator** también implementa la interfaz **Iterable**, por lo que se le puede usar en una instrucción *foreach*. Sin embargo, requiere un “indicador de fin” para saber cuándo parar, y esto se crea utilizando el segundo constructor.

He aquí una segunda implementación de **Generator<T>**, que esta vez se utiliza para generar números de Fibonacci:

```

//: generics/Fibonacci.java
// Generar una secuencia de Fibonacci.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
    private int count = 0;
    public Integer next() { return fib(count++); }
    private int fib(int n) {
        if(n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }
}

```

```

    }
    public static void main(String[] args) {
        Fibonacci gen = new Fibonacci();
        for(int i = 0; i < 18; i++)
            System.out.print(gen.next() + " ");
    }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:-
```

Aunque estamos trabajando con valores **int** tanto dentro como fuera de la clase, el parámetro de tipo es **Integer**. Esto nos plantea una de las limitaciones de los genéricos de Java. No se pueden utilizar primitivas como parámetros de tipo. Sin embargo, Java SE5 ha añadido, afortunadamente, la funcionalidad de conversión automática entre primitivas y tipos envolvente, para poder efectuar las conversiones fácilmente. Podemos ver el efecto en este ejemplo porque los valores **int** se utilizan, en general, en la clase de manera transparente.

Podemos ir un paso más allá y crear un generador de Fibonacci de tipo **Iterable**. Una opción consiste en reimplementar la clase y añadir la interfaz **Iterable**, pero no siempre tenemos control sobre el código original, y no merece la pena reescribir código a menos que nos veamos obligados a hacerlo. En lugar de ello, podemos crear un *adaptador* para obtener la interfaz deseada; este patrón de diseño ya fue presentado anteriormente en el libro.

Los adaptadores pueden implementarse de múltiples formas. Por ejemplo, podemos utilizar el mecanismo de herencia para generar la clase adaptada:

```

//: generics/IterableFibonacci.java
// Adaptar la clase Fibonacci para hacerla de tipo Iterable.
import java.util.*;

public class IterableFibonacci
    extends Fibonacci implements Iterable<Integer> {
    private int n;
    public IterableFibonacci(int count) { n = count; }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public boolean hasNext() { return n > 0; }
            public Integer next() {
                n--;
                return IterableFibonacci.this.next();
            }
            public void remove() { // No implementado
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        for(int i : new IterableFibonacci(18))
            System.out.print(i + " ");
    }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:-
```

Para utilizar **IterableFibonacci** en una instrucción *foreach*, hay que proporcionar al constructor un límite para que **hasNext()** sepa cuándo devolver **false**.

Ejercicio 7: (2) Utilice el mecanismo de composición en lugar del mecanismo de herencia para adaptar **Fibonacci** con el fin de hacerla de tipo **Iterable**.

Ejercicio 8: (2) Siguiendo la forma del ejemplo **Coffee**, cree una jerarquía de personajes (**StoryCharacter**) de su película favorita, dividiéndolos en buenos (**GoodGuys**) y malos (**BadGuys**). Cree un generador para **StoryCharacter**, siguiendo la forma de **CoffeeGenerator**.

Métodos genéricos

Hasta ahora, hemos estado analizando la parametrización de clases enteras, pero también podemos parametrizar métodos de una clase. La propia clase puede ser o no genérica; esto no influye en la posibilidad de disponer de métodos genéricos.

Un método genérico permite que el método varie independientemente de la clase. Como directriz, deberemos usar los métodos genéricos “siempre que podamos”. En otras palabras: si es posible hacer que un método sea genérico, en lugar de que lo sea la clase completa, probablemente el programa sea más claro si hacemos genérico el método. Además, si un método es estático, no tiene acceso a los parámetros genéricos de tipo de la clase, por lo que si esa genericidad es necesaria en el método, deberemos definirlo como un método genérico.

Para definir un método genérico, simplemente colocamos una lista de parámetros genéricos delante del valor retorno del modo siguiente:

```
//: generics/GenericMethods.java

public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }
    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        gm.f("");
        gm.f(1);
        gm.f(1.0);
        gm.f(1.0F);
        gm.f('c');
        gm.f(gm);
    }
}

} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*///:-
```

La clase **GenericMethods** no está parametrizada, aunque es perfectamente posible parametrizar simultáneamente tanto una clase como sus métodos. Pero en este caso, solo el método **f()** tiene un parámetro de tipo, indicado por la lista de parámetros antes del tipo de retorno del método.

Observe que con una clase genérica es preciso especificar los parámetros de tipo en el momento de instanciar la clase. Pero con un método genérico, usualmente no hace falta especificar los tipos de parámetro, porque el compilador puede determinar esos tipos por nosotros. Este mecanismo se denomina *inferencia del argumento de tipo*. Por tanto, las llamadas a **f()** parecen llamadas a método normales, y en la práctica **f()** se comporta como si estuviera infinitamente sobrecargado. El método admitirá incluso un argumento del tipo **GenericMethods**.

Para las llamadas a **f()** que usen tipos primitivos entra en acción el mecanismo de conversión de tipos automática, envolviendo de manera transparente los tipos primitivos en sus objetos asociados. De hecho, los métodos genéricos y el mecanismo de conversión automática de tipos permiten eliminar parte del código que anteriormente requería utilizar conversiones de tipos manuales.

Ejercicio 9: (1) Modifique **GenericMethods.java** de modo que **f()** acepte tres argumentos, cada uno de los cuales tiene que ser de un tipo parametrizado distinto.

Ejercicio 10: (1) Modifique el ejercicio anterior de modo que uno de los argumentos de **f()** no sea parametrizado.

Aprovechamiento de la inferencia del argumento de tipo

Una de las quejas acerca de los genéricos es que añaden todavía más texto a nuestro código. Considere el programa **holding/MapOfList.java** del Capítulo 11, *Almacenamiento de objetos*. La creación del contenedor **Map** de **List** tiene el aspecto siguiente:

```
Map<Person, List<? extends Pet>> petPeople =
    new HashMap<Person, List<? extends Pet>>();
```

(Esta utilización de **extends** y los signos de interrogación se explicarán posteriormente en el capítulo). Parece, por el ejemplo, que nos estamos repitiendo y que el compilador debería deducir una de las listas de argumentos genéricos a partir de la otra. En realidad, no puede deducirla, pero la inferencia del argumento de tipo en un método genérico permite realizar algunas simplificaciones. Por ejemplo, podemos crear una utilidad que contenga varios métodos estáticos y con la que se generen las implementaciones de los diversos contenedores más comúnmente utilizadas:

```
//: net/mindview/util/New.java
// Utilidades para simplificar la creación de contenedores
// genéricos empleando la inferencia del argumento de tipo.
package net.mindview.util;
import java.util.*;

public class New {
    public static <K,V> Map<K,V> map() {
        return new HashMap<K,V>();
    }
    public static <T> List<T> list() {
        return new ArrayList<T>();
    }
    public static <T> LinkedList<T> lList() {
        return new LinkedList<T>();
    }
    public static <T> Set<T> set() {
        return new HashSet<T>();
    }
    public static <T> Queue<T> queue() {
        return new LinkedList<T>();
    }
    // Ejemplos:
    public static void main(String[] args) {
        Map<String, List<String>> sls = New.map();
        List<String> ls = New.list();
        LinkedList<String> lls = New.lList();
        Set<String> ss = New.set();
        Queue<String> qs = New.queue();
    }
} ///:-
```

En **main()** podemos ver ejemplos de cómo se emplea esta herramienta: la inferencia del argumento de tipo elimina la necesidad de repetir la lista de parámetros genéricos. Podemos aplicar esto a **holding/MapOfList.java**:

```
//: generics/SimplerPets.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class SimplerPets {
    public static void main(String[] args) {
        Map<Person, List<? extends Pet>> petPeople = New.map();
        // El resto del código es igual...
    }
} ///:-
```

Aunque se trata de un ejemplo interesante del mecanismo de inferencia del argumento de tipo, resulta difícil determinar las ventajas de este mecanismo. A la persona que lea el código la obligamos a analizar y a comprender esta biblioteca adicional y sus implicaciones; por lo que sería igual de productivo dejar la definición original (que es bastante repetitiva) precisamente para simplificar. Sin embargo, si la biblioteca estándar de Java incluyera algo similar a la utilidad **New.java** que hemos presentado, tendría bastante sentido utilizarla.

El mecanismo de inferencia de tipos no funciona más que en las asignaciones. Si pasamos el resultado de una llamada a un método, tal como **New.map()**, como argumento a otro método, el compilador *no intentará* realizar una inferencia de tipos. En lugar de ello, lo que hará es tratar la llamada al método como si el valor de retorno se asignara a una variable de tipo **Object**. He aquí un ejemplo con el que se genera un error de compilación:

```
//: generics/LimitsOfInference.java
import typeinfo.pets.*;
import java.util.*;

public class LimitsOfInference {
    static void
    f(Map<Person, List<? extends Pet>> petPeople) {}
    public static void main(String[] args) {
        // f(New.map()); // No se compila
    }
} //:-
```

Ejercicio 11: (1) Pruebe **New.java** creando sus propias clases y verificando que **New** funcione adecuadamente con las mismas.

Especificación explícita de tipos

Es posible especificar explícitamente el tipo de un método genérico, aunque esta sintaxis raramente es necesaria. Para hacer esto, se coloca el tipo entre corchetes angulares después del punto e inmediatamente antes del nombre del método. A la hora de invocar a un método desde dentro de la misma clase, hay que utilizar **this** antes del punto; y cuando se trabaje con métodos estáticos hay que emplear el nombre de la clase antes del punto. El problema mostrado en **LimitsOfInference.java** puede resolverse utilizando esta sintaxis:

```
//: generics/ExplicitTypeSpecification.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class ExplicitTypeSpecification {
    static void f(Map<Person, List<Pet>> petPeople) {}
    public static void main(String[] args) {
        f(New.<Person, List<Pet>>map());
    }
} //:-
```

Por supuesto, esto elimina la ventaja de utilizar la clase **New** para reducir la cantidad de texto tecleado, pero esta sintaxis adicional sólo será necesaria cuando no estemos escribiendo una instrucción de asignación.

Ejercicio 12: (1) Repita el ejercicio anterior utilizando la especificación explícita de tipos.

Varargs y métodos genéricos

Los métodos genéricos y las listas de argumentos variables pueden coexistir perfectamente:

```
//: generics/GenericVarargs.java
import java.util.*;

public class GenericVarargs {
    public static <T> List<T> makeList(T... args) {
```

```

List<T> result = new ArrayList<T>();
for(T item : args)
    result.add(item);
return result;
}
public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList("ABCDEFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
}
/* Output:
[A]
[A, B, C]
[ , A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
*///:-
```

El método **makeList()** mostrado aquí tiene la misma funcionalidad que el método **java.util.Arrays.asList()** de la biblioteca estándar.

Un método genérico para utilizar con generadores

Resulta bastante cómodo utilizar un generador para llenar un objeto **Collection**, y también tiene bastante sentido hacer genérica esta operación:

```

//: generics/Generators.java
// Una utilidad para utilizar con generadores.
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;

public class Generators {
    public static <T> Collection<T>
        fill(Collection<T> coll, Generator<T> gen, int n) {
        for(int i = 0; i < n; i++)
            coll.add(gen.next());
        return coll;
    }
    public static void main(String[] args) {
        Collection<Coffee> coffee = fill(
            new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
        for(Coffee c : coffee)
            System.out.println(c);
        Collection<Integer> fnumbers = fill(
            new ArrayList<Integer>(), new Fibonacci(), 12);
        for(int i : fnumbers)
            System.out.print(i + ", ");
    }
}
/* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
*///:-
```

Observe que el método genérico **fill()** puede aplicarse de forma transparente a contenedores y generadores de objetos **Coffee** e **Integer**.

Ejercicio 13: (4) Sobreträgue el método `fill()` de modo que los argumentos y tipos de retorno sean los subtipos específicos de `Collection`: `List`, `Queue` y `Set`. De esta forma, no perdemos el tipo de contenedor. ¿Podemos utilizar el mecanismo de sobreträgura para distinguir entre `List` y `LinkedList`?

Un generador de propósito general

He aquí una clase que produce un objeto `Generator` para cualquier clase que disponga de un constructor predeterminado. Para reducir la cantidad de texto tecleado, también incluye un método genérico para crear un objeto `BasicGenerator`:

```
//: net/mindview/util/BasicGenerator.java
// Crear automáticamente un generador, dada una clase con un
// constructor predeterminado (sin argumentos).
package net.mindview.util;

public class BasicGenerator<T> implements Generator<T> {
    private Class<T> type;
    public BasicGenerator(Class<T> type){ this.type = type; }
    public T next() {
        try {
            // Asume que el tipo es una clase pública:
            return type.newInstance();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    // Producir un generador predeterminado dado un indicador de tipo:
    public static <T> Generator<T> create(Class<T> type) {
        return new BasicGenerator<T>(type);
    }
} //:-
```

Esta clase proporciona una implementación básica que producirá objetos de una clase que (1) sea pública (ya que `BasicGenerator` está en un paquete separado, la clase en cuestión debe tener acceso público y no simplemente de paquete) y (2) tenga un constructor predeterminado (uno que no tome ningún argumento). Para crear uno de estos objetos `BasicGenerator`, invocamos el método `create()` y pasamos el indicador de tipo para el tipo que queramos generar. El método genérico `create()` permite escribir `BasicGenerator.create(MyType.class)` en lugar de la instrucción `new BasicGenerator<MyType>(MyType.class)` que es más complicada.

Por ejemplo, he aquí una clase simple que dispone de un constructor predeterminado:

```
//: generics/CountedObject.java

public class CountedObject {
    private static long counter = 0;
    private final long id = counter++;
    public long id() { return id; }
    public String toString() { return "CountedObject " + id; }
} //:-
```

La clase `CountedObject` lleva la cuenta de cuántas instancias de sí misma se han creado e informa de la cantidad total mediante `toString()`.

Utilizando `BasicGenerator`, podemos crear fácilmente un objeto `Generator` para `CountedObject`:

```
//: generics/BasicGeneratorDemo.java
import net.mindview.util.*;

public class BasicGeneratorDemo {
    public static void main(String[] args) {
        Generator<CountedObject> gen =
            BasicGenerator.create(CountedObject.class);
```

```

        for(int i = 0; i < 5; i++)
            System.out.println(gen.nextInt());
    }
} /* Output:
CountedObject 0
CountedObject 1
CountedObject 2
CountedObject 3
CountedObject 4
*///:-
```

Podemos ver cómo el método genérico reduce la cantidad de texto necesaria para crear el objeto **Generator**. Los genéricos de Java nos fuerzan a pasar de todos modos el objeto **Class**, por lo que también podríamos utilizarlo para la inferencia de tipos en el método **create()**.

Ejercicio 14: (1) Modifique **BasicGeneratorDemo.java** para utilizar la forma explícita de creación del objeto **Generator** (es decir, utilice el constructor explícito en lugar del método genérico **create()**).

Simplificación del uso de las tuplas

El mecanismo de inferencia del argumento de tipo, junto con las importaciones de tipo **static**, nos permite reescribir las tuplas que hemos presentado anteriormente, para obtener una biblioteca de propósito más general. Aquí, las tuplas pueden crearse utilizando un método estático sobrecargado:

```

//: net/mindview/util/Tuple.java
// Biblioteca de tuplas utilizando el mecanismo de
// inferencia del argumento de tipo,
package net.mindview.util;

public class Tuple {
    public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
        return new TwoTuple<A,B>(a, b);
    }
    public static <A,B,C> ThreeTuple<A,B,C>
        tuple(A a, B b, C c) {
        return new ThreeTuple<A,B,C>(a, b, c);
    }
    public static <A,B,C,D> FourTuple<A,B,C,D>
        tuple(A a, B b, C c, D d) {
        return new FourTuple<A,B,C,D>(a, b, c, d);
    }
    public static <A,B,C,D,E>
        FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {
        return new FiveTuple<A,B,C,D,E>(a, b, c, d, e);
    }
} //:-
```

He aquí una modificación de **TupleTest.java** para probar **Tuple.java**:

```

//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

public class TupleTest2 {
    static TwoTuple<String, Integer> f() {
        return tuple("hi", 47);
    }
    static TwoTuple f2() { return tuple("hi", 47); }
    static ThreeTuple<Amphibian, String, Integer> g() {
        return tuple(new Amphibian(), "hi", 47);
    }
}
```

```

static
FourTuple<Vehicle,Amphibian,String,Integer> h() {
    return tuple(new Vehicle(), new Amphibian(), "hi", 47);
}
static
FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
    return tuple(new Vehicle(), new Amphibian(),
        "hi", 47, 11.1);
}
public static void main(String[] args) {
    TwoTuple<String,Integer> ttsi = f();
    System.out.println(ttsi);
    System.out.println(f2());
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
}
/* Output: (80% match)
(hi, 47)
(hi, 47)
(Amphibian@7d772e, hi, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
(Vehicle@1a46e30, Amphibian@3e25a5, hi, 47, 11.1)
*///:-
```

Observe que `f()` devuelve un objeto parametrizado `TwoTuple`, mientras que `f2()` devuelve un objeto `TwoTuple` no parametrizado. El compilador no proporciona ninguna advertencia acerca de `f2()` en este caso porque el valor de retorno no está siendo utilizado de forma parametrizada; en un cierto sentido, está siendo “generalizado” a un objeto `TwoTuple` no parametrizado. Sin embargo, si quisieramos calcular el resultado de `f2()` en un objeto parametrizado `TwoTuple`, el compilador generaría una advertencia.

Ejercicio 15: (1) Verifique la afirmación anterior.

Ejercicio 16: (2) Añada una tupla `SixTuple` a `Tuple.java` y pruébelas mediante `TupleTest2.java`.

Una utilidad Set

Vamos a ver otro ejemplo del uso de método genérico. Considere las relaciones matemáticas que pueden expresarse utilizando conjuntos. Estos conjuntos pueden definirse de forma cómoda como método genérico, para utilizarlos con todos los diferentes tipos:

```

//: net/mindview/util/Sets.java
package net.mindview.util;
import java.util.*;

public class Sets {
    public static <T> Set<T> union(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.addAll(b);
        return result;
    }
    public static <T>
    Set<T> intersection(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.retainAll(b);
        return result;
    }
    // Restar subconjunto de un superconjunto:
    public static <T> Set<T>
    difference(Set<T> superset, Set<T> subset) {
```

```

        Set<T> result = new HashSet<T>(superset);
        result.removeAll(subset);
        return result;
    }
    // Reflexivo--todo lo que no esté en la intersección:
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {
        return difference(union(a, b), intersection(a, b));
    }
} //:-)

```

Los primeros tres métodos duplican el primer argumento copiando sus referencias en un nuevo objeto **HashSet**, de modo que los conjuntos utilizados como argumentos no se modifican directamente. El valor de retorno será, por tanto, un nuevo objeto **Set**.

Los cuatro métodos representan las operaciones matemáticas de conjuntos: **union()** devuelve un objeto **Set** que contiene la combinación de los dos argumentos, **intersection()** devuelve un objeto **Set** que contiene los elementos comunes a los dos argumentos, **difference()** resta los elementos **subset** de **superset** y **complement()** devuelve un objeto **Set** con todos los elementos que no formen parte de la intersección. Para crear un ejemplo simple que muestre los efectos de estos métodos, he aquí una enumeración que contiene diferentes nombres de acuarelas:

```

//: generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
    ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
    BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
    CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
    COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
    SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
    BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} //:-)

```

Por comodidad (para no tener que cualificar todos los nombres) importamos esta enumeración estáticamente en el ejemplo siguiente. Este ejemplo utiliza **EnumSet**, que es una herramienta de Java SE5 que permite crear conjuntos fácilmente a partir de enumeraciones (aprenderemos más de **EnumSet** en el Capítulo 19, *Tipos enumerados*). Aquí, al método estático **EnumSet.range()** se le pasan el primer y el último elemento del rango que hay que utilizar para crear el objeto **Set** resultante:

```

//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
    public static void main(String[] args) {
        Set<Watercolors> set1 =
            EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
        Set<Watercolors> set2 =
            EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
        print("set1: " + set1);
        print("set2: " + set2);
        print("union(set1, set2): " + union(set1, set2));
        Set<Watercolors> subset = intersection(set1, set2);
        print("intersection(set1, set2): " + subset);
        print("difference(set1, subset): " +
              difference(set1, subset));
        print("difference(set2, subset): " +
              difference(set2, subset));
        print("complement(set1, set2): " +

```

```

        complement(set1, set2));
    }
} /* Output: (Sample)
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET, CERULEAN_BLUE_HUE,
PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE, PERMANENT_GREEN,
VIRIDIAN_HUE, SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER, BURNT_UMBER]
union(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE, PERMANENT_GREEN, BURNT_UMBER,
COBALT_BLUE_HUE, VIOLET, BRILLIANT_RED, RAW_UMBER, ULTRAMARINE, BURNT_SIENNA, CRIMSON,
CERULEAN_BLUE_HUE, PHTHALO_BLUE, MAGENTA, VIRIDIAN_HUE]
intersection(set1, set2): [ULTRAMARINE, PERMANENT_GREEN, COBALT_BLUE_HUE, PHTHALO_BLUE,
CERULEAN_BLUE_HUE, VIRIDIAN_HUE]
difference(set1, subset): [ROSE_MADDER, CRIMSON, VIOLET, MAGENTA, BRILLIANT_RED]
difference(set2, subset): [RAW_UMBER, SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, BURNT_UMBER]
complement(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE, BURNT_UMBER, VIOLET, BRILLIANT_RED,
RAW_UMBER, BURNT_SIENNA, CRIMSON, MAGENTA]
*///:-
```

Analizando la salida, puede ver el resultado de cada una de las operaciones.

El siguiente ejemplo utiliza `Sets.difference()` para mostrar las diferencias de métodos entre diversas clases `Collection` y `Map` de `java.util`:

```

//: net/mindview/util/ContainerMethodDifferences.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class ContainerMethodDifferences {
    static Set<String> methodSet(Class<?> type) {
        Set<String> result = new TreeSet<String>();
        for(Method m : type.getMethods())
            result.add(m.getName());
        return result;
    }
    static void interfaces(Class<?> type) {
        System.out.print("Interfaces in " +
            type.getSimpleName() + ": ");
        List<String> result = new ArrayList<String>();
        for(Class<?> c : type.getInterfaces())
            result.add(c.getSimpleName());
        System.out.println(result);
    }
    static Set<String> object = methodSet(Object.class);
    static { object.add("clone"); }
    static void
    difference(Class<?> superset, Class<?> subset) {
        System.out.print(superset.getSimpleName() +
            " extends " + subset.getSimpleName() + ", adds: ");
        Set<String> comp = Sets.difference(
            methodSet(superset), methodSet(subset));
        comp.removeAll(object); // No mostrar métodos 'Object'
        System.out.println(comp);
        interfaces(superset);
    }
    public static void main(String[] args) {
        System.out.println("Collection: " +
            methodSet(Collection.class));
        interfaces(Collection.class);
        difference(Set.class, Collection.class);
        difference(HashSet.class, Set.class);
    }
}
```

```

        difference(LinkedHashSet.class, HashSet.class);
        difference(TreeSet.class, Set.class);
        difference(List.class, Collection.class);
        difference(ArrayList.class, List.class);
        difference(LinkedList.class, List.class);
        difference(Queue.class, Collection.class);
        difference(PriorityQueue.class, Queue.class);
        System.out.println("Map: " + methodSet(Map.class));
        difference(HashMap.class, Map.class);
        difference(LinkedHashMap.class, HashMap.class);
        difference(SortedMap.class, Map.class);
        difference(TreeMap.class, Map.class);
    }
}
//;-

```

La salida de este programa fue utilizada en la sección “Resumen” del Capítulo 11, *Almacenamiento de objetos*.

Ejercicio 17: (4) Analice la documentación del JDK correspondiente a **EnumSet**. Verá que hay definido un método **clone()**, clonar. Sin embargo, no podemos efectuar una clonación a partir de la referencia a la interfaz **Set** que se pasa en **Sets.java**. ¿Podría modificar **Sets.java** para tratar tanto el caso general de una interfaz **Set**, tal como se muestra, como el caso especial de un objeto **EnumSet**, utilizando **clone()** en lugar de crear un nuevo objeto **HashSet**?

Clases internas anónimas

Los genéricos también pueden utilizarse con las clases internas y con las clases internas anónimas. He aquí un ejemplo que implementa la interfaz **Generator** utilizando clases internas anónimas:

```

//: generics/BankTeller.java
// Una simulación muy simple de un cajero automático.
import java.util.*;
import net.mindview.util.*;

class Customer {
    private static long counter = 1;
    private final long id = counter++;
    private Customer() {}
    public String toString() { return "Customer " + id; }
    // A method to produce Generator objects:
    public static Generator<Customer> generator() {
        return new Generator<Customer>() {
            public Customer next() { return new Customer(); }
        };
    }
}

class Teller {
    private static long counter = 1;
    private final long id = counter++;
    private Teller() {}
    public String toString() { return "Teller " + id; }
    // Un único objeto Generator:
    public static Generator<Teller> generator =
        new Generator<Teller>() {
            public Teller next() { return new Teller(); }
        };
}

public class BankTeller {

```

```

public static void serve(Teller t, Customer c) {
    System.out.println(t + " serves " + c);
}
public static void main(String[] args) {
    Random rand = new Random(47);
    Queue<Customer> line = new LinkedList<Customer>();
    Generators.fill(line, Customer.generator(), 15);
    List<Teller> tellers = new ArrayList<Teller>();
    Generators.fill(tellers, Teller.generator(), 4);
    for(Customer c : line)
        serve(tellers.get(rand.nextInt(tellers.size())), c);
}
/* Output:
Teller 3 serves Customer 1
Teller 2 serves Customer 2
Teller 3 serves Customer 3
Teller 1 serves Customer 4
Teller 1 serves Customer 5
Teller 3 serves Customer 6
Teller 1 serves Customer 7
Teller 2 serves Customer 8
Teller 3 serves Customer 9
Teller 3 serves Customer 10
Teller 2 serves Customer 11
Teller 4 serves Customer 12
Teller 2 serves Customer 13
Teller 1 serves Customer 14
Teller 1 serves Customer 15
*///:-
```

Tanto **Customer** como **Teller** tienen constructores privados, lo que nos obliga a utilizar objetos **Generator**. **Customer** tiene un método **generator()** que genera un nuevo objeto **Generator<Customer>** cada vez que lo invocamos. Puede que no necesitemos múltiples objetos **Generator**, y **Teller** crea un único objeto **generator** público. Si analiza **main()** verá que ambas técnicas se utilizan en los métodos **fill()**.

Puesto que tanto el método **generator()** de **Customer** como el objeto **Generator** de **Teller** son estáticos, no pueden formar parte de una interfaz, así que no hay forma de hacer genérica esta función concreta. A pesar de ello funciona razonablemente bien con el método **fill()**.

Examinaremos otras versiones de este problema de versiones de colas en el Capítulo 21, *Concurrencia*.

Ejercicio 18: (3) Siguiendo la forma de **BankTeller.java**, cree un ejemplo donde el pez grande (**BigFish**) se coma al chico (**LittleFish**) en el océano (**Ocean**).

Construcción de modelos complejos

Una ventaja importante de los genéricos es la capacidad de crear modelos complejos de forma simple y segura. Por ejemplo, podemos crear fácilmente una lista de tuplas:

```

//: generics/TupleList.java
// Combinación de tipos genéricos para crear tipos genéricos complejos.
import java.util.*;
import net.mindview.util.*;

public class TupleList<A,B,C,D>
extends ArrayList<FourTuple<A,B,C,D>> {
    public static void main(String[] args) {
        TupleList<Vehicle, Amphibian, String, Integer> tl =
            new TupleList<Vehicle, Amphibian, String, Integer>();
        tl.add(TupleTest.h());
```

```

t1.add(TupleTest.h());
for(FourTuple<Vehicle,Amphibian,String,Integer> i: t1)
    System.out.println(i);
}
} /* Output: (75# match)
(Vehicle@11b86e7, Amphibian@35ce36, hi, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
*///:-

```

Aunque hace falta bastante texto (especialmente en la creación del iterador), obtenemos una estructura de datos bastante potente sin necesidad de utilizar demasiado código.

He aquí otro ejemplo donde se muestra lo fácil que es crear modelos complejos utilizando tipos genéricos. Cada clase se crea como un bloque componente y la solución total tiene múltiples partes. En este caso, el modelo corresponde a un comercio de venta al por menor con pasillos, estanterías y productos:

```

//: generics/Store.java
// Construcción de un modelo complejo utilizando contenedores genéricos.
import java.util.*;
import net.mindview.util.*;

class Product {
    private final int id;
    private String description;
    private double price;
    public Product(int IDnumber, String descr, double price){
        id = IDnumber;
        description = descr;
        this.price = price;
        System.out.println(toString());
    }
    public String toString() {
        return id + ": " + description + ", price: $" + price;
    }
    public void priceChange(double change) {
        price += change;
    }
    public static Generator<Product> generator =
        new Generator<Product>() {
            private Random rand = new Random(47);
            public Product next() {
                return new Product(rand.nextInt(1000), "Test",
                    Math.round(rand.nextDouble() * 1000.0) + 0.99);
            }
        };
}

class Shelf extends ArrayList<Product> {
    public Shelf(int nProducts) {
        Generators.fill(this, Product.generator, nProducts);
    }
}

class Aisle extends ArrayList<Shelf> {
    public Aisle(int nShelves, int nProducts) {
        for(int i = 0; i < nShelves; i++)
            add(new Shelf(nProducts));
    }
}

class CheckoutStand {}

```

```

class Office {}

public class Store extends ArrayList<Aisle> {
    private ArrayList<CheckoutStand> checkouts =
        new ArrayList<CheckoutStand>();
    private Office office = new Office();
    public Store(int nAisles, int nShelves, int nProducts) {
        for(int i = 0; i < nAisles; i++)
            add(new Aisle(nShelves, nProducts));
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Aisle a : this)
            for(Shelf s : a)
                for(Product p : s) {
                    result.append(p);
                    result.append("\n");
                }
        return result.toString();
    }
    public static void main(String[] args) {
        System.out.println(new Store(14, 5, 10));
    }
} /* Output:
258: Test, price: $400.99
861: Test, price: $160.99
868: Test, price: $417.99
207: Test, price: $268.99
551: Test, price: $114.99
278: Test, price: $804.99
520: Test, price: $554.99
140: Test, price: $530.99
...
*/

```

Como puede ver en **Store.toString()**, el resultado son muchos niveles de contenedores que, a pesar de la complejidad, resultan manejables y son seguros en lo que al tratamiento de tipos se refiere. Lo más impresionante es que no resulta demasiado complejo, desde el punto de vista intelectual, construir dicho tipo de modelos.

Ejercicio 19: (2) Siguiendo la forma de **Store.java**, construya un modelo de un buque de carga donde se utilicen contenedores metálicos para las mercancías.

El misterio del borrado

A medida que nos sumergimos más profundamente en los genéricos, aparecen una serie de aspectos que parecen no tener sentido a primera vista. Por ejemplo, aunque podemos escribir **ArrayList.class**, no podemos escribir **ArrayList<Integer>.class**. Considere también lo siguiente:

```

//: generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {
    public static void main(String[] args) {
        Class c1 = new ArrayList<String>().getClass();
        Class c2 = new ArrayList<Integer>().getClass();
        System.out.println(c1 == c2);
    }
} /* Output:
true
*/

```

`ArrayList<String>` y `ArrayList<Integer>` son claramente de tipos distintos. Los diferentes tipos se comportan de forma distinta, y si tratamos de almacenar un objeto `Integer` en un contenedor `ArrayList<String>`, obtendremos un comportamiento diferente (la operación falla) que si tratamos de almacenar ese objeto `Integer` en un contenedor `ArrayList<Integer>` (la operación si está permitida). Sin embargo, el programa anterior sugiere que ambos tipos son iguales.

He aquí otro ejemplo que aumenta todavía más la confusión:

```
//: generics/LostInformation.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
    public static void main(String[] args) {
        List<Frob> list = new ArrayList<Frob>();
        Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
        Quark<Fnorkle> quark = new Quark<Fnorkle>();
        Particle<Long,Double> p = new Particle<Long,Double>();
        System.out.println(Arrays.toString(
            list.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            map.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            quark.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(
            p.getClass().getTypeParameters()));
    }
}

} /* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*///:-
```

De acuerdo con la documentación del JDK, `Class.getTypeParameters()` “devuelve una matriz de objetos `TypeVariable` que representan las variables de tipo definidas en la declaración genérica ...” Esto parece sugerir que podríamos ser capaces de averiguar cuáles son los tipos de parámetro. Sin embargo, como podemos ver analizando la salida, lo único que podemos averiguar son los contenedores utilizados como variables para los parámetros, lo cual no constituye una información muy interesante.

La cruda realidad es que:

No hay información disponible acerca de los tipos de parámetros genéricos dentro del código genérico.

Por tanto, podemos llegar a determinar cosas como el identificador del parámetro de tipo y los límites del tipo genérico, pero no podemos llegar a determinar los parámetros de tipo reales utilizados para crear una instancia concreta. Este hecho, que resulta especialmente frustrante para los que tienen experiencia previa con C++, constituye el problema fundamental al que hay que enfrentarse cuando se trabaja con genéricos de Java.

Los genéricos de Java se implementan utilizando el mecanismo de *borrado*. Esto significa que toda la información específica de tipos se borra cuando se utiliza un genérico. Dentro de un genérico, la única cosa que sabemos es que estamos usando un objeto. Por tanto, `List<String>` y `List<Integer>` son, de hecho, el mismo tipo en tiempo de ejecución. Ambas formas se “borran” sustituyéndolas por su *tipo de origen List*. Entender este mecanismo de borrado y cómo hay que tratar con él constituye uno de los principales problemas a la hora de aprender el concepto de genéricos en Java, éste es precisamente el problema que analizaremos a lo largo de esta sección.

La técnica usada en C++

He aquí un ejemplo C++ que utiliza *plantillas*. Observará que la sintaxis para los tipos parametrizados es bastante similar, ya que Java se ha inspirado precisamente en C++:

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
    T obj;
public:
    Manipulator(T x) { obj = x; }
    void manipulate() { obj.f(); }
};

class HasF {
public:
    void f() { cout << "HasF::f()" << endl; }
};

int main() {
    HasF hf;
    Manipulator<HasF> manipulator(hf);
    manipulator.manipulate();
} /* Output:
HasF::f()
///:-
```

La clase **Manipulator** almacena un objeto de tipo **T**. Lo interesante es el método **manipulate()** que invoca un método **f()** sobre **obj**. ¿Cómo puede saber que el método **f()** existe para el parámetro de tipo **T**? El compilador C++ efectúa la comprobación cuando instanciamos la plantilla, por lo que el punto de instantación de **Manipulator<HasF>** comprueba que **HasF** tiene un método **f()**. Si no fuera así, se obtendría un error en tiempo de compilación, preservándose por tanto la seguridad referente a los tipos.

Escribir este tipo de código en C++ resulta sencillo, porque cuando se instancia una plantilla, el código de la plantilla conoce el tipo de sus parámetros de plantilla. Los genéricos de Java son distintos. He aquí la traducción de **HasF**:

```
//: generics/HasF.java

public class HasF {
    public void f() { System.out.println("HasF.f()"); }
} ///:-
```

Si tomamos el resto del ejemplo y lo traducimos a Java no se podrá compilar:

```
//: generics/Manipulation.java
// {CompileTimeError} (Won't compile)

class Manipulator<T> {
    private T obj;
    public Manipulator(T x) { obj = x; }
    // Error: no se puede encontrar el símbolo: método f():
    public void manipulate() { obj.f(); }
}

public class Manipulation {
    public static void main(String[] args) {
        HasF hf = new HasF();
        Manipulator<HasF> manipulator =
            new Manipulator<HasF>(hf);
```

```

        manipulator.manipulate();
    }
} //:-
```

Dado al mecanismo de borrado, el compilador de Java no puede relacionar el requisito de que `manipulate()` debe ser capaz de invocar `f()` sobre `obj` con el hecho de que `HasF` tiene un método `f()`. Para poder invocar `f()`, debemos ayudar a la clase genérica, proporcionándola un *límite* que indique al compilador que sólo debe aceptar los tipos que se conformen con dicho límite. Para esto se utiliza la palabra clave `extends`. Una vez que se incluye el límite, si que se puede realizar la compilación:

```

//: generics/Manipulator2.java

class Manipulator2<T extends HasF> {
    private T obj;
    public Manipulator2(T x) { obj = x; }
    public void manipulate() { obj.f(); }
} //:-
```

El límite `<T extends HasF>` dice que `T` debe ser de tipo `HasF` o algo derivado de `HasF`. Si es así, entonces resulta seguro invocar `f()` sobre `obj`.

Decimos, a este respecto, que el parámetro de tipo genérico *se borra de acuerdo con su primer límite* (es posible tener múltiples límites, como veremos posteriormente). También hablamos en relación con esto, del *borrado del parámetro de tipo*. El compilador, en la práctica, sustituye al parámetro de tipo por lo que el límite indique, de modo que en el caso anterior `T` se borra y se sustituye por `HasF`, lo cual es lo mismo que sustituir `T` por `HasF` en el cuerpo de la clase.

El lector podría pensar, correctamente, que en `Manipulator2.java`, los genéricos no proporcionan ninguna ventaja. Podríamos perfectamente realizar el borrado de tipos nosotros y crear una clase sin genéricos:

```

//: generics/Manipulator3.java

class Manipulator3 {
    private HasF obj;
    public Manipulator3(HasF x) { obj = x; }
    public void manipulate() { obj.f(); }
} //:-
```

Esto nos plantea una cuestión importante: los genéricos sólo son útiles cuando deseamos utilizar parámetros de tipo que sean más “genéricos” que un tipo específico (y todos sus subtipos); en otras palabras, cuando queramos escribir código que funcione con múltiples clases. Como resultado, los parámetros de tipo y su aplicación dentro de un fragmento útil de código genérico serán normalmente más complejos que una simple sustitución de clases. Sin embargo, no debemos concluir por ello que cualquier cosa de la forma `<T extends HasF>` no tiene ningún sentido. Por ejemplo, si una clase tiene un método que devuelve `T`, entonces los genéricos son útiles, porque permitirán devolver el tipo exacto:

```

//: generics/ReturnGenericType.java

class ReturnGenericType<T extends HasF> {
    private T obj;
    public ReturnGenericType(T x) { obj = x; }
    public T get() { return obj; }
} //:-
```

Es preciso examinar todo el código y determinar si es lo suficientemente complejo como para merecer el uso de genéricos. Examinaremos el tema de los límites con más detalle más adelante en el capítulo.

Ejercicio 20: (1) Cree una interfaz con dos métodos y una clase que implemente dicha interfaz y añada un tercer método. En otra clase, cree un método genérico con un tipo de argumento que esté limitado por la interfaz y demuestre que los métodos de la interfaz son invocables dentro de este método genérico. En `main()`, pase una instancia de la clase implementadora al método genérico.

Compatibilidad de la migración

Para eliminar cualquier potencial confusión acerca del mecanismo de borrado de tipos, es necesario entender claramente que *no se trata* de una característica del lenguaje. Se trata de un compromiso en la implementación de los genéricos de Java, compromiso que es necesario porque los genéricos no han formado parte del lenguaje desde el principio. Este compromiso puede crearnos algunos quebraderos de cabeza, por lo que es necesario acostumbrarse a él lo antes posible y entender a qué se debe.

Si los genéricos hubieran formado parte de Java 1.0, esta funcionalidad no se habría implementado utilizando el mecanismo de borrado de tipos, sino que se habría empleado el mecanismo de la *reificación* para retener los parámetros de tipo como entidades de primera clase, de modo que seríamos capaces de realizar, con los parámetros de tipo, operaciones de reflexión y operaciones del lenguaje basadas en tipos. Veremos posteriormente en el capítulo que el mecanismo de borrado reduce el aspecto “genérico” de los genéricos. Los genéricos siguen siendo útiles en Java, pero lo que pasa es que no son tan útiles como podrían ser, y la razón de ello es precisamente el mecanismo de borrado de tipos.

En una implementación basada en dicho mecanismo, los tipos genéricos se tratan como tipos de segunda clase que no pueden utilizarse en algunos contextos importantes. Los tipos genéricos sólo están presentes durante la comprobación estática de tipos, después de lo cual todo tipo genérico del programa se borra, sustituyéndolo por un tipo límite no genérico. Por ejemplo, las anotaciones de tipos como `List<T>` se borran sustituyéndolas por `List`, y las variables de tipos normales se borran sustituyéndolas por `Object` a menos que se especifique un límite.

La principal motivación para el mecanismo de borrado de tipos es que permite utilizar clientes de código genérico con bibliotecas no genéricas, y viceversa. Esto se denomina a menudo *compatibilidad de la migración*. En un mundo ideal, existiría un punto de partida en el que todo hubiera sido hecho genérico a la vez. En la realidad, incluso aunque los programadores sólo estén escribiendo código genérico, se verán forzados a tratar con bibliotecas no genéricas que hayan sido escritas antes de la aparición de Java SE5. Los autores de esas bibliotecas puede que no lleguen nunca a tener ningún motivo para hacer su código más genérico, o puede simplemente que tarden algún tiempo en ponerse manos a la obra.

Por ello, los genéricos de Java no sólo deben soportar la *compatibilidad descendente* (el código y los archivos de clase existentes siguen siendo legales y continúan significando lo que antes significaban) sino que también tienen que soportar la compatibilidad de migración, de modo que las bibliotecas puedan llegar a ser genéricas a su propio ritmo y de modo también que, cuando una biblioteca se reescriba en forma genérica, no haga que dejen de funcionar el código y las aplicaciones que dependen de ella. Después de decidir que el objeto era éste, los diseñadores de Java y diversos grupos que estaban trabajando en el problema, decidieron que el borrado de tipos era la única solución factible. El mecanismo de borrado de tipos permite esta migración hacia el código genérico, al conseguir que el código no genérico pueda coexistir con el que sí lo es.

Por ejemplo, suponga que una aplicación utiliza dos bibliotecas, `X` e `Y`, y que `Y` utiliza la biblioteca `Z`. Con la aparición de Java SE5, los creadores de esta aplicación y de estas bibliotecas probablemente terminen por efectuar una migración hacia código genérico. Sin embargo, cada uno de esos diseñadores tendrá diferentes motivaciones y diferentes restricciones en lo que respecta a dicha migración. Para conseguir la compatibilidad de migración, cada biblioteca y aplicación tiene que ser independiente de todas las demás en lo que respecta a la utilización de genéricos. Por tanto, no deben ser capaces de detectar si las otras bibliotecas están utilizando genéricos o no. En consecuencia, la evidencia de que una biblioteca concreta está usando genéricos debe ser “borrada”.

Sin algún tipo de ruta de migración, todas las bibliotecas que hubieran sido diseñadas a lo largo del tiempo correrían el riesgo de no poder ser utilizadas por los desarrolladores que decidieran comenzar a utilizar los genéricos de Java. Pero como las bibliotecas son la parte del lenguaje de programación que mayor impacto tiene sobre la productividad, este coste no resultaba aceptable. Si el mecanismo de borrado de tipos era la mejor ruta de migración posible o la única existente, es algo que sólo el tiempo nos dirá.

El problema del borrado de tipos

Por tanto, la justificación principal para el borrado de tipos es el proceso de transición de código no genérico a código genérico, y la incorporación de genéricos dentro del lenguaje sin hacer que dejen de funcionar las bibliotecas existentes. El borrado de tipos permite que el código de cliente existente, no genérico, continúe pudiendo ser usado sin modificación, hasta que los clientes estén listos para reescribir el código de cara a utilizar genéricos. Se trata de una motivación muy noble, porque no hace que de repente deje de funcionar todo el código existente.

El coste del borrado de tipos es significativo. Los tipos genéricos no pueden utilizarse en operaciones que hagan referencia explícita a tipos de tiempo de ejecución; como ejemplo de estas operaciones podemos citar las proyecciones de tipos, las operaciones `instanceof` y las expresiones `new`. Como toda la información de tipos acerca de los parámetros se pierde, cada vez que escribamos código genérico debemos estar perpetuamente acordándonos de que la idea de que disponemos de información de tipos es solo *aparente*. Por tanto, cuando escribimos un fragmento de código como éste:

```
class Foo<T> {
    T var;
}
```

podría parecer que al crear una instancia de `Foo`:

```
Foo<Cat> f = new Foo<Cat>();
```

el código de la clase `Foo` debería saber que ahora está trabajando con un objeto `Cat`. La sintaxis sugiere de manera directa que el tipo `T` está siendo sustituido a lo largo de toda la clase. Pero en realidad no es así y debemos siempre tener presente, cuando estemos escribiendo el código para la clase, que se trata simplemente de un objeto de tipo `Object`.

Además, el borrado de tipos y la compatibilidad de migración significan que el uso de genéricos no se impone en aquellas ocasiones en que sería bueno que se impusiera:

```
//: generics/ErasureAndInheritance.java

class GenericBase<T> {
    private T element;
    public void set(T arg) { arg = element; }
    public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // Ninguna advertencia

// class Derived3 extends GenericBase<?> {}
// Extraño error:
//   unexpected type found : ?
//   required: class or interface without bounds

public class ErasureAndInheritance {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Derived2 d2 = new Derived2();
        Object obj = d2.get();
        d2.set(obj); // ¡Advertencia aquí!
    }
} ///:-
```

`Derived2` hereda de `GenericBase` sin ningún parámetro genérico y el compilador no genera ninguna advertencia. La advertencia no se genera hasta que se invoca `set()`.

Para que no aparezca la advertencia, Java proporciona una anotación, que es la que podemos ver en el listado (esta anotación no estaba soportada en las versiones anteriores a Java SE5):

```
@SuppressWarnings("unchecked")
```

Observe que esta anotación se coloca en el método que genera la advertencia, en lugar de en la clase completa. Es mejor "enfocar" lo máximo posible a la hora de desactivar una advertencia, para no ocultar accidentalmente un problema real al desactivar las advertencias en un contexto demasiado amplio.

Presumiblemente, el error producido por `Derived3` indica que el compilador espera una clase base pura.

Añadamos a esto el esfuerzo adicional de gestionar los límites cuando queramos tratar el parámetro de tipo como algo más que simplemente un objeto de tipo `Object` y la conclusión de todo ello es que hace falta un esfuerzo mucho mayor con unas

ventajas mucho menores que cuando se utilizan tipos parametrizados en lenguajes tales como C++, Ada o Eiffel. Esto no quiere decir que dichos lenguajes tengan en general más ventajas que Java a la hora de abordar la mayoría de los problemas de programación, sino simplemente que sus mecanismos de tipos parametrizados son más flexibles y potentes que los de Java.

El efecto de los límites

Debido al mecanismo de borrado de tipos, el aspecto más confuso de los genéricos es el hecho de que podemos representar cosas que no tienen ningún significado. Por ejemplo:

```
//: generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
    private Class<T> kind;
    public ArrayMaker(Class<T> kind) { this.kind = kind; }
    @SuppressWarnings("unchecked")
    T[] create(int size) {
        return (T[])Array.newInstance(kind, size);
    }
    public static void main(String[] args) {
        ArrayMaker<String> stringMaker =
            new ArrayMaker<String>(String.class);
        String[] stringArray = stringMaker.create(9);
        System.out.println(Arrays.toString(stringArray));
    }
} /* Output:
[null, null, null, null, null, null, null, null, null]
*///:-
```

Aunque **kind** se almacena como **Class<T>**, el mecanismo de borrado de tipos significa que en realidad se está almacenando simplemente como un objeto **Class** sin ningún parámetro. Por tanto, cuando hacemos algo con ese objeto, como por ejemplo crear una matriz, **Array.newInstance()** no dispone en la práctica de la información de tipos que está implícita en **kind**; como consecuencia, no puede producir el resultado específico, lo que obliga a realizar una proyección de tipo que genera una advertencia que no se puede corregir.

Observe que la utilización de **Array.newInstance()** es la técnica recomendada para la creación de matrices dentro de genéricos.

Si creamos un contenedor en lugar de una matriz, las cosas son distintas:

```
//: generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
    List<T> create() { return new ArrayList<T>(); }
    public static void main(String[] args) {
        ListMaker<String> stringMaker= new ListMaker<String>();
        List<String> stringList = stringMaker.create();
    }
} //:-
```

El compilador no genera ninguna advertencia, aún cuando sabemos perfectamente (debido al mecanismo de borrado de tipos) que la **<T>** en **new ArrayList<T>()** dentro de **create()** se elimina: en tiempo de ejecución no hay ninguna **<T>** dentro de la clase, por lo que parece que no tiene ningún significado. Pero si hacemos caso de esta idea y cambiamos la expresión a **new ArrayList()**, el compilador generará una advertencia.

¿Carece realmente de significado en este caso? ¿Qué pasaría si pusiéramos algunos objetos en la lista antes de devolverla, como en el siguiente ejemplo?

```
//: generics/FilledListMaker.java
import java.util.*;

public class FilledListMaker<T> {
    List<T> create(T t, int n) {
        List<T> result = new ArrayList<T>();
        for(int i = 0; i < n; i++)
            result.add(t);
        return result;
    }
    public static void main(String[] args) {
        FilledListMaker<String> stringMaker =
            new FilledListMaker<String>();
        List<String> list = stringMaker.create("Hello", 4);
        System.out.println(list);
    }
} /* Output:
[Hello, Hello, Hello, Hello]
*///:-
```

Aunque el compilador es incapaz de tener ninguna información acerca de **T** en **create()**, sigue pudiendo garantizar (en tiempo de compilación) que lo que pongamos dentro de **result** es de tipo **T**, de modo que concuerde con **ArrayList<T>**. Por tanto, aún cuando el mecanismo de borrado de tipos elimine la información acerca del tipo real dentro de un método o de una clase, el compilador sigue pudiendo garantizar la coherencia interna en lo que respecta a la forma en que se utiliza el tipo dentro del método o de la clase.

Puesto que el mecanismo de borrado de tipos elimina la información de tipos en el cuerpo de un método, lo que importa en tiempo de ejecución son los *límites*: los puntos en los que los objetos entran y salen de un método. Estos son los puntos en los que el compilador realiza las comprobaciones de tipos en tiempo de compilación e inserta código de proyección de tipos. Considere el siguiente ejemplo no genérico:

```
//: generics/SimpleHolder.java

public class SimpleHolder {
    private Object obj;
    public void set(Object obj) { this.obj = obj; }
    public Object get() { return obj; }
    public static void main(String[] args) {
        SimpleHolder holder = new SimpleHolder();
        holder.set("Item");
        String s = (String)holder.get();
    }
} //:-
```

Si descompilamos el resultado con **javap -c SimpleHolder**, obtenemos (después de editar la salida):

```
public void set(java.lang.Object);
  0:   aload_0
  1:   aload_1
  2:   putfield #2; //Campo obj:Object;
  5:   return

public java.lang.Object get();
  0:   aload_0
  1:   getfield #2; //Campo obj:Object;
  4:   areturn

public static void main(java.lang.String[]);
  0:   new #3; //Clase SimpleHolder
  3:   dup
  4:   invokespecial #4; //Método "<init>":()V
```

```

7:      astore_1
8:      aload_1
9:      ldc #5; //String Item
11:     invokevirtual #6; // Método set:(Object;)V
14:     aload_1
15:     invokevirtual #7; // Método get:()Object;
18:     checkcast #8; //class java/lang/String
21:     astore_2
22:     return

```

Los métodos `set()` y `get()` simplemente almacenan y producen el valor, y la precisión de tipos se comprueba en el lugar donde se produce la llamada a `get()`.

Ahora vamos a incorporar genéricos al código anterior:

```

//: generics/GenericHolder.java

public class GenericHolder<T> {
    private T obj;
    public void set(T obj) { this.obj = obj; }
    public T get() { return obj; }
    public static void main(String[] args) {
        GenericHolder<String> holder =
            new GenericHolder<String>();
        holder.set("Item");
        String s = holder.get();
    }
} //:-

```

La necesidad de efectuar una proyección de tipos para `get()` ha desaparecido, pero también sabemos que el valor pasado a `set()` sufre una comprobación de tipos en tiempo de compilación. He aquí el código intermedio relevante:

```

public void set(java.lang.Object);
0:      aload_0
1:      aload_1
2:      putfield #2; //Campo obj:Object;
5:      return

public java.lang.Object get();
0:      aload_0
1:      getfield #2; //Campo obj:Object;
4:      areturn

public static void main(java.lang.String[]);
0:      new #3; //Clase GenericHolder
3:      dup
4:      invokespecial #4; //Método "<init>":()V
7:      astore_1
8:      aload_1
9:      ldc #5; //String Item
11:     invokevirtual #6; // Método set:(Object;)V
14:     aload_1
15:     invokevirtual #7; // Método get:()Object;
18:     checkcast #8; //class java/lang/String
21:     astore_2
22:     return

```

El código resultante es idéntico. El trabajo adicional de comprobar el tipo entrante en `set()` es nulo, ya que es el compilador quien se encarga de realizarlo. Y la proyección de tipos para el valor saliente de `get()` sigue estando ahí, pero ahora no tenemos que hacerlo nosotros explícitamente; el compilador se encarga de insertar esa proyección automáticamente, de modo que el código que escribamos (y que tengamos que leer) estará mucho más libre de “ruido”.

Puesto que `get()` y `set()` generan el mismo código intermedio, toda la acción referida a los genéricos tiene lugar en los límites; en concreto, se trata de la comprobación adicional en tiempo de compilación para los valores entrantes y de la proyección de tipos que se inserta para los valores salientes. Para contrarrestar la confusión en lo que respecta al tema del mecanismo de borrado de tipos, recuerde siempre que “los límites son los lugares en los que la acción se produce”.

Compensación del borrado de tipos

Como hemos visto, el borrado de tipos hace que perdamos la posibilidad de realizar ciertas operaciones en el código genérico. En concreto, no funcionará ninguna cosa que requiera conocer el tipo exacto en tiempo de ejecución:

```
//: generics/Erased.java
// {CompileTimeError} (no se compilará)

public class Erased<T> {
    private final int SIZE = 100;
    public static void f(Object arg) {
        if(arg instanceof T) {}           // Error
        T var = new T();                 // Error
        T[] array = new T[SIZE];         // Error
        T[] array = (T)new Object[SIZE]; // Advertencia de no comprobación
    }
} ///:-
```

Ocasionalmente, podemos solventar mediante programa estos problemas, pero en ocasiones nos vemos forzados a compensar el mecanismo de borrado de tipos introduciendo lo que se denomina un *marcador de tipos*. Esto quiere decir que pasamos explícitamente el objeto **Class** correspondiente a nuestro tipo para poder utilizarlo en expresiones donde los tipos entran en juego.

Por ejemplo, el intento de utilizar `instanceof` en el programa anterior falla porque la información de tipos ha sido borrada. Si introducimos un marcador de tipos, podemos utilizar en su lugar un método `isInstance()` dinámico:

```
//: generics/ClassTypeCapture.java

class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
    Class<T> kind;
    public ClassTypeCapture(Class<T> kind) {
        this.kind = kind;
    }
    public boolean f(Object arg) {
        return kind.isInstance(arg);
    }
    public static void main(String[] args) {
        ClassTypeCapture<Building> ctt1 =
            new ClassTypeCapture<Building>(Building.class);
        System.out.println(ctt1.f(new Building()));
        System.out.println(ctt1.f(new House()));
        ClassTypeCapture<House> ctt2 =
            new ClassTypeCapture<House>(House.class);
        System.out.println(ctt2.f(new Building()));
        System.out.println(ctt2.f(new House()));
    }
} /* Output:
true
true
false
true
*///:-
```

El compilador garantiza que el marcador de tipos se corresponda con el argumento genérico.

Ejercicio 21: (4) Modifique `ClassTypeCapture.java` añadiendo un contenedor `Map<String, Class<?>>`, un método `addType(String typename, Class<?> kind)` y un método `createNew(String typename)`. `createNew()` generará una nueva instancia de la clase asociada con la cadena de caracteres que se le proporcione como argumento, o producirá un mensaje de error.

Creación de instancias de tipos

El intento de crear un objeto con `new T()` en `Erased.java` no funciona, en parte debido al mecanismo de borrado de tipos y en parte porque el compilador no puede verificar que `T` tenga un constructor predeterminado (sin argumentos). Pero en C++ esta operación es natural, sencilla y segura (se comprueba en tiempo de compilación):

```
///: generics/InstantiateGenericType.cpp
// ;C++, no Java!

template<class T> class Foo {
    T x; // Crear un campo de tipo T
    T* y; // Puntero a T
public:
    // Inicializar el puntero:
    Foo() { y = new T(); }
};

class Bar {};

int main() {
    Foo<Bar> fb;
    Foo<int> fi; // ... y funciona con primitivas
} //:-
```

La solución en Java consiste en pasar un objeto factoría y utilizarlo para crear la nueva instancia. Un objeto factoría muy adecuado es el propio objeto `Class`, por lo que si utilizamos un marcador de tipos, podemos emplear `newInstance()` para crear un nuevo objeto de dicho tipo:

```
///: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
    T x;
    public ClassAsFactory(Class<T> kind) {
        try {
            x = kind.newInstance();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Employee {}

public class InstantiateGenericType {
    public static void main(String[] args) {
        ClassAsFactory<Employee> fe =
            new ClassAsFactory<Employee>(Employee.class);
        print("ClassAsFactory<Employee> succeeded");
        try {
            ClassAsFactory<Integer> fi =
                new ClassAsFactory<Integer>(Integer.class);
        }
```

```

    } catch(Exception e) {
        print("ClassAsFactory<Integer> failed");
    }
}
/* Output:
ClassAsFactory<Employee> succeeded
ClassAsFactory<Integer> failed
*///:-

```

Este ejemplo se puede compilar, pero falla si utilizamos `ClassAsFactory<Integer>` porque `Integer` no dispone de ningún constructor predeterminado. Como el error no se detecta en tiempo de compilación, la gente de Sun desaconseja utilizar esta técnica. Lo que sugieren, en su lugar, es que se utilice una factoría explícita y que se restrinja el tipo de modo que sólo admite una clase que implemente dicha factoría:

```

//: generics/FactoryConstraint.java

interface FactoryI<T> {
    T create();
}

class Foo2<T> {
    private T x;
    public <F extends FactoryI<T>> Foo2(F factory) {
        x = factory.create();
    }
    // ...
}

class IntegerFactory implements FactoryI<Integer> {
    public Integer create() {
        return new Integer(0);
    }
}

class Widget {
    public static class Factory implements FactoryI<Widget> {
        public Widget create() {
            return new Widget();
        }
    }
}

public class FactoryConstraint {
    public static void main(String[] args) {
        new Foo2<Integer>(new IntegerFactory());
        new Foo2<Widget>(new Widget.Factory());
    }
} //://:-

```

Observe que esto no es más que una variación del hecho de pasar `Class<T>`. Ambas técnicas pasan objetos factoría; pero `Class<T>` resulta ser el objeto factoría predefinido, mientras que en el ejemplo anterior creamos un objeto factoría explícito. Lo importante es que conseguimos que se realice una comprobación en tiempo de compilación.

Otra técnica consiste en utilizar el patrón de diseño *basado en plantillas*. En el siguiente ejemplo, `get()` es el método plantilla y `create()` se define en la subclase para generar un objeto de dicho tipo:

```

//: generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
    final T element;
    GenericWithCreate() { element = create(); }

```

```

    abstract T create();
}

class X {}

class Creator extends GenericWithCreate<X> {
    X create() { return new X(); }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
}

public class CreatorGeneric {
    public static void main(String[] args) {
        Creator c = new Creator();
        c.f();
    }
} /* Output:
X
*///:-

```

Ejercicio 22: (6) Utilice un marcador de tipos junto con el mecanismo de reflexión para crear un método que emplee la versión con argumentos de `newInstance()` con el fin de crear un objeto de una clase con un constructor que tenga argumentos.

Ejercicio 23: (1) Modifique `FactoryConstraint.java` para que `create()` admita un argumento.

Ejercicio 24: (3) Modifique el Ejercicio 21 para que los objetos factoría se almacenen en el mapa en lugar de en `Class<?>`.

Matrices de genéricos

Como hemos visto en `Erased.java`, no se pueden crear matrices de genéricos. La solución consiste en utilizar un contenedor `ArrayList` en todos aquellos lugares donde necesitemos crear una matriz de genéricos:

```

//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
    private List<T> array = new ArrayList<T>();
    public void add(T item) { array.add(item); }
    public T get(int index) { return array.get(index); }
} //://:-

```

Aquí obtenemos el comportamiento de una matriz, sin renunciar por ello a las comprobaciones de tipos en tiempo de compilación que los genéricos permiten.

En ocasiones, seguiremos necesitando crear una matriz de tipos genéricos (`ArrayList`, por ejemplo, utiliza matrices internamente). Resulta interesante saber que podemos definir una referencia de una forma tal que haga que el compilador no se queje. Por ejemplo:

```

//: generics/ArrayOfGenericReference.java

class Generic<T> {}

public class ArrayOfGenericReference {
    static Generic<Integer>[] gia;
} //://:-

```

El compilador acepta esta sintaxis sin generar ninguna advertencia. Pero no podemos crear nunca una matriz de ese tipo exacto (incluyendo los parámetros de tipo), por lo que la cuestión resulta algo confusa. Puesto que todas las matrices tienen

la misma estructura (tamaño de cada posición de la matriz y disposición de la matriz) independientemente del tipo que almacene, parece que deberíamos poder crear una matriz de objetos **Object** y proyectarla sobre el tipo de matriz deseado. De hecho, esta solución podrá compilarse, pero no se podrá ejecutar, ya que se generará una excepción **ClassCastException**:

```
//: generics/ArrayOfGeneric.java

public class ArrayOfGeneric {
    static final int SIZE = 100;
    static Generic<Integer>[] gia;
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        // Se compila; genera ClassCastException:
        ///!gia = (Generic<Integer>[])new Object[SIZE];
        // El tipo en tiempo de ejecución es el raw (después del borrado):
        gia = (Generic<Integer>[])new Generic[SIZE];
        System.out.println(gia.getClass().getSimpleName());
        gia[0] = new Generic<Integer>();
        ///!gia[1] = new Object(); // Error de tiempo de compilación
        // Descubre la discordancia de tipos en tiempo de compilación:
        ///!gia[2] = new Generic<Double>();
    }
} /* Output:
Generic[]
*///:-
```

El problema es que las matrices controlan su tipo real y ese tipo se establece en el momento de creación de la matriz. Por tanto, aunque **gia** haya sido proyectado sobre **Generic<Integer>[]**, dicha información sólo existe en tiempo de compilación (y sin la anotación **@SuppressWarnings**, obtendremos una advertencia debido a dicha proyección). En tiempo de ejecución, sigue siendo una matriz de tipo **Object**, y eso hace que se produzcan problemas. La única forma de crear adecuadamente una matriz de un tipo genérico es crear una nueva matriz del tipo resultante del borrado de tipos y efectuar una proyección de tipos con dicha matriz.

Veamos un ejemplo algo más sofisticado. Consideré un envoltorio genérico simple para una matriz:

```
//: generics/GenericArray.java

public class GenericArray<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArray(int sz) {
        array = (T[])new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Método que expone la representación subyacente:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArray<Integer> gai =
            new GenericArray<Integer>(10);
        // Esto provoca una excepción ClassCastException:
        ///! Integer[] ia = gai.rep();
        // Esto es correcto:
        Object[] oa = gai.rep();
    }
} /*:-
```

Como antes, no podemos decir **T[] array = new T[sz]**, por lo que creamos una matriz de objetos y la proyectamos.

El método `rep()` devuelve una matriz `T[]`, que en `main()` debe ser una matriz `Integer[]` para `gai`, pero si llamamos a ese método y tratamos de capturar el resultado como una referencia a `Integer[]`, obtenemos una excepción `ClassCastException`, de nuevo debido a que el tipo real en tiempo de ejecución es `Object[]`.

Si compilamos `GenericArray.java` después de desactivar mediante comentarios la anotación `@SuppressWarnings`, el compilador genera una advertencia:

```
Note: GenericArray.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

En este caso, hemos obtenido una única advertencia y parece que se refiere a la operación de proyección de tipos. Pero si queremos asegurarnos, debemos realizar la compilación con `-Xlint:unchecked`:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
  found   : java.lang.Object[]
  required: T[]
      array = (T[])new Object[sz];
                           ^
1 warning
```

Ciertamente, el compilador se está quejando sobre la proyección de tipos. Puesto que las advertencias introducen ruido dentro del proceso de diseño, lo mejor que podemos hacer, una vez que verifiquemos que se produce una advertencia, es desactivarla utilizando `@SuppressWarnings`. De esa forma, cuando aparezca alguna otra advertencia podremos investigarla adecuadamente.

Debido al mecanismo de borrado de tipos, el tipo en tiempo de ejecución de la matriz sólo puede ser `Object[]`. Si lo proyectamos inmediatamente sobre `T[]`, entonces el tipo real de la matriz se perderá en tiempo de compilación y el compilador podría no aplicar algunas comprobaciones de potenciales errores. Debido a esto, es mejor utilizar una matriz `Object[]` dentro de la colección y añadir una proyección a `T` cuando la usemos como elemento de la matriz. Veamos cómo se aplicaría esta solución con el ejemplo `GenericArray.java`:

```
//: generics/GenericArray2.java

public class GenericArray2<T> {
    private Object[] array;
    public GenericArray2(int sz) {
        array = new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    @SuppressWarnings("unchecked")
    public T get(int index) { return (T)array[index]; }
    @SuppressWarnings("unchecked")
    public T[] rep() {
        return (T[])array; // Advertencia: proyección no comprobada
    }
    public static void main(String[] args) {
        GenericArray2<Integer> gai =
            new GenericArray2<Integer>(10);
        for(int i = 0; i < 10; i++)
            gai.put(i, i);
        for(int i = 0; i < 10; i++)
            System.out.print(gai.get(i) + " ");
        System.out.println();
        try {
            Integer[] ia = gai.rep();
        } catch(Exception e) { System.out.println(e); }
    }
} /* Output: (Sample)
0 1 2 3 4 5 6 7 8 9
```

```
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to [Ljava.lang.Integer;
*///:-
```

Inicialmente, parece que las cosas no son muy distintas, salvo por el hecho de que hemos desplazado de lugar la proyección de tipos. Sin las anotaciones `@SuppressWarnings`, seguimos obteniendo advertencias que nos dicen que faltan comprobaciones. Sin embargo, la representación interna es ahora `Object[]` en lugar de `T[]`. Cuando invocamos `get()`, se efectúa la proyección del objeto sobre `T`, que es de hecho el tipo correcto, por lo que la operación es segura. Sin embargo, si invocamos `rep()`, el método vuelve a intentar proyectar `Object[]` sobre `T[]`, lo que sigue siendo incorrecto y genera una advertencia en tiempo de compilación y una excepción en tiempo de ejecución. Por tanto, no hay ninguna manera de cambiar el tipo de la matriz subyacente, que sólo puede ser `Object[]`. La ventaja de tratar `array` internamente como `Object[]` en lugar de como `T[]` es que resulta menos probable que nos olvidemos del tipo de la matriz en tiempo de ejecución e introduzcamos accidentalmente un error (aunque la mayoría de esos errores, y quizás todos, se detectarían rápidamente en tiempo de ejecución).

Para el nuevo código que desarrollemos, lo que debemos hacer es pasar un testigo de tipos. En dicho caso, `GenericArray` tendría el aspecto siguiente:

```
//: generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;

public class GenericArrayWithTypeToken<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArrayWithTypeToken(Class<T> type, int sz) {
        array = (T[])Array.newInstance(type, sz);
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Exponer la representación subyacente:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArrayWithTypeToken<Integer> gai =
            new GenericArrayWithTypeToken<Integer>(
                Integer.class, 10);
        // Esto ahora funciona:
        Integer[] ia = gai.rep();
    }
} //:-
```

El testigo de tipos `Class<T>` se pasa al constructor para compensar el mecanismo de borrado de tipos, con el fin de poder crear el tipo real de matriz que necesitemos, aunque el mensaje de advertencia referido a la proyección de tipos deberá ser suprimido mediante `@SuppressWarnings`. Una vez que obtengamos el tipo real, podemos devolverlo y obtener los resultados deseados como podemos ver en `main()`. El tipo de la matriz en tiempo de ejecución es el tipo exacto `T[]`.

Lamentablemente, si examinamos el código fuente en las bibliotecas estándar de Java SE5, podremos ver que existen por todas partes proyecciones de matrices de tipo `Object` a tipos parametrizados. Por ejemplo, he aquí el constructor `ArrayList` que utiliza como argumento un contenedor `Collection`, después de simplificar y limpiar el código un poco:

```
public ArrayList(Collection c) {
    size = c.size();
    elementData = (E[])new Object[size];
    c.toArray(elementData);
}
```

Si examina el código de `ArrayList.java`, podrá encontrar multitud de estas proyecciones. ¿Y qué es lo que sucede cuando compilamos este código?

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Como puede ver, las bibliotecas estándar generan una multitud de advertencias. Si el lector ha trabajado antes con C, y especialmente con el C anterior al estándar ANSI, recordará un efecto muy concreto de las advertencias: en el momento en que descubrimos que las podemos ignorar, las ignoramos completamente. Por esa razón, lo mejor es que intentemos que el compilador no genere ningún tipo de mensaje, a menos que el programador deba hacer algo con ese mensaje.

En su bitácora web,³ Neal Gafter (uno de los principales desarrolladores de Java SE5) apunta que le daba bastante pereza reescribir las bibliotecas Java, y que los buenos programadores no deben imitar lo que él hizo. Neal también señala que le hubiera resultado imposible corregir parte del código de la biblioteca Java sin modificar la interfaz existente. Por tanto, aunque en los archivos de código fuente de la biblioteca Java aparezcan ciertas técnicas, eso no quiere decir que esa sea la forma correcta de hacer las cosas. Cuando examine el código de biblioteca no dé por sentado que se trate de un ejemplo que haya de seguir en su propio código.

Límites

Ya hemos presentado brevemente los *límites* anteriormente en el capítulo. Los límites nos permiten imponer restricciones a los tipos de parámetros que pueden utilizarse con los genéricos. Aunque esto nos permite imponer reglas acerca de los tipos a los que pueden aplicarse los genéricos, un efecto quizás más importante es que podemos invocar métodos pertenecientes a los tipos definidos como límite.

Puesto que el mecanismo de borrado de tipos elimina la información de tipos, los únicos métodos que podemos invocar para un parámetro genérico al que no se le hayan impuesto límites son aquellos disponibles para **Object**. Sin embargo, si somos capaces de restringir el parámetro para que se corresponda con un subconjunto de tipos, entonces podemos invocar todos los métodos de dicho subconjunto. Para implementar esta restricción, el mecanismo de genéricos en Java utiliza la palabra clave **extends**. Es importante comprender que **extends** tiene un significado bastante distinto al normal dentro del contexto de los límites de genéricos. El siguiente ejemplo ilustra los fundamentos básicos de los mecanismos de límites:

```
//: generics/BasicBounds.java

interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
    T item;
    Colored(T item) { this.item = item; }
    T getItem() { return item; }
    // El límite nos permite invocar un método:
    java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }

// Esto no funcionará -- primero hay que definir la clase
// y luego las interfaces:
// class ColoredDimension<T extends HasColor & Dimension> {

// Múltiples límites:
class ColoredDimension<T extends Dimension & HasColor> {
    T item;
    ColoredDimension(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}
```

³ <http://gaffer.blogspot.com/2004/09/puzzling-through-erasure-answer.html>

```

interface Weight { int weight(); }

// Al igual que con la herencia, sólo se puede tener una
// clase concreta, pero puede haber múltiples interfaces:
class Solid<T extends Dimension & HasColor & Weight> {
    T item;
    Solid(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
    int weight() { return item.weight(); }
}

class Bounded
extends Dimension implements HasColor, Weight {
    public java.awt.Color getColor() { return null; }
    public int weight() { return 0; }
}

public class BasicBounds {
    public static void main(String[] args) {
        Solid<Bounded> solid =
            new Solid<Bounded>(new Bounded());
        solid.color();
        solid.getY();
        solid.weight();
    }
} //:-)

```

Habrá observado que **BasicBounds.java** parece contener redundancias que podrían eliminarse recurriendo al mecanismo de herencia. En el siguiente ejemplo, podemos ver cómo cada nivel de herencia añade también restricciones de límite:

```

//: generics/InheritBounds.java

class HoldItem<T> {
    T item;
    HoldItem(T item) { this.item = item; }
    T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
    Colored2(T item) { super(item); }
    java.awt.Color color() { return item.getColor(); }
}

class ColoredDimension2<T extends Dimension & HasColor>
extends Colored2<T> {
    ColoredDimension2(T item) { super(item); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}

class Solid2<T extends Dimension & HasColor & Weight>
extends ColoredDimension2<T> {
    Solid2(T item) { super(item); }
    int weight() { return item.weight(); }
}

```

```

public class InheritBounds {
    public static void main(String[] args) {
        Solid2<Bounded> solid2 =
            new Solid2<Bounded>(new Bounded());
        solid2.color();
        solid2.getY();
        solid2.weight();
    }
} //:-)

```

HoldItem simplemente almacena un objeto, por lo que este comportamiento es heredado dentro de **Colored2**, que requiere que también su parámetro se corresponda con **HasColor**. **ColoredDimension2** y **Solid2** extienden todavía más la jerarquía y añaden límites en cada nivel. Ahora, los métodos son heredados y no tienen por qué repetirse en cada clase.

He aquí un ejemplo con más niveles:

```

//: generics/EpicBattle.java
// Ejemplo de límites para genéricos de Java.
import java.util.*;

interface SuperPower {}
interface XRayVision extends SuperPower {
    void seeThroughWalls();
}
interface SuperHearing extends SuperPower {
    void hearSubtleNoises();
}
interface SuperSmell extends SuperPower {
    void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
    POWER power;
    SuperHero(POWER power) { this.power = power; }
    POWER getPower() { return power; }
}

class SuperSleuth<POWER extends XRayVision>
extends SuperHero<POWER> {
    SuperSleuth(POWER power) { super(power); }
    void see() { power.seeThroughWalls(); }
}

class CanineHero<POWER extends SuperHearing & SuperSmell>
extends SuperHero<POWER> {
    CanineHero(POWER power) { super(power); }
    void hear() { power.hearSubtleNoises(); }
    void smell() { power.trackBySmell(); }
}

class SuperHearSmell implements SuperHearing, SuperSmell {
    public void hearSubtleNoises() {}
    public void trackBySmell() {}
}

class DogBoy extends CanineHero<SuperHearSmell> {
    DogBoy() { super(new SuperHearSmell()); }
}

public class EpicBattle {

```

```

// Límites en métodos genéricos:
static <POWER extends SuperHearing>
void useSuperHearing(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
}
static <POWER extends SuperHearing & SuperSmell>
void superFind(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
    hero.getPower().trackBySmell();
}
public static void main(String[] args) {
    DogBoy dogBoy = new DogBoy();
    useSuperHearing(dogBoy);
    superFind(dogBoy);
    // Podemos hacer esto:
    List<? extends SuperHearing> audioBoys;
    // Pero no podemos hacer esto:
    // List<? extends SuperHearing & SuperSmell> dogBoys;
}
} //:-)

```

Observe que los comodines (de los que hablaremos a continuación) están limitados a un único límite.

Ejercicio 25: (2) Cree dos interfaces y una clase que implemente ambas. Cree dos métodos genéricos, uno cuyo argumento de parámetro esté limitado por la primera interfaz y otro cuyo argumento de parámetro esté limitado por la segunda interfaz. Cree una instancia de la clase que implementa ambas interfaces y demuestre que se puede utilizar con ambos métodos genéricos.

Comodines

Ya hemos visto algunos usos simples de los *comodines* (símbolos de interrogación dentro de las expresiones de argumentos genéricos) en el Capítulo 11, *Almacenamiento de objetos* y en el Capítulo 14, *Información de tipos*. En esta sección vamos a explorar esta cuestión con más detalle.

Empezaremos con un ejemplo que demuestra un comportamiento concreto de las matrices. Podemos asignar una matriz de un tipo derivado a una referencia de matriz del tipo base:

```

//: generics/CovariantArrays.java

class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
    public static void main(String[] args) {
        Fruit[] fruit = new Apple[10];
        fruit[0] = new Apple(); // OK
        fruit[1] = new Jonathan(); // OK
        // El tipo en tiempo de ejecución es Apple[], no Fruit[] ni Orange[]:
        try {
            // El compilador permite añadir Fruit:
            fruit[0] = new Fruit(); // ArrayStoreException
        } catch(Exception e) { System.out.println(e); }
        try {
            // El compilador permite añadir Oranges:
            fruit[0] = new Orange(); // ArrayStoreException
        } catch(Exception e) { System.out.println(e); }
    }
}

```

```

} /* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
*/:-
```

La primera línea de `main()` crea una matriz de objetos `Apple` y la asigna una referencia a una matriz de objetos `Fruit`. Esto tiene bastante sentido, ya que `Apple` es un tipo de `Fruit`, por lo que una matriz de `Apple` tiene que ser una matriz de `Fruit`.

Sin embargo, si el tipo real de la matriz es `Apple[]`, sólo deberíamos poder insertar en la matriz un objeto `Apple` o un subtipo de `Apple`, lo que de hecho funciona tanto en tiempo de compilación como en tiempo de ejecución. Pero observe que el compilador nos permite insertar un objeto `Fruit` dentro de la matriz. Esto tiene sentido para el compilador, porque dispone de una referencia a `Fruit[]`; como dispone de esa referencia, ¿por qué no debería permitir colocar en la matriz un objeto `Fruit`, o cualquier cosa que descienda de `Fruit`, como por ejemplo `Orange`? Por tanto, la operación se permite en tiempo de compilación. Sin embargo, el mecanismo de tiempo de ejecución para las matrices sabe que está tratando con una matriz `Apple[]` y genera una excepción cuando se inserta un tipo incorrecto de la matriz.

El término “generalización” resulta confuso dentro de este contexto. Lo que estamos realmente haciendo es asignar una matriz a otra. El comportamiento de las matrices es tal que permite almacenar otros objetos, pero como podemos realizar una generalización, resulta claro que los objetos matriz pueden preservar las reglas acerca del tipo de objetos que contienen. Es como si las matrices fueran conscientes de qué es lo que están almacenando, por lo que entre las comprobaciones realizadas en tiempo de compilación y las realizadas en tiempo de ejecución, no podemos tratar de abusar del mecanismo de matrices.

Esta forma de comportarse de las matrices no resulta tan terrible, porque al final sí que detectamos en tiempo de ejecución que hemos insertado un tipo incorrecto. Pero uno de los objetivos principales de los genéricos era precisamente mover esos mecanismos de detección de errores a tiempo de compilación. Por tanto, ¿qué sucede si tratamos de utilizar contenedores genéricos en lugar de matrices?

```

//: generics/NonCovariantGenerics.java
// {CompileTimeError} (Won't compile)
import java.util.*;

public class NonCovariantGenerics {
    // Error de compilación: tipos incompatibles:
    List<Fruit> fList = new ArrayList<Apple>();
}
```

Aunque pudiéramos sentirnos tentados de concluir, a la vista de este ejemplo, que no se puede asignar a un contenedor de objetos `Apple` a un contenedor de objetos `Fruit`, recuerde que los genéricos no se refieren sólo a los contenedores. Lo que este ejemplo nos dice realmente es que no se puede asignar un genérico *relacionado* con objetos `Apple` a un genérico *relacionado* con objetos `Fruit`. Si el compilador, como sucede en el caso de las matrices, supiera lo suficiente acerca del código como para determinar que hay contenedores implicados, quizás podría ser algo más permisivo. Pero el compilador no sabe que es así, por lo que rehusará permitir la “generalización”. De todos modos, tampoco se trata de una “generalización real”: una lista de objetos `Apple` no es una lista de objetos `Fruit`. Una lista de objetos `Apple` permitirá almacenar objetos `Apple` y subtipos de `Apple`, mientras que una lista de objetos `Fruit` permitirá almacenar cualquier clase de objetos `Fruit`. Es cierto que esto incluye a los objetos `Apple`, pero eso no la hace una lista de objetos `Apple`; seguirá siendo una lista de objetos `Fruit`. Una lista de objetos `Apple` no es equivalente en lo que respecta a tipos a una lista de objetos `Fruit`, aún cuando un objeto `Apple` sea un tipo de objeto `Fruit`.

La cuestión real es que de lo que estamos hablando es del tipo de contenedor, no del tipo de los objetos que el contenedor almacena. A diferencia de las matrices, los genéricos no tienen mecanismo de covarianza integrado. Esto se debe a que las matrices están definidas completamente en el lenguaje y pueden incorporar, por tanto, comprobaciones tanto en tiempo de compilación como en tiempo de ejecución; sin embargo, con los genéricos, el compilador y el sistema de ejecución no pueden saber lo que queremos hacer con los tipos y cuáles son las reglas que deberían aplicarse.

En ocasiones, sin embargo, puede que queramos establecer algún tipo de relación de generalización entre los dos, y esto es, precisamente, lo que los comodines permiten.

```

//: generics/GenericsAndCovariance.java
import java.util.*;
```

```

public class GenericsAndCovariance {
    public static void main(String[] args) {
        // Los comodines permiten la covarianza;
        List<? extends Fruit> flist = new ArrayList<Apple>();
        // Error de compilación: no se puede añadir cualquier tipo de objeto;
        // flist.add(new Apple());
        // flist.add(new Fruit());
        // flist.add(new Object());
        flist.add(null); // Legal pero poco interesante
        // Sabemos que devuelve al menos Fruit;
        Fruit f = flist.get(0);
    }
} //:-.

```

El tipo de **flist** es ahora **List<? extends Fruit>**, lo cual puede leerse como “una lista de cualquier tipo que herede de **Fruit**”. Sin embargo, esto no significa que la lista pueda almacenar cualquier tipo de objeto **Fruit**. El comodín hace referencia a un tipo concreto, por lo que significa “algun tipo específico que la referencia **flist** no especifique”. Por tanto, la lista que se asigne tiene que estar almacenando algún tipo especificado como **Fruit** o **Apple**, aunque para poder realizar la generalización a **flist**, ese tipo se especifica como “no importa cuál sea”.

Si la única restricción es que la lista almacene un tipo o subtipo de **Fruit** específico, pero no nos importa en realidad cuál sea éste, entonces ¿qué es lo que podemos hacer con dicha lista? Si no sabemos el tipo de objeto que la lista está almacenando, ¿cómo podemos añadir con seguridad un objeto? Al igual que sucede con la “generalización” de la matriz **CovariantArrays.java**, no podemos añadir cualquier objeto que queramos, lo único que sucede es que, en este caso, el compilador prohíbe las operaciones no permitidas antes, en lugar de que sea el sistema de ejecución el que se encargue de prohibirlas. En otras palabras, esto nos permite descubrir el problema bastante antes.

Podríamos pensar que las cosas se han ido un poco de las manos, porque ahora ni siquiera podemos añadir un objeto **Apple** a una lista que habíamos dicho que sí podía almacenar objetos **Apple**. En efecto, así es, pero es que el compilador no tiene ningún conocimiento de eso. Un contenedor **List<? extends Fruit>** puede apuntar a una lista **List<Orange>**. Una vez que hacemos este tipo de “generalización”, perdemos la posibilidad de pasar ningún objeto, ni siquiera de tipo **Object**.

Por otro lado, si invocamos un método que devuelva **Fruit**, esa operación sí es segura porque sabemos que cualquier cosa que haya en la lista deberá ser al menos de tipo **Fruit**, por lo que el compilador permitirá la operación.

Ejercicio 26: (2) Ilustre la covarianza de matrices utilizando objetos de tipo **Number** e **Integer**.

Ejercicio 27: (2) Demuestre que la covarianza no funciona con las listas utilizando objetos de tipo **Number** e **Integer**, y luego introduzca comodines.

¿Hasta qué punto es inteligente el compilador?

Ahora, podríamos pensar que no podemos invocar ningún método que admita argumentos, pero vamos a analizar este ejemplo:

```

//: generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
    public static void main(String[] args) {
        List<? extends Fruit> flist =
            Arrays.asList(new Apple());
        Apple a = (Apple)flist.get(0); // Ninguna advertencia
        flist.contains(new Apple()); // El argumento es 'Object'
        flist.indexOf(new Apple()); // El argumento es 'Object'
    }
} //:-.

```

Podemos ver llamadas a **contains()** e **indexOf()** que toman objetos **Apple** como argumentos, y esas llamadas son perfectamente válidas. ¿Significa esto que el compilador sí que examina el código para ver si un método concreto modifica su objeto?

Examinando la documentación de **ArrayList**, podemos comprobar que el compilador no es tan inteligente. Mientras que **add()** toma un argumento del tipo de parámetro genérico, **contains()** e **indexOf()** toman argumentos de tipo **Object**. Por tanto, cuando especificamos un contenedor **ArrayList<? extends Fruit>**, el argumento de **add()** se convierte en '**? extends Fruit**'. A partir de dicha descripción, el compilador no puede saber qué tipo específico de **Fruit** se requiere, por lo que no aceptará ningún tipo de **Fruit**. No importa si primero generalizamos el objeto **Apple** a **Fruit**: el compilador se negará a invocar un método (como **add()**) si hay un comodín en la lista de argumentos.

Con **contains()** e **indexOf()**, los argumentos son de tipo **Object**, por lo que no hay ningún comodín implicado y el compilador sí que permite la llamada. Esto significa que es responsabilidad del diseñador de clases genéricas definir qué clases son "seguras" y utilizar el tipo **Object** para sus argumentos. Para prohibir una llamada cuando el tipo se utilice con comodines, utilice el parámetro de tipo dentro de la lista de argumentos.

Podemos ilustrar esto con una clase **Holder** muy simple:

```
//: generics/Holder.java

public class Holder<T> {
    private T value;
    public Holder() {}
    public Holder(T val) { value = val; }
    public void set(T val) { value = val; }
    public T get() { return value; }
    public boolean equals(Object obj) {
        return value.equals(obj);
    }
    public static void main(String[] args) {
        Holder<Apple> Apple = new Holder<Apple>(new Apple());
        Apple d = Apple.get();
        Apple.set(d);
        // Holder<Fruit> Fruit = Apple; // No se puede generalizar
        Holder<? extends Fruit> fruit = Apple; // OK
        Fruit p = fruit.get();
        d = (Apple)fruit.get(); // Devuelve 'Object'
        try {
            Orange c = (Orange)fruit.get(); // Ninguna advertencia
        } catch(Exception e) { System.out.println(e); }
        // fruit.set(new Apple()); // No se puede invocar set()
        // fruit.set(new Fruit()); // No se puede invocar set()
        System.out.println(fruit.equals(d)); // OK
    }
} /* Output: (Sample)
java.lang.ClassCastException: Apple cannot be cast to Orange
true
*///:-
```

Holder tiene un método **set()** que toma un argumento **T**, un método **get()** que devuelve **T**, y un método **equals()** que toma un argumento de tipo **Object**. Como ya hemos visto, si creamos un contenedor **Holder<Apple>**, no podemos generalizarlo a **Holder<Fruit>**, pero sí que podemos generalizarlo a **Holder<? extends Fruit>**. Si invocamos **get()**, sólo devuelve un objeto **Fruit**, que es lo único que el compilador sabe, teniendo en cuenta que el límite que hemos hecho es "cualquier cosa que extienda **Fruit**". Si el programador tiene más información acerca de los objetos implicados, puede efectuar una predicción sobre un tipo específico de **Fruit** y no se generará ninguna advertencia, aunque correremos el riesgo de que se genere una excepción **ClassCastException**. El método **set()** no funcionará ni con un objeto **Apple** ni con un objeto **Fruit**, porque el argumento de **set()** es también "**? Extends Fruit**", lo que significa que puede ser cualquier cosa y el compilador no puede verificar que "cualquier cosa" sea segura en lo que a tipos respecta.

Sin embargo, el método **equals()** funciona correctamente, porque toma un objeto **Object** en lugar de **T** como argumento. Por tanto, el compilador sólo presta atención a los tipos de objetos que se pasan a los métodos y que se devuelven desde éstos. El compilador no analiza el código para ver si efectuamos ninguna escritura o lectura real.

Contravarianza

También es posible proceder en sentido inverso y utilizar *comodines de supertipo*. Con esto, lo que expresamos es que el comodín está limitado por cualquier clase base de una clase concreta, especificando `<? super MyClass>` o incluso utilizando un parámetro de tipo: `<? super T>` (aunque no se puede dar un límite de supertipo a un parámetro genérico, es decir, no podemos escribir `<T super MyClass>`). Esto nos permite pasar con seguridad un objeto de un cierto a un tipo genérico. De este modo, con los comodines de supertipo podemos escribir dentro de un contenedor de tipo **Collection**:

```
//: generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
        apples.add(new Apple());
        apples.add(new Jonathan());
        // apples.add(new Fruit()); // Error
    }
} ///:-
```

El argumento **apples** es una lista de algún tipo que es el tipo base de **Apple**; por tanto, sabemos que resulta seguro añadir un objeto **Apple** o un subtipo de **Apple**. Sin embargo, puesto que el *límite inferior* es **Apple**, no sabemos si resulta seguro añadir un objeto **Fruit** a dicha lista, porque eso permitiría que la lista se abriera a la adición de tipos distintos de **Apple**, lo que violaría la seguridad estática de tipos.

Por tanto, podemos pensar en los límites de subtipo y de supertipo en términos de cómo se puede “escribir” (pasar a un método) en un tipo genérico y cómo se puede “leer” (devolver desde un método) de un tipo genérico.

Los límites de supertipo relajan las restricciones de lo que podemos pasar a un método:

```
//: generics/GenericWriting.java
import java.util.*;

public class GenericWriting {
    static <T> void writeExact(List<T> list, T item) {
        list.add(item);
    }
    static List<Apple> apples = new ArrayList<Apple>();
    static List<Fruit> fruit = new ArrayList<Fruit>();
    static void f1() {
        writeExact(apples, new Apple());
        // writeExact(fruit, new Apple()); // Error:
        // Tipos incompatibles: se encontró Fruit, se requiere Apple
    }
    static <T> void
    writeWithWildcard(List<? super T> list, T item) {
        list.add(item);
    }
    static void f2() {
        writeWithWildcard(apples, new Apple());
        writeWithWildcard(fruit, new Apple());
    }
    public static void main(String[] args) { f1(); f2(); }
} ///:-
```

El método **writeExact()** utiliza un tipo de parámetro exacto (comodines). En **f1()** podemos ver que esto funciona correctamente, siempre y cuando nos limitemos a insertar un objeto **Apple** en un contenedor **List<Apple>**. Sin embargo, **writeExact()** no permite insertar un objeto **Apple** dentro de un contenedor **List<Fruit>**, aún cuando nosotros sepamos que eso debería ser posible.

En **writeWithWildcard()**, el argumento es ahora `List<? super T>`, por lo que la lista almacena un tipo específico derivado de `T`; por tanto, resulta seguro pasar `T` o cualquier cosa que se derive `T` como argumento a los métodos de la lista.

Podemos ver esto en `f2()`, donde sigue siendo posible insertar un objeto `Apple` en una lista `List<Apple>`, como antes, pero ahora resulta posible insertar un objeto `Apple` en una lista `List<Fruit>`, tal como cabría esperar.

Podríamos realizar este mismo tipo de análisis como revisión del tema de la covarianza y de los comodines:

```
//: generics/GenericReading.java
import java.util.*;

public class GenericReading {
    static <T> T readExact(List<T> list) {
        return list.get(0);
    }
    static List<Apple> apples = Arrays.asList(new Apple());
    static List<Fruit> fruit = Arrays.asList(new Fruit());
    // Un método estático se adapta a cada llamada:
    static void f1() {
        Apple a = readExact(apples);
        Fruit f = readExact(fruit);
        f = readExact(apples);
    }
    // Sin embargo, si tenemos una clase, su tipo se
    // establece en el momento de instanciarla:
    static class Reader<T> {
        T readExact(List<T> list) { return list.get(0); }
    }
    static void f2() {
        Reader<Fruit> fruitReader = new Reader<Fruit>();
        Fruit f = fruitReader.readExact(fruit);
        // Fruit a = fruitReader.readExact(apples); // Error:
        // readExact(List<Fruit>) no puede
        // aplicarse a (List<Apple>).
    }
    static class CovariantReader<T> {
        T readCovariant(List<? extends T> list) {
            return list.get(0);
        }
    }
    static void f3() {
        CovariantReader<Fruit> fruitReader =
            new CovariantReader<Fruit>();
        Fruit f = fruitReader.readCovariant(fruit);
        Fruit a = fruitReader.readCovariant(apples);
    }
    public static void main(String[] args) {
        f1(); f2(); f3();
    }
} ///:-
```

Como antes, el primer método `readExact()` utiliza el tipo concreto. Por tanto, si utilizamos el tipo concreto sin ningún comodín, podemos escribir y leer de dicho tipo concreto en una lista. Además, para el valor de retorno, el método genérico estático `readExact()` "se adapta" efectivamente a cada llamada a método y devuelve un objeto `Apple` de una lista `List<Apple>` y un objeto `Fruit` de una lista `List<Fruit>`, como puede verse en `f1()`. Por tanto, si podemos resolver el problema con un método genérico estático, no necesitamos recurrir a la covarianza si únicamente nos estamos limitando a leer.

Sin embargo, si tenemos una clase genérica, el parámetro se establece para la clase en el momento de crear una instancia de dicha clase. Como podemos ver en `f2()`, la instancia `fruitReader` puede leer un objeto `Fruit` de una lista `List<Fruit>`, puesto que ese es su tipo concreto. Pero una lista `List<Apple>` también debería producir objetos `Fruit` y el objeto `fruitReader` no permite esto.

Para corregir el problema, el método `CovariantReader.readCovariant()` toma una lista `List<? extends T>`, de modo que resulta seguro leer un objeto `T` de dicha lista (sabemos que todo lo que hay en esa lista es cuando menos de tipo `T`,

y posiblemente algo derivado de T). En f3() podemos ver que ahora sí es posible leer un objeto Fruit de una lista List<Apple>.

Ejercicio 28: (4) Cree una clase genérica Generic1<T> con un único método que tome un argumento de tipo T. Cree una segunda clase genérica Generic2<T> con un único método que devuelva un argumento de tipo T. Escriba un método genérico con un argumento contravariante de la primera clase genérica que invoque al método de dicha clase. Escriba un segundo método con un argumento covariante de la segunda clase genérica que invoque al método de dicha clase. Pruebe el diseño utilizando la biblioteca typeinfo.pets.

Comodines no limitados

El comodín no limitado <?> parece que quiere significar “cualquier cosa”, por lo que utilizar un comodín no limitado parece equivalente a emplear un tipo normal. Ciertamente, el compilador parece aceptar, a primera vista, esta interpretación:

```
//: generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
    static List list1;
    static List<?> list2;
    static List<? extends Object> list3;
    static void assign1(List list) {
        list1 = list;
        list2 = list;
        // list3 = list; // Advertencia: conversión no comprobada
        // Found: List, Required: List<? extends Object>
    }
    static void assign2(List<?> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    static void assign3(List<? extends Object> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    public static void main(String[] args) {
        assign1(new ArrayList());
        assign2(new ArrayList());
        // assign3(new ArrayList()); // Advertencia:
        // conversión no comprobada. No encontrado: ArrayList
        // Requerido: List<? extends Object>
        assign1(new ArrayList<String>());
        assign2(new ArrayList<String>());
        assign3(new ArrayList<String>());
        // Ambas formas son aceptables como List<?>:
        List<?> wildList = new ArrayList();
        wildList = new ArrayList<String>();
        assign1(wildList);
        assign2(wildList);
        assign3(wildList);
    }
} ///:-
```

Hay muchos casos como los de este ejemplo en que al compilador no le importa si utilizamos un tipo normal o <?>. En dichos casos, <?> parece completamente superfluo. Sin embargo, sigue teniendo sentido utilizar este comodín, porque lo que dicho comodín dice en la práctica es “he escrito este código teniendo presente los genéricos de Java y lo que estoy utilizando aquí no es un tipo normal, aunque en este caso el parámetro genérico puede referirse a cualquier tipo”.

Un segundo ejemplo muestra un uso importante de los comodines no limitados. Cuando estemos tratando con múltiples parámetros genéricos, a menudo es importante permitir que un parámetro sea de cualquier tipo al mismo tiempo que se asigna un tipo concreto al otro parámetro:

```
//: generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
    static Map map1;
    static Map<?,?> map2;
    static Map<String,?> map3;
    static void assign1(Map map) { map1 = map; }
    static void assign2(Map<?,?> map) { map2 = map; }
    static void assign3(Map<String,?> map) { map3 = map; }
    public static void main(String[] args) {
        assign1(new HashMap());
        assign2(new HashMap());
        // assign3(new HashMap()); // Advertencia:
        // Conversión no comprobada. Encontrado: HashMap
        // Requerido: Map<String,?>
        assign1(new HashMap<String, Integer>());
        assign2(new HashMap<String, Integer>());
        assign3(new HashMap<String, Integer>());
    }
} //:~
```

Pero de nuevo, cuando lo que tenemos son todos comodines no limitados, como por ejemplo en `Map<?,?>`, el compilador parece no efectuar distinciones con respecto a un mapa normal y corriente. Además, `UnboundedWildcards1.java` muestra que el compilador trata `List<?>` y `List<? extends Object>` de manera diferente.

Lo que resulta más confuso es que el compilador no siempre diferencia entre, por ejemplo, `List` y `List<?>`, por lo que ambas expresiones podrían parecer equivalentes. De hecho, puesto que los argumentos genéricos se ven sometidos al mecanismo de borrado de tipos, `List<?>` podría parecer equivalente a `List<Object>`, y `List` es de hecho equivalente a `List<Object>`. Sin embargo, estas afirmaciones no son completamente ciertas. `List` quiere decir en la práctica “una lista simple que almacena cualquier tipo de objeto `Object`”, mientras que `List<?>` significa “una lista no simple de *algún tipo específico*, aunque no sabemos qué tipo es ese”.

¿En qué casos se preocupa el compilador de las diferencias entre los tipos normales y los tipos que incluyen comodines no limitados? El siguiente ejemplo utiliza la clase `Holder<T>` que hemos definido anteriormente. Contiene métodos que toman `Holder` como argumentos, pero de distintas maneras: como tipo normal, con un parámetro de un tipo específico y con un parámetro con un comodín no limitado:

```
//: generics/Wildcards.java
// Exploración del significado de los comodines.

public class Wildcards {
    // Argumento normal:
    static void rawArgs(Holder holder, Object arg) {
        // holder.set(arg); // Advertencia:
        // Llamada no comprobada a (T) como miembro
        // del tipo normal Holder
        // holder.set(new Wildcards()); // Misma advertencia

        // No se puede hacer esto; no se dispone de ninguna 'T':
        // T t = holder.get();

        // OK, pero la información de tipos se ha perdido:
        Object obj = holder.get();
    }
    // Similar a rawArgs(), pero con errores en lugar de advertencias:
```

```

static void unboundedArg(Holder<?> holder, Object arg) {
    // holder.set(arg); // Error:
    //   set(capture of ?) en Holder<capture of ?>
    //   no puede aplicarse a (Object)
    // holder.set(new Wildcards()); // Mismo error

    // No se puede hacer esto: no se dispone de ninguna 'T':
    // T t = holder.get();

    // OK, pero la información de tipos se ha perdido:
    Object obj = holder.get();
}

static <T> T exact1(Holder<T> holder) {
    T t = holder.get();
    return t;
}

static <T> T exact2(Holder<T> holder, T arg) {
    holder.set(arg);
    T t = holder.get();
    return t;
}

static <T>
T wildSubtype(Holder<? extends T> holder, T arg) {
    // holder.set(arg); // Error:
    //   set(capture of ? extends T) en
    //   Holder<capture of ? extends T>
    //   no se puede aplicar a (T)
    T t = holder.get();
    return t;
}

static <T>
void wildSupertype(Holder<? super T> holder, T arg) {
    holder.set(arg);
    // T t = holder.get(); // Error:
    //   Tipos incompatibles: encontrado Object, requerido T

    // OK, pero la información de tipos se ha perdido:
    Object obj = holder.get();
}

public static void main(String[] args) {
    Holder raw = new Holder<Long>();
    // O bien:
    raw = new Holder();
    Holder<Long> qualified = new Holder<Long>();
    Holder<?> unbounded = new Holder<Long>();
    Holder<? extends Long> bounded = new Holder<Long>();
    Long lng = 1L;

    rawArgs(raw, lng);
    rawArgs(qualified, lng);
    rawArgs(unbounded, lng);
    rawArgs(bounded, lng);

    unboundedArg(raw, lng);
    unboundedArg(qualified, lng);
    unboundedArg(unbounded, lng);
    unboundedArg(bounded, lng);

    // Object rl = exact1(raw); // Advertencias:
}

```

```

// Conversión no comprobada de Holder a Holder<T>
// Invocación de método no comprobado: exact1(Holder<T>)
// aplicada a (Holder)
Long r2 = exact1(qualified);
Object r3 = exact1(unbounded); // Debe devolver Object
Long r4 = exact1(bounded);

// Long r5 = exact2(raw, lng); // Advertencias:
// Conversión no comprobada de Holder a Holder<Long>
// Invocación de método no comprobado: exact2(Holder<T>, T)
// aplicada a (Holder, Long)
Long r6 = exact2(qualified, lng);
// Long r7 = exact2(unbounded, lng); // Error:
// exact2(Holder<T>, T) no puede aplicarse a
// (Holder<capture of ?>, Long)
// Long r8 = exact2(bounded, lng); // Error:
// exact2(Holder<T>, T) no puede aplicarse a
// to (Holder<capture of ? extends Long>, Long)

// Long r9 = wildSubtype(raw, lng); // Advertencias:
// Conversión no comprobada de Holder
// a Holder<? extends Long>
// Invocación de método no comprobado:
// wildSubtype(Holder<? extends T>, T)
// aplicada a (Holder, Long)
Long r10 = wildSubtype(qualified, lng);
// OK, pero sólo puede devolver Object:
Object r11 = wildSubtype(unbounded, lng);
Long r12 = wildSubtype(bounded, lng);

// wildSupertype(raw, lng); // Advertencias:
// Conversión no comprobada de Holder
// a Holder<? super Long>
// Invocación de método no comprobado:
// wildSupertype(Holder<? super T>, T)
// aplicada a (Holder, Long)
wildSupertype(qualified, lng);
// wildSupertype(unbounded, lng); // Error:
// wildSupertype(Holder<? super T>, T) no puede
// aplicarse a (Holder<capture of ?>, Long)
// wildSupertype(bounded, lng); // Error:
// wildSupertype(Holder<? super T>, T) no puede
// aplicarse a (Holder<capture of ? extends Long>, Long)
}
} //:-
```

En `rawArgs()`, el compilador sabe que `Holder` es un tipo genérico, por lo que aún cuando se lo exprese aquí como un tipo normal, el compilador sabe que pasar un objeto de tipo `Object` a `set()` no resulta seguro. Puesto que se trata de un tipo normal, podemos pasar un objeto de cualquier tipo a `set()` y dicho objeto se generaliza a `Object`. Por tanto, siempre que tengamos un tipo normal, estaremos sacrificando posibilidades de comprobación en tiempo de compilación. La llamada a `get()` muestra el mismo problema: no hay ningún `T`, por lo que el resultado sólo puede ser de tipo `Object`.

Resulta tentador pensar que el tipo `Holder` y `Holder<?>` son aproximadamente iguales. Pero `unboundedArg()` demuestra que son distintos: este método hace aflorar el mismo tipo de problemas, pero informa de ellos como de errores en lugar de como advertencias, porque el tipo `Holder` permitirá almacenar una combinación de objetos de cualquier tipo, mientras que `Holder<?>` almacena una colección homogénea de *algún tipo específico*, y no podemos limitarnos a pasar un objeto de tipo `Object`.

En `exact1()` y `exact2()`, podemos ver los parámetros genéricos exactos utilizados sin comodines. Como vemos, `exact2()` tiene limitaciones distintas que `exact1()`, debido al argumento adicional.

En `wildSubtype()`, las restricciones en el tipo de `Holder` se relajan para incluir un contenedor `Holder` de cualquier cosa que extienda `T`. De nuevo, esto significa que `T` podría ser `Fruit`, mientras que `holder` podría ser perfectamente el contenedor `Holder<Apple>`. Para impedir que se inserte un objeto `Orange` en un contenedor `Holder<Apple>`, la llamada a `set()` (o cualquier método que tome un argumento referido al parámetro de tipo) no está permitida. Sin embargo, seguimos sabiendo que cualquier cosa que extraigamos de un contenedor `Holder<? extends Fruit>` será al menos de tipo `Fruit`, por lo que sí está permitido invocar `get()` (o cualquier método que genere un valor de retorno que se corresponda con el parámetro de tipo).

Los comodines de supertipo se muestran en `wildSupertype()`, que tiene el comportamiento opuesto a `wildSubtype()`: `holder` puede ser un contenedor que almacene cualquier tipo que sea una clase base de `T`. Por tanto, `set()` puede aceptar un objeto `T`, puesto que cualquier cosa que funcione con un tipo base también funcionará, polimórficamente, con un tipo derivado (y por tanto con `T`). Sin embargo, no resulta útil tratar de invocar `get()`, porque el tipo almacenado por `holder` puede ser cualquier supertipo, de modo que el único tipo seguro es `Object`.

Este ejemplo también muestra las limitaciones relativas a lo que se puede y no se puede hacer respecto a un parámetro no limitado. El método que ilustra esta situación es `unbounded()`: no se puede invocar `get()` o `set()` con `T` porque no disponemos de ningún `T`.

En `main()`, podemos ver cuáles de estos métodos pueden aceptar cada uno de estos métodos sin errores ni advertencias. Para garantizar la compatibilidad de migración, `rawArgs()` admite todas las diferentes variaciones de `Holder` sin generar advertencias. El método `unboundedArg()` también acepta todos los tipos, aunque, como hemos indicado anteriormente, los gestiona de manera distinta dentro del cuerpo del método.

Si pasamos una referencia `Holder` normal a un método que tome un tipo genérico “exacto” (comodines) obtendremos una advertencia, porque el argumento exacto está esperando información que no existe en el tipo normal. Y si pasamos una referencia no limitada a `exact1()`, no existe la suficiente información de tipos como para establecer el tipo de retorno.

Podemos ver que `exact2()` tiene el mayor número de restricciones, ya que desea disponer exactamente de un `Holder<T>` y de un argumento de tipo `T`, y debido a ello genera errores o advertencias a menos que le entreguemos los argumentos exactos. En ocasiones, esto es perfectamente admisible, pero se trata de una restricción excesiva; en ese caso, podemos utilizar comodines, dependiendo de si queremos obtener valores de retorno de un tipo determinado a partir de nuestro argumento genérico (como puede verse en `wildSubtype()`) o de si queremos pasar argumentos con un tipo determinado a nuestro argumento genérico (como puede verse en `wildSupertype()`).

Por tanto, la ventaja de utilizar tipos exactos en lugar de tipos con comodín es que podemos hacer más cosas con los parámetros genéricos. Sin embargo, utilizar comodines nos permite aceptar como argumentos un rango más amplio de tipos parametrizados. El programador deberá decidir cuál es el compromiso más adecuado examinando caso por caso.

Conversión de captura

Hay una situación concreta que *exige* el uso de `<?>` en lugar de un tipo normal. Si se pasa un tipo normal a un método que utilice `<?>`, al compilador le resulta posible inferir el parámetro de tipo real, de modo que el método puede a su vez invocar otro método que utilice el tipo exacto. El siguiente ejemplo muestra esta técnica que se denomina *conversión de captura* porque el tipo no especificado con comodín se captura y se convierte en un tipo exacto. Aquí, los comentarios acerca de las advertencias sólo se aplican si se elimina la anotación `@SuppressWarnings`:

```
//: generics/CaptureConversion.java

public class CaptureConversion {
    static <T> void f1(Holder<T> holder) {
        T t = holder.get();
        System.out.println(t.getClass().getSimpleName());
    }
    static void f2(Holder<?> holder) {
        f1(holder); // Llamada con tipo capturado
    }
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Holder raw = new Holder<Integer>(1);
```

```

// f1(raw); // Genera advertencias
f2(raw); // Sin advertencias
Holder rawBasic = new Holder();
rawBasic.set(new Object()); // Advertencia
f2(rawBasic); // Sin advertencias
// Generalización a Holder<?>, sigue sabiendo cómo manejarla:
Holder<?> wildcarded = new Holder<Double>(1.0);
f2(wildcarded);
}
} /* Output:
Integer
Object
Double
*///:-
```

Los parámetros de tipo **f1()** son todos exactos, sin comodines ni límites. En **f2()**, el parámetro **Holder** es un comodín no limitado, por lo que podría parecer que es desconocido en la práctica. Sin embargo, dentro de **f2()**, se invoca **f1()** y **f1()** requiere un parámetro conocido. Lo que está sucediendo es que el tipo de parámetro se captura en el proceso de invocación de **f2()**, de modo que puede utilizarse en la llamada a **f1()**.

El lector podría preguntarse si esta técnica podría utilizarse para escribir, pero eso requeriría que pasáramos un tipo específico junto con **Holder<?>**. La conversión de captura sólo funciona en aquellas situaciones donde necesitamos, dentro del método, trabajar con el tipo exacto. Observe que no se puede devolver **T** desde **f2()**, porque **T** es desconocido para **f2()**. La conversión de captura resulta interesante, pero bastante limitada.

Ejercicio 29: (5) Cree un método genérico que tome como argumento un contenedor **Holder<List<?>>**. Determine qué métodos puede o no invocar para el contenedor **Holder** y para la lista **List**. Repita el ejercicio para un argumento de tipo **List<Holder<?>>**.

Problemas

Esta sección analiza diversos tipos de problemas que surgen cuando se intentan utilizar los genéricos de Java.

No pueden usarse primitivas como parámetros de tipo

Como se ha mencionado anteriormente en este capítulo, una de las limitaciones existentes en los genéricos de Java es que no pueden utilizarse primitivas como parámetros de tipo. No podemos, por ejemplo, crear un contenedor **ArrayList<int>**.

La solución consiste en utilizar las clases envoltorio de las primitivas en conjunción con el mecanismo de conversión automática de Java SE5. Si creamos un contenedor **ArrayList<Integer>** y utilizamos enteros primitivos con este contenedor, descubriremos que el mecanismo de conversión automática entra en acción y se encarga de convertir entre enteros y objetos **Integer** automáticamente, por tanto, es como si dispusiéramos de un contenedor **ArrayList<int>**:

```

//: generics/ListOfInt.java
// El mecanismo de conversión automática compensa la incapacidad
// de utilizar primitivas en los genéricos.
import java.util.*;

public class ListOfInt {
    public static void main(String[] args) {
        List<Integer> li = new ArrayList<Integer>();
        for(int i = 0; i < 5; i++)
            li.add(i);
        for(int i : li)
            System.out.print(i + " ");
    }
} /* Output:
0 1 2 3 4
*///:-
```

Observe que el mecanismo de conversión automática admite incluso la utilización de la sintaxis *foreach* para generar valores `int`.

En general, esta solución funciona adecuadamente, ya que podemos almacenar y extraer apropiadamente valores primitivos enteros. Se producen ciertas conversiones, pero éstas se llevan a cabo de forma transparente. Sin embargo, si las cuestiones de rendimiento son un problema, podemos utilizar una versión especializada de los contenedores adaptada para tipos primitivos; un ejemplo de versión de código fuente abierto para este tipo de contenedores especializados es `org.apache.commons.collections.primitives`.

He aquí otro enfoque, que crea un conjunto de bytes:

```
//: generics/ByteSet.java
import java.util.*;

public class ByteSet {
    Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
    Set<Byte> mySet =
        new HashSet<Byte>(Arrays.asList(possibles));
    // Pero no se puede hacer esto:
    // Set<Byte> mySet2 = new HashSet<Byte>(
    //     Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
}
```

Observe que el mecanismo de conversión automática resuelve algunos problemas, pero no todos ellos. El siguiente ejemplo muestra una interfaz **Generator** genérica que especifica un método `next()` que devuelve un objeto con el tipo especificado del parámetro. La clase **FArray** contiene un método genérico que utiliza un generador para llenar una matriz con objetos (hacer la clase genérica no funcionaría en este caso porque el método es estático). Las implementaciones de **Generator** provienen del Capítulo 16, *Matrices*, y en `main()` podemos ver cómo se utiliza `FArray.fill()` para llenar matrices con objetos:

```
//: generics/PrimitiveGenericTest.java
import net.mindview.util.*;

// Rellenar una matriz utilizando un generador:
class FArray {
    public static <T> T[] fill(T[] a, Generator<T> gen) {
        for(int i = 0; i < a.length; i++)
            a[i] = gen.next();
        return a;
    }
}

public class PrimitiveGenericTest {
    public static void main(String[] args) {
        String[] strings = FArray.fill(
            new String[7], new RandomGenerator.String(10));
        for(String s : strings)
            System.out.println(s);
        Integer[] integers = FArray.fill(
            new Integer[7], new RandomGenerator.Integer());
        for(int i: integers)
            System.out.println(i);
        // El mecanismo de conversión automática no funciona en este caso.
        // Lo siguiente no podrá compilarse:
        // int[] b =
        //     FArray.fill(new int[7], new RandIntGenerator());
    }
} /* Output:
YNzbrnyGcF
OWZnTcQrGs
```

```
eGZMmJMRoE
suEcUOneOE
dLsmwHLGEa
hKcxrEqUCB
bkInaMesbt
7052
6665
2654
3909
5202
2209
5458
*///:-
```

Puesto que `RandomGenerator.Integer` implementa `Generator<Integer>`, cabría esperar que el mecanismo de conversión automática se encargara de convertir el valor de `next()` de `Integer` a `int`. Sin embargo, el mecanismo de conversión automática no se aplica a las matrices, así que esta solución no funciona.

Ejercicio 30: (2) Cree un contenedor `Holder` para cada uno de los tipos envoltorio de las primitivas y demuestre que el mecanismo de conversión automática funciona para los métodos `set()` y `get()` de cada instancia.

Implementación de interfaces parametrizadas

Una clase no puede implementar dos variantes de la misma interfaz genérica. Debido al mecanismo de borrado de tipos, ambas serían la misma interfaz. He aquí una situación donde se produce este tipo de colisión:

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (No se compilará)

interface Payable<T> {}

class Employee implements Payable<Employee> {}
class Hourly extends Employee
    implements Payable<Hourly> {} //://:-
```

`Hourly` no se compilará debido a que el mecanismo de borrado de tipos reduce `Payable<Employee>` y `Payable<Hourly>` a la misma clase, `Payable`, y el código anterior significaría que estaríamos tratando de implementar la misma interfaz dos veces. Lo que resulta interesante es que, si eliminamos los parámetros genéricos en ambos usos de `Payable` (como hace el compilador durante el borrado de tipos), el código sí que puede compilarse.

Esta cuestión puede resultar bastante frustrante cuando estemos trabajando con alguna de las interfaces más fundamentales de Java, como `Comparable<T>`, como podremos ver más adelante en esta sección.

Ejercicio 31: (1) Elimine todos los genéricos de `MultipleInterfaceVariants.java` y modifique el ejemplo para que el código pueda compilarse.

Proyecciones de tipos y advertencias

Utilizar una proyección de tipos o `instanceof` con un parámetro de tipo genérico no tiene ningún efecto. El siguiente contenedor almacena los valores internamente como objetos de tipo `Object` y los proyecta de nuevo sobre `T` en el momento de extraerlos:

```
//: generics/GenericCast.java

class FixedSizeStack<T> {
    private int index = 0;
    private Object[] storage;
    public FixedSizeStack(int size) {
        storage = new Object[size];
    }
}
```

```

public void push(T item) { storage[index++] = item; }
@SuppressWarnings("unchecked")
public T pop() { return (T)storage[--index]; }
}

public class GenericCast {
    public static final int SIZE = 10;
    public static void main(String[] args) {
        FixedSizeStack<String> strings =
            new FixedSizeStack<String>(SIZE);
        for(String s : "A B C D E F G H I J".split(" "))
            strings.push(s);
        for(int i = 0; i < SIZE; i++) {
            String s = strings.pop();
            System.out.print(s + " ");
        }
    }
} /* Output:
J I H G F E D C B A
*///:-

```

Sin la anotación `@SuppressWarnings`, el compilador generaría una advertencia de “proyección de tipos no comprobada” para `pop()`. Debido al mecanismo de borrado de tipos, no puede saber si la proyección de tipos es segura, y el método `pop()` no realiza en la práctica ninguna proyección. `T` se borra para sustituirla por su primer límite que es `Object` de manera pre-determinada, por lo que `pop()` está, de hecho, proyectando un objeto de tipo `Object` sobre `Object`.

Hay veces en que los genéricos no eliminan la necesidad de efectuar la proyección, y esto genera una advertencia por parte del compilador, que es inapropiada. Por ejemplo:

```

//: generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        List<Widget> shapes = (List<Widget>)in.readObject();
    }
} //://:-

```

Como veremos en el siguiente capítulo, `readObject()` no puede saber lo que está leyendo, por lo que devuelve un objeto que habrá que proyectar. Pero cuando desactivamos mediante comentarios la anotación `@SuppressWarnings` y compilamos el programa, se obtiene una advertencia:

```

Note: NeedCasting.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

```

Y si seguimos las instrucciones y recompilamos con `-Xlint:unchecked`:

```

NeedCasting.java:12: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.List<Widget>
    List<Shape> shapes = (List<Widget>)in.readObject();

```

Estamos obligados a realizar la proyección y a pesar de ello se nos dice que no podemos hacerlo. Para resolver el problema, debemos utilizar una nueva forma de proyección de tipos introducida en Java SE5, que es la proyección de tipos a través de una clase genérica:

```

//: generics/ClassCasting.java
import java.io.*;
import java.util.*;

```

```

public class ClassCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        // No se compilará:
        // List<Widget> lw1 =
        // List<Widget>.class.cast(in.readObject());
        List<Widget> lw2 = List.class.cast(in.readObject());
    }
} //:-~

```

Sin embargo, no podemos efectuar una proyección sobre el tipo real (`List<Widget>`). En otras palabras, no podemos escribir:

```
List<Widget>.class.cast(in.readObject())
```

e incluso si añadimos otra proyección como la siguiente:

```
(List<Widget>)List.class.cast(in.readObject())
```

seguiremos obteniendo una advertencia.

Ejercicio 32: (1) Verifique que `FixedSizeStack` en `GenericCast.java` genera excepciones si intentamos salimos de los límites fijados. ¿Quiere esto decir que no se necesita el código de comprobación de límites?

Ejercicio 33: (3) Corrija `GenericCast.java` utilizando un contenedor `ArrayList`.

Sobrecarga

El siguiente ejemplo no se podrá compilar, aunque parece que se trata de algo bastante razonable:

```

//: generics/UseList.java
// {CompileTimeError} (no se compilará)
import java.util.*;

public class UseList<W,T> {
    void f(List<T> v) {}
    void f(List<W> v) {}
} //:-~

```

Al sobrecargar el método se produce, debido al mecanismo de borrado de tipos, un conflicto debido a la existencia de signaturas idénticas.

En lugar de ello, debemos proporcionar nombres de métodos distintos en aquellos casos en que el borrado de tipos en los argumentos no genere listas de argumentos diferenciadas:

```

//: generics/UseList2.java
import java.util.*;

public class UseList2<W,T> {
    void f1(List<T> v) {}
    void f2(List<W> v) {}
} //:-~

```

Afortunadamente, el compilador detecta este tipo de problemas.

Secuestro de una interfaz por parte de la clase base

Suponga que disponemos de una clase `Pet` que es `Comparable` con otros objetos `Pet`:

```

//: generics/ComparablePet.java
public class ComparablePet

```

```
implements Comparable<ComparablePet> {
    public int compareTo(ComparablePet arg) { return 0; }
} //:-
```

Resulta razonable tratar de enfocar mejor el tipo con el que pueda compararse una subclase de **ComparablePet**. Por ejemplo, un objeto **Cat** sólo debería ser **Comparable** con otros objetos **Cat**:

```
//: generics/HijackedInterface.java
// {CompileTimeError} (no se compilará)

class Cat extends ComparablePet implements Comparable<Cat>{
    // Error: Comparable no puede heredarse con
    // diferentes argumentos: <Cat> y <Pet>
    public int compareTo(Cat arg) { return 0; }
} //:-
```

Lamentablemente, esta solución no funciona. Una vez que se establece el argumento **ComparablePet** para **Comparable**, no puede ya compararse ninguna otra clase implementadora con ninguna cosa, salvo con **ComparablePet**:

```
//: generics/RestrictedComparablePets.java

class Hamster extends ComparablePet
implements Comparable<ComparablePet> {
    public int compareTo(ComparablePet arg) { return 0; }
}

// O simplemente:

class Gecko extends ComparablePet {
    public int compareTo(ComparablePet arg) { return 0; }
} //:-
```

Hamster demuestra que es posible reimplementar la misma interfaz que podemos encontrar en **ComparablePet**, siempre y cuando sea exactamente la misma, incluyendo los tipos de parámetro. Sin embargo, esto es lo mismo que limitarse a sustituir los métodos en la clase base, como puede verse en **Gecko**.

Tipos autolimitados

Existe una sintaxis que provoca bastante confusión y que aparece con bastante asiduidad en el caso de los genéricos de Java. He aquí el aspecto:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

Esto es algo comparable a tener dos espejos enfrentados, lo que genera una especie de reflexión infinita. La clase **SelfBounded** toma un argumento genérico **T**, por su parte, **T** está restringido por un límite, y dicho límite es **SelfBounded**, con **T** como argumento.

Este tipo de sintaxis resulta difícil de entender cuando se ve por primera vez, y nos permite enfatizar el hecho de que la palabra clave **extends**, cuando se utiliza con los límites de los genéricos, es completamente distinta que cuando se la usa para crear subclases.

Genéricos curiosamente recurrentes

Para comprender lo que significa un tipo autolimitado, comenzemos con una versión más simple de esa sintaxis, sin el auto-límite.

No podemos heredar directamente de un parámetro genérico. Sin embargo, *sí que podemos* heredar de una clase que utilice dicho parámetro genérico en su propia definición. En otras palabras, podemos escribir:

```
//: generics/CuriouslyRecurringGeneric.java
class GenericType<T> {}
```

```
public class CuriouslyRecurringGeneric
    extends GenericType<CuriouslyRecurringGeneric> {} //:-~
```

Este tipo de estructura podría denominarse *genérico curiosamente recurrente* siguiendo el título del artículo *Curiously Recurring Template Pattern* de Jim Coplien aplicado a C++. La parte “curiosamente recurrente” hace referencia al hecho de que nuestra clase, lo cual resulta muy curioso, en su propia clase base.

Para comprender lo que esto significa, tratemos de enunciarlo en voz alta: “Estamos creando una nueva clase que hereda de un tipo genérico que toma el nombre de nuestra clase como parámetro”. ¿Qué es lo que puede hacer el tipo base genérico cuando se le da el nombre de la clase derivada? A este respecto, tenemos que tener en cuenta que los genéricos en Java están relacionados con los argumentos y los tipos de retorno, así que se puede definir una clase base que utilice el tipo derivado en sus argumentos y en sus tipos de retorno. También puede utilizar el tipo derivado para definir el tipo de los campos, aún cuando esos tipos serán borrados y sustituidos por **Object**. He aquí una clase genérica que nos permite expresar esto:

```
//: generics/BasicHolder.java

public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
} //:-~
```

Se trata de un tipo genérico normal con métodos que aceptan y generan objetos del tipo especificado en el parámetro, junto con un método que opera sobre el campo almacenado (aunque únicamente realiza operaciones de tipo **Object** con ese campo).

Podemos utilizar **BasicHolder** en un genérico curiosamente recurrente:

```
//: generics/CRGWithBasicHolder.java

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
} /* Output:
Subtype
*//:-~
```

Observe en este ejemplo algo muy importante: la nueva clase **Subtype** toma argumentos y valores de retorno de tipo **Subtype**, no simplemente de la clase base **BasicHolder**. Ésta es la esencia de los genéricos curiosamente recurrentes: *la clase derivada sustituye a la clase base en sus parámetros*. Esto significa que la clase base genérica se convierte en una cierta clase de plantilla para describir la funcionalidad común de todas sus clases derivadas, pero esta funcionalidad utilizará el tipo derivado en todos los argumentos y tipos de retorno. En otras palabras, se utilizará el tipo exacto en lugar del tipo base en la clase resultante. Por tanto, en **Subtype**, tanto el argumento de **set()** como el tipo de retorno de **get()** son exactamente **Subtype**.

Autolimitación

El contenedor **BasicHolder** puede utilizar cualquier tipo como su parámetro genérico, como puede verse aquí:

```
//: generics/Unconstrained.java

class Other {}
```

```

class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
    public static void main(String[] args) {
        BasicOther b = new BasicOther(), b2 = new BasicOther();
        b.set(new Other());
        Other other = b.get();
        b.f();
    }
} /* Output:
Other
*///:-

```

La autolimitación realiza el paso adicional de *obligar* a que el genérico se utilice como su propio argumento límite. Examinemos cómo puede utilizarse y cómo no puede utilizarse la clase resultante:

```

//: generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
    T element;
    SelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // También OK

class C extends SelfBounded<C> {
    C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// No se puede hacer esto:
// class E extends SelfBounded<D> {}
// Error de compilación: el parámetro de tipo D no está dentro de su límite

// Sin embargo, podemos hacer esto, así que no se puede forzar la sintaxis:
class F extends SelfBounded {}

public class SelfBounding {
    public static void main(String[] args) {
        A a = new A();
        a.set(new A());
        a = a.set(new A()).get();
        a = a.get();
        C c = new C();
        c = c.setAndGet(new C());
    }
} /*/

```

Lo que la autolimitación hace es requerir el uso de la clase en una relación de herencia como ésta:

```
class A extends SelfBounded<A> {}
```

Esto nos fuerza a pasar la clase que estemos definiendo como parámetro a la clase base.

¿Cuál es el valor añadido que obtenemos al autolimitar el parámetro? El parámetro de tipo debe ser el mismo que la clase que se esté definiendo. Como podemos ver en la definición de la clase **B**, también podemos heredar de una clase

SelfBounded que utilice un parámetro **SelfBounded**, aunque el uso predominante parece ser el que podemos ver en la clase **A**. El intento de definir **E** demuestra que no se puede utilizar un parámetro de tipo que no sea **SelfBounded**.

Lamentablemente, **F** se compila sin ninguna advertencia, por lo que la sintaxis de autolimitación no se puede imponer. Si fuera realmente importante, se necesitaría una herramienta externa para garantizar que los tipos normales no se utilizan en lugar de los tipos parametrizados.

Observe que se puede eliminar la restricción y todas las clases se seguirán compilando, pero entonces **E** también se compilaría:

```
//: generics/NotSelfBounded.java

public class NotSelfBounded<T> {
    T element;
    NotSelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
    C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Ahora esto es OK:
class E2 extends NotSelfBounded<D2> {} //:-
```

Así que, obviamente, la restricción de autolimitación sólo sirve para imponer la relación de herencia. Si se utiliza la autolimitación, sabemos que el parámetro de tipo utilizado por la clase será el mismo tipo básico que la clase que esté utilizando dicho parámetro. Esto obliga a cualquiera que utilice dicha clase a ajustarse a ese formato.

También resulta posible utilizar la autolimitación en los métodos genéricos:

```
//: generics/SelfBoundingMethods.java

public class SelfBoundingMethods {
    static <T extends SelfBounded<T>> T f(T arg) {
        return arg.set(arg).get();
    }
    public static void main(String[] args) {
        A a = f(new A());
    }
} //:-
```

Esto evita que el método se aplique a ningún tipo, excepto un argumento autolimitado de la forma mostrada.

Covarianza de argumentos

El valor de los tipos autolimitados es que producen *tipos de argumentos covariantes*, es decir, tipos de argumentos de los métodos que varían con el fin de ajustarse a las subclases.

Aunque los tipos autolimitados también producen tipos de retorno que coinciden con el tipo de la subclase, esto no es tan importante, porque Java SE5 ya introduce la funcionalidad de *tipos de retorno covariantes*:

```
//: generics/CovariantReturnTypes.java

class Base {}
```

```

class Derived extends Base {}

interface OrdinaryGetter {
    Base get();
}

interface DerivedGetter extends OrdinaryGetter {
    // El tipo de retorno del método sustituido puede variar:
    Derived get();
}

public class CovariantReturnTypes {
    void test(DerivedGetter d) {
        Derived d2 = d.get();
    }
} //:-

```

El método `get()` en `DerivedGetter` sustituye a `get()` en `OrdinaryGetter` y devuelve un tipo que se deriva del tipo devuelto por `OrdinaryGetter.get()`. Aunque se trata de algo perfectamente lógico (un método de un tipo derivado debería poder devolver el tipo más específico que el método del tipo base que esté sustituyendo), no podía hacerse en las versiones anteriores de Java.

Un genérico autolimitado produce, de hecho, el tipo exacto derivado como valor de retorno, como podemos ver aquí con el método `get()`:

```

//: generics/GenericsAndReturnTypes.java

interface GenericGetter<T extends GenericGetter<T>> {
    T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
    void test(Getter g) {
        Getter result = g.get();
        GenericGetter gg = g.get(); // También el tipo base
    }
} //:-

```

Observe que este código no podría haberse compilado de no haberse incluido los tipos de retorno covariantes en Java SE5.

Sin embargo, en el código no genérico, los *tipos de argumento* no pueden variar con los subtipos:

```

//: generics/OrdinaryArguments.java

class OrdinarySetter {
    void set(Base base) {
        System.out.println("OrdinarySetter.set(Base)");
    }
}

class DerivedSetter extends OrdinarySetter {
    void set(Derived derived) {
        System.out.println("DerivedSetter.set(Derived)");
    }
}

public class OrdinaryArguments {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
    }
}

```

```

DerivedSetter ds = new DerivedSetter();
ds.set(derived);
ds.set(base); // Se compila: sobrecargado, no sustituido
}
} /* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*///:-

```

Tanto `set(derived)` como `set(base)` son legales, por lo que `DerivedSetter.set()` no está sustituyendo `OrdinarySetter.set()`, sino que está *sobrecargando* dicho método. Analizando la salida, puede ver que hay dos métodos en `DerivedSetter`, por lo que la versión de la clase base sigue estando disponible, lo que confirma que se ha producido una sobrecarga del método.

Sin embargo, con los tipos autolimitados, sólo hay un único método en la clase derivada y dicho método toma el tipo derivado como argumento, no el tipo base:

```

//: generics/SelfBoundingAndCovariantArguments.java

interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
    void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
    void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
        s1.set(s2);
        // s1.set(sbs); // Error:
        // set(Setter) en SelfBoundSetter<Setter>
        // no puede aplicarse a (SelfBoundSetter)
    }
} //://:-

```

El compilador no reconoce el intento de pasar el tipo de base como argumento a `set()`, porque no existe ningún método con dicha firma. El argumento ha sido, en la práctica, sustituido.

Si el mecanismo de autolimitación, entra en acción el mecanismo normal de herencia y lo que obtenemos es una sobrecarga, al igual que sucede en el caso no genérico:

```

//: generics/PlainGenericInheritance.java

class GenericSetter<T> { // Sin autolimitación
    void set(T arg) {
        System.out.println("GenericSetter.set(Base)");
    }
}

class DerivedGS extends GenericSetter<Base> {
    void set(Derived derived) {
        System.out.println("DerivedGS.set(Derived)");
    }
}

public class PlainGenericInheritance {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedGS dgs = new DerivedGS();
        dgs.set(derived);
        dgs.set(base); // Se compila: sobrecargado, no sustituido
    }
} /* Output:

```

```
DerivedGS.set(Derived)
GenericSetter.set(Base)
*///:-
```

Este código se asemeja a `OrdinaryArguments.java`; en dicho ejemplo, `DerivedSetter` hereda de `OrdinarySetter` que contiene un conjunto `set(Base)`. Aquí, `DerivedGS` hereda de `GenericSetter<Base>` que también contiene un conjunto `set(Base)`, creado por el genérico. Y al igual que en `OrdinaryArguments.java`, podemos ver analizando la salida que `DerivedGS` contiene dos versiones sobrecargadas de `set()`. Sin el mecanismo de la autolimitación, lo que se hace es sobrecargar los tipos de argumentos. Si se utiliza la autolimitación, se termina disponiendo de una única versión de un método, que admite el tipo exacto de argumento.

Ejercicio 34: (4) Cree un tipo genérico autolimitado que contenga un método abstracto que admita un argumento con el tipo del parámetro genérico y genere un valor de retorno con el tipo del parámetro genérico. En un método no abstracto de la clase, invoque dicho método abstracto y devuelva su resultado. Defina otra clase que herede del tipo autolimitado y compruebe el funcionamiento de la clase resultante.

Seguridad dinámica de los tipos

Dado que podemos pasar contenedores genéricos a los programas anteriores a Java SE5, sigue existiendo la posibilidad de que código escrito con el estilo antiguo corrompa nuestros contenedores. Java SE5 dispone de un conjunto de utilidades en `java.util.Collections` para resolver el problema de comprobación de tipos en esta situación: los métodos estáticos `checkedCollection()`, `checkedList()`, `checkedMap()`, `checkedSet()`, `checkedSortedMap()` y `checkedSortedSet()`. Cada uno de estos métodos toma como primer argumento el contenedor que queremos comprobar dinámicamente y como segundo argumento el tipo que queremos imponer que se utilice.

Un contenedor comprobado generará una excepción `ClassCastException` en cualquier lugar en el que tratemos de *insertar* un objeto inapropiado, a diferencia de los contenedores anteriores a la introducción del mecanismo de genéricos (contenedores normales y corrientes), que lo que harían sería informarnos de que hay un problema en el momento de tratar de *extraer* el objeto. En este último caso, sabremos que existe un problema, pero no podremos determinar quién es el culpable; por el contrario, con los contenedores comprobados, sí que podemos averiguar quién es el que ha tratado de insertar el objeto erróneo.

Examinemos este problema de “inscripción de un gato en una lista de perros” utilizando un contenedor comprobado. Aquí, `oldStyleMethod()` representa un cierto código heredado, porque admite un objeto `List` normal, siendo necesaria la anotación `@SuppressWarnings("unchecked")` para suprimir la advertencia resultante:

```
//: generics/CheckedList.java
// Using Collection.checkedList().
import typeinfo.pets.*;
import java.util.*;

public class CheckedList {
    @SuppressWarnings("unchecked")
    static void oldStyleMethod(List probablyDogs) {
        probablyDogs.add(new Cat());
    }
    public static void main(String[] args) {
        List<Dog> dogs1 = new ArrayList<Dog>();
        oldStyleMethod(dogs1); // Acepta sin rechistar un objeto Cat
        List<Dog> dogs2 = Collections.checkedList(
            new ArrayList<Dog>(), Dog.class);
        try {
            oldStyleMethod(dogs2); // Genera una excepción
        } catch(Exception e) {
            System.out.println(e);
        }
        // Los tipos derivados funcionan correctamente:
        List<Pet> pets = Collections.checkedList(
            new ArrayList<Pet>(), Pet.class);
        pets.add(new Dog());
    }
}
```

```

    pets.add(new Cat());
}
} /* Output:
java.lang.ClassCastException: Attempt to insert class typeinfo.pets.Cat element into
collection with element type class typeinfo.pets.Dog
*///:-

```

Cuando ejecutamos el programa, vemos que la inserción de un objeto **Cat** no provoca ninguna queja por parte **dogs1**, pero **dogs2** genera inmediatamente una excepción al tratar de insertar un tipo incorrecto. También podemos ver que resulta perfectamente posible introducir objetos de un tipo derivado dentro de un contenedor comprobado donde se esté haciendo la comprobación según el tipo base.

Ejercicio 35: (1) Modifique **CheckedList.java** para que utilice las clases **Coffee** definidas en este capítulo.

Excepciones

Debido al mecanismo de borrado de tipos, la utilización de genéricos con excepciones es extremadamente limitada. Una cláusula **catch** no puede tratar una excepción de un tipo genérico, porque es necesario conocer el tipo exacto de la excepción tanto en tiempo de compilación como en tiempo de ejecución. Asimismo, una clase genérica no puede heredar directa ni indirectamente de **Throwable** (esto impide que tratemos de definir excepciones genéricas que no puedan ser atrapadas).

Sin embargo, los parámetros de tipo sí que pueden usarse en la cláusula **throws** de la declaración de un método. Esto nos permite escribir código genérico que varíe según el tipo de una excepción comprobada:

```

//: generics/ThrowGenericException.java
import java.util.*;

interface Processor<T,E extends Exception> {
    void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T,E extends Exception>
extends ArrayList<Processor<T,E>> {
    List<T> processAll() throws E {
        List<T> resultCollector = new ArrayList<T>();
        for(Processor<T,E> processor : this)
            processor.process(resultCollector);
        return resultCollector;
    }
}

class Failure1 extends Exception {}

class Processor1 implements Processor<String,Failure1> {
    static int count = 3;
    public void
    process(List<String> resultCollector) throws Failure1 {
        if(count-- > 1)
            resultCollector.add("Hep!");
        else
            resultCollector.add("Ho!");
        if(count < 0)
            throw new Failure1();
    }
}

class Failure2 extends Exception {}

class Processor2 implements Processor<Integer,Failure2> {

```

```

static int count = 2;
public void
process(List<Integer> resultCollector) throws Failure2 {
    if(count-- == 0)
        resultCollector.add(47);
    else {
        resultCollector.add(11);
    }
    if(count < 0)
        throw new Failure2();
}
}

public class ThrowGenericException {
    public static void main(String[] args) {
        ProcessRunner<String,Failure1> runner =
            new ProcessRunner<String,Failure1>();
        for(int i = 0; i < 3; i++)
            runner.add(new Processor1());
        try {
            System.out.println(runner.processAll());
        } catch(Failure1 e) {
            System.out.println(e);
        }

        ProcessRunner<Integer,Failure2> runner2 =
            new ProcessRunner<Integer,Failure2>();
        for(int i = 0; i < 3; i++)
            runner2.add(new Processor2());
        try {
            System.out.println(runner2.processAll());
        } catch(Failure2 e) {
            System.out.println(e);
        }
    }
} //:-

```

Un objeto **Processor** ejecuta un método **process()** y puede generar una excepción de tipo E. El resultado de **process()** se almacena en el contenedor **List<T> resultCollector** (esto se denomina *parámetro de recolección*). Un objeto **ProcessRunner** tiene un método **processAll()** que ejecuta todo objeto **Processor** que almacene y devuelve el objeto **resultCollector**.

Si no pudiéramos parametrizar las excepciones generadas, no podríamos escribir este código en forma genérica, debido a la existencia de las excepciones comprobadas.

Ejercicio 36: (2) Añada una segunda excepción parametrizada a la clase **Processor** y demuestre que las excepciones pueden variar independientemente.

Mixins

El término *mixin* parece haber adquirido distintos significados a lo largo del tiempo, pero el concepto fundamental se refiere a la mezcla (*mixing*) de capacidades de múltiples clases con el fin de producir una clase resultante que represente a todos los tipos del *mixin*. Este tipo de labor suele realizarse en el último minuto, lo que hace que resulte bastante conveniente para ensamblar fácilmente unas clases con otras.

Una de las ventajas de los mixins es que permiten aplicar coherentemente una serie de características y comportamientos a múltiples clase. Como ventaja añadida, si queremos cambiar algo en una clase *mixin*, dichos cambios se aplicarán a todas las clases a las que se les haya aplicado el *mixin*. Debido a esto, los *mixins* parecen orientados a lo que se denomina *programación orientada a aspectos* (AOP, *aspect-oriented programming*), y diversos autores sugieren precisamente que se utilice el concepto de aspectos para resolver el problema de los *mixins*.

Mixins en C++

Uno de los argumentos más sólidos en favor de la utilización de los mecanismos de herencia múltiples en C++ es, precisamente, el poder utilizar *mixins*. Sin embargo, un enfoque más interesante y más elegante a la hora de tratar los *mixins* es el que se basa en la utilización de tipos parametrizados; según este enfoque, un *mixin* es una clase que hereda de su parámetro de tipo. En C++, podemos crear *mixins* fácilmente debido a que C++ recuerda el tipo de sus parámetros de plantilla.

He aquí un ejemplo de C++ con dos tipos de *mixin*: uno que permite añadir la propiedad de disponer de una marca temporal y otro que añade un número de serie para cada instancia de objeto:

```
//: generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
    long timeStamp;
public:
    TimeStamped() { timeStamp = time(0); }
    long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
    long serialNumber;
    static long counter;
public:
    SerialNumbered() { serialNumber = counter++; }
    long getSerialNumber() { return serialNumber; }
};

// Definir e inicializar el almacenamiento estático:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
    string value;
public:
    void set(string val) { value = val; }
    string get() { return value; }
};

int main() {
    TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
    mixin1.set("test string 1");
    mixin2.set("test string 2");
    cout << mixin1.get() << " " << mixin1.getStamp() <<
        " " << mixin1.getSerialNumber() << endl;
    cout << mixin2.get() << " " << mixin2.getStamp() <<
        " " << mixin2.getSerialNumber() << endl;
} /* Output: (Sample)
test string 1 1129840250 1
test string 2 1129840250 2
*///:-
```

En **main()**, el tipo resultante de **mixin1** y **mixin2** tiene todos los métodos de los tipos que se han usado en la mezcla. Podemos considerar un *mixin* como una especie de función que establece una correspondencia entre clases existentes y una serie de nuevas subclases. Observe lo fácil que resulta crear un *mixin* utilizando esta técnica; básicamente, nos limitamos a decir lo que queremos y el compilador se encarga del resto:

```
TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
```

Lamentablemente, los genéricos de Java no permiten esto. El mecanismo de borrado de tipos hace que se pierda la información acerca del tipo de la clase base, por lo que una clase genérica no puede heredar directamente de un parámetro genérico.

Mezclado de clases utilizando interfaces

Una solución comúnmente sugerida consiste en utilizar interfaces para generar el efecto de los *mixins*, de la forma siguiente:

```
//: generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
    private final long timeStamp;
    public TimeStampedImp() {
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public long getSerialNumber() { return serialNumber; }
}

interface Basic {
    public void set(String val);
    public String get();
}

class BasicImp implements Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
    private TimeStamped timeStamp = new TimeStampedImp();
    private SerialNumbered serialNumber =
        new SerialNumberedImp();
    public long getStamp() { return timeStamp.getStamp(); }
    public long getSerialNumber() {
        return serialNumber.getSerialNumber();
    }
}

public class Mixins {
    public static void main(String[] args) {
        Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
        mixin1.set("test string 1");
        mixin2.set("test string 2");
        System.out.println(mixin1.get() + " " +
            mixin1.getStamp() + " " + mixin1.getSerialNumber());
        System.out.println(mixin2.get() + " " +
            mixin2.getStamp() + " " + mixin2.getSerialNumber());
    }
}
```

```

    }
} /* Output: (Sample)
test string 1 1132437151359 1
test string 2 1132437151359 2
*///:-
```

La clase **Mixin** está utilizando, básicamente, el mecanismo de *delegación*, por lo que cada uno de los tipos mezclados requiere un campo en **Mixin**, y es necesario escribir todos los métodos que se precisan en **Mixin** para redirigir las llamadas al objeto apropiado. Este ejemplo utiliza clases triviales, pero en un *mixin* más complejo el código puede llegar a crecer de tamaño de forma bastante rápida.⁴

Ejercicio 37: (2) Añada una nueva clase *mixin Colored* a Mixins.java, mézclela en **Mixin** y demuestre que funciona.

Utilización del patrón Decorador

Cuando examinamos la forma en que se utiliza, el concepto de *mixin* parece estar estrechamente relacionado con el patrón de diseño *Decorador*.⁵ Los decoradores se utilizan a menudo en aquellas ocasiones donde, para poder satisfacer cada una de las combinaciones posibles, el mecanismo simple de creación de subclases produce tantas clases que llega a resultar poco práctico.

El patrón Decorador utiliza objetos en distintos niveles para añadir responsabilidades, de forma dinámica y transparente, a distintos objetos individuales. El Decorador especifica que todos los objetos que envuelven al objeto inicial de partida tienen la misma interfaz básica. En cierto modo, lo que hacemos es definir que un cierto objeto es “decorable” y luego agregarle funcionalidad por el método de envolver alrededor de ese objeto otra serie de clases. Esto hace que el uso de los decoradores sea transparente: existe un conjunto de mensajes comunes que podemos enviar a un objeto, independientemente de si éste ha sido decorado o no. La clase decoradora también tener métodos pero, como veremos, esta posibilidad tiene sus limitaciones.

Los decoradores se implementan utilizando los mecanismos de composición junto con estructuras formales (la jerarquía de objetos decorables/decoradores), mientras que los *mixins* están basados en el mecanismo de herencia. Por tanto, podríamos decir que los *mixins* basados en tipos parametrizados son una especie de mecanismo genérico decorador que no requiere que se utilice la estructura de herencias definida en el patrón de diseño Decorador.

El ejemplo anterior puede rehacerse con el patrón de diseño Decorador:

```

//: generics/decorator/Decoration.java
package generics.decorator;
import java.util.*;

class Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Decorator extends Basic {
    protected Basic basic;
    public Decorator(Basic basic) { this.basic = basic; }
    public void set(String val) { basic.set(val); }
    public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
    private final long timeStamp;
    public TimeStamped(Basic basic) {

```

⁴ Observe que algunos entornos de programación, como Eclipse e IntelliJ Idea, generan automáticamente el código de delegación.

⁵ Los patrones se tratan en *Thinking in Patterns (with Java)*, que puede encontrar en www.MindView.net. Consulte también *Design Patterns*, de Erich Gamma *et al.* (Addison-Wesley, 1995).

```

        super(basic);
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public SerialNumbered(Basic basic) { super(basic); }
    public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
    public static void main(String[] args) {
        TimeStamped t = new TimeStamped(new Basic());
        TimeStamped t2 = new TimeStamped(
            new SerialNumbered(new Basic()));
        //! t2.getSerialNumber(); // No disponible
        SerialNumbered s = new SerialNumbered(new Basic());
        SerialNumbered s2 = new SerialNumbered(
            new TimeStamped(new Basic()));
        //! s2.getStamp(); // No disponible
    }
} //:-

```

La clase resultante de un *mixin* contiene todos los métodos de interés, pero el tipo de objeto que resulta de la utilización de decoradores es el tipo con que el objeto haya sido decorado. En otras palabras, aunque es posible añadir más de un nivel, el tipo real será el último de esos niveles, de modo que sólo serán visibles los métodos de ese nivel final; por el contrario, el tipo de un *mixin* es *todos* los tipos que se hayan mezclado. En consecuencia, una desventaja significativa del patrón de diseño Decorador es que sólo trabaja, en la práctica, con uno de los niveles de decoración (el nivel final) mientras que la técnica basada en *mixin* resulta bastante más natural. Por tanto, el patrón de diseño Decorador sólo constituye una solución limitada para el problema que los *mixins* abordan.

Ejercicio 38: (4) Cree un sistema Decorador simple comenzando con una clase que represente un café normal y luego proporcionando una serie de decoradores que representen la leche, la espuma, el chocolate, el caramelo y la crema batida.

Mixins con proxies dinámicos

Resulta posible utilizar un *proxy* dinámico para un mecanismo que permita modelar los *mixins* de forma más precisa que lo que se puede conseguir utilizando el patrón de diseño Decorador (consulte el Capítulo 14, *Información de tipos*, para ver una explicación acerca de cómo funcionan los *proxies* dinámicos en Java). Con un *proxy* dinámico, el tipo *dinámico* de la clase resultante es igual a los tipos combinados que hayamos mezclado.

Debido a las restricciones de los *proxies* dinámicos, cada una de las clases que intervengan en la mezcla deberá ser la implementación de una interfaz:

```

//: generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

class MixinProxy implements InvocationHandler {
    Map<String, Object> delegatesByMethod;
    public MixinProxy(TwoTuple<Object, Class<?>>... pairs) {
        delegatesByMethod = new HashMap<String, Object>();
        for(TwoTuple<Object, Class<?>> pair : pairs) {

```

```

        for(Method method : pair.second.getMethods()) {
            String methodName = method.getName();
            // La primera interfaz del mapa
            // implementa el método.
            if (!delegatesByMethod.containsKey(methodName))
                delegatesByMethod.put(methodName, pair.first);
        }
    }

    public Object invoke(Object proxy, Method method,
        Object[] args) throws Throwable {
        String methodName = method.getName();
        Object delegate = delegatesByMethod.get(methodName);
        return method.invoke(delegate, args);
    }

    @SuppressWarnings("unchecked")
    public static Object newInstance(TwoTuple... pairs) {
        Class[] interfaces = new Class[pairs.length];
        for(int i = 0; i < pairs.length; i++) {
            interfaces[i] = (Class)pairs[i].second;
        }
        ClassLoader cl =
            pairs[0].first.getClass().getClassLoader();
        return Proxy.newProxyInstance(
            cl, interfaces, new MixinProxy(pairs));
    }

}

public class DynamicProxyMixin {
    public static void main(String[] args) {
        Object mixin = MixinProxy.newInstance(
            tuple(new BasicImpl(), Basic.class),
            tuple(new TimeStampedImpl(), TimeStamped.class),
            tuple(new SerialNumberedImpl(), SerialNumbered.class));
        Basic b = (Basic)mixin;
        TimeStamped t = (TimeStamped)mixin;
        SerialNumbered s = (SerialNumbered)mixin;
        b.set("Hello");
        System.out.println(b.get());
        System.out.println(t.getStamp());
        System.out.println(s.getSerialNumber());
    }
} /* Output: (Sample)
Hello
1132519137015
1
*///:-

```

Puesto que el único que incluye todos los tipos mezclados es el tipo dinámico y no es tipo estático, esta solución sigue sin ser tan elegante como la utilizada en C++, ya que nos vemos obligados a realizar una especialización sobre el tipo apropiado antes de que podamos invocar ningún método del mismo. Sin embargo, esta solución está significativamente más próxima que las anteriores a lo que es el concepto de un verdadero *mixin*.

Es mucho el trabajo que se ha realizado para poder soportar *mixins* en Java, incluyendo la creación de una extensión del lenguaje (el lenguaje Jam) que se ha definido específicamente con el objeto de soportar los *mixins*.

Ejercicio 39: (1) Añada una nueva clase mixin **Colored** a **DynamicProxyMixin.java**, mézclela en **Mixin**, y demuestre que funciona.

Tipos latentes

Al principio de este capítulo hemos introducido la idea de escribir código que pueda ser aplicado de la forma más general posible. Para hacer esto, necesitamos formas de relajar las restricciones que afectan a los tipos con los que nuestro código puede trabajar, sin por ello perder los beneficios proporcionados por el mecanismo de comprobación estática de tipos. Por eso, somos capaces de escribir código que puede utilizarse, sin modificaciones, en el número mayor de situaciones; en otras palabras, código “más genérico”.

Los genéricos de Java parecen dar un paso adicional en esta dirección. Cuando escribimos o utilizamos genéricos que simplemente se emplean para almacenar objetos, el código funciona con cualquier tipo de datos (salvo con las primitivas, aunque como hemos visto, el mecanismo de conversión automática solventa este problema). Dicho de otra manera, los genéricos de “almacenamiento” son capaces de decir: “No me importa de qué tipo eres”. El código al que no le preocupa el tipo de dato con el que funciona puede aplicarse, ciertamente, en cualquier situación y es, por tanto, bastante “genérico”.

Como también hemos visto, surge un problema en el momento en el que queremos realizar manipulaciones con los tipos genéricos (distintas de la invocación de métodos de `Object`), porque el mecanismo de borrado de tipos requiere que especifiquemos los límites que pueden usarse para los tipos genéricos, con el fin de poder invocar de manera segura métodos específicos para los objetos genéricos dentro del código. Ésta es una limitación significativa al concepto de “genérico”, porque es necesario restringir los tipos genéricos de modo que puedan heredar de clases concretas o implementar interfaces particulares. En algunos casos, puede que terminemos generando en lugar de genéricos una clase o interfaz normales, debido a que un genérico con límites puede no diferir de la especificación de una clase o una interfaz.

Una solución que proporcionan algunos lenguajes de programación se denomina *tipos latentes* o *tipos estructurales*. Un término más coloquial es el de *tipos pato* (*duck typing*); este término ha llegado a ser muy popular debido a que no acarrea el bagaje que los otros términos si acarrean. El término proviene de la frase “si camina como un pato y se comporta como un pato, podemos tratarlo como un pato”.

El código genérico, normalmente, sólo invoca unos cuantos métodos de un tipo genérico, y los lenguajes con tipos latentes relajan la restricción (teniendo que producir código más genérico) requiriendo que tan sólo un subconjunto de los métodos sea implementado, *no* una clase o interfaz concretas. Debido a esto, los tipos latentes permiten saltar las fronteras de las jerarquías de clase, invocando métodos que no forman parte de una interfaz común. En la práctica, un fragmento de código podría decir: “No me importa de qué tipo eres, siempre y cuando dispongas de los métodos `speak()` y `sit()`”. Al no requerir un tipo específico, el código puede ser más genérico.

Los tipos latentes son un mecanismo de organización y reutilización del código. Con ellos, podemos escribir fragmentos de código que pueden reutilizarse más fácilmente que si no se empleara el mecanismo. La organización y reutilización del código son las herramientas fundamentales de la programación informática: escribir el código una sola vez, utilizarlo más de una vez y mantener el código en un único lugar. Al no vernos obligados a especificar una interfaz exacta con la que el código pueda operar, los tipos latentes nos permiten escribir menos código y aplicar dicho código más fácilmente y en más lugares.

Dos ejemplos de lenguajes que soportan el mecanismo de tipos latentes son Python (que puede descargarse gratuitamente de www.Python.org) y C++.⁶ Python es un lenguaje con tipos dinámicos (prácticamente todas las comprobaciones de tipos se realizan en tiempo de ejecución) y C++ es un lenguaje con tipos estáticos (las comprobaciones de tipos se realizan en tiempo de compilación), por lo que los tipos latentes pueden utilizarse tanto con las comprobaciones de tipo estáticas como con las dinámicas.

Si tomamos la descripción anterior y la expresamos en Python, tendría el aspecto siguiente:

```
#: generics/DogsAndRobots.py

class Dog:
    def speak(self):
        print "Arf!"
    def sit(self):
        print "Sitting"
    def reproduce(self):
        pass
```

⁶ Los lenguajes Ruby y Smalltalk también soportan los tipos latentes.

```

class Robot:
    def speak(self):
        print "Click!"
    def sit(self):
        print "Clank!"
    def oilChange(self):
        pass

def perform(anything):
    anything.speak()
    anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)
#:-

```

Python utiliza el sangrado para determinar el ámbito (así que no hacen falta símbolos de llaves) y un símbolo de dos puntos para dar comienzo a un nuevo ámbito. Un símbolo '#' indica un comentario que se extiende hasta el final de la línea, al igual que '//' en Java. Los métodos de una clase especifican explícitamente, como primer argumento, el equivalente de la referencia **this**, que en este lenguaje se denomina **self** por convenio. Las llamadas a constructor no requieren ninguna clase de palabra clave "**new**". Y Python permite la existencia de funciones normales (es decir, funciones que no son miembro de una clase), como pone de manifiesto la función **perform()**.

En **perform(anything)**, observe que no se indica ningún tipo para **for anything**, y que **anything** es simplemente un identificador. Esta variable debe ser capaz de realizar las operaciones que **perform()** le pida, por lo que hay una interfaz implícita. Pero nunca hace falta escribir explícitamente dicha interfaz, ya que está latente. **perform()** no se preocupa de cuál sea el tipo de su argumento, así que podemos pasárselo a esa función cualquier objeto siempre y cuando éste soporte los métodos **speak()** y **sit()**. Si pasamos a **perform()** un objeto que no soporte estas operaciones, obtendremos una excepción en tiempo de ejecución.

Podemos producir el mismo efecto en C++:

```

//: generics/DogsAndRobots.cpp

class Dog {
public:
    void speak() {}
    void sit() {}
    void reproduce() {}
};

class Robot {
public:
    void speak() {}
    void sit() {}
    void oilChange() {}
};

template<class T> void perform(T anything) {
    anything.speak();
    anything.sit();
}

int main() {
    Dog d;
    Robot r;
    perform(d);
    perform(r);
} //:-_

```

Tanto en Python como en C++, **Dog** y **Robot** no tienen nada en común, salvo que ambos disponen de dos métodos con signatures idénticas. Desde el punto de vista de los tipos, se trata de tipos completamente distintos. Sin embargo, a `perform()` no le preocupa el tipo específico de su argumento y el mecanismo de tipos latentes le permite aceptar ambos tipos de objeto.

C++ verifica que pueda enviar dichos mensajes. El compilador proporcionará un mensaje de error si tratamos de pasar el tipo incorrecto (estos mensajes de error han sido, históricamente, bastante amenazantes y muy extensos, y son la razón principal de que las plantillas C++ tengan una reputación tan mala). Aunque ambos lo hacen en instantes distintos (C++ en tiempo de compilación y Python en tiempo de ejecución), los dos lenguajes garantizan que no se puedan utilizar incorrectamente los tipos, y esa es la razón por la que decimos que los dos lenguajes son *fueramente tipados*.⁷ Los tipos latentes no afectan al tipado fuerte.

Como el mecanismo de genéricos se añadió a Java en una etapa tardía, no hubo la oportunidad de implementar un mecanismo de tipos latentes, así que Java no tiene soporte para esta funcionalidad. Como resultado, puede parecer al principio que el mecanismo de genéricos de Java es “menos genérico” que el de otros lenguajes que sí soportan los tipos latentes.⁸ Por ejemplo, si tratamos de implementar el ejemplo anterior en Java, estaremos obligados a utilizar una clase o una interfaz y a especificarlas dentro de una expresión de límite:

```
//: generics/Performs.java

public interface Performs {
    void speak();
    void sit();
} //:-

//: generics/DogsAndRobots.java
// No hay tipos latentes en Java
import typeinfo.pets.*;
import static net.mindview.util.Print.*;

class PerformingDog extends Dog implements Performs {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class Robot implements Performs {
    public void speak() { print("Click!"); }
    public void sit() { print("Clank!"); }
    public void oilChange() {}
}

class Communicate {
    public static <T extends Performs>
    void perform(T performer) {
        performer.speak();
        performer.sit();
    }
}

public class DogsAndRobots {
    public static void main(String[] args) {
        PerformingDog d = new PerformingDog();
```

⁷ Dado que se pueden utilizar proyecciones de tipos, que deshabilitan en la práctica el sistema de tipos, algunos autores argumentan que C++ es un lenguaje débilmente tipado, pero creo que esa opinión es exagerada. Probablemente sea más justo decir que C++ es un lenguaje “fueramente tipado” con una puerta trasera.⁴

⁸ La implementación de los mecanismos de Java utilizando el borrado de tipos se denomina, en ocasiones, mecanismo de tipos genéricos de *segunda clase*.

```

Robot r = new Robot();
Communicate.perform(d);
Communicate.perform(r);
}
} /* Output:
Woof!
Sitting
Click!
Clank!
*///:-
```

Sin embargo, observe que `perform()` no necesita utilizar genéricos para poder funcionar. Podemos simplemente especificar que acepte un objeto `Performs`:

```

//: generics/SimpleDogsAndRobots.java
// Eliminación del genérico, el código sigue funcionando.

class CommunicateSimply {
    static void perform(Performs performer) {
        performer.speak();
        performer.sit();
    }
}

public class SimpleDogsAndRobots {
    public static void main(String[] args) {
        CommunicateSimply.perform(new PerformingDog());
        CommunicateSimply.perform(new Robot());
    }
} /* Output:
Woof!
Sitting
Click!
Clank!
*///:-
```

En este caso, los genéricos eran simplemente innecesarios, ya que las clases estaban ya obligadas a implementar la interfaz `Performs`.

Compensación de la carencia de tipos latentes

Aunque Java no soporta el mecanismo de tipos latentes, resulta que esto no significa que el código genérico con límites no pueda aplicarse a través de diferentes jerarquías de tipos. En otras palabras: sigue siendo posible crear código verdaderamente genérico, aunque hace falta algo de esfuerzo adicional.

Reflexión

Una de las técnicas que podemos utilizar es el mecanismo de reflexión. He aquí el método `perform()` que utiliza tipos latentes:

```

//: generics/LatentReflection.java
// Utilización del mecanismo de reflexión para generar tipos latentes.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// No implementa Performs:
class Mime {
    public void walkAgainstTheWind() {}
    public void sit() { print("Pretending to sit"); }
```

```

public void pushInvisibleWalls() {}
public String toString() { return "Mime"; }
}

// No implementa Performs:
class SmartDog {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class CommunicateReflectively {
    public static void perform(Object speaker) {
        Class<?> spkr = speaker.getClass();
        try {
            try {
                Method speak = spkr.getMethod("speak");
                speak.invoke(speaker);
            } catch(NoSuchMethodException e) {
                print(speaker + " cannot speak");
            }
            try {
                Method sit = spkr.getMethod("sit");
                sit.invoke(speaker);
            } catch(NoSuchMethodException e) {
                print(speaker + " cannot sit");
            }
        } catch(Exception e) {
            throw new RuntimeException(speaker.toString(), e);
        }
    }
}

public class LatentReflection {
    public static void main(String[] args) {
        CommunicateReflectively.perform(new SmartDog());
        CommunicateReflectively.perform(new Robot());
        CommunicateReflectively.perform(new Mime());
    }
} /* Output:
Woof!
Sitting
Click!
Clank!
Mime cannot speak
Pretending to sit
*///:-
```

Aquí, las clases son completamente disjuntas y no tenemos clases base (distintas de `Object`) ni interfaces en común. Gracias al mecanismo de reflexión, `CommunicateReflectively.perform()` puede establecer dinámicamente si los métodos deseados están disponibles e invocarlos. Incluso es capaz de gestionar el hecho de que `Mime` sólo tiene uno de los métodos necesarios, cumpliendo parcialmente con su objetivo.

Aplicación de un método a una secuencia

El mecanismo de reflexión proporciona algunas posibilidades interesantes, pero relegan todas las comprobaciones de tipos a tiempo de ejecución, por lo que resulta indeseable en muchas situaciones. Normalmente, siempre es preferible conseguir que las comprobaciones de tipos se realicen en tiempo de compilación. Pero, ¿es posible tener una comprobación de tipos en tiempo de compilación y tipos latentes?

Examinemos un ejemplo donde se analiza este problema. Suponga que desea crear un método `apply()` que pueda aplicar cualquier método a todos los objetos de una secuencia. Ésta es una situación en la que las interfaces parecen no encajar. Lo que queremos es aplicar cualquier método a una colección de objetos, y las interfaces introducen demasiadas restricciones como para poder describir el concepto de "cualquier método". ¿Cómo podemos hacer esto en Java?

Inicialmente, podemos resolver el problema mediante el mecanismo de reflexión, que resulta ser bastante elegante gracias a los `varargs` de Java SE5:

```
//: generics/Apply.java
// {main: ApplyTest}
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Apply {
    public static <T, S extends Iterable<? extends T>>
        void apply(S seq, Method f, Object... args) {
        try {
            for(T t: seq)
                f.invoke(t, args);
        } catch(Exception e) {
            // Los fallos son errores del programador
            throw new RuntimeException(e);
        }
    }
}

class Shape {
    public void rotate() { print(this + " rotate"); }
    public void resize(int newSize) {
        print(this + " resize " + newSize);
    }
}

class Square extends Shape {}

class FilledList<T> extends ArrayList<T> {
    public FilledList(Class<? extends T> type, int size) {
        try {
            for(int i = 0; i < size; i++)
                // Presupone un constructor predeterminado:
                add(type.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class ApplyTest {
    public static void main(String[] args) throws Exception {
        List<Shape> shapes = new ArrayList<Shape>();
        for(int i = 0; i < 10; i++)
            shapes.add(new Shape());
        Apply.apply(shapes, Shape.class.getMethod("rotate"));
        Apply.apply(shapes,
            Shape.class.getMethod("resize", int.class), 5);
        List<Square> squares = new ArrayList<Square>();
        for(int i = 0; i < 10; i++)
            squares.add(new Square());
        Apply.apply(squares, Shape.class.getMethod("rotate"));
    }
}
```

```

        Apply.apply(squares,
            Shape.class.getMethod("resize", int.class), 5);
        Apply.apply(new FilledList<Shape>(Shape.class, 10),
            Shape.class.getMethod("rotate"));
        Apply.apply(new FilledList<Shape>(Square.class, 10),
            Shape.class.getMethod("rotate"));

        SimpleQueue<Shape> shapeQ = new SimpleQueue<Shape>();
        for(int i = 0; i < 5; i++) {
            shapeQ.add(new Shape());
            shapeQ.add(new Square());
        }
        Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
    }
} /* (Execute to see output) */:-
```

En **Apply**, tenemos suerte, porque se da la circunstancia de que Java incorpora una interfaz **Iterable** que es utilizada por la biblioteca de contenedores de Java. Debido a esto, el método **apply()** puede aceptar cualquier cosa que implemente la interfaz **Iterable**, lo que incluye todas las clases **Collection**, como **List**. Pero también puede aceptar cualquier otra cosa, siempre y cuando hagamos esa cosa de tipo **Iterable**: por ejemplo, la clase **SimpleQueue** definida a continuación y que se utiliza en el ejemplo anterior en **main()**:

```

//: generics/SimpleQueue.java
// Un tipo diferente de contenedor que es Iterable
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void add(T t) { storage.offer(t); }
    public T get() { return storage.poll(); }
    public Iterator<T> iterator() {
        return storage.iterator();
    }
} /*:-
```

En **Apply.java**, las excepciones se convierten a **RuntimeException** porque no hay mucha posibilidad de recuperarse de las excepciones, en este caso, representan realmente errores del programador.

Observe que hemos tenido que incluir límites y comodines para poder utilizar **Apply** y **FilledList** en todas las situaciones deseadas. Pruebe a experimentar quitando los límites comodines y descubrirá que **Apply** y **FilledList** no funcionan en algunas situaciones.

FilledList representa un cierto dilema. Para poder utilizar un tipo, éste debe disponer de un constructor predeterminado (sin argumentos). Java no tiene ninguna forma de descubrir eso en tiempo de compilación, así que el problema se pasa a tiempo de ejecución. Una sugerencia muy común para poder garantizar la comprobación de tipos en tiempo de ejecución consiste en definir una interfaz factoría que disponga de un método que genere objetos, entonces **FilledList** aceptaría dicha interfaz en lugar de la “factoría normal” correspondiente al tipo especificado. El problema con esto es que todas las clases que se utilicen en **FilledList** deben entonces implementar esa interfaz factoría. Y el caso es que la mayoría de las clases se crean sin tener conocimiento de nuestra interfaz, por lo que no pueden implementarla. Posteriormente veremos una posible solución utilizando adaptadores.

Pero la técnica utilizada, consistente en emplear un indicador de tipo, constituye probablemente un compromiso razonable (al menos como primera solución). Con esta técnica, utilizar algo como **FilledList** es lo suficientemente fácil como para que el programador se sienta tentado de utilizarlo, en lugar de ignorarlo. Por supuesto, dado que los errores se descubren en tiempo de ejecución, será necesario preocuparse de que dichos errores aparezcan lo antes posible durante el proceso de desarrollo.

Observe que esta técnica basada en un indicador de tipos es la que se recomienda en diversos libros y artículos, como por ejemplo el artículo *Generics in the Java Programming Language*, de Gilad Bracha⁹. En dicho artículo, el autor señala que:

⁹ Véase la cita al final de este capítulo.

"Se trata de una sintaxis que se utiliza intensivamente en las nuevas API para manipulación de anotaciones, por ejemplo". Sin embargo, en mi opinión, no todo el mundo se siente igual de cómodo al utilizar esta técnica; algunas personas prefieren emplear el método de la factoría que fue presentado anteriormente en este capítulo.

Asimismo, aunque la solución de Java resulta bastante elegante, debemos observar que el uso del mecanismo de reflexión (aunque se ha mejorado significativamente en las últimas versiones de Java) puede hacer que el programa sea más lento que las implementaciones no basadas en la reflexión, ya que hay demasiadas tareas que llevar a cabo en tiempo de ejecución. Esto no debería impedir que empleáramos esta solución, al menos como primera solución al problema (salvo que queramos caer en el error de la optimización prematura), pero representa ciertamente una diferencia entre las dos técnicas.

Ejercicio 40: (3) Añada un método `speak()` a todas las clases de `typeinfo.pets`. Modifique `Apply.java` para invocar al método `speak()` para una colección heterogénea de objetos Pet.

¿Qué pasa cuando no disponemos de la interfaz correcta?

El ejemplo anterior aprovechaba el hecho de que la interfaz `Iterable` ya está disponible, siendo esa interfaz precisamente lo que necesitábamos. Pero ¿qué sucede en el caso general, cuando no existe todavía una interfaz que se ajuste a nuestras necesidades?

Por ejemplo, vamos a generalizar la idea de `FilledList` y a crear un método `fill()` parametrizado que admita una secuencia y la rellene utilizando un objeto `Generator`. Al tratar de escribir esto en Java nos encontramos con un problema, porque no existe ninguna interfaz adecuada "`Addable`" (una interfaz que permita añadir objetos), mientras que en el caso anterior sí que teníamos una interfaz `Iterable`. Por tanto, en lugar de decir "cualquier cosa para la cual podamos invocar el método `add()`", nos vemos forzados a decir "un subtipo de `Collection`". El código resultante no es particularmente genérico, ya que debe restringirse para funcionar con implementaciones de `Collection`. Si tratamos de utilizar una clase que no implemente `Collection`, el código genérico no funcionará. He aquí el aspecto que tendría este ejemplo:

```
//: generica/Fill.java
// Generalización de la idea de FilledList
// {main: FillTest}
import java.util.*;

// No funciona con "cualquier cosa que tenga un método add()". No hay
// una interfaz "Addable", por lo que nos vemos limitados a
// utilizar un contenedor Collection. No podemos generalizar
// empleando genéricos en este caso.

public class Fill {
    public static <T> void fill(Collection<T> collection,
        Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            // Presupone un constructor predeterminado:
            try {
                collection.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
}

class Contract {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getName() + " " + id;
    }
}
class TitleTransfer extends Contract {}
```

```

class FillTest {
    public static void main(String[] args) {
        List<Contract> contracts = new ArrayList<Contract>();
        Fill.fill(contracts, Contract.class, 3);
        Fill.fill(contracts, TitleTransfer.class, 2);
        for(Contract c: contracts)
            System.out.println(c);
        SimpleQueue<Contract> contractQueue =
            new SimpleQueue<Contract>();
        // No funciona. fill() no es lo suficientemente genérico:
        // Fill.fill(contractQueue, Contract.class, 3);
    }
} /* Output:
Contract 0
Contract 1
Contract 2
TitleTransfer 3
TitleTransfer 4
*///:-
```

Es en estas situaciones donde resulta ventajoso disponer de un mecanismo parametrizado con tipos latentes, porque de esa forma no estaremos a merced de las decisiones de diseño que hubiera tomado en el pasado cualquier diseñador concreto de bibliotecas; gracias a eso no tendremos que reescribir nuestro código cada vez que nos encontramos con una biblioteca que no hubiera tenido en cuenta nuestra situación concreta (así que el código será verdaderamente “genérico”). En el caso anterior, como los diseñadores de Java no vieron la necesidad (lo cual resulta bastante natural) de agregar una interfaz “**Addable**”, estamos obligados a movernos dentro de la jerarquía **Collection**, y **SimpleQueue** no funcionará, aún cuando disponga de un método **add()**. Dado que ahora está restringido a trabajar con **Collection**, el código no es particularmente “genérico”. Con los tipos latentes este problema no se presentaría.

Simulación de tipos latentes mediante adaptadores

De modo que los genéricos de Java no disponen de tipos latentes y necesitamos algo como los tipos latentes para poder escribir código que pueda aplicarse traspasando las fronteras entre las clases (es decir, código “genérico”). ¿Hay alguna forma de salvar esta limitación?

¿Qué es lo que no permitiría hacer los tipos latentes? Los tipos latentes implicarían que podríamos escribir código que dijera: “No me importa qué tipo estoy usando, siempre y cuando ese tipo disponga de estos métodos”. En la práctica, los tipos latentes crean una *interfaz implícita* que contiene los métodos deseados. Por tanto, si escribimos la interfaz necesaria a mano (ya que Java no lo hace por nosotros), eso debería resolver el problema.

Escribir código para obtener una interfaz que necesitamos a partir de otra interfaz de la que disponemos constituye un ejemplo del patrón de diseño Adaptador. Podemos utilizar adaptadores para adaptar las clases existentes con el fin de producir la interfaz deseada, utilizando para ello una cantidad de código relativamente pequeña. La solución, que utiliza la jerarquía **Coffee** anteriormente definida, ilustra las diferentes formas de escribir adaptadores:

```

//: generics/Fill2.java
// Utilización de adaptadores para simular tipos latentes.
// {main: Fill2Test}
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface Addable<T> { void add(T t); }

public class Fill2 {
    // Versión con indicador de clase:
    public static <T> void fill(Addable<T> addable,
        Class<? extends T> classToken, int size) {
```

```

for(int i = 0; i < size; i++)
    try {
        addable.add(classToken.newInstance());
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
}
// Versión con generador:
public static <T> void fill(Addable<T> addable,
Generator<T> generator, int size) {
    for(int i = 0; i < size; i++)
        addable.add(generator.next());
}

// Para adaptar un tipo base, es necesario utilizar composición.
// Definir Addable como contenedor Collection usando composición:
class AddableCollectionAdapter<T> implements Addable<T> {
    private Collection<T> c;
    public AddableCollectionAdapter(Collection<T> c) {
        this.c = c;
    }
    public void add(T item) { c.add(item); }
}

// Un método auxiliar para capturar el tipo automáticamente:
class Adapter {
    public static <T>
    Addable<T> collectionAdapter(Collection<T> c) {
        return new AddableCollectionAdapter<T>(c);
    }
}

// Para adaptar un tipo específico, podemos usar la herencia.
// Hacer Addable un contenedor SimpleQueue utilizando la herencia:
class AddableSimpleQueue<T>
extends SimpleQueue<T> implements Addable<T> {
    public void add(T item) { super.add(item); }
}

class Fill2Test {
    public static void main(String[] args) {
        // Adaptar una colección:
        List<Coffee> carrier = new ArrayList<Coffee>();
        Fill2.fill(
            new AddableCollectionAdapter<Coffee>(carrier),
            Coffee.class, 3);
        // El método auxiliar captura el tipo:
        Fill2.fill(Adapter.collectionAdapter(carrier),
            Latte.class, 2);
        for(Coffee c: carrier)
            print(c);
        print("-----");
        // Utilizar una clase adaptada:
        AddableSimpleQueue<Coffee> coffeeQueue =
            new AddableSimpleQueue<Coffee>();
        Fill2.fill(coffeeQueue, Mocha.class, 4);
        Fill2.fill(coffeeQueue, Latte.class, 1);
        for(Coffee c: coffeeQueue)
            print(c);
    }
}

```

```

    }
} /* Output:
Coffee 0
Coffee 1
Coffee 2
Latte 3
Latte 4
-----
Mocha 5
Mocha 6
Mocha 7
Mocha 8
Latte 9
*///:-
```

Fill2 no requiere un objeto **Collection** a diferencia de **Fill**. En lugar de ello, sólo necesita algo que implemente **Addable**, y **Addable** ha sido escrita precisamente para **Fill**, este ejemplo es una manifestación del tipo latente que queríamos que el compilador construyera por nosotros.

En esta versión, también hemos añadido un método **fill()** sobrecargado que toma un objeto **Generator** en lugar de un indicador de tipo. El objeto **Generator** no presenta problemas de seguridad de tipos en tiempo de compilación: el compilador garantiza que lo que pasemos sea un objeto **Generator** válido, así que no puede generarse ninguna excepción.

El primer adaptador, **AddableCollectionAdapter**, funciona con el tipo base **Collection**, lo que significa que puede utilizarse cualquier implementación de **Collection**. Esta versión simplemente almacena la referencia a **Collection** y la utiliza para implementar **add()**.

Si disponemos de un tipo específico en lugar de disponer de la clase base de una jerarquía, podemos escribir algo menos de código a la hora de crear el adaptador empleando el mecanismo de herencia, como puede verse en **AddableSimpleQueue**.

En **Fill2Test.main()**, podemos ver cómo funcionan los diversos tipos de adaptadores. En primer lugar, se adapta un tipo **Collection** con **AddableCollectionAdapter**. Una segunda versión de esto utiliza el método auxiliar genérico, y podemos ver cómo el método genérico captura el tipo, así que no es necesario escribirlo explícitamente; se trata de un truco bastante conveniente, que nos permite escribir código más elegante.

A continuación, se utiliza la clase **AddableSimpleQueue** pre-adaptada. Observe que en ambos casos los adaptadores permiten utilizar con **Fill2.fill()** las clases que previamente no implementaba **Addable**.

La utilización de adaptadores de esta forma parece compensar la falta de un mecanismo de tipos latentes, por lo que podría pensarse que podemos escribir código genuinamente genérico. Sin embargo, se trata de un paso de programación adicional y es necesario que lo comprendan tanto el creador de la biblioteca como el consumidor de la misma; y este concepto puede no ser entendido fácilmente por los programadores menos expertos. Los verdaderos mecanismos de tipos latentes permiten, al eliminar este paso adicional, aplicar el código genérico más fácilmente, y ahí radica precisamente su valor.

Ejercicio 41: (1) Modifique **Fill2.java** para utilizar las clases **typeinfo.pets** en lugar de las clases **Coffee**.

Utilizando los objetos de función como estrategias

Este ejemplo final nos permitirá código verdaderamente genérico utilizando la técnica de adaptadores descrita en la sección anterior. El ejemplo comenzó con un intento de crear una suma de una secuencia de elementos (de cualquier tipo que pueda sumarse), pero terminó evolucionando hacia la realización de operaciones generales, usando un estilo de programación *funcional*.

Si nos fijamos exclusivamente en el proceso de sumar objetos, podemos ver que este es un caso en el que tenemos operaciones comunes entre clases, pero dichas operaciones no están representadas en ninguna clase base que podamos especificar: en ocasiones se puede incluso utilizar un operador "+" y otras veces puede haber algún tipo de método "suma". Ésta es, generalmente la situación con la que nos encontramos cuando tratamos de escribir código genérico, porque lo que queremos es que el código se pueda aplicar a múltiples clases; especialmente, como en este caso, múltiples clases que ya existan y que no tengamos posibilidad de "corregir". Incluso si restringiéramos este ejemplo a las subclases de **Number**, dicha superclase no incluye nada acerca de la "sumabilidad".

La solución consiste en utilizar el patrón de diseño de *Estrategia*, que permite obtener código más elegante porque aisla completamente "las cosas que cambian" dentro de un *objeto de función*¹⁰. Un objeto de función es un objeto que se comporta, en cierta manera, como una función: normalmente, existe un método de interés (en los lenguajes que soportan el mecanismo de sobrecarga de operadores, podemos hacer que la llamada a este método parezca una llamada a método normal). El valor de los objetos de función es que, a diferencia de un método normal, los podemos pasar de un sitio a otro y también pueden tener un estado que persista entre sucesivas llamadas. Por supuesto, podemos conseguir algo como esto con cualquier método de una clase, pero (al igual que con cualquier patrón de diseño) el objeto de función se distingue principalmente por su intención original. Aquí, la intención es crear algo que se comporte como un único método que podemos pasar de un sitio a otro; por tanto, está estrechamente acoplado (y en ocasiones es indistinguible de él), con el patrón de diseño de *Estrategia*.

De acuerdo con mi experiencia con distintos patrones de diseño, las fronteras son un tanto difusas en este caso: lo que vamos a hacer es crear objetos de función que realicen una cierta adaptación, y esos objetos se van a pasar a una serie de métodos para utilizarlos como estrategias.

Adoptando este enfoque, en este ejemplo se añaden los diversos tipos de métodos genéricos que originalmente queríamos crear, así como algunos otros. He aquí el resultado:

```
//: generics/Functional.java
import java.math.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Distintos tipos de objetos de función:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
interface Collector<T> extends UnaryFunction<T,T> {
    T result(); // Extraer resultado del parámetro de recopilación
}
interface UnaryPredicate<T> { boolean test(T x); }

public class Functional {
    // Invoca al objeto Combiner para cada elemento con el fin de
    // combinarlo con un resultado dinámico, devolviéndose
    // al final el resultante:
    public static <T> T
    reduce(Iterable<T> seq, Combiner<T> combiner) {
        Iterator<T> it = seq.iterator();
        if(it.hasNext()) {
            T result = it.next();
            while(it.hasNext())
                result = combiner.combine(result, it.next());
            return result;
        }
        // Si seq es la lista vacía:
        return null; // O generar una excepción
    }
    // Tomar un objeto de función e invocarlo para cada objeto de
    // la lista, ignorando el valor de retorno. El objeto de función
    // puede actuar como un parámetro de recopilación, así que
    // se lo devuelve al final.
    public static <T> Collector<T>
    forEach(Iterable<T> seq, Collector<T> func) {
        for(T t : seq)
            func.function(t);
    }
}
```

¹⁰ En ocasiones, podrás ver que a estos objetos se los denomina *functores*. En este texto, utilizaremos el término *objeto de función* en lugar de *functor*, ya que el término "functor" tiene un significado diferente y muy específico en matemáticas.

```

        return func;
    }
    // Crea una lista de resultados invocando un objeto
    // de función para cada objeto de la lista:
    public static <R,T> List<R>
    transform(Iterable<T> seq, UnaryFunction<R,T> func) {
        List<R> result = new ArrayList<R>();
        for(T t : seq)
            result.add(func.function(t));
        return result;
    }

    // Aplica un predicado unario a cada elemento de una secuencia y devuelve
    // una lista con los elementos que han dado como resultado "true":
    public static <T> List<T>
    filter(Iterable<T> seq, UnaryPredicate<T> pred) {
        List<T> result = new ArrayList<T>();
        for(T t : seq)
            if(pred.test(t))
                result.add(t);
        return result;
    }

    // Para utilizar los anteriores métodos genéricos, necesitamos crear
    // objetos de función para adaptarlos a nuestras necesidades concretas:
    static class IntegerAdder implements Combiner<Integer> {
        public Integer combine(Integer x, Integer y) {
            return x + y;
        }
    }
    static class
    IntegerSubtracter implements Combiner<Integer> {
        public Integer combine(Integer x, Integer y) {
            return x - y;
        }
    }
    static class
    BigDecimalAdder implements Combiner<BigDecimal> {
        public BigDecimal combine(BigDecimal x, BigDecimal y) {
            return x.add(y);
        }
    }
    static class
    BigIntegerAdder implements Combiner<BigInteger> {
        public BigInteger combine(BigInteger x, BigInteger y) {
            return x.add(y);
        }
    }
    static class
    AtomicLongAdder implements Combiner<AtomicLong> {
        public AtomicLong combine(AtomicLong x, AtomicLong y) {
            // No está claro si esto es significativo:
            return new AtomicLong(x.addAndGet(y.get()));
        }
    }
    // Podemos incluso hacer una función unaria con un "ulp"
    // (Units in the last place, las unidades en el último lugar):
    static class BigDecimalUlp
    implements UnaryFunction<BigDecimal,BigDecimal> {

```

```

public BigDecimal function(BigDecimal x) {
    return x.ulp();
}
}
static class GreaterThan<T extends Comparable<T>>
implements UnaryPredicate<T> {
    private T bound;
    public GreaterThan(T bound) { this.bound = bound; }
    public boolean test(T x) {
        return x.compareTo(bound) > 0;
    }
}
static class MultiplyingIntegerCollector
implements Collector<Integer> {
    private Integer val = 1;
    public Integer function(Integer x) {
        val *= x;
        return val;
    }
    public Integer result() { return val; }
}
public static void main(String[] args) {
    // Genéricos, varargs y conversión automática funcionando conjuntamente:
    List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
    Integer result = reduce(li, new IntegerAdder());
    print(result);

    result = reduce(li, new IntegerSubtracter());
    print(result);

    print(filter(li, new GreaterThan<Integer>(4)));

    print(forEach(li,
        new MultiplyingIntegerCollector()).result());

    print(forEach(filter(li, new GreaterThan<Integer>(4)),
        new MultiplyingIntegerCollector()).result());

    MathContext mc = new MathContext(7);
    List<BigDecimal> lbd = Arrays.asList(
        new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
        new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
    BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
    print(rbd);

    print(filter(lbd,
        new GreaterThan<BigDecimal>(new BigDecimal(3))));

    // Utilizar la funcionalidad de generación de primos de BigInteger:
    List<BigInteger> lbi = new ArrayList<BigInteger>();
    BigInteger bi = BigInteger.valueOf(11);
    for(int i = 0; i < 11; i++) {
        lbi.add(bi);
        bi = bi.nextProbablePrime();
    }
    print(lbi);

    BigInteger rbi = reduce(lbi, new BigIntegerAdder());
    print(rbi);
}

```

```

// La suma de esta lista de primos también es prima:
print(rbi.isProbablePrime(5));
List<AtomicLong> lal = Arrays.asList(
    new AtomicLong(11), new AtomicLong(47),
    new AtomicLong(74), new AtomicLong(133));
AtomicLong ral = reduce(lal, new AtomicLongAdder());
print(ral);

print(transform(lbd,new BigDecimalUlp()));
}
} /* Output:
28
-26
[5, 6, 7]
5040
210
11.000000
[3.300000, 4.400000]
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
311
true
265
[0.000001, 0.000001, 0.000001, 0.000001]
*///:-

```

El ejemplo comienza definiendo interfaces para distintos tipos de objetos de función. Estas interfaces se han creado a medida que eran necesarias mientras se desarrollaban los diferentes métodos y se descubría la necesidad de cada una. La clase **Combiner** me fue sugerida por un contribuidor anónimo a uno de los artículos que publiqué en mi sitio web. **Combiner** abstrae los detalles específicos relativos a tratar de sumar dos objetos y se limita a enunciar que esos objetos están siendo combinados de alguna manera. Como resultado, podemos ver que **IntegerAdder** y **IntegerSubtracter** pueden ser tipos de **Combiner**.

Una función unario (**UnaryFunction**) toma un único argumento y produce un resultado, el argumento y el resultado no tienen por qué ser del mismo tipo. Se utiliza un elemento **Collector** como “parámetro de recopilación” y podemos extraer el resultado una vez que hayamos acabado. Un predicado **UnaryPredicate** produce un resultado de tipo **boolean**. Hay otros tipos de objetos de función que pueden definirse, pero estos son suficientes para entender el concepto.

La clase **Functional** contiene una serie de métodos genéricos que aplican objetos de función a secuencia. El método **reduce()** aplica la función contenida en un objeto **Combiner** a cada elemento de una secuencia con el fin de producir un único resultado.

forEach() toma un objeto **Collector** y aplica su función a cada elemento, ignorando el resultado de cada llamada a función. Podemos invocar este método simplemente debido a su efecto colateral (lo que no encajaría con un estilo de programación “funcional” pero puede, a pesar de todo, ser útil), o bien el objeto **Collector** puede mantener el estado interno para actuar como un parámetro de recopilación, como es el caso en este ejemplo.

transform() genera una lista invocando una función **UnaryFunction** sobre cada objeto de la secuencia y capturando el resultado.

Finalmente, **filter()** aplica un predicado **UnaryPredicate** a cada objeto de una secuencia y almacena los objetos que producen **true** en un contenedor **List**, que luego se devuelve.

Podemos definir funciones genéricas adicionales. La biblioteca estándar de plantillas de C++, por ejemplo, dispone de multitud de ellas. El problema también se ha resuelto con unas bibliotecas de código abierto, como por ejemplo JGA (*Generic Algorithms for Java*).

En C++, el mecanismo de tipos latentes se encarga de establecer la correspondencia entre las operaciones cuando se invocan las funciones, pero en Java necesitamos escribir los objetos de función para adaptar los métodos genéricos a nuestras necesidades concretas. Por tanto, la siguiente parte de la clase muestra diferentes implementaciones de los objetos de función. Observe, por ejemplo, que **IntegerAdder** y **BigDecimalAdder** resuelven el mismo problema, (sumar dos objetos),

invocando las operaciones apropiadas para su tipo concreto. Éste es un ejemplo de combinación de los patrones de diseño Adaptador y Estrategia.

En `main()`, podemos ver que en cada llamada a método se pasa una secuencia junto con el objeto de función apropiado. Asimismo, vemos que hay expresiones que pueden llegar a ser bastante complejas, como por ejemplo:

```
forEach(filter(li, new GreaterThan(4)),
        new MultiplyingIntegerCollector()).result()
```

Esta expresión genera una lista seleccionando todos los elementos de `li` que sean mayores que 4, y luego aplica el método `MultiplyingIntegerCollector()` a la lista resultante y extrae el resultado con `result()`. No vamos a explicar los detalles del resto del código, aunque el lector no debería tener problemas en comprenderlo sin más que analizarlo.

Ejercicio 42: (5) Cree dos clases separadas, que no tengan nada en común. Cada clase debe almacenar un valor y disponer al menos de métodos que produzcan dicho valor y permitan realizar una modificación del mismo. Modifique `Functional.java` para que realice operaciones funcionales sobre colecciones de dichas dos clases (estas operaciones no tienen que ser aritméticas como son las de `Functional.java`).

Resumen: ¿realmente es tan malo el mecanismo de proyección?

Habiendo estado trabajando en explicar las plantillas de C++ desde que éstas fueron concebidas, probablemente yo haya estado haciendo la pregunta que da título a esta sección durante más tiempo que la mayoría de los demás autores. Pero sólo recientemente me he detenido realmente a pensar hasta qué punto esta pregunta es válida en muchas situaciones: ¿cuántas veces merece la pena complicar las cosas para el problema que estamos tratando de describir?

Podríamos argumentar de la forma siguiente. Uno de los lugares más obvios para utilizar un mecanismo de tipos genéricos es con clases contenedores tales como `List`, `Set`, `Map`, etc., es decir con las clases que ya hemos visto en el Capítulo 11, *Almacenamiento de objetos*, y de lo que hablaremos más en detalle en el Capítulo 17, *Análisis detallado de los contenedores*. Antes de Java SE5, cuando incluíamos un objeto en un contenedor, éste se generalizaba a `Object`, perdiéndose así la información de tipos. Cuando se quería extraerlo de nuevo para hacer algo con él, era necesario volver a proyectarlo sobre el tipo apropiado. Un ejemplo sería una lista `List` de objetos `Cat` (gatos). Si la versión genérica de los contenedores introducida en Java SE5, lo que haríamos sería introducir objetos de tipo `Object` y extraer objetos de tipo `Object`, con lo cual resulta perfectamente posible insertar un perro (`Dog`) en una lista de objetos `Cat`.

Sin embargo, lo que las versiones de Java anteriores a la aparición de genéricos no nos permitían era *mal utilizar* los objetos introducidos en un contenedor. Si introducimos un objeto `Dog` en un contenedor donde estamos almacenando objetos `Cat` y luego intentamos tratar todo lo que hay en el contenedor como si fuera un objeto `Cat`, se obtiene una excepción `RuntimeException` al extraer la referencia `Dog` del contenedor `Cat` y tratar de proyectarla sobre `Cat`. Así pues, el problema termina por descubrirse; la única desventaja es que se descubra el problema en tiempo de ejecución en lugar de en tiempo de compilación.

En las ediciones anteriores del libro, yo decía que:

Esto no es sólo una molestia. En ocasiones, puede dar lugar a errores de programación difíciles de detectar. Si una parte de un programa (o varias partes de un programa) inserta objetos en un contenedor y lo único que podemos descubrir en una parte separada del programa, por la generación de una excepción, es que se ha introducido un objeto incorrecto dentro del contenedor, entonces nos vemos obligados a averiguar en qué punto se ha producido la inserción incorrecta.

Sin embargo, después de examinar de nuevo la cuestión, comencé a pensar en ella. En primer lugar, ¿con qué frecuencia se produce este problema? Personalmente, no recuerdo que nunca me haya pasado algo así y, cuando he preguntado a la gente que asistía a mis conferencias, tampoco he logrado encontrar a nadie que me dijera que a ellos le había pasado. En otro libro sobre el tema, se incluía un ejemplo de una lista denominada `files` que contenía objetos `String`; en este ejemplo, parecía completamente natural añadir un objeto `File` (archivo) a `files`, por lo que habría sido mejor denominar al objeto `fileNames` (nombres de archivo). Independientemente de lo exhaustivos que sean los mecanismos de comprobación de tipos de Java, sigue siendo posible escribir programas enrevesados, y el hecho de que un programa mal escrito se pueda compilar no quiere decir que deje de ser un programa mal escrito. Quizá la mayoría de los programadores utilicen contenedores con nombres apro-

piados como “**cats**” que proporcionan una advertencia visual al programador que esté intentado añadir un objeto que no sea del tipo **Cat**. E incluso si llegara a darse el caso de que alguien introduzca el objeto incorrecto, ¿durante cuánto tiempo permanecería oculto ese problema antes de ser descubierto? Parece lógico pensar que, tan pronto como empezáramos a ejecutar pruebas con datos reales, se generaría rápidamente una excepción.

Un determinado autor ha llegado a decir que dicho problema podría “permanecer oculto durante años”, pero yo no he podido encontrar informes que hablen de personas que tengan grandes dificultades para encontrar errores del tipo “perro en una lista de gatos”, ni tampoco he encontrado informes donde se diga que ese problema se produce muy a menudo. Mientras que con las hebras de programación, como veremos en el Capítulo 21, *Concurrencia*, resulta bastante sencillo y común que haya errores que sólo se manifiesten de forma muy infrecuente, y que sólo nos proporcionan una vaga indicación de qué es lo que anda mal, en este otro ejemplo de la inserción de objetos erróneos, las cosas no son así. Por tanto, ¿es el problema de la inserción de objetos erróneos la razón de que todas estas funcionalidades tan significativas y complejas hayan sido añadidas a Java?

En mi opinión, la *intención* de esa funcionalidad del lenguaje de propósito general denominada “genéricos” (no necesariamente de la implementación concreta que Java hace) es la *expresividad*, no simplemente la creación de contenedores que sean seguros en lo que respecta a los tipos de datos. La posibilidad de disponer de contenedores que sean seguros respecto a los tipos de datos se obtiene como efecto colateral de la capacidad de crear código que tenga un propósito más general.

Por tanto, aunque el argumento de la inserción de tipos incorrectos en una lista se utiliza a menudo para justificar la existencia de los genéricos, dicho argumento resulta cuestionable. Y, como decíamos al principio del capítulo, no creo que el *concepto* de genéricos tenga nada que ver con ese problema. Por el contrario, los genéricos, como su propio nombre indica, constituyen una forma de escribir código más “genérico” y que esté menos restringido por los tipos de datos con los que pueda trabajar, de manera que un mismo fragmento de código pueda aplicarse a una mayor cantidad de tipos de datos. Como hemos visto en este capítulo, resulta fácil escribir clases “contenedoras” verdaderamente genéricas (es decir, lo que son los contenedores de Java), pero escribir código que manipule sus tipos genéricos requiere un esfuerzo adicional tanto por parte del creador de la clase, como por parte del consumidor de la misma, que debe comprender el concepto y la implementación del patrón de diseño Adaptador. Dicho esfuerzo adicional reduce la facilidad de uso de esta funcionalidad y puede, por tanto, hacer que sea menos aplicable en diversos lugares en los que podría sin embargo representar un valor añadido.

Observe también que como los genéricos fueron introducidos de manera bastante tardía en Java en lugar de haber sido incluidos en el lenguaje desde el principio, algunos de los contenedores no pueden ser tan robustos como deberían. Por ejemplo, fíjese en **Map**, y en particular en los métodos **containsKey(Object key)** y **get(Object key)**. Si estas clases hubieran sido diseñadas con genéricos previamente existentes, estos métodos hubieran utilizado tipos parametrizados en lugar de **Object**, permitiendo así las comprobaciones en tiempo de compilación que se supone que los genéricos deben proporcionar. En los mapas de C++ por ejemplo, el tipo de la clave se comprueba siempre en tiempo de compilación.

Hay una cosa muy clara: introducir cualquier tipo de mecanismo genérico en una versión posterior del lenguaje después de que ese lenguaje haya llegado a ser de uso general, conduce a situaciones extremadamente liosas, y es imposible cumplir el objetivo sin un esfuerzo enorme. En C++, las plantillas fueron introducidas en la versión ISO inicial del lenguaje (aunque incluso eso fue causa de cierta confusión, porque ya se estaba usando una versión anterior, sin plantillas, antes de que el primer estándar de C++ apareciese), por lo que, en la práctica, las plantillas fueron *siempre* una parte del lenguaje. En Java, los genéricos no se introdujeron hasta casi 10 años después de que el lenguaje empezara a utilizarse, por lo que los problemas con la migración hacia los genéricos son considerables y han tenido un impacto significativo en el diseño del propio mecanismo de genéricos. El resultado es que nosotros, los programadores, tenemos que sufrir ahora las consecuencias derivadas de la falta de visión de los diseñadores de Java que crearon la versión 1.0. Cuando Java se diseñó originalmente, los diseñadores tenían conocimiento, por supuesto, acerca de las plantillas C++. E incluso consideraron incluirlas en el lenguaje, pero por alguna razón decidieron dejarlas fuera (lo que probablemente indica que tenían prisa por terminar el diseño). Como resultado, tanto el lenguaje como los programadores que lo emplean tienen que enfrentarse con una serie de problemas derivados de esa omisión. Sólo el tiempo nos dirá cuál es el impacto final sobre el lenguaje que tendrá la solución que Java ha adoptado para el tema de los genéricos.

Algunos lenguajes, y en especial *Nice* (véase <http://nice.sourceforge.net>; este lenguaje genera código intermedio Java y funciona con las bibliotecas Java existentes) y *NextGen* (véase <http://japan.cs.rice.edu/nextgen>) han incorporado otras soluciones más limpias y menos problemáticas para el tema de los tipos parametrizados. No resulta posible imaginar que uno de estos lenguajes llegue a ser el sucesor de Java, porque ambos han adoptado el mismo enfoque exacto que C++ adoptó con respecto a C: usar aquello que estaba disponible y mejorarlo.

Lecturas adicionales

El documento de carácter introductorio para los genéricos es *Generics in the Java Programming Language*, de Gilad Bracha, que puede encontrar en <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>.

Java Generics FAQs de Angelika Langer es un recurso muy útil. Puede encontrarlo en www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html.

Puede ver detalles acerca de los comodines en *Adding Wildcards to the Java Programming Language*, de Torgerson, Ernst, Hansen, von der Ahe, Bracha y Gafter, que podrá encontrar en www.jot.fm/issues/issue_2004_12/article5.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Matrices

16

Al final del Capítulo 5, *Inicialización y limpieza*, vimos cómo definir e inicializar una matriz.

Podríamos tratar de describir de manera simple las matrices diciendo que lo que hacemos es crearlas y rellenarlas, luego seleccionar elementos de las mismas utilizando índices enteros y diciendo, además, que las matrices no cambian de tamaño. La mayoría de las veces, eso es todo lo que necesitamos saber pero en ocasiones hay que realizar operaciones más sofisticadas con las matrices, y también puede que tengamos que comparar la utilización de una matriz con la de otros contenedores más flexibles. En este capítulo veremos cómo analizar las matrices con un mayor detalle.

Por qué las matrices son especiales

Existen diferentes maneras de almacenar objetos, así que ¿qué es lo que hace que las matrices sean especiales?

Existen tres aspectos que distinguen a las matrices de otros tipos de contenedores: la eficiencia, el tipo y la capacidad de almacenar primitivas. La matriz es la forma más eficiente en Java para almacenar una secuencia de referencias a objetos y para acceder aleatoriamente a ellas. Una matriz es una secuencia lineal simple, lo que hace que el acceso a los elementos sea rápido. El coste que hay que pagar por esta velocidad es que el tamaño de un objeto matriz es fijo y no puede cambiarse a lo largo de la vida de la matriz. Podríamos pensar, como alternativa, en utilizar un contenedor de tipo **ArrayList** (consulte el Capítulo 11, *Almacenamiento de objetos*), que se encargará de asignar automáticamente más espacio, creando un nuevo contenedor y desplazando todas las referencias desde el contenedor antiguo hasta el nuevo. Aunque generalmente resulta preferible emplear un contenedor **ArrayList** en lugar de una matriz, esta flexibilidad adicional tiene un cierto coste asociado, de manera que un contenedor **ArrayList** es perceptiblemente menos eficiente que una matriz.

Tanto las matrices como los contenedores incluyen mecanismos necesarios para garantizar que no podamos abusar de ellos. Independientemente de si estamos utilizando una matriz o un contenedor obtendremos una excepción **RuntimeException** si nos pasamos de los límites, un hecho que indica que se ha producido un error del programador.

Antes de la adopción de los genéricos, las otras clases de contenedores trataban con los objetos como si éstos no tuvieran ningún tipo específico. En otras palabras, trataban con ellos como si fueran de tipo **Object**, la clase raíz de todas las clases de Java. Las matrices son más convenientes que los contenedores anteriores a los mecanismos de genéricos, porque podemos crear una matriz para almacenar un tipo específico. Esto significa que se dispone de una comprobación de tipos en tiempo de compilación, con el fin de impedir que insertemos un objeto de tipo incorrecto o que confundamos el tipo de los objetos que estemos extrayendo. Por supuesto, Java impedirá que envíemos un mensaje inapropiado a cualquier objeto, tanto en tiempo de compilación como en tiempo de ejecución, de modo que ninguno de los dos enfoques es más arriesgado que el otro. Simplemente resulta mucho más cómodo que el compilador nos avise de los errores, y con las matrices existe una menor probabilidad de que el usuario final pueda verse sorprendido por la generación de una excepción.

Una matriz puede almacenar primitivas mientras que un contenedor de los anteriores a la adopción de mecanismo de genéricos no puede albergar primitivas. Sin embargo, con los genéricos, los contenedores tienen que especificar y comprobar el tipo de los objetos que almacenan y, gracias a los mecanismos de conversión automática, los contenedores pueden actuar como si fueran capaces de almacenar primitivas, ya que la conversión es transparente. He aquí un ejemplo donde se comparan las matrices con los contenedores genéricos:

```
//: arrays/ContainerComparison.java
import java.util.*;
import static net.mindview.util.Print.*;
```

```

class BerylliumSphere {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Sphere " + id; }
}

public class ContainerComparison {
    public static void main(String[] args) {
        BerylliumSphere[] spheres = new BerylliumSphere[10];
        for(int i = 0; i < 5; i++)
            spheres[i] = new BerylliumSphere();
        print(Arrays.toString(spheres));
        print(spheres[4]);

        List<BerylliumSphere> sphereList =
            new ArrayList<BerylliumSphere>();
        for(int i = 0; i < 5; i++)
            sphereList.add(new BerylliumSphere());
        print(sphereList);
        print(sphereList.get(4));

        int[] integers = { 0, 1, 2, 3, 4, 5 };
        print(Arrays.toString(integers));
        print(integers[4]);

        List<Integer> intList = new ArrayList<Integer>(
            Arrays.asList(0, 1, 2, 3, 4, 5));
        intList.add(97);
        print(intList);
        print(intList.get(4));
    }
} /* Output:
[Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4, null, null, null, null, null]
Sphere 4
[Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
Sphere 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*///:-
```

En ambas maneras de almacenar los objetos se comprueban los tipos de los datos y la única diferencia aparente es que las matrices utilizan [] para acceder a los elementos, mientras que un contenedor de tipo **List** utiliza métodos como **add()** y **get()**. La similitud entre las matrices y el contenedor **ArrayList** es intencionada, de manera que resulte conceptualmente fácil commutar entre ambas soluciones. Pero, como vimos en el Capítulo 11, *Almacenamiento de los objetos*, los contenedores tienen una funcionalidad mucho más rica que las matrices.

Con la aparición de los mecanismos de conversión automática, los contenedores son casi tan fáciles de utilizar con primitivas como las matrices. La única ventaja que le queda, en consecuencia, a las matrices es la eficiencia. Sin embargo, cuando lo que estemos tratando de resolver sea un problema más general, las matrices pueden ser demasiado restrictivas, y en esos casos lo que hacemos es utilizar una clase de contenedor.

Las matrices son objetos de primera clase

Independientemente del tipo de matriz con el que estemos trabajando, el identificador de la matriz es de hecho una referencia a un verdadero objeto que se crea dentro del círculo de memoria. Éste es el objeto que almacena las referencias a los otros objetos (los que están almacenados en la matriz) y puede crearse tanto implícitamente como parte de la sintaxis de ini-

cialización de la matriz, cuanto explicitamente mediante una expresión `new`. Parte del objeto matriz (de hecho, el único campo o método al que podemos acceder) es el miembro `length` (longitud) de sólo lectura que nos dice cuántos elementos pueden almacenarse en dicho objeto matriz. La sintaxis '`[]`' es la única otra forma que tenemos de acceder al objeto matriz.

El siguiente ejemplo resume las diversas formas en que puede inicializarse una matriz, y las maneras en que las referencias de matriz pueden asignarse a diferentes objetos matriz. El ejemplo también muestra que las matrices de objetos y las matrices de primitivas son casi idénticas en lo que a su uso se refiere. La única diferencia es que las matrices de objetos almacenan referencias, mientras que las matrices de primitivas almacenan directamente valores primitivos.

```
//: arrays/ArrayOptions.java
// Inicialización y reasignación de matrices.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayOptions {
    public static void main(String[] args) {
        // Matrices de objetos:
        BerylliumSphere[] a; // Variable local no inicializada
        BerylliumSphere[] b = new BerylliumSphere[5];
        // Las referencias dentro de la matriz se inicializan
        // automáticamente con null:
        print("b: " + Arrays.toString(b));
        BerylliumSphere[] c = new BerylliumSphere[4];
        for(int i = 0; i < c.length; i++)
            if(c[i] == null) // Se puede comprobar si es una referencia nula
                c[i] = new BerylliumSphere();
        // Inicialización agregada:
        BerylliumSphere[] d = { new BerylliumSphere(),
                               new BerylliumSphere(), new BerylliumSphere()
                           };
        // Inicialización agregada dinámica:
        a = new BerylliumSphere[]{ new BerylliumSphere(), new BerylliumSphere(),
                               };
        // (La coma final es opcional en ambos casos)
        print("a.length = " + a.length);
        print("b.length = " + b.length);
        print("c.length = " + c.length);
        print("d.length = " + d.length);
        a = d;
        print("a.length = " + a.length);

        // Matrices de primitivas:
        int[] e; // Referencia nula
        int[] f = new int[5];
        // Las primitivas contenidas en la matriz se
        // inicializan automáticamente con cero:
        print("f: " + Arrays.toString(f));
        int[] g = new int[4];
        for(int i = 0; i < g.length; i++)
            g[i] = i*i;
        int[] h = { 11, 47, 93 };
        // Error de compilación: variable e no inicializada:
        // print("e.length = " + e.length);
        print("f.length = " + f.length);
        print("g.length = " + g.length);
        print("h.length = " + h.length);
        e = h;
        print("e.length = " + e.length);
```

```

e = new int[]{ 1, 2 };
print("e.length = " + e.length);
}
/* Output:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*///:-
```

La matriz **a** es una variable local no inicializada y el compilador nos impide que hagamos nada con esta referencia hasta que la hayamos inicializado adecuadamente. La matriz **b** se inicializa para que apunte a una matriz de referencias **BerylliumSphere**, pero en esa matriz nunca se llegan a almacenar objetos **BerylliumSphere** directamente. De todos modos, podemos seguir preguntando cuál es el tamaño de la matriz, ya que **b** está apuntando a un objeto legítimo. Esto nos revela una cierta desventaja: no podemos averiguar cuántos elementos hay realmente almacenados *en* la matriz, ya que **length** nos dice sólo cuántos elementos *pueden* almacenarse; en otras palabras, dicho campo nos dice el tamaño del objeto matriz no el número de elementos que está almacenando en un momento determinado. Sin embargo, cuando se crea un objeto matriz sus referencias se inicializan automáticamente con el valor **null**, por lo que podemos ver si una posición concreta de una matriz tiene un objeto almacenado, comprobando si esa posición tiene un valor **null**. De forma similar, las matrices de primitivas se inicializan automáticamente con cero para los tipos numéricos, con **(char)0** para **char**, y con **false** para **boolean**.

La matriz **c** permite ilustrar la creación del objeto matriz seguida de la asignación de objetos **BerylliumSphere** a todas las posiciones de la matriz. La matriz **d** muestra la sintaxis de “inicialización agregada” que hace que el objeto matriz se cree (implícitamente con **new** en el cúmulo de memoria, al igual que la matriz **e**) e inicialice con objetos **BerylliumSphere**, todo ello en una única instrucción.

La siguiente inicialización de matriz puede considerarse como una especie de “inicialización agregada dinámica”. La inicialización agregada utilizada por **d** debe utilizarse en el lugar donde se define **d**, pero con la segunda sintaxis podemos crear e inicializar un objeto matriz en cualquier parte. Por ejemplo, suponga que **hide()** es un método que toma como argumento una matriz de objetos **BerylliumSphere**. Podriamos invocar ese método escribiendo:

```
hide(d);
```

pero también podemos crear dinámicamente la matriz que queramos pasar como argumento:

```
hide(new BerylliumSphere[] { new BerylliumSphere(),
    new BerylliumSphere() } );
```

En muchas situaciones, esta sintaxis proporciona una forma mucho más conveniente de escribir el código.

La expresión:

```
a = d;
```

muestra cómo podemos tomar una referencia asociada a un objeto matriz y asignarla a otro objeto matriz, al igual que podemos hacer con otro tipo de referencia a objetos. Ahora, tanto **a** como **d** están apuntando al mismo objeto matriz situado en el cúmulo de memoria.

La segunda parte de **ArrayOptions.java** muestra que las matrices de primitivas funcionan igual que las matrices de objetos, *salvo* porque las matrices de primitivas almacenan directamente los valores primitivos.

Ejercicio 1: (2) Cree un método que tome como argumento una matriz de objetos **BerylliumSphere**. Invoque el método creando el argumento dinámicamente. Demuestre que la inicialización agregada normal de matrices no funciona en este caso. Descubra las únicas situaciones en las que funciona la inicialización agregada de matrices y en las que la inicialización agregada dinámica es redundante.

Devolución de una matriz

Suponga que estamos escribiendo un método y que no queremos devolver un único valor, sino un conjunto de ellos. Los lenguajes como C y C++ hacen que esto sea difícil, porque no se puede devolver una matriz, sino sólo un puntero a una matriz. Esto genera problemas, porque resulta complicado controlar el tiempo de vida de la matriz, lo que a su vez conduce a fugas de memoria.

En Java, basta con devolver directamente la matriz. Nunca tenemos por qué preocuparnos por esa matriz: la matriz pervivirá mientras que sea necesaria y el depurador de memoria se encargará de borrarla una vez que hayamos terminado de utilizarla.

Como ejemplo, vamos a ver cómo se devolvería una matriz de objetos **String**:

```
//: arrays/IceCream.java
// Devolución de matrices desde métodos.
import java.util.*;

public class IceCream {
    private static Random rand = new Random(47);
    static final String[] FLAVORS = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    public static String[] flavorSet(int n) {
        if(n > FLAVORS.length)
            throw new IllegalArgumentException("Set too big");
        String[] results = new String[n];
        boolean[] picked = new boolean[FLAVORS.length];
        for(int i = 0; i < n; i++) {
            int t;
            do
                t = rand.nextInt(FLAVORS.length);
            while(picked[t]);
            results[i] = FLAVORS[t];
            picked[t] = true;
        }
        return results;
    }
    public static void main(String[] args) {
        for(int i = 0; i < 7; i++)
            System.out.println(Arrays.toString(flavorSet(3)));
    }
} /* Output:
[Rum Raisin, Mint Chip, Mocha Almond Fudge]
[Chocolate, Strawberry, Mocha Almond Fudge]
[Strawberry, Mint Chip, Mocha Almond Fudge]
[Rum Raisin, Vanilla Fudge Swirl, Mud Pie]
[Vanilla Fudge Swirl, Chocolate, Mocha Almond Fudge]
[Praline Cream, Strawberry, Mocha Almond Fudge]
[Mocha Almond Fudge, Strawberry, Mint Chip]
*///:-
```

El método **flavorSet()** crea una matriz de objetos **String** denominada **results**. El tamaño de esta matriz es **n**, que está determinado por el argumento que le pasemos al método. A continuación, el método selecciona una serie de valores aleatoriamente de entre la matriz **FLAVORS** y los coloca en **results**, devolviendo después esta matriz. La devolución de una matriz es exactamente igual que la devolución de cualquier otro objeto: se trata simplemente de una referencia. No es importante que la matriz haya sido creada dentro de **flavorSet()**, o en cualquier otro lugar. El depurador de memoria se encargará de borrar la matriz cuando hayamos terminado de usarla y esa matriz persistirá durante todo el tiempo que la necesitemos.

Como nota adicional, observe que cuando `flavorSet()` selecciona valores aleatoriamente, se encarga de comprobar que un valor concreto no haya sido previamente seleccionado. Esto se hace en un bucle `do` que continúa realizando selecciones aleatorias hasta que encuentre un valor que no esté ya en la matriz `picked` (por supuesto, también podría haberse realizado una comparación `String` para ver si el valor aleatorio está ya en la matriz `results`). Si tiene éxito, añade la nueva entrada y localiza la siguiente (`i` se incrementa).

Puede ver analizando la salida que `flavorSet()` selecciona los valores en orden aleatorio cada una de las veces:

Ejercicio 2: (1) Escriba un método que tome un argumento `int` y devuelva una matriz de dicho tamaño rellenada con objetos `BerylliumSphere`.

Matrices multidimensionales

Podemos crear fácilmente matrices multidimensionales. Para las matrices multidimensionales de primitivas, delimitamos cada vector de la matriz mediante llaves:

```
//: arrays/MultidimensionalPrimitiveArray.java
// Creación de matrices multidimensionales.
import java.util.*;

public class MultidimensionalPrimitiveArray {
    public static void main(String[] args) {
        int[][] a = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[1, 2, 3], [4, 5, 6]]
*///:-
```

Cada conjunto anidado de llaves nos desplaza al siguiente nivel de la matriz.

Este ejemplo utiliza el método `Arrays.deepToString()` de Java SE5, que transforma matrices multidimensionales en objetos `String`, como podemos ver a la salida.

También podemos asignar una matriz con `new`. He aquí una matriz tridimensional asignada en una expresión `new`:

```
//: arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
    public static void main(String[] args) {
        // Matriz 3-D con longitud fija:
        int[][][] a = new int[2][2][4];
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
*///:-
```

Podemos ver que los valores primitivos de la matriz se inicializan automáticamente si no proporcionamos un valor de inicialización explícito. Las matrices de objetos se inicializan con `null`.

Cada vector de las matrices que forman la matriz total puede tener cualquier longitud (esto se denomina *matriz desigual*):

```
//: arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
    public static void main(String[] args) {
```

```

Random rand = new Random(47);
// Matriz 3-D con vectores de longitud variable:
int[][][] a = new int[rand.nextInt(7)][][];
for(int i = 0; i < a.length; i++) {
    a[i] = new int[rand.nextInt(5)][];
    for(int j = 0; j < a[i].length; j++)
        a[i][j] = new int[rand.nextInt(5)];
}
System.out.println(Arrays.deepToString(a));
}
} /* Output:
[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]],
[0, 0, 0], [0], [0, 0, 0, 0], [[0, 0, 0], [0, 0, 0]],
[0], [[], [], [0]]]
*///:-
```

La primera instrucción **new** crea una matriz con un primer elemento de longitud aleatoria y el resto indeterminado. La segunda instrucción **new** dentro del bucle **for** rellena los elementos, pero dejando el tercer índice indeterminado hasta que nos encontramos con la tercera instrucción **new**.

Podemos tratar matrices de objetos no primitivos de una forma similar. En el siguiente ejemplo podemos ver cómo agrupar varias expresiones **new** mediante llaves:

```

//: arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
    public static void main(String[] args) {
        BerylliumSphere[][] spheres = {
            { new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),
                new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),
                new BerylliumSphere(), new BerylliumSphere(),
                new BerylliumSphere(), new BerylliumSphere(),
                new BerylliumSphere(), new BerylliumSphere() },
        };
        System.out.println(Arrays.deepToString(spheres));
    }
} /* Output:
[[Sphere 0, Sphere 1], [Sphere 2, Sphere 3, Sphere 4, Sphere 5],
 [Sphere 6, Sphere 7, Sphere 8, Sphere 9, Sphere 10,
 Sphere 11, Sphere 12, Sphere 13]]
*///:-
```

Podemos ver que **spheres** es otra matriz desigual, siendo la longitud de cada lista de objetos diferente.

El mecanismo de conversión automática también funciona con los inicializadores de matrices:

```

//: arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
    public static void main(String[] args) {
        Integer[][] a = { // Conversión automática:
            { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
            { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
            { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
            { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
        };
        System.out.println(Arrays.deepToString(a));
    }
}
```

```

    }
} /* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26,
27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*///:-
```

He aquí cómo podríamos construir por partes una matriz de objetos no primitivos:

```

//: arrays/AssemblingMultidimensionalArrays.java
// Creación de matrices multidimensionales.
import java.util.*;

public class AssemblingMultidimensionalArrays {
    public static void main(String[] args) {
        Integer[][] a;
        a = new Integer[3][];
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer[3];
            for(int j = 0; j < a[i].length; j++)
                a[i][j] = i * j; // Conversión automática
        }
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*///:-
```

La expresión `i*j` sólo tiene por objeto asignar un valor interesante al objeto `Integer`.

El método `Arrays.deepToString()` funciona tanto con matrices de primitivas como con matrices de objetos:

```

//: arrays/MultiDimWrapperArray.java
// Matrices multidimensionales de objetos "envoltorio".
import java.util.*;

public class MultiDimWrapperArray {
    public static void main(String[] args) {
        Integer[][] a1 = { // Conversión automática
            { 1, 2, 3 },
            { 4, 5, 6 },
        };
        Double[][] a2 = { // Conversión automática
            { { 1.1, 2.2 }, { 3.3, 4.4 } },
            { { 5.5, 6.6 }, { 7.7, 8.8 } },
            { { 9.9, 1.2 }, { 2.3, 3.4 } },
        };
        String[][] a3 = {
            { "The", "Quick", "Sly", "Fox" },
            { "Jumped", "Over" },
            { "The", "Lazy", "Brown", "Dog", "and", "friend" },
        };
        System.out.println("a1: " + Arrays.deepToString(a1));
        System.out.println("a2: " + Arrays.deepToString(a2));
        System.out.println("a3: " + Arrays.deepToString(a3));
    }
} /* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]]]
[9.9, 1.2], [[2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The, Lazy,
Brown, Dog, and, friend]]
*///:-
```

De nuevo, en las matrices **Integer** y **Double**, el mecanismo de conversión automática de Java SE5 se encarga de crear por nosotros los objetos envoltorio.

- Ejercicio 3:** (4) Escriba un método que cree e inicialice una matriz bidimensional de valores **double**. El tamaño de la matriz estará determinado por los argumentos del método y los valores de inicialización serán un rango determinado por sendos valores inicial y final que también serán argumentos del método. Cree un segundo método que imprima la matriz generada por el primer método. En **main()** compruebe los métodos creando e imprimiendo varias matrices de tamaños diferentes.
- Ejercicio 4:** (2) Repita el ejercicio anterior para una matriz tridimensional.
- Ejercicio 5:** (1) Demuestre que las matrices multidimensionales de tipos primitivos se inicializan automáticamente con **null**.
- Ejercicio 6:** (1) Escriba un método que tome dos argumentos **int**, indicando las dos dimensiones de una matriz 2-D. El método debe crear y llenar una matriz 2-D de objetos **BerylliumSphere** de acuerdo con los argumentos de dimensión.
- Ejercicio 7:** (1) Repita el ejercicio anterior para una matriz 3-D.

Matrices y genéricos

En general, las matrices y los genéricos no se combinan demasiado bien. No se pueden instanciar matrices de tipos parametrizados:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegal
```

El mecanismo de borrado automático de tipos elimina la información del parámetro de tipo, y las matrices deben conocer el tipo exacto que almacenan, con el fin de poder imponer los mecanismos de seguridad de tipos.

Sin embargo, lo que sí se puede es parametrizar el tipo de la propia matriz:

```
//: arrays/ParameterizedArrayType.java

class ClassParameter<T> {
    public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
    public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
    public static void main(String[] args) {
        Integer[] ints = { 1, 2, 3, 4, 5 };
        Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Integer[] ints2 =
            new ClassParameter<Integer>().f(ints);
        Double[] doubles2 =
            new ClassParameter<Double>().f(doubles);
        ints2 = MethodParameter.f(ints);
        doubles2 = MethodParameter.f(doubles);
    }
} //:-
```

Observe la comodidad que se deriva de utilizar un método parametrizado en lugar de una clase parametrizada: no hace falta instanciar una clase con un parámetro para cada tipo diferente al que necesitemos aplicar, y además el método se puede definir como estático. Por supuesto, no siempre podemos utilizar un método parametrizado en lugar de una clase parametrizada, aunque sí hay muchas ocasiones en las que puede ser preferible.

En realidad, no es del todo correcto decir que no se pueden crear matrices de tipos genéricos. El compilador no permite, en efecto, *instanciar* una matriz de un tipo genérico, sin embargo, lo que sí permite es crear una referencia a dicha matriz. Por ejemplo:

```
List<String>[] ls;
```

El compilador admite este tipo de sintaxis sin emitir ninguna queja. Y, aunque no podemos crear un objeto matriz real que almacene genéricos, sí que podemos crear una matriz del tipo no genérico y efectuar una proyección de tipos:

```
//: arrays/ArrayOfGenerics.java
// Es posible crear matrices de genéricos.
import java.util.*;

public class ArrayOfGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        List<String>[] ls;
        List[] la = new List[10];
        ls = (List<String>[])la; // Advertencia no comprobada
        ls[0] = new ArrayList<String>();
        // La comprobación en tiempo de compilación produce un error:
        // ls[1] = new ArrayList<Integer>();

        // El problema: List<String> es un subtipo de Object
        Object[] objects = ls; // Por lo que la asignación es correcta
        // Se compila y se ejecuta sin ningún problema:
        objects[1] = new ArrayList<Integer>();

        // Sin embargo, si nuestras necesidades son simples se
        // puede crear una matriz de genéricos, aunque con una
        // advertencia no comprobada:
        List<BerylliumSphere>[] spheres =
            (List<BerylliumSphere>[])new List[10];
        for(int i = 0; i < spheres.length; i++)
            spheres[i] = new ArrayList<BerylliumSphere>();
    }
} ///:-
```

Una vez que disponemos de una referencia a `List<String>[]`, podemos ver que se obtiene una cierta comprobación en tiempo de compilación. El problema es que las matrices son covariantes, por lo que una matriz `List<String>[]` es también una matriz `Object[]`, y podemos utilizar esto para asignar un objeto `ArrayList<Integer>` a nuestra matriz, sin que se produzca ningún error ni en tiempo de compilación ni en tiempo de ejecución.

Sin embargo, si sabemos que no vamos a efectuar ninguna generalización y nuestras necesidades son relativamente simples, es posible crear una matriz de genéricos, lo que nos proporciona una cierta comprobación de tipos básica en tiempo de compilación. No obstante, casi siempre un contenedor genérico será preferible a una matriz de genéricos.

En general, nos encontraremos con que los genéricos son efectivos en los *límites* de una clase o método. En los interiores, el mecanismo de borrado de tipos suele hacer inutilizables los genéricos. De este modo, no podemos, por ejemplo, crear una matriz de un tipo genérico:

```
//: arrays/ArrayOfGenericType.java
// Las matrices de tipos genéricos no se pueden compilar.

public class ArrayOfGenericType<T> {
    T[] array; // OK
    @SuppressWarnings("unchecked")
    public ArrayOfGenericType(int size) {
        // array = new T[size]; // Illegal
        array = (T[])new Object[size]; // Advertencia no comprobada
    }
    // Illegal:
    // public <U> U[] makeArray() { return new U[10]; }
} ///:-
```

De nuevo, el mecanismo de borrado de tipos interfiere con nuestros propósitos; en este ejemplo, se intenta crear matrices de tipos que se han visto sometidos al mecanismo de borrado de tipos y que son, por tanto, de tipo desconocido. Observe que *podemos* crear una matriz de tipo **Object**, y proyectarla, pero si quitamos la anotación `@SuppressWarnings` obtendremos una advertencia “no comprobada” en tiempo de compilación, porque la matriz no almacena realmente, ni comprueba de forma dinámica el tipo **T**. En otras palabras, si creamos una matriz **String[]**, Java impondrá tanto en tiempo de compilación como en tiempo de ejecución que sólo podemos colocar objetos **String** en dicha matriz. Sin embargo, si creamos una matriz **Object[]**, podemos almacenar en ella cualquier cosa menos tipos primitivos.

Ejercicio 8: (1) Demuestre las afirmaciones del párrafo anterior.

Ejercicio 9: (3) Cree las clases necesarias para el ejemplo **Peel<Banana>** y demuestre que el compilador no lo acepta. Corrija el problema utilizando un contenedor **ArrayList**.

Ejercicio 10: (2) Modifique **ArrayOfGenerics.java** para emplear contenedores en lugar de matrices. Demuestre que puede eliminar las advertencias de tiempo de compilación.

Creación de datos de prueba

Cuando se experimenta con las matrices y con los programas en general, resulta útil poder generar fácilmente matrices llenas de datos de prueba. Las herramientas de esta sección permiten llenar una matriz con valores u objetos.

Arrays.fill()

La clase **Arrays** de la biblioteca estándar de Java tiene un método **fill()** bastante trivial: se limita a duplicar un mismo valor en cada posición o, en el caso de los objetos, inserta en cada posición copias de la misma referencia. He aquí un ejemplo:

```
//: arrays/FillingArrays.java
// Utilización de Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class FillingArrays {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays.fill(a1, true);
        print("a1 = " + Arrays.toString(a1));
        Arrays.fill(a2, (byte)11);
        print("a2 = " + Arrays.toString(a2));
        Arrays.fill(a3, 'x');
        print("a3 = " + Arrays.toString(a3));
        Arrays.fill(a4, (short)17);
        print("a4 = " + Arrays.toString(a4));
        Arrays.fill(a5, 19);
        print("a5 = " + Arrays.toString(a5));
        Arrays.fill(a6, 23);
        print("a6 = " + Arrays.toString(a6));
        Arrays.fill(a7, 29);
        print("a7 = " + Arrays.toString(a7));
        Arrays.fill(a8, 47);
```

```

print("a8 = " + Arrays.toString(a8));
Arrays.fill(a9, "Hello");
print("a9 = " + Arrays.toString(a9));
// Manipulating ranges:
Arrays.fill(a9, 3, 5, "World");
print("a9 = " + Arrays.toString(a9));
}
} /* Output:
a1 = [true, true, true, true, true]
a2 = [11, 11, 11, 11, 11, 11]
a3 = [x, x, x, x, x, x]
a4 = [17, 17, 17, 17, 17, 17]
a5 = [19, 19, 19, 19, 19, 19]
a6 = [23, 23, 23, 23, 23, 23]
a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9 = [Hello, Hello, Hello, Hello, Hello, Hello]
a9 = [Hello, Hello, Hello, World, World, Hello]
*///:-
```

Podemos llenar la matriz completa o, como muestran las dos últimas instrucciones, llenar tan solo un rango de elementos. Pero como sólo se puede invocar `Arrays.fill()` con un único valor de datos, los resultados no son especialmente útiles.

Generadores de datos

Para crear matrices de datos más interesantes, pero de una manera flexible utilizaremos el concepto de **Generador** introducido en el Capítulo 15, *Genéricos*. Si una herramientas utiliza un objeto **Generador**, podemos generar cualquier clase de datos eligiendo el objeto **Generador** adecuado (se trata de un ejemplo del patrón de diseño basado en *estrategia*), cada uno de los diferentes generadores representa una estrategia diferente.¹

En esta sección proporcionaremos algunos generadores y, como ya hemos visto en ejemplos anteriores, también podemos definir fácilmente otros generadores que deseemos.

En primer lugar, he aquí un conjunto básico de generadores de recuento para todos los tipos envoltorio de primitivas y para las cadenas de caracteres. Las clases generadoras están anidadas dentro de la clase **CountingGenerator**, de modo que pueden utilizar el mismo nombre que los tipos de objeto que estén generando; por ejemplo, un generador que cree objetos de tipo **Integer** podría generarse mediante la expresión `new CountingGenerator.Integer()`:

```

//: net/mindview/util/CountingGenerator.java
// Implementaciones simples de generadores.
package net.mindview.util;

public class CountingGenerator {
    public static class Boolean implements Generator<java.lang.Boolean> {
        private boolean value = false;
        public java.lang.Boolean next() {
            value = !value; // sólo salta hacia atrás y hacia adelante
            return value;
        }
    }
    public static class Byte implements Generator<java.lang.Byte> {
        private byte value = 0;
        public java.lang.Byte next() { return value++; }
    }
}
```

¹ Si bien hay que recalcar que en este tema las fronteras resultan un tanto borrosas. También podríamos argumentar que un objeto **Generador** representa el patrón de diseño de *Comando*; sin embargo, en mi opinión, la tarea consiste en llenar una matriz y el objeto **Generador** lleva a cabo parte de dicha tarea, así que se trata más de una estrategia que de un comando.

```

static char[] chars = {"abcdefghijklmnopqrstuvwxyz" +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
public static class Character implements Generator<java.lang.Character> {
    int index = -1;
    public java.lang.Character next() {
        index = (index + 1) % chars.length;
        return chars[index];
    }
}
public static class String implements Generator<java.lang.String> {
    private int length = 7;
    Generator<java.lang.Character> cg = new Character();
    public String() {}
    public String(int length) { this.length = length; }
    public java.lang.String next() {
        char[] buf = new char[length];
        for(int i = 0; i < length; i++)
            buf[i] = cg.next();
        return new java.lang.String(buf);
    }
}
public static class Short implements Generator<java.lang.Short> {
    private short value = 0;
    public java.lang.Short next() { return value++; }
}
public static class Integer implements Generator<java.lang.Integer> {
    private int value = 0;
    public java.lang.Integer next() { return value++; }
}
public static class Long implements Generator<java.lang.Long> {
    private long value = 0;
    public java.lang.Long next() { return value++; }
}
public static class Float implements Generator<java.lang.Float> {
    private float value = 0;
    public java.lang.Float next() {
        float result = value;
        value += 1.0;
        return result;
    }
}
public static class Double implements Generator<java.lang.Double> {
    private double value = 0.0;
    public java.lang.Double next() {
        double result = value;
        value += 1.0;
        return result;
    }
}
}

```

Cada clase implementa su propio significado del término “recuento”. En el caso de `CountingGenerator.Character`, se trata simplemente de las letras mayúsculas y minúsculas repetidas una y otra vez. La clase `CountingGenerator.String` utiliza

CountingGenerator.Character para llenar una matriz de caracteres, que luego se transforma en un objeto de tipo **String**. El tamaño de la matriz está determinado por el argumento del constructor. Observe que **CountingGenerator.String** utiliza un objeto **Generator<java.lang.Character>** básico en lugar de una referencia específica a **CountingGenerator.Character**. Posteriormente, este generador puede sustituirse para generar **RandomGenerator.String** en **RandomGenerator.java**.

He aquí una herramienta de prueba que utiliza el mecanismo de reflexión con la sintaxis de generadores anidados, de manera que pueda utilizarse para probar cualquier conjunto de generadores que se adapten a esta estructura:

```
//: arrays/GeneratorsTest.java
import net.mindview.util.*;

public class GeneratorsTest {
    public static int size = 10;
    public static void test(Class<?> surroundingClass) {
        for(Class<?> type : surroundingClass.getClasses()) {
            System.out.print(type.getSimpleName() + ": ");
            try {
                Generator<?> g = (Generator<?>)type.newInstance();
                for(int i = 0; i < size; i++)
                    System.out.printf(g.next() + " ");
                System.out.println();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public static void main(String[] args) {
        test(CountingGenerator.class);
    }
} /* Output:
Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Long: 0 1 2 3 4 5 6 7 8 9
Integer: 0 1 2 3 4 5 6 7 8 9
Short: 0 1 2 3 4 5 6 7 8 9
String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP QRSTUVW XYZabcd efghijk lmnopqr
Character: a b c d e f g h i j
Byte: 0 1 2 3 4 5 6 7 8 9
Boolean: true false true false true false true false
*/://:-
```

Esto presupone que la clase que estemos probando contiene un conjunto de objetos **Generator** anidados, cada uno de los cuales dispone de un constructor predeterminado (sin argumentos). El método de reflexión **getClasses()** genera todas las clases anidadas. Entonces, el método **test()** crea una instancia de cada uno de estos generadores e imprime el resultado producido invocando **next()** diez veces.

He aquí un conjunto de objetos **Generator** que utilizan el generador de números aleatorios. Puesto que el constructor **Random** se inicializa con un valor constante, la salida es repetible cada vez que ejecutemos un programa empleando uno de estos generadores:

```
//: net/mindview/util/RandomGenerator.java
// Generadores que producen valores aleatorios.
package net.mindview.util;
import java.util.*;

public class RandomGenerator {
    private static Random r = new Random(47);
    public static class Boolean implements Generator<java.lang.Boolean> {
```

```

public java.lang.Boolean next() {
    return r.nextBoolean();
}
}
public static class
Byte implements Generator<java.lang.Byte> {
    public java.lang.Byte next() {
        return (byte)r.nextInt();
    }
}
public static class
Character implements Generator<java.lang.Character> {
    public java.lang.Character next() {
        return CountingGenerator.chars[
            r.nextInt(CountingGenerator.chars.length)];
    }
}
public static class
String extends CountingGenerator.String {
    // Insertar el generador aleatorio de caracteres:
    { cg = new Character(); } // Inicializador de instancia
    public String() {}
    public String(int length) { super(length); }
}
public static class
Short implements Generator<java.lang.Short> {
    public java.lang.Short next() {
        return (short)r.nextInt();
    }
}
public static class
Integer implements Generator<java.lang.Integer> {
    private int mod = 10000;
    public Integer() {}
    public Integer(int modulo) { mod = modulo; }
    public java.lang.Integer next() {
        return r.nextInt(mod);
    }
}
public static class
Long implements Generator<java.lang.Long> {
    private int mod = 10000;
    public Long() {}
    public Long(int modulo) { mod = modulo; }
    public java.lang.Long next() {
        return new java.lang.Long(r.nextInt(mod));
    }
}
public static class
Float implements Generator<java.lang.Float> {
    public java.lang.Float next() {
        // Eliminar todos los decimales salvo los dos primeros:
        int trimmed = Math.round(r.nextFloat() * 100);
        return ((float)trimmed) / 100;
    }
}
public static class
Double implements Generator<java.lang.Double> {
    public java.lang.Double next() {

```

```

        long trimmed = Math.round(r.nextDouble() * 100);
        return ((double)trimmed) / 100;
    }
}
} //:-

```

Puede ver que **RandomGenerator.String** hereda de **CountingGenerator.String** y simplemente inserta el nuevo generador de tipo **Character**.

Para generar números que no sean demasiado grandes, **RandomGenerator.Integer** utiliza de forma predeterminada un módulo igual a 10.000, pero el constructor sobrecargado nos permite elegir un valor más pequeño. La misma técnica se usa para **RandomGenerator.Long**. Para los generadores de tipo **Float** y **Double**, los valores situados detrás del punto decimal se recortan.

Podemos reutilizar **GeneratorsTest** para probar **RandomGenerator**:

```

//: arrays/RandomGeneratorsTest.java
import net.mindview.util.*;

public class RandomGeneratorsTest {
    public static void main(String[] args) {
        GeneratorsTest.test(RandomGenerator.class);
    }
} /* Output:
Double: 0.73 0.53 0.16 0.19 0.52 0.27 0.26 0.05 0.8 0.76
Float: 0.53 0.16 0.53 0.4 0.49 0.25 0.8 0.11 0.02 0.8
Long: 7674 8804 8950 7826 4322 896 8033 2984 2344 5810
Integer: 8303 3141 7138 6012 9966 8689 7185 6992 5746 3976
Short: 3358 20592 284 26791 12834 -8092 13656 29324 -1423 5327
String: bklInaMe sbtWHkj UrUkZPg wsqPzDy CyRFJQA HxxHvHq
XumcXZJ oogoyWM NvqeutP nXsgqia
Character: x x E A J J m z M s
Byte: -60 -17 55 -14 -5 115 39 -37 79 115
Boolean: false true false false true true true true true
*//:-

```

Podemos modificar el número de valores producidos cambiando el valor **GeneratorsTest.size**, que es de tipo **public**.

Creación de matrices a partir de generadores

Para tomar un objeto **Generator** y generar una matriz, necesitamos dos herramientas de conversión. La primera utiliza cualquier generador para producir una matriz de subtipos **Object**. Para resolver el problema de las primitivas, la segunda herramienta toma cualquier matriz de tipos envoltorio de primitivas y produce la matriz de primitivas asociada.

La primera herramienta tiene dos opciones, representadas por un método estático sobrecargado que se denomina **array()**. La primera versión del método toma una matriz existente y la rellena utilizando un generador, mientras que la segunda versión toma un objeto **Class**, un generador y el número deseado de elementos y crea una nueva matriz, de nuevo llenándola mediante el generador elegido. Observe que esta herramienta sólo produce matrices de subtipos **Object** y no permite crear matrices de primitivas:

```

//: net/mindview/util/Generated.java
package net.mindview.util;
import java.util.*;

public class Generated {
    // Rellenar una matriz existente
    public static <T> T[] array(T[] a, Generator<T> gen) {
        return new CollectionData<T>(gen, a.length).toArray(a);
    }
    // Crear una nueva matriz:
    @SuppressWarnings("unchecked")

```

```

public static <T> T[] array(Class<T> type,
    Generator<T> gen, int size) {
    T[] a =
        (T[])java.lang.reflect.Array.newInstance(type, size);
    return new CollectionData<T>(gen, size).toArray(a);
}
} //:-

```

La clase **CollectionData** se definirá en el Capítulo 17, *Análisis detallado de los contenedores*. Esta clase crea un objeto **Collection** lleno con elementos producidos por el generador **gen**. El número de elementos está determinado por el segundo argumento del constructor. Todos los subtipos de **Collection** tienen un método **toArray()** que llena la matriz argumento con los elementos del objeto **Collection**.

El segundo método emplea el mecanismo de reflexión para crear dinámicamente una matriz del tipo y tamaño apropiados. Entonces esta matriz se llena empleando la misma técnica que con el primer método.

Podemos probar **Generated** utilizando una de las clases **CountingGenerator** definidas en la sección anterior:

```

//: arrays/TestGenerated.java
import java.util.*;
import net.mindview.util.*;

public class TestGenerated {
    public static void main(String[] args) {
        Integer[] a = { 9, 8, 7, 6 };
        System.out.println(Arrays.toString(a));
        a = Generated.array(a, new CountingGenerator.Integer());
        System.out.println(Arrays.toString(a));
        Integer[] b = Generated.array(Integer.class,
            new CountingGenerator.Integer(), 15);
        System.out.println(Arrays.toString(b));
    }
} /* Output:
[9, 8, 7, 6]
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*//:-

```

Aún cuando la matriz **a** está inicializada, dichos valores se sobreescreiben al pasársela a través de **Generated.array()**, que sustituye los valores (pero deja la matriz original en su lugar). La inicialización de **b** muestra cómo puede crearse una matriz llena partiendo de cero.

Los genéricos no funcionan con valores primitivos, y queremos utilizar los generadores para llenar matrices de primitivas. Para resolver este problema, creamos un convertidor que toma cualquier matriz de objetos envoltorio y la convierte en una matriz de los tipos primitivos asociados. Sin esta herramienta, tendríamos que crear generadores especiales para todas las primitivas.

```

//: net/mindview/util/ConvertTo.java
package net.mindview.util;

public class ConvertTo {
    public static boolean[] primitive(Boolean[] in) {
        boolean[] result = new boolean[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i]; // Conversión automática
        return result;
    }
    public static char[] primitive(Character[] in) {
        char[] result = new char[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
    }
}

```

```

        return result;
    }
    public static byte[] primitive(Byte[] in) {
        byte[] result = new byte[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static short[] primitive(Short[] in) {
        short[] result = new short[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static int[] primitive(Integer[] in) {
        int[] result = new int[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static long[] primitive(Long[] in) {
        long[] result = new long[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static float[] primitive(Float[] in) {
        float[] result = new float[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static double[] primitive(Double[] in) {
        double[] result = new double[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
}
} //:-

```

Cada versión de **primitive()** crea una matriz de primitivas apropiada con la longitud correcta, y luego copia los elementos desde la matriz **in** de tipos envoltorio. Observe que el mecanismo de conversión automática entra en acción en la expresión:

```
result[i] = in[i];
```

He aquí un ejemplo que muestra cómo puede utilizarse **ConvertTo** con ambas versiones de **Generated.array()**:

```

//: arrays/PrimitiveConversionDemonstration.java
import java.util.*;
import net.mindview.util.*;

public class PrimitiveConversionDemonstration {
    public static void main(String[] args) {
        Integer[] a = Generated.array(Integer.class,
            new CountingGenerator.Integer(), 15);
        int[] b = ConvertTo.primitive(a);
        System.out.println(Arrays.toString(b));
        boolean[] c = ConvertTo.primitive(
            Generated.array(Boolean.class,
                new CountingGenerator.Boolean(), 7));
        System.out.println(Arrays.toString(c));
    }
}

```

```

    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[true, false, true, false, true, false, true]
*///:-
```

Finalmente, he aquí un programa que prueba las herramientas de generación de matrices utilizando clases **RandomGenerator**:

```

//: arrays/TestArrayGeneration.java
// Comprobar las herramientas que utilizan generadores
// para llenar matrices.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestArrayGeneration {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = ConvertTo.primitive(Generated.array(
            Boolean.class, new RandomGenerator.Boolean(), size));
        print("a1 = " + Arrays.toString(a1));
        byte[] a2 = ConvertTo.primitive(Generated.array(
            Byte.class, new RandomGenerator.Byte(), size));
        print("a2 = " + Arrays.toString(a2));
        char[] a3 = ConvertTo.primitive(Generated.array(
            Character.class,
            new RandomGenerator.Character(), size));
        print("a3 = " + Arrays.toString(a3));
        short[] a4 = ConvertTo.primitive(Generated.array(
            Short.class, new RandomGenerator.Short(), size));
        print("a4 = " + Arrays.toString(a4));
        int[] a5 = ConvertTo.primitive(Generated.array(
            Integer.class, new RandomGenerator.Integer(), size));
        print("a5 = " + Arrays.toString(a5));
        long[] a6 = ConvertTo.primitive(Generated.array(
            Long.class, new RandomGenerator.Long(), size));
        print("a6 = " + Arrays.toString(a6));
        float[] a7 = ConvertTo.primitive(Generated.array(
            Float.class, new RandomGenerator.Float(), size));
        print("a7 = " + Arrays.toString(a7));
        double[] a8 = ConvertTo.primitive(Generated.array(
            Double.class, new RandomGenerator.Double(), size));
        print("a8 = " + Arrays.toString(a8));
    }
} /* Output:
a1 = [true, false, true, false, false, true]
a2 = [104, -79, -76, 126, 33, -64]
a3 = [Z, n, T, c, Q, r]
a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
a5 = [7704, 7383, 7706, 575, 8410, 6342]
a6 = [7674, 8804, 8950, 7826, 4322, 896]
a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:-
```

Esto garantiza también que cada versión de **ConvertTo.primitive()** funcione correctamente.

Ejercicio 11: (2) Demuestre que el mecanismo de conversión automática (*autoboxing*) no funciona con las matrices.

Ejercicio 12: (1) Cree una matriz inicializada de valores **double** utilizando **CountingGenerator**. Imprima los resultados.

- Ejercicio 13:** (2) Rellene un objeto `String` utilizando `CountingGenerator.Character`.
- Ejercicio 14:** (6) Cree una matriz de cada tipo primitivo y luego rellene cada matriz utilizando `CountingGenerator`. Imprima cada matriz.
- Ejercicio 15:** (2) Modifique `ContainerComparison.java` creando un generador para `BerylliumSphere` y efectúe los cambios necesarios en `main()` para utilizar dicho generador con `Generated.array()`.
- Ejercicio 16:** (3) Comenzando con `CountingGenerator.java`, cree una clase `SkipGenerator` que produzca nuevos valores por el procedimiento de aplicar un incremento que se fijará de acuerdo con un argumento del constructor. Modifique `TestArrayGeneration.java` para demostrar que la nueva clase funciona correctamente.
- Ejercicio 17:** (5) Cree y pruebe un objeto `Generator` para `BigDecimal`, y compruebe que funciona con los métodos `Generated`.

Utilidades para matrices

En `java.util`, podrá encontrar la clase `Arrays`, que contiene un conjunto de métodos estáticos de utilidad para matrices. Hay seis métodos básicos: `equals()`, para comprobar si dos matrices son iguales (y un método `deepEquals()` para matrices multidimensionales); `fill()`, del que ya hemos hablado en este capítulo; `sort()`, para ordenar una matriz; `binarySearch()`, para encontrar un elemento en una matriz ordenada; `toString()`, para generar una representación de tipo `String` para una matriz; y `hashCode()`, para generar el valor *hash* de una matriz (veremos qué significa esto en el Capítulo 17, *Análisis detallado de los contenedores*). Todos estos métodos están sobrecargados para poder usarlos con todos los tipos primitivos y con objetos. Además, `Arrays.asList()` toma cualquier secuencia o matriz y la transforma en un contenedor de tipo `List`; ya hemos hablado de este método en el Capítulo 11, *Almacenamiento de objetos*.

Antes de analizar los métodos de `Arrays`, existe otro método útil que no forma parte de `Arrays`.

Copia en una matriz

La biblioteca estándar de Java proporciona un método estático, `System.arraycopy()`, que permite copiar matrices de forma bastante más rápida que se si utiliza un bucle `for` para realizar la copia manualmente. `System.arraycopy()` está sobrecargado para aceptar todos los tipos. He aquí un ejemplo donde se manipulan matrices de valores `int`:

```
//: arrays/CopyingArrays.java
// Utilización de System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class CopyingArrays {
    public static void main(String[] args) {
        int[] i = new int[7];
        int[] j = new int[10];
        Arrays.fill(i, 47);
        Arrays.fill(j, 99);
        print("i = " + Arrays.toString(i));
        print("j = " + Arrays.toString(j));
        System.arraycopy(i, 0, j, 0, i.length);
        print("j = " + Arrays.toString(j));
        int[] k = new int[5];
        Arrays.fill(k, 103);
        System.arraycopy(i, 0, k, 0, k.length);
        print("k = " + Arrays.toString(k));
        Arrays.fill(k, 103);
        System.arraycopy(k, 0, i, 0, k.length);
        print("i = " + Arrays.toString(i));
        // Objetos:
        Integer[] u = new Integer[10];
        Integer[] v = new Integer[5];
```

```

        Arrays.fill(u, new Integer(47));
        Arrays.fill(v, new Integer(99));
        print("u = " + Arrays.toString(u));
        print("v = " + Arrays.toString(v));
        System.arraycopy(v, 0, u, u.length/2, v.length);
        print("u = " + Arrays.toString(u));
    }
} /* Output:
i = [47, 47, 47, 47, 47, 47]
j = [99, 99, 99, 99, 99, 99, 99, 99, 99]
j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]
k = [47, 47, 47, 47, 47]
i = [103, 103, 103, 103, 103, 47, 47]
u = [47, 47, 47, 47, 47, 47, 47, 47, 47]
v = [99, 99, 99, 99, 99]
u = [47, 47, 47, 47, 47, 99, 99, 99, 99]
*///:-
```

Los argumentos de `arraycopy()` son la matriz de origen, o el desplazamiento dentro de la matriz de origen a partir del cual hay que empezar a copiar, la matriz de destino, el desplazamiento dentro de la matriz de destino donde debe empezar la copia y el número de elementos que hay que copiar. Naturalmente, cualquier violación de las fronteras de las matrices generará una excepción.

El ejemplo muestra que pueden copiarse tanto matrices de primitivas como matrices de objetos. Sin embargo, si copiamos matrices de objetos, entonces sólo se copian las referencias, no produciéndose ninguna duplicación de los propios objetos. Esto se denomina *copia superficial* (consulte los suplementos en línea del libro para conocer más detalles).

`System.arraycopy()` no realiza conversiones automáticas para los tipos primitivos: las dos matrices deben tener exactamente el mismo tipo.

Ejercicio 18: (3) Cree y rellene una matriz de objetos `BerylliumSphere`. Copie esta matriz en otra nueva y demuestre que se trata de una copia superficial.

Comparación de matrices

`Arrays` proporciona el método `equals()` para comprobar si dos matrices son iguales; dicho método está sobrecargado para poder trabajar con todos los tipos de primitivas y con `Object`. Para ser iguales, las matrices deben tener el mismo número de elementos y cada elemento tiene que ser equivalente al elemento correspondiente de la otra matriz, utilizándose el método `equals()` para cada elemento (para las primitivas, se utiliza el método `equals()` de la correspondiente clase envoltorio, por ejemplo, `Integer.equals()` para `int`). Por ejemplo:

```

//: arrays/ComparingArrays.java
// Utilización de Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class ComparingArrays {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        print(Arrays.equals(a1, a2));
        a2[3] = 11;
        print(Arrays.equals(a1, a2));
        String[] s1 = new String[4];
        Arrays.fill(s1, "Hi");
        String[] s2 = { new String("Hi"), new String("Hi"),
            new String("Hi"), new String("Hi") };
        print(Arrays.equals(s1, s2));
    }
}
```

```

    }
} /* Output:
true
false
true
****:-
```

Originalmente, **a1** y **a2** son exactamente iguales, por lo que la salida es “true”, pero después se cambia uno de los elementos, lo que hace que el resultado sea “false”. En el último caso, todos los elementos de **s1** apuntan al mismo objeto, pero **s2** tiene cinco objetos diferentes. Sin embargo, la igualdad de matrices está basada en los contenidos (a través de **Object.equals()**), por lo que el resultado es “true”.

Ejercicio 19: (2) Cree una clase con un campo **int** que se inicialice a partir de un argumento de un constructor. Cree dos matrices de estos objetos, utilizando valores de inicialización idénticos para cada matriz, y demuestre que **Arrays.equals()** dice que son distintas. Añada el método **equals()** a la clase para corregir el problema.

Ejercicio 20: (4) Ilustre mediante un ejemplo el uso de **deepEquals()** para matrices multidimensionales.

Comparaciones de elementos de matriz

Las operaciones de ordenación deben realizar comparaciones basadas en el tipo real de los objetos. Por supuesto, una solución consiste en escribir un método de ordenación distinto para cada uno de los posibles tipos, pero dicho código no será reutilizable para tipos nuevos.

Uno de los objetivos principales del diseño en el campo de la programación consiste en “separar las cosas que cambian de las cosas que no lo hacen”, y aquí el código que no varía es el algoritmo de ordenación general, siendo la única cosa que cambia entre un uso y el siguiente la forma en que se comparan los objetos. Por tanto, en lugar de incluir el código de comparación en muchas rutinas de ordenación diferentes, se utiliza el patrón de diseño basado en estrategia². Con una Estrategia, la parte del código que varía se encapsula dentro de una clase separada, (el objeto Estrategia). Lo que se hace es entregar un objeto Estrategia al código que permanece invariable, el cual utiliza dicha Estrategia para implementar su algoritmo. De esa forma podemos hacer que los diferentes objetos expresen diferentes formas de comparación y entregarles a todos ellos el mismo código de ordenación.

Java dispone de dos maneras para proporcionar la funcionalidad de comparación. La primera es con el método de comparación “natural” que se añade a una clase implementando la interfaz **java.lang.Comparable**. Se trata de una interfaz muy simple con un único método, **compareTo()**. Este método toma como argumento otro objeto del mismo tipo y produce un valor negativo si el objeto actual es inferior al argumento, cero si el argumento es igual y un valor positivo si el objeto actual es superior al argumento.

He aquí una clase que implementa **Comparable** e ilustra la comparabilidad empleando el método de la biblioteca estándar de Java **Arrays.sort()**:

```

//: arrays/CompType.java
// Implementación de Comparable en una clase.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CompType implements Comparable<CompType> {
    int i;
    int j;
    private static int count = 1;
    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }
    public String toString() {
        String result = "[i = " + i + ", j = " + j + "]";
        Print.out(result);
        return result;
    }
}
```

² Design Patterns, Erich Gamma et al. (Addison-Wesley, 1995). Consulte *Thinking in Patterns (with Java)* en www.MindView.net.

```

        if(count++ % 3 == 0)
            result += "\n";
        return result;
    }
    public int compareTo(CompType rv) {
        return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
    }
    private static Random r = new Random(47);
    public static Generator<CompType> generator() {
        return new Generator<CompType>() {
            public CompType next() {
                return new CompType(r.nextInt(100),r.nextInt(100));
            }
        };
    }
    public static void main(String[] args) {
        CompType[] a =
            Generated.array(new CompType[12], generator());
        print("before sorting:");
        print(Arrays.toString(a));
        Arrays.sort(a);
        print("after sorting:");
        print(Arrays.toString(a));
    }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
, [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
, [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
, [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
]
*///:~

```

Cuando definimos el método de comparación somos responsables de decidir qué quiere decir comparar uno de nuestros objetos con otro. Aquí, sólo se utilizan los valores *i* para la comparación, mientras que los valores *j* se ignoran.

El método **generator()** produce un objeto que implementa la interfaz **Generator** creando una clase interna anónima. Ésta construye objetos **CompType** inicializándolos con valores aleatorios. En **main()**, se utiliza el generador para llenar una matriz de objetos **CompType**, que se ordena a continuación. Si no hubiéramos implementado **Comparable** obtendríamos una excepción **ClassCastException** en tiempo de ejecución cuando tratáramos de invocar **sort()**. Esto se debe a que **sort()** proyecta su argumento sobre **Comparable**.

Ahora suponga que alguien nos entrega una clase que no implementa **Comparable**, o que alguien nos entrega una clase que *sí* que implementa **Comparable**, pero decidimos que no nos gusta la forma en que funciona y que preferiríamos disponer de un método de comparación distinto para ese tipo de objeto. Para resolver el problema, creamos una clase separada que implementa una interfaz llamada **Comparator** (ya la hemos presentado brevemente en el Capítulo 11, *Almacenamiento de objetos*). Se trata de un ejemplo del patrón de diseño basado en Estrategia. Tiene dos métodos, **compare()** y **equals()**. Sin embargo, no tenemos necesidad de implementar **equals()** salvo por necesidades especiales de rendimiento, ya que cada vez que se crea una clase, ésta hereda implícitamente de **Object**, que tiene un método **equals()**. Por tanto, podemos limitarnos a utilizar el método **equals()** predeterminado de **Object** sin por ello dejar de satisfacer las imposiciones de la interfaz.

La clase **Collections** (que examinaremos con más detalle en el siguiente capítulo) contiene un método **reverseOrder()** que produce un objeto **Comparator** para invertir el sistema natural de ordenación. Esto puede aplicarse a **CompType**:

```

//: arrays/Reverse.java
// El comparador Collections.reverseOrder()
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Reverse {
    public static void main(String[] args) {
        CompType[] a = Generated.array(
            new CompType[12], CompType.generator());
        print("before sorting:");
        print((Arrays.toString(a)));
        Arrays.sort(a, Collections.reverseOrder());
        print("after sorting:");
        print((Arrays.toString(a)));
    }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
, [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
, [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
, [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
]
*/

```

También podemos escribir nuestro propio objeto **Comparator**. El siguiente ejemplo compara objetos **CompType** basados en sus valores **j**, en lugar de en sus valores **i**:

```

//: arrays/ComparatorTest.java
// Implementación de un comparador para una clase.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class CompTypeComparator implements Comparator<CompType> {
    public int compare(CompType o1, CompType o2) {
        return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = Generated.array(
            new CompType[12], CompType.generator());
        print("before sorting:");
        print((Arrays.toString(a)));
        Arrays.sort(a, new CompTypeComparator());
        print("after sorting:");
        print((Arrays.toString(a)));
    }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]

```

```

        , [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
        , [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
    ]
    after sorting:
    [[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
    , [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
    , [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
    , [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
    ]
*///:-
```

Ejercicio 21: (3) Trate de ordenar una matriz de objetos del Ejercicio 18. Implemente **Comparable** para corregir el problema. Ahora cree un objeto **Comparator** para disponer los objetos en orden inverso.

Ordenación de una matriz

Con los métodos de ordenación predefinidos, podemos ordenar cualquier matriz de primitivas o cualquier matriz de objetos que implementen **Comparable** o dispongan de un objeto **Comparator** asociado.³ He aquí un ejemplo que genera objetos **String** aleatorios y los ordena:

```

//: arrays/StringSorting.java
// Ordenación de una matriz de objetos String.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class StringSorting {
    public static void main(String[] args) {
        String[] sa = Generated.array(new String[20],
            new RandomGenerator.String(5));
        print("Before sort: " + Arrays.toString(sa));
        Arrays.sort(sa);
        print("After sort: " + Arrays.toString(sa));
        Arrays.sort(sa, Collections.reverseOrder());
        print("Reverse sort: " + Arrays.toString(sa));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sort: " + Arrays.toString(sa));
    }
} /* Output:
Before sort: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMRoE,
suEcU, OneOE, dLsmw, HLGEa, hKcxr, EqUCB, bkIna, Mesbt,
WHkjU, rUkZP, gwsqP, zDyCy, RFJQA, HxxHv]
After sort: [EqUCB, HLGEa, HxxHv, JMRoE, Mesbt, OWZnT,
OneOE, RFJQA, WHkjU, YNzbr, bkIna, cQrGs, dLsmw, eGZMm,
gwsqP, hKcxr, nyGcF, rUkZP, suEcU, zDyCy]
Reverse sort: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP,
eGZMm, dLsmw, cQrGs, bkIna, YNzbr, WHkjU, RFJQA, OneOE,
OWZnT, Mesbt, JMRoE, HxxHv, HLGEa, EqUCB]
Case-insensitive sort: [bkIna, cQrGs, dLsmw, eGZMm, EqUCB,
gwsqP, hKcxr, HLGEa, HxxHv, JMRoE, Mesbt, nyGcF, OneOE,
OWZnT, RFJQA, rUkZP, suEcU, WHkjU, YNzbr, zDyCy]
*///:-
```

Una de las cosas que observamos al analizar la salida en los algoritmos de ordenación para objetos **String** es que la ordenación es *lexicográfica*, por lo que se ponen primero todas las palabras que comienzan por letras mayúsculas y después todas las palabras que comienzan con minúscula. Si queremos agrupar las palabras con independencia de si las letras son mayúsculas o minúsculas, se utiliza **String.CASE_INSENSITIVE_ORDER**, como se muestra en la última llamada a **sort()** del ejemplo anterior.

³ Sorprendentemente, Java 1.0 y 1.1 no incluían ningún soporte para la ordenación de objetos **String**.

El algoritmo de ordenación que se utiliza en la biblioteca estándar de Java está diseñado para comportarse de manera óptima para el tipo concreto que se esté ordenando: un algoritmo Quicksort para primitivas y una ordenación estable por combinación para objetos. No es necesario preocuparse acerca de las cuestiones de rendimiento a menos que la herramienta de perfilado nos indique que el proceso de ordenación está actuando como un cuello de botella.

Búsquedas en una matriz ordenada

Una vez ordenada una matriz, podemos realizar una búsqueda rápida de un elemento concreto invocando **Arrays.binarySearch()**. Sin embargo, si tratamos de utilizar **binarySearch()** en una matriz desordenada los resultados serán impredecibles. El siguiente ejemplo utiliza un objeto **RandomGenerator.Integer** para llenar una matriz y luego emplea el mismo generador para producir valores de búsqueda:

```
//: arrays/ArraySearching.java
// Utilización de Arrays.binarySearch().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ArraySearching {
    public static void main(String[] args) {
        Generator<Integer> gen =
            new RandomGenerator.Integer(1000);
        int[] a = ConvertTo.primitive(
            Generated.array(new Integer[25], gen));
        Arrays.sort(a);
        print("Sorted array: " + Arrays.toString(a));
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);
            if(location >= 0) {
                print("Location of " + r + " is " + location +
                    ", a[" + location + "] = " + a[location]);
                break; // Salir del bucle while
            }
        }
    }
} /* Output:
Sorted array: [128, 140, 200, 207, 258, 258, 278, 288, 322, 429, 511, 520, 522, 551, 555,
589, 693, 704, 809, 861, 861, 868, 916, 961, 998]
Location of 322 is 8, a[8] = 322
*///:-
```

En el bucle **while**, se generan elementos de búsqueda con valores aleatorios hasta que se encuentra uno de ellos.

Arrays.binarySearch() devuelve un valor mayor o igual que cero si se encuentra el elemento que se está buscando. En caso contrario, devuelve un valor negativo que representa el lugar en el que debería insertarse el elemento, si se está manteniendo la ordenación de la matriz de forma manual. El valor devuelto es:

- (punto de inserción) - 1

El punto de inserción es el índice del primer elemento superior a la clave, o bien **a.size()**, si todos los elementos de la matriz son inferiores a la clave especificada.

Si una matriz contiene elementos duplicados, no hay ninguna garantía en lo qué respecta a cuál de esos duplicados se encontrará. El algoritmo de búsqueda no está diseñado para soportar elementos duplicados, aunque sí que los tolera. Si necesitamos una lista ordenada de elementos no duplicados, hay que emplear **TreeSet** (para mantener la ordenación) o **LinkedHashSet** (para mantener el orden de inserción). Estas clases se encargan de resolver por nosotros todos los detalles, de manera automática. Sólo en aquellos casos en los que aparezcan cuellos de botella de rendimiento debería sustituirse una de estas clases por una matriz cuyo mantenimiento se realizará de forma manual.

Si ordenamos una matriz de objetos utilizando un objeto **Comparator** (las matrices primitivas no permiten realizar ordenaciones con un objeto **Comparator**), hay que incluir ese mismo objeto **Comparator** cuando se invoque a **binarySearch()** (empleando la versión sobrecargada de este método). Por ejemplo, podemos modificar el programa **StringSorting.java** para realizar una búsqueda:

```
//: arrays/AlphabeticSearch.java
// Búsqueda con un comparador.
import java.util.*;
import net.mindview.util.*;

public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa = Generated.array(new String[30],
            new RandomGenerator.String(5));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        System.out.println(Arrays.toString(sa));
        int index = Arrays.binarySearch(sa, sa[10],
            String.CASE_INSENSITIVE_ORDER);
        System.out.println("Index: " + index + "\n" + sa[index]);
    }
} /* Output:
[bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr, HLGEa, HqXum, HxxHv, JMROE,
JmzMs, Mesbt, MNvqe, nyGcF, ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia, slJrl, suEcU,
uTpNt, vpfFv, WHkjU, xxEAJ, YNzbr, zDyCyl]
Index: 10
HxxHv
*///:-
```

El objeto **Comparator** debe pasarse como tercer argumento al método **binarySearch()** sobrecargado. En este ejemplo, el éxito de la operación de búsqueda está garantizado porque el elemento de búsqueda se selecciona a partir de la propia matriz.

Ejercicio 22: (2) Demuestre que los resultados de realizar una búsqueda binaria con **binarySearch()** en una matriz desordenada son impredecibles.

Ejercicio 23: (2) Cree una matriz de objetos **Integer**, rellénela con valores **int** aleatorios (utilizando conversión automática) y dispóngala en orden inverso utilizando **Comparator**.

Ejercicio 24: (3) Demuestre que se pueden realizar búsquedas en la clase del Ejercicio 19.

Resumen

En este capítulo, hemos visto que Java proporciona un soporte razonable para las matrices de tamaño fijo y bajo nivel. Este tipo de matriz pone énfasis en el rendimiento más que en la flexibilidad, de forma similar al modelo de matrices de C y C++. En la versión inicial de Java, las matrices de tamaño fijo y bajo nivel eran absolutamente necesarias, no sólo porque los diseñadores de Java eligieron incluir tipos primitivos (también por razones de rendimiento), sino también porque el soporte para contenedores en dicha versión era mínimo. Por esa razón, en las primeras versiones de Java siempre era razonable elegir las matrices como forma de implementación.

En las versiones subsiguientes de Java, el soporte de contenedores mejoró significativamente y ahora los contenedores tienden a ser preferibles a las matrices desde todos los puntos de vista, salvo el de rendimiento. Incluso en lo que al rendimiento respecta, el de los contenedores ha mejorado significativamente. Como hemos indicado en otros puntos del libro, los problemas de rendimiento nunca suelen presentarse, de todos modos, en los lugares donde nos los imaginamos.

Con la adición del mecanismo de conversión automática de tipos primitivos y del mecanismo de genéricos, resulta muy sencillo almacenar primitivas en los contenedores, lo que constituye un argumento adicional para sustituir las matrices de bajo nivel por contenedores. Puesto que los genéricos permiten producir contenedores seguros en lo que respecta a los tipos de objetos, las matrices han dejado también de suponer una ventaja a ese respecto.

Como se ha indicado en este capítulo, y como podrá ver cuando trate de utilizarlos, los genéricos resultan bastante hostiles en lo que a las matrices se refiere. A menudo, incluso cuando conseguimos que los genéricos y las matrices funcionen con-

juntamente de alguna manera (como veremos en el siguiente capítulo), seguimos teniendo advertencias “no comprobadas” durante la compilación.

En muchas ocasiones, los diseñadores del lenguaje Java me han dicho directamente que se deberían utilizar contenedores en lugar de matrices; esos comentarios surgían a la hora de discutir ejemplos concretos (yo he estado empleando matrices para ilustrar técnicas específicas y no tenía, por tanto, la opción de utilizar contenedores).

Todas estas cuestiones indican que los contenedores son preferibles, por regla general, a las matrices a la hora de programar con las versiones recientes de Java. Sólo cuando esté demostrado que el rendimiento constituye un problema (y que utilizar una matriz permitirá resolverlo) deberíamos recurrir a la utilización de matrices.

Puede ser una afirmación demasiado categórica, pero algunos lenguajes no disponen en absoluto de matrices de tamaño fijo y bajo nivel. Sólo disponen de contenedores de tamaño variable con una funcionalidad significativamente superior a la de las matrices estilo C/C++/Java. Python,⁴ por ejemplo, tiene un tipo `list` que utiliza la sintaxis básica de matrices, pero dispone de una funcionalidad muy superior; podemos incluso heredar a partir de ese tipo:

```
#: arrays/PythonLists.py

aList = [1, 2, 3, 4, 5]
print type(aList) # <type 'list'>
print aList # [1, 2, 3, 4, 5]
print aList[4] # 5. Indexación básica de la lista
aList.append(6) # Añadir a cambiar el tamaño de las listas
aList += [7, 8] # Añadir una lista a una lista
print aList # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print aSlice # [3, 4]

class MyList(list): # Heredar de una lista
    # Definir un método, puntero 'this' es explícito:
    def getReversed(self):
        reversed = self[:] # Copiar lista utilizando fragmentos
        reversed.reverse() # Método predefinido de la lista
        return reversed

list2 = MyList(aList) # No hace falta 'new' para crear objetos
print type(list2) # <class '__main__.MyList'>
print list2.getReversed() # [8, 7, 6, 5, 4, 3, 2, 1]
#:=
```

La sintaxis básica de Python se ha presentado en el capítulo anterior. En este ejemplo, se crea una lista simplemente insertando una secuencia de objetos separados por comas entre corchetes. El resultado es un objeto cuyo tipo en tiempo de ejecución es `list` (la salida de las instrucciones `print` se muestra en forma de comentarios en la misma línea). El resultado de imprimir un objeto `list` es el mismo que si utilizáramos `Arrays.toString()` en Java.

La creación de una subsecuencia de un objeto `list` se lleva a cabo a partir de los “fragmentos”, incluyendo el operador `:` dentro de la operación de índice. El tipo `list` tiene muchas otras operaciones predefinidas.

`MyList` es una definición de `class`; las clases base se indican entre paréntesis. Dentro de la clase, las instrucciones `def` especifican métodos y el primer argumento al método es automáticamente equivalente a `this` en Java, salvo porque en Python es explícito y se utiliza el identificador `self` por convenio (no se trata de una palabra clave). Observe que el constructor se hereda automáticamente.

Aunque todo en Python es realmente un objeto (incluyendo los tipos enteros y de coma flotante), seguimos disponiendo de una puerta de escape, en el sentido de que se pueden optimizar partes del código cuyo rendimiento sea crítico escribiendo extensiones en C, C++ o con una herramienta especial llamada Pyrex, que está diseñada para acelerar fácilmente la ejecución del código. De esta forma, podemos mantener la pureza de la orientación a objetos sin que ello nos impida realizar mejoras de rendimiento.

⁴ Véase www.Python.org.

El lenguaje PHP⁵ va un paso más allá todavía al disponer de un único tipo de matriz, que actúa tanto como una matriz con índices de tipo entero cuanto como una matriz asociativa (un mapa).

Resulta interesante preguntarse después de todos estos años de evolución del lenguaje Java, si los diseñadores incluirían las primitivas y las matrices de bajo nivel en el lenguaje si tuvieran que comenzar de nuevo con la tarea de diseño. Si se dejaran estas funcionalidades fuera del lenguaje, sería posible construir un lenguaje orientado a objetos verdaderamente puro (a pesar de lo que se diga, Java no es un lenguaje orientado a objetos puro, precisamente debido a los remanentes de bajo nivel). El argumento de la eficiencia siempre parece bastante convincente, pero el paso del tiempo ha ido mostrando una evolución en el sentido de alejarse de esta idea para utilizar componentes de más nivel, como los contenedores. Y hay que tener en cuenta, que si se pudieran incluir los contenedores en el cuerpo principal del lenguaje, como sucede en otros lenguajes, entonces el compilador dispondría de mejores oportunidades para mejorar el código.

Dejando aparte estas especulaciones teóricas, lo cierto es que las matrices siguen estando presentes y que nos encontraremos con ellas una y otra vez a la hora de leer código. Sin embargo, los contenedores son casi siempre una mejor opción.

Ejercicio 25: (3) Reescriba `PythonLists.py` en Java.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

⁵ Véase www.php.net.

Análisis detallado de los contenedores

17

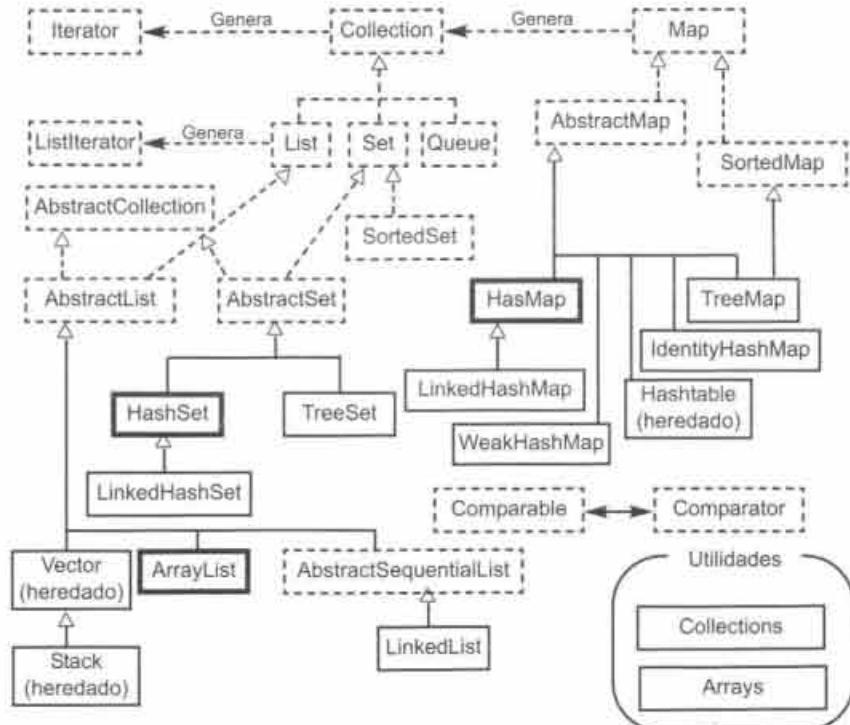
En el Capítulo 11, *Almacenamiento de objetos*, hemos introducido los conceptos y la funcionalidad básica de la biblioteca de contenedores de Java, y aquellas explicaciones son suficientes para poder empezar a utilizar contenedores. En este capítulo se analiza dicha importante biblioteca con más profundidad.

Para poder sacar el máximo contenido de la biblioteca de contenedores necesitamos conocer más detalles de los que se presentaron en el Capítulo 11, *Almacenamiento de los objetos*, pero el material que en este capítulo se presenta depende de temas más avanzados (como los genéricos), lo cual es la razón de que se pospusiera hasta este momento.

Después de incluir una panorámica completa de los contenedores veremos cómo funcionan los mecanismos de *hash* y cómo escribir métodos `hashCode()` y `equals()` que funcionen con los contenedores a los que se les haya aplicado un mecanismo de *hash*. Veremos también por qué hay diferentes versiones de algunos contenedores y qué criterios usar para elegir una versión u otra. El capítulo finaliza con una exploración de las utilidades de propósito general y de una serie de clases especiales.

Taxonomía completa de los contenedores

La sección “Resumen” del Capítulo 11, *Almacenamiento de objetos*, mostraba un diagrama simplificado de la biblioteca de contenedores de Java. He aquí un diagrama más completo de la biblioteca de colecciones, incluyendo las clases abstractas y los componentes heredados (con la excepción de las implementaciones de `Queue`):



Taxonomía completa de los contenedores

Java SE5 añade:

- La interfaz **Queue** (para implementarla se ha modificado **LinkedList** como vimos en el Capítulo 11, *Almacenamiento de objetos*) y sus implementaciones **PriorityQueue** (cola con prioridad) y diversas variantes de **BlockingQueue** (cola bloqueante) que se verán en Capítulo 21, *Concurrencia*.
- Una interfaz **ConcurrentMap** y su implementación **ConcurrentHashMap** (mapa hash concurrente), que también se utiliza para el mecanismo de hebras de programación que se aborda en el Capítulo 21, *Concurrencia*.
- **CopyOnWriteArrayList** y **CopyOnWriteArraySet**, también para concurrencia.
- **EnumSet** y **EnumMap**, implementaciones especiales de **Set** y **Map** para utilizar con enumeraciones, como se explica en el Capítulo 19, *Tipos enumerados*.
- Varias utilidades en la clase **Collections**.

Los recuadros de trazos representan clases abstractas y podemos ver varias clases cuyos nombres comienzan por “**Abstract**”. Estas clases pueden resultar un poco confusas al principio, pero son simplemente herramientas que implementan parcialmente una interfaz concreta. Si estuviéramos creando nuestra propia clase **Set**, por ejemplo, no comenzaríamos con la interfaz **Set** e implementaríamos todos los métodos; en lugar de ello heredariamos de **AbstractSet** y haríamos el trabajo mínimo necesario para definir nuestra nueva clase. Sin embargo, la biblioteca de contenedores contiene la suficiente funcionalidad como para satisfacer nuestras necesidades casi siempre, por lo que normalmente podemos ignorar cualquier clase que comience con “**Abstract**”.

Relleno de contenedores

Aunque el problema de imprimir contenedores está resuelto, el proceso de llenar contenedores sufre la misma deficiencia que **java.util.Arrays**. Al igual que con **Arrays**, existe una clase de acompañamiento denominada **Collections** que contiene métodos de utilidad estáticos, incluyendo uno que se llama **fill()** y que sirve para llenar colecciones. Al igual que la versión de **Arrays**, este método **fill()** se limita a duplicar una única referencia a objeto por todo el contenedor. Además, sólo funciona para objetos **List**, aunque la lista resultante puede pasarse a un constructor o a un método **addAll()**:

```
//: containers/FillingLists.java
// Los métodos Collections.fill() y Collections.nCopies().
import java.util.*;

class StringAddress {
    private String s;
    public StringAddress(String s) { this.s = s; }
    public String toString() {
        return super.toString() + " " + s;
    }
}

public class FillingLists {
    public static void main(String[] args) {
        List<StringAddress> list= new ArrayList<StringAddress>(
            Collections.nCopies(4, new StringAddress("Hello")));
        System.out.println(list);
        Collections.fill(list, new StringAddress("World!"));
        System.out.println(list);
    }
} /* Output: (Sample)
[StringAddress@82ba41 Hello, StringAddress@82ba41 Hello,
StringAddress@82ba41 Hello, StringAddress@82ba41 Hello]
[StringAddress@923e30 World!, StringAddress@923e30 World!,
StringAddress@923e30 World!, StringAddress@923e30 World!]
*///:-
```

Este ejemplo muestra dos formas de llenar un objeto **Collection** con referencias a un único objeto. La primera, **Collections.nCopies()**, crea un objeto **List** que se pasa al constructor; el cual llena el objeto **ArrayList**.

El método **toString()** en **StringAddress** llama a **Object.toString()**, que genera el nombre de la clase seguido por la representación hexadecimal sin signo del código *hash* del objeto (generada por un método **hashCode()**). Podemos ver, analizando la salida, que todas las referencias apuntan al mismo objeto y éste también se cumple después de invocar un segundo método, **Collections.fill()**. El método **fill()** es todavía menos útil debido al hecho de que sólo puede sustituir elementos que ya se encuentren dentro del objeto **List** no permitiendo añadir nuevos elementos.

Una solución basada en generador

Casi todos los subtipos de **Collection** tienen un constructor que toma como argumento otro objeto **Collection**, a partir del cual puede llenar el nuevo contenedor. Por tanto, para poder crear fácilmente datos de prueba, todo lo que necesitamos es construir una clase que tome como argumentos del constructor un objeto **Generator** (definido en el Capítulo 15, *Genéricos*, y analizado con más detalles en el Capítulo 16, *Matrices*) y un valor **quantity** (cantidad):

```
//: net/mindview/util/CollectionData.java
// Una colección rellena con datos utilizando un objeto generador.
package net.mindview.util;
import java.util.*;

public class CollectionData<T> extends ArrayList<T> {
    public CollectionData(Generator<T> gen, int quantity) {
        for(int i = 0; i < quantity; i++)
            add(gen.next());
    }
    // Un método genérico de utilidad:
    public static <T> CollectionData<T>
    list(Generator<T> gen, int quantity) {
        return new CollectionData<T>(gen, quantity);
    }
} //:-
```

Este ejemplo utiliza **Generator** para insertar en el contenedor tantos objetos como necesitemos. El contenedor resultante puede entonces pasar al constructor de cualquier tipo **Collection**, y dicho constructor copiará los datos dentro de la instancia. También puede utilizarse el método **addAll()** que forma parte de todos los subtipos de **Collection** para llenar un objeto **Collection** existente.

El método genérico de utilidad reduce la cantidad de texto que hay que escribir a la hora de utilizar la clase.

CollectionData es un ejemplo del patrón de diseño *Adaptador*¹; adapta un objeto **Generator** al constructor de un tipo **Collection**.

He aquí un ejemplo que inicializa un contenedor **LinkedHashSet**:

```
//: containers/CollectionDataTest.java
import java.util.*;
import net.mindview.util.*;

class Government implements Generator<String> {
    String[] foundation = ("strange women lying in ponds " +
    "distributing swords is no basis for a system of " +
    "government").split(" ");
    private int index;
    public String next() { return foundation[index++]; }
}
```

¹ Puede que esto no encaje estrictamente en la definición de adaptador, tal como se define en el libro de *Design Patterns*, pero creo que cumple con el espíritu de este patrón.

```

public class CollectionDataTest {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>(
            new CollectionData<String>(new Government(), 15));
        // Utilización del método de utilidad:
        set.addAll(CollectionData.list(new Government(), 15));
        System.out.println(set);
    }
} /* Output:
[strange, women, lying, in, ponds, distributing, swords,
is, no, basis, for, a, system, of, government]
*///:-
```

Los elementos están en el mismo orden en que se insertaron, porque un contenedor **LinkedHashSet** mantiene una lista enlazada que preserva el orden de inserción.

Todos los generadores definidos en el Capítulo 16, *Matrices*, están ahora disponibles a través del adaptador **CollectionData**. He aquí un ejemplo donde se utilizan dos de ellos:

```

//: containers/CollectionDataGeneration.java
// Utilización de los generadores definidos en el Capítulo 16, Matrices.
import java.util.*;
import net.mindview.util.*;

public class CollectionDataGeneration {
    public static void main(String[] args) {
        System.out.println(new ArrayList<String>(
            CollectionData.list( // Convenience method
                new RandomGenerator.String(9), 10)));
        System.out.println(new HashSet<Integer>(
            new CollectionData<Integer>(
                new RandomGenerator.Integer(), 10)));
    }
} /* Output:
[YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw,
HLGEahKox, rEqUCBbkI, naMesbtWH, kjUrUrkZPg, wsqPzDyCy]
[573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
*///:-
```

La longitud de la cadena de caracteres producida por **RandomGenerator.String** se controla mediante el argumento del constructor.

Generadores de mapas

Podemos usar la misma técnica para un objeto **Map**, pero eso requiere una clase **Pair** (par), ya que es necesario producir una pareja de objetos (una clave y un valor) en cada llamada al método **next()** de un generador, con el fin de llenar un contenedor **Map**:

```

//: net/mindview/util/Pair.java
package net.mindview.util;

public class Pair<K,V> {
    public final K key;
    public final V value;
    public Pair(K k, V v) {
        key = k;
        value = v;
    }
} //:-
```

Los campos **key** (clave) y **value** (valor) son de tipo público y final, de modo que **Pair** se convierte en un *objeto de transferencia de datos* de sólo lectura (o *mensajero*).

El adaptador para **Map** puede ahora utilizar distintas combinaciones de generadores, iterables y valores constantes para rellenar objetos de inicialización de **Map**:

```
//: net/mindview/util/MapData.java
// Un mapa lleno con datos utilizando un objeto generador.
package net.mindview.util;
import java.util.*;

public class MapData<K,V> extends LinkedHashMap<K,V> {
    // Generador de un único par:
    public MapData(Generator<Pair<K,V>> gen, int quantity) {
        for(int i = 0; i < quantity; i++) {
            Pair<K,V> p = gen.next();
            put(p.key, p.value);
        }
    }
    // Dos generadores separados:
    public MapData(Generator<K> genK, Generator<V> genV,
        int quantity) {
        for(int i = 0; i < quantity; i++) {
            put(genK.next(), genV.next());
        }
    }
    // Un generador de clave y un único valor:
    public MapData(Generator<K> genK, V value, int quantity) {
        for(int i = 0; i < quantity; i++) {
            put(genK.next(), value);
        }
    }
    // Un iterable y un generador de valor:
    public MapData(Iterable<K> genK, Generator<V> genV) {
        for(K key : genK) {
            put(key, genV.next());
        }
    }
    // Un iterable y un único valor:
    public MapData(Iterable<K> genK, V value) {
        for(K key : genK) {
            put(key, value);
        }
    }
    // Métodos genéricos de utilidad:
    public static <K,V> MapData<K,V>
    map(Generator<Pair<K,V>> gen, int quantity) {
        return new MapData<K,V>(gen, quantity);
    }
    public static <K,V> MapData<K,V>
    map(Generator<K> genK, Generator<V> genV, int quantity) {
        return new MapData<K,V>(genK, genV, quantity);
    }
    public static <K,V> MapData<K,V>
    map(Generator<K> genK, V value, int quantity) {
        return new MapData<K,V>(genK, value, quantity);
    }
    public static <K,V> MapData<K,V>
    map(Iterable<K> genK, Generator<V> genV) {
        return new MapData<K,V>(genK, genV);
    }
    public static <K,V> MapData<K,V>
```

```

    map<Iterable<K>, V value) {
        return new MapData<K,V>(genK, value);
    }
} //:-
```

Esto nos permite decidir si queremos utilizar un único objeto **Generator<Pair<K,V>>**, dos generadores separados, un generador y un valor constante, un objeto **Iterable** (lo que incluye cualquier tipo **Collection**) y un generador, o un objeto **Iterable** y un único valor. Los métodos genéricos de utilidad reducen la cantidad de texto que es necesario escribir a la hora de crear un objeto **MapData**.

He aquí un ejemplo en el que se utiliza **MapData**. El generador **Letters** también implementa **Iterable** produciendo un objeto **Iterator**; de esta forma, puede utilizarse para probar los métodos **MapData.map()** que funcionan con un objeto **Iterable**:

```

//: containers/MapDataTest.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Letters implements Generator<Pair<Integer, String>>,
    Iterable<Integer> {
    private int size = 9;
    private int number = 1;
    private char letter = 'A';
    public Pair<Integer, String> next() {
        return new Pair<Integer, String>(
            number++, "" + letter++);
    }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public Integer next() { return number++; }
            public boolean hasNext() { return number < size; }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}

public class MapDataTest {
    public static void main(String[] args) {
        // Generador de un par:
        print(MapData.map(new Letters(), 11));
        // Dos generadores separados:
        print(MapData.map(new CountingGenerator.Character(),
            new RandomGenerator.String(3), 8)));
        // Un generador de clave y un único valor:
        print(MapData.map(new CountingGenerator.Character(),
            "Value", 6));
        // Un Iterable y un generador de valor:
        print(MapData.map(new Letters(),
            new RandomGenerator.String(3)));
        // Un Iterable y un único valor:
        print(MapData.map(new Letters(), "Pop"));
    }
} /* Output:
{l=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
{a=YNz, b=brn, c=yGc, d=FOw, e=ZnT, f=cQr, g=Gse, h=GZM}
{a=Value, b=Value, c=Value, d=Value, e=Value, f=Value}
{l=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
{l=Pop, 2=Pop, 3=Pop, 4=Pop, 5=Pop, 6=Pop, 7=Pop, 8=Pop}
*///:-
```

Este ejemplo también utiliza los generadores del Capítulo 16, *Matrices*.

Podemos crear cualquier conjunto de datos generados para mapas o colecciones usando estas herramientas, y luego inicializar un objeto **Map** o **Collection** utilizando el constructor o los métodos **Map.putAll()** o **Collection.addAll()**.

Utilización de clases abstractas

Una solución alternativa al problema de generar datos de prueba para contenedores consiste en crear implementaciones personalizadas de **Collection** y **Map**. Cada contenedor de **java.util** tiene su propia clase abstracta que proporciona una implementación parcial de dicho contenedor, por lo que lo único que hace falta es implementar los métodos necesarios para obtener el contenedor deseado. Si el contenedor resultante es de sólo lectura, como suele suceder para los datos de prueba, el número de métodos que es necesario proporcionar se minimiza.

Aunque no resulta particularmente necesario en este caso, la siguiente solución también proporciona la oportunidad de ilustrar otro patrón de diseño: el patrón de diseño denominado *Peso mosca*. Utilizamos este patrón de diseño cuando la solución normal requiere demasiados objetos, o cuando la producción de objetos normales requiere demasiado espacio. El patrón de diseño *Peso mosca* externaliza parte del objeto de modo que, en lugar de que todo lo del objeto esté contenido en el propio objeto, parte del objeto o la totalidad del mismo se busca en una tabla externa, más eficiente (o se genera mediante algún otro cálculo que permita ahorrar espacio).

Un aspecto importante de este ejemplo consiste en demostrar lo relativamente simple que es crear sendos objetos **Map** y **Collection** personalizados heredando a partir de las clases de **java.util.Abstract**. Para crear un mapa **Map** de sólo lectura heredamos de **AbstractMap** e implementamos **entrySet()**. Para crear un conjunto **Set** de sólo lectura, heredamos de **AbstractSet** e implementamos **iterator()** y **size()**.

El conjunto de datos de este ejemplo es un mapa de los países del mundo y sus capitales². El método **capitals()** genera un mapa de países y capitales. El método **names()** genera una lista de los nombres de países. En ambos casos, podemos obtener un listado parcial proporcionando un argumento **int** que indique el tamaño deseado:

```
//: net/mindview/util/Countries.java
// Mapas y listas "Peso mosca" de datos de ejemplo.
package net.mindview.util;
import java.util.*;
import static net.mindview.util.Print.*;

public class Countries {
    public static final String[][] DATA = {
        // África
        {"ALGERIA", "Algiers"}, {"ANGOLA", "Luanda"}, {"BENIN", "Porto-Novo"}, {"BOTSWANA", "Gaberone"}, {"BULGARIA", "Sofia"}, {"BURKINA FASO", "Ouagadougou"}, {"BURUNDI", "Bujumbura"}, {"CAMEROON", "Yaounde"}, {"CAPE VERDE", "Praia"}, {"CENTRAL AFRICAN REPUBLIC", "Bangui"}, {"CHAD", "N'djamena"}, {"COMOROS", "Moroni"}, {"CONGO", "Brazzaville"}, {"DJIBOUTI", "Djibouti"}, {"EGYPT", "Cairo"}, {"EQUATORIAL GUINEA", "Malabo"}, {"ERITREA", "Asmara"}, {"ETHIOPIA", "Addis Ababa"}, {"GABON", "Libreville"}, {"THE GAMBIA", "Banjul"}, {"GHANA", "Accra"}, {"GUINEA", "Conakry"}, {"BISSAU", "Bissau"}, {"COTE D'IVOIR (IVORY COAST)", "Yamoussoukro"}, {"KENYA", "Nairobi"}, {"LESOTHO", "Maseru"}, {"LIBERIA", "Monrovia"}, {"LIBYA", "Tripoli"}, {"MADAGASCAR", "Antananarivo"}, {"MALAWI", "Lilongwe"}, {"MALI", "Bamako"}, {"MAURITANIA", "Nouakchott"}, {"MAURITIUS", "Port Louis"}, {"MOROCCO", "Rabat"}};
}
```

² Estos datos se han extraído de Internet. A lo largo del tiempo diversos lectores me han enviado correcciones.

```

("MOZAMBIQUE", "Maputo"), {"NAMIBIA", "Windhoek"},  

("NIGER", "Niamey"), {"NIGERIA", "Abuja"},  

("RWANDA", "Kigali"),  

("SAO TOME E PRINCIPE", "Sao Tome"),  

("SENEGAL", "Dakar"), {"SEYCHELLES", "Victoria"},  

("SIERRA LEONE", "Freetown"), {"SOMALIA", "Mogadishu"},  

("SOUTH AFRICA", "Pretoria/Cape Town"),  

("SUDAN", "Khartoum"),  

("SWAZILAND", "Mbabane"), {"TANZANIA", "Dodoma"},  

("TOGO", "Lome"), {"TUNISIA", "Tunis"},  

("UGANDA", "Kampala"),  

("DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",  

"Kinshasa"),  

("ZAMBIA", "Lusaka"), {"ZIMBABWE", "Harare"},  

// Asia  

("AFGHANISTAN", "Kabul"), {"BAHRAIN", "Manama"},  

("BANGLADESH", "Dhaka"), {"BHUTAN", "Thimphu"},  

("BRUNEI", "Bandar Seri Begawan"),  

("CAMBODIA", "Phnom Penh"),  

("CHINA", "Beijing"), {"CYPRUS", "Nicosia"},  

("INDIA", "New Delhi"), {"INDONESIA", "Jakarta"},  

("IRAN", "Tehran"), {"IRAQ", "Baghdad"},  

("ISRAEL", "Jerusalem"), {"JAPAN", "Tokyo"},  

("JORDAN", "Amman"), {"KUWAIT", "Kuwait City"},  

("LAOS", "Vientiane"), {"LEBANON", "Beirut"},  

("MALAYSIA", "Kuala Lumpur"), {"THE MALDIVES", "Male"},  

("MONGOLIA", "Ulan Bator"),  

("MYANMAR (BURMA)", "Rangoon"),  

("NEPAL", "Katmandu"), {"NORTH KOREA", "P'yongyang"},  

("OMAN", "Muscat"), {"PAKISTAN", "Islamabad"},  

("PHILIPPINES", "Manila"), {"QATAR", "Doha"},  

("SAUDI ARABIA", "Riyadh"), {"SINGAPORE", "Singapore"},  

("SOUTH KOREA", "Seoul"), {"SRI LANKA", "Colombo"},  

("SYRIA", "Damascus"),  

("TAIWAN (REPUBLIC OF CHINA)", "Taipei"),  

("THAILAND", "Bangkok"), {"TURKEY", "Ankara"},  

("UNITED ARAB EMIRATES", "Abu Dhabi"),  

("VIETNAM", "Hanoi"), {"YEMEN", "Sana'a"},  

// Australia y Oceania  

("AUSTRALIA", "Canberra"), {"FIJI", "Suva"},  

("KIRIBATI", "Bairiki"),  

("MARSHALL ISLANDS", "Dalap-Uliga-Darrit"),  

("MICRONESIA", "Palikir"), {"NAURU", "Yaren"},  

("NEW ZEALAND", "Wellington"), {"PALAU", "Koror"},  

("PAPUA NEW GUINEA", "Port Moresby"),  

("SOLOMON ISLANDS", "Honaira"), {"TONGA", "Nuku'alofa"},  

("TUVALU", "Fongafale"), {"VANUATU", "< Port-Vila"},  

("WESTERN SAMOA", "Apia"),  

// Europa del Este y la Unión Soviética  

("ARMENIA", "Yerevan"), {"AZERBAIJAN", "Baku"},  

("BELARUS (BYELORUSSIA)", "Minsk"),  

("GEORGIA", "Tbilisi"),  

("KAZAKSTAN", "Almaty"), {"KYRGYZSTAN", "Alma-Ata"},  

("MOLDOVA", "Chisinau"), {"RUSSIA", "Moscow"},  

("TAJIKISTAN", "Dushanbe"), {"TURKMENISTAN", "Ashkabad"},  

("UKRAINE", "Kyiv"), {"UZBEKISTAN", "Tashkent"},  

// Europa  

("ALBANIA", "Tirana"), {"ANDORRA", "Andorra la Vella"},  

("AUSTRIA", "Vienna"), {"BELGIUM", "Brussels"},

```

```

    {"BOSNIA", "-"}, {"HERZEGOVINA", "Sarajevo"},  

    {"CROATIA", "Zagreb"}, {"CZECH REPUBLIC", "Prague"},  

    {"DENMARK", "Copenhagen"}, {"ESTONIA", "Tallinn"},  

    {"FINLAND", "Helsinki"}, {"FRANCE", "Paris"},  

    {"GERMANY", "Berlin"}, {"GREECE", "Athens"},  

    {"HUNGARY", "Budapest"}, {"ICELAND", "Reykjavik"},  

    {"IRELAND", "Dublin"}, {"ITALY", "Rome"},  

    {"LATVIA", "Riga"}, {"LIECHTENSTEIN", "Vaduz"},  

    {"LITHUANIA", "Vilnius"}, {"LUXEMBOURG", "Luxembourg"},  

    {"MACEDONIA", "Skopje"}, {"MALTA", "Valletta"},  

    {"MONACO", "Monaco"}, {"MONTENEGRERO", "Podgorica"},  

    {"THE NETHERLANDS", "Amsterdam"}, {"NORWAY", "Oslo"},  

    {"POLAND", "Warsaw"}, {"PORTUGAL", "Lisbon"},  

    {"ROMANIA", "Bucharest"}, {"SAN MARINO", "San Marino"},  

    {"SERBIA", "Belgrade"}, {"SLOVAKIA", "Bratislava"},  

    {"SLOVENIA", "Ljubljana"}, {"SPAIN", "Madrid"},  

    {"SWEDEN", "Stockholm"}, {"SWITZERLAND", "Berne"},  

    {"UNITED KINGDOM", "London"}, {"VATICAN CITY", "---"},  

    // América del Norte y América Central  

    {"ANTIGUA AND BARBUDA", "Saint John's"},  

    {"BAHAMAS", "Nassau"},  

    {"BARBADOS", "Bridgetown"}, {"BELIZE", "Belmopan"},  

    {"CANADA", "Ottawa"}, {"COSTA RICA", "San Jose"},  

    {"CUBA", "Havana"}, {"DOMINICA", "Roseau"},  

    {"DOMINICAN REPUBLIC", "Santo Domingo"},  

    {"EL SALVADOR", "San Salvador"},  

    {"GRENADA", "Saint George's"},  

    {"GUATEMALA", "Guatemala City"},  

    {"HAITI", "Port-au-Prince"},  

    {"HONDURAS", "Tegucigalpa"}, {"JAMAICA", "Kingston"},  

    {"MEXICO", "Mexico City"}, {"NICARAGUA", "Managua"},  

    {"PANAMA", "Panama City"}, {"ST. KITTS", "-"},  

    {"NEVIS", "Basseterre"}, {"ST. LUCIA", "Castries"},  

    {"ST. VINCENT AND THE GRENADINES", "Kingstown"},  

    {"UNITED STATES OF AMERICA", "Washington, D.C."},  

    // América del Sur  

    {"ARGENTINA", "Buenos Aires"},  

    {"BOLIVIA", "Sucre (legal)/La Paz (administrative)"},  

    {"BRAZIL", "Brasilia"}, {"CHILE", "Santiago"},  

    {"COLOMBIA", "Bogota"}, {"ECUADOR", "Quito"},  

    {"GUYANA", "Georgetown"}, {"PARAGUAY", "Asuncion"},  

    {"PERU", "Lima"}, {"SURINAME", "Paramaribo"},  

    {"TRINIDAD AND TOBAGO", "Port of Spain"},  

    {"URUGUAY", "Montevideo"}, {"VENEZUELA", "Caracas"}  

};

// Utilización de AbstractMap implementando entrySet()
private static class FlyweightMap
extends AbstractMap<String, String> {
    private static class Entry
        implements Map.Entry<String, String> {
            int index;
            Entry(int index) { this.index = index; }
            public boolean equals(Object o) {
                return DATA[index][0].equals(o);
            }
            public String getKey() { return DATA[index][0]; }
            public String getValue() { return DATA[index][1]; }
            public String setValue(String value) {
                throw new UnsupportedOperationException();
            }
        }
    }
}

```

```

        }
        public int hashCode() {
            return DATA[index][0].hashCode();
        }
    }
    // Utilizar AbstractSet implementando size() e iterator()
    static class EntrySet
    extends AbstractSet<Map.Entry<String, String>> {
        private int size;
        EntrySet(int size) {
            if(size < 0)
                this.size = 0;
            // No puede tener un tamaño mayor que la matriz:
            else if(size > DATA.length)
                this.size = DATA.length;
            else
                this.size = size;
        }
        public int size() { return size; }
        private class Iter
        implements Iterator<Map.Entry<String, String>> {
            // Sólo un objeto Entry por cada Iterator:
            private Entry entry = new Entry(-1);
            public boolean hasNext() {
                return entry.index < size - 1;
            }
            public Map.Entry<String, String> next() {
                entry.index++;
                return entry;
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        }
        public
        Iterator<Map.Entry<String, String>> iterator() {
            return new Iter();
        }
    }
    private static Set<Map.Entry<String, String>> entries =
        new EntrySet(DATA.length);
    public Set<Map.Entry<String, String>> entrySet() {
        return entries;
    }
}
// Crear un mapa parcial de un número 'size' de países:
static Map<String, String> select(final int size) {
    return new FlyweightMap() {
        public Set<Map.Entry<String, String>> entrySet() {
            return new EntrySet(size);
        }
    };
}
static Map<String, String> map = new FlyweightMap();
public static Map<String, String> capitals() {
    return map; // El mapa completo
}
public static Map<String, String> capitals(int size) {
    return select(size); // Un mapa parcial
}

```

```

static List<String> names =
    new ArrayList<String>(map.keySet());
// Todos los nombres:
public static List<String> names() { return names; }
// Un lista parcial:
public static List<String> names(int size) {
    return new ArrayList<String>(select(size).keySet());
}
public static void main(String[] args) {
    print(capitals(10));
    print(names(10));
    print(new HashMap<String, String>(capitals(3)));
    print(new LinkedHashMap<String, String>(capitals(3)));
    print(new TreeMap<String, String>(capitals(3)));
    print(new Hashtable<String, String>(capitals(3)));
    print(new HashSet<String>(names(6)));
    print(new LinkedHashSet<String>(names(6)));
    print(new TreeSet<String>(names(6)));
    print(new ArrayList<String>(names(6)));
    print(new LinkedList<String>(names(6)));
    print(capitals().get("BRAZIL"));
}
} /* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo,
BOTSWANA=Gaberone, BULGARIA=Sofia, BURKINA FASO
=Ouagadougou, BURUNDI=Bujumbura, CAMEROON=Yaounde,
CAPE VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
Brasilia
*///:-

```

La matriz bidimensional de cadenas de caracteres **DATA** es pública, por lo que se la puede emplear en cualquier otro lugar. **FlyweightMap** debe implementar el método **entrySet()**, que requiere tanto una implementación personalizada de **Set** como una clase **Map.Entry** personalizada. He aquí parte de la solución “peso mosca”: cada objeto **Map.Entry** simplemente almacena su índice, en lugar de almacenar la clave y el valor reales. Cuando invocamos **getKey()** o **getValue()**, utiliza el índice para devolver el elemento de **DATA** apropiado. El contenedor **EntrySet** asegura que su tamaño (**size**) no sea superior al de **DATA**.

Podemos ver la otra parte de la solución “peso mosca” implementada en **EntrySetIterator**. En lugar de crear un objeto **Map.Entry** para cada pareja de datos en **DATA**, sólo existe un objeto **Map.Entry** por cada iterador. El objeto **Entry** se utiliza como una ventana a los datos: sólo contiene un índice (**index**) a la matriz estática de cadenas de caracteres. Cada vez que invocamos **next()** para iterador, se incrementa la variable índice de un objeto **Entry**, de modo que apunte a la siguiente pareja de elementos y luego el único objeto **Entry** de ese objeto **Iterator** es devuelto por **next()**.³

El método **select()** genera un contenedor **FlyweightMap** que contiene un conjunto **EntrySet** del tamaño deseado, el cual se usa en los métodos **capitals()** y **names()** sobrecargados que se ilustran en **main()**.

³ Los mapas de **java.util** realizan copias masivas utilizando **getKey()** y **getValue()**, de modo que esta solución funciona. Si un mapa personalizado fuera a copiar simplemente el objeto **Map.Entry** completo entonces esta técnica causaría problemas.

Para algunas pruebas, el tamaño limitado de **Countries** es un problema. Podemos usar la misma técnica para generar contenadores personalizados inicializados que tengan un conjunto de datos de cualquier tamaño. A continuación se muestra una clase de tipo **List** que puede tener cualquier tamaño y que se preinicializa con datos de tipo **Integer**:

```
//: net/mindview/util/CountingIntegerList.java
// Lista de cualquier longitud con datos de ejemplo.
package net.mindview.util;
import java.util.*;

public class CountingIntegerList
extends AbstractList<Integer> {
    private int size;
    public CountingIntegerList(int size) {
        this.size = size < 0 ? 0 : size;
    }
    public Integer get(int index) {
        return Integer.valueOf(index);
    }
    public int size() { return size; }
    public static void main(String[] args) {
        System.out.println(new CountingIntegerList(30));
    }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:-
```

Para crear una lista de sólo lectura a partir de **AbstractList**, hay que implementar **get()** y **size()**. De nuevo, se utiliza una solución de tipo “peso mosca”: **get()** produce el valor cuando lo pedimos, por lo que la lista no tiene, en la práctica, que estar rellena.

He aquí un mapa que contiene valores de tipo **Integer** y **String** univocos preinicializados; puede tener cualquier tamaño:

```
//: net/mindview/util/CountingMapData.java
// Mapa de longitud limitada con datos de ejemplo.
package net.mindview.util;
import java.util.*;

public class CountingMapData
extends AbstractMap<Integer, String> {
    private int size;
    private static String[] chars =
        "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
        .split(" ");
    public CountingMapData(int size) {
        if(size < 0) this.size = 0;
        this.size = size;
    }
    private static class Entry
    implements Map.Entry<Integer, String> {
        int index;
        Entry(int index) { this.index = index; }
        public boolean equals(Object o) {
            return Integer.valueOf(index).equals(o);
        }
        public Integer getKey() { return index; }
        public String getValue() {
            return
                chars[index % chars.length] +
                Integer.toString(index / chars.length);
        }
    }
}
```

```

public String setValue(String value) {
    throw new UnsupportedOperationException();
}
public int hashCode() {
    return Integer.valueOf(index).hashCode();
}
}
public Set<Map.Entry<Integer, String>> entrySet() {
    // LinkedHashSet mantiene el orden de inicialización:
    Set<Map.Entry<Integer, String>> entries =
        new LinkedHashSet<Map.Entry<Integer, String>>();
    for(int i = 0; i < size; i++)
        entries.add(new Entry(i));
    return entries;
}
public static void main(String[] args) {
    System.out.println(new CountingMapData(60));
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0, 10=K0, 11=L0, 12=M0, 13=N0,
14=O0, 15=P0, 16=Q0, 17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0, 25=Z0,
26=A1, 27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1, 33=H1, 34=I1, 35=J1, 36=K1, 37=L1,
38=M1, 39=N1, 40=O1, 41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1, 49=X1,
50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2, 57=F2, 58=G2, 59=H2}
*///:-
```

Aquí, se utiliza un contenedor **LinkedHashSet** en lugar de crear una clase **Set** personalizada, por lo que la solución de tipo “peso mosca” no está completamente implementada.

- Ejercicio 1:** (1) Cree una lista (inténtelo tanto con **ArrayList** como con **LinkedList**) y rellénela utilizando **Countries**. Ordene la lista e imprimala, luego aplique **Collections.shuffle()** a la lista repetidamente, imprimiéndola cada vez de modo que pueda ver cómo el método **shuffle()** aleatoriza la lista de manera diferente cada vez.
- Ejercicio 2:** (2) Defina un mapa y un conjunto que contengan todos los países que comiencen por ‘A’.
- Ejercicio 3:** (1) Utilizando **Countries**, rellene un conjunto múltiples veces con los mismos datos y verifique que el conjunto termina conteniendo una única copia de cada instancia. Pruebe a hacer lo mismo con **HashSet**, **LinkedHashSet** y **TreeSet**.
- Ejercicio 4:** (2) Cree un inicializador de **Collection** que abra un archivo y lo descomponga en palabras usando **TextFile**, y luego emplee las palabras como origen de los datos para el contenedor de **Collection** resultante. Demuestre que la solución funciona.
- Ejercicio 5:** (3) Modifique **CountingMapData.java** para implementar completamente la solución de tipo “peso mosca” añadiendo una clase **EntrySet** personalizada como la de **Countries.java**.

Funcionalidad de las colecciones

La tabla de la página siguiente muestra todo lo que se puede hacer con un contenedor de tipo **Collection** (sin incluir los métodos que provienen automáticamente de **Object**), y, por tanto, todo lo que se puede hacer con un contenedor de tipo **Set** o **List** (**List** tiene también funcionalidad adicional). Los contenedores de tipo **Map** no heredan de **Collection**, por lo que los trataremos por separado.

Observe que no existe ningún método **get()** para seleccionar elementos mediante acceso aleatorio. Eso se debe a que **Collection** también incluye **Set**, que mantiene su propia ordenación interna (y que hace, por tanto, que las búsquedas por acceso aleatorio no tengan ningún significado). Por tanto, si queremos examinar los elementos de un contenedor **Collection**, debemos utilizar un iterador.

El siguiente ejemplo ilustra todos estos métodos. Aunque estos métodos funcionen con cualquier cosa que implemente **Collection**, se utiliza un contenedor **ArrayList** como “mínimo denominador común”:

| | |
|--|---|
| boolean add(T) | Garantiza que el contenedor almacene el argumento que es del tipo genérico T. Devuelve false si no añade el argumento (éste es un método “opcional”, descrito en la sección siguiente). |
| boolean addAll(Collection<? extends T>) | Añade todos los elementos del argumento. Devuelve true si se añade algún elemento. (“Opcional”). |
| void clear() | Elimina todos los elementos del contenedor (“Opcional”). |
| boolean contains(T) | Devuelve true si el contenedor almacena el argumento que es de tipo genérico T. |
| Boolean containsAll(Collection<?>) | Devuelve true si el contenedor almacena todos los elementos del argumento. |
| boolean isEmpty() | Devuelve true si el contenedor no tiene ningún elemento. |
| Iterator<T> iterator() | Devuelve un objeto Iterator<T> que puede utilizarse para recorrer los elementos del contenedor. |
| Boolean remove(Object) | Si el argumento está en el contenedor, se elimina una instancia de dicho elemento. Devuelve true si se ha producido alguna eliminación (“Opcional”). |
| boolean removeAll(Collection<?>) | Elimina todos los elementos contenidos en el argumento. Devuelve true si se ha producido alguna eliminación (“Opcional”). |
| Boolean retainAll(Collection<?>) | Retiene únicamente los elementos que estén contenidos en el argumento (una “intersección”, según la teoría de conjuntos). Devuelve true si se ha producido algún cambio. (“Opcional”). |
| int size() | Devuelve el número de elementos del contenedor. |
| Object[] toArray() | Devuelve una matriz que contiene todos los elementos del contenedor. |
| <T> T[] toArray(T[] a) | Devuelve una matriz que contiene todos los elementos del contenedor. El tipo en tiempo de ejecución del resultado es el que corresponde a la matriz argumento a en lugar de ser simplemente Object . |

```

//: containers/CollectionMethods.java
// Cosas que se pueden hacer con todas las colecciones.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CollectionMethods {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        c.addAll(Countries.names(6));
        c.add("ten");
        c.add("eleven");
        print(c);
        // Hacer una matriz a partir de la lista:
        Object[] array = c.toArray();
        // Hacer una matriz de objetos String a partir de la lista:
        String[] str = c.toArray(new String[0]);
        // Encontrar elementos max y min elements; esto significa
        // diferentes cosas dependiendo de la forma
        // en que esté implementada la interfaz Comparable:
        print("Collections.max(c) = " + Collections.max(c));
        print("Collections.min(c) = " + Collections.min(c));
    }
}

```

```

// Añadir una colección a otra colección
Collection<String> c2 = new ArrayList<String>();
c2.addAll(Countries.names(6));
c.addAll(c2);
print(c);
c.remove(Countries.DATA[0][0]);
print(c);
c.remove(Countries.DATA[1][0]);
print(c);
// Eliminar todos los componentes contenidos
// en la colección proporcionada como argumento:
c.removeAll(c2);
print(c);
c.addAll(c2);
print(c);
// ¿Está un elemento en esta colección?
String val = Countries.DATA[3][0];
print("c.contains(" + val + ") = " + c.contains(val));
// ¿Está una colección dentro de esta colección?
print("c.containsAll(c2) = " + c.containsAll(c2));
Collection<String> c3 =
    ((List<String>)c).subList(3, 5);
// Conservar todos los elementos que estén tanto
// en c2 como en c3 (una intersección de conjuntos):
c2.retainAll(c3);
print(c2);
// Eliminar todos los elementos de
// c2 que también aparezcan en c3:
c2.removeAll(c3);
print("c2.isEmpty() = " + c2.isEmpty());
c = new ArrayList<String>();
c.addAll(Countries.names(6));
print(c);
c.clear(); // Eliminar todos los elementos
print("after c.clear():" + c);
}
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven]
Collections.max(c) = ten
Collections.min(c) = ALGERIA
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven, ALGERIA,
ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven, ALGERIA, ANGOLA,
BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ten, eleven]
[ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA,
BURKINA FASO]
c.contains(BOTSWANA) = true
c.containsAll(c2) = true
[ANGOLA, BENIN]
c2.isEmpty() = true
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
after c.clear(): []
*///:-

```

Se crean contenedores **ArrayList** que contienen diferentes conjuntos de datos y se los generaliza a objetos **Collection**, por lo que queda claro que no se está utilizando más que la interfaz **Collection**. **main()** utiliza una serie de ejercicios simples para ilustrar todos los métodos de **Collection**.

Las siguientes secciones del capítulo describen las diversas implementaciones de **List**, **Set** y **Map** y se indica en cada caso (con un asterisco) cuál debería ser la opción preferida. Las descripciones de las clases heredadas **Vector**, **Stack** y **Hashtable** se dejan para el final del capítulo; aunque no deberían utilizarse estas clases, lo más probable es que tengamos oportunidad de verlas al leer código antiguo.

Operaciones opcionales

Los métodos que realizan diversos tipos de operaciones de adición y eliminación son *operaciones opcionales* en la interfaz **Collection**. Esto significa que la clase implementadora no está obligada a proporcionar definiciones de estos métodos que funcionen.

Se trata de una forma bastante inusual de definir una interfaz. Como hemos visto, una interfaz es una especie de contrato dentro del diseño orientado a objetos. Ese contrato dice: “Independientemente de cómo decidas implementar esta interfaz, te garantizo que puedes enviar estos mensajes al objeto”⁴. Pero el hecho de que exista una operación “opcional” viola este principio fundamental; ya que implica que al invocar algunos métodos *no* se obtendrá un comportamiento con significado. En lugar de ello, esos métodos generarán excepciones. Podría parecer que estamos renunciando a la seguridad de tipos en tiempo de compilación.

Pero las cosas no son en realidad así. Si una operación es opcional, el compilador sigue imponiendo la restricción de que sólo se puedan invocar los métodos especificados en dicha interfaz. Esto no se parece a los lenguajes dinámicos, en los que se puede invocar cualquier método para cualquier objeto y averiguar en tiempo de ejecución si una llamada concreta funciona⁵. Además, la mayoría de los métodos que toman un contenedor **Collection** como argumento sólo *leen* de dicha colección, y todos los métodos de “lectura” de **Collection** *no* son opcionales.

¿Para qué queríamos definir métodos como “opcionales”? Al hacerlo así, evitamos una explosión de interfaces en el diseño. Otros diseños de bibliotecas de contenedores siempre parecen terminar en una confusa plétora de interfaces, para describir cada una de las variantes del tema principal. Ni siquiera resulta posible capturar todos los casos especiales en las interfaces, porque alguien puede siempre inventar una nueva interfaz. La técnica de “operaciones no soportadas” permite conseguir un objetivo importante de la biblioteca de contenedores de Java: los contenedores son simples de aprender y de utilizar. Las operaciones no soportadas son un caso especial que pueden retardarse hasta que sean necesarias. Sin embargo, para que esta técnica funcione:

1. La excepción **UnsupportedOperationException** debe ser un suceso raro. En otras palabras, para la mayoría de las clases, todas las operaciones deben funcionar, y sólo en casos especiales esa operación no estará soportada. Esto es así en la biblioteca de contenedores de Java, ya que las clases que se utilizan el 99 por ciento del tiempo (**ArrayList**, **LinkedList**, **HashSet** y **HashMap**, así como las otras implementaciones concretas) soportan todas las operaciones. El diseño proporciona una “puerta trasera” si queremos crear un nuevo contenedor de tipo **Collection** sin proporcionar definiciones significativas para todos los métodos de la interfaz **Collection**, sin que por ello ese nuevo contenedor deje de encajar dentro de la biblioteca existente.
2. Cuando una operación *no esté soportada*, puede existir una probabilidad razonable de que aparezca una excepción **UnsupportedOperationException** en tiempo de implementación, en lugar de después de haber enviado el producto al cliente. Después de todo, esa excepción indica que hay un error de programación: se ha empleado una implementación incorrectamente.

Merece la pena reseñar que las operaciones no soportadas sólo son detectables en tiempo de ejecución y representan, por tanto, una comprobación dinámica de tipos. Si el lector proviene de un lenguaje con tipos estáticos como C++, Java puede parecer simplemente otro lenguaje con tipos estáticos. Por supuesto que Java *tiene* comprobación estática de tipos, pero también tiene una cantidad significativa de comprobación de tipos dinámica, por lo que resulta difícil decir si Java es un tipo de lenguaje u otro. Una vez que comience a entender esto, verá otros ejemplos de comprobación dinámica de tipos en Java.

⁴ Utilizo aquí el término **interfaz** tanto para describir la palabra clave **interface** como el significado más general de “los métodos soportados por una clase o subclase”.

⁵ Aunque esto suene extraño y posiblemente sea inútil al ser descrito de esta forma, hemos visto, especialmente en el Capítulo 14, *Información de tipos*, que esta especie de comportamiento dinámico puede ser importante.

Operaciones no soportadas

Un origen bastante común de la aparición de operaciones no soportadas es cuando disponemos de un contenedor que está respaldado por una estructura de datos de tamaño fijo. Obtenemos dichos contenedores cuando transformamos una matriz en una lista con el método `Arrays.asList()`. También podemos *decidir* que algún contenedor (incluyendo los mapas) genere la excepción `UnsupportedOperationException` utilizando los métodos “no modificables” de la clase `Collections`. Este ejemplo ilustra ambos casos:

```
//: containers/Unsupported.java
// Operaciones no soportadas en los contenedores de Java.
import java.util.*;

public class Unsupported {
    static void test(String msg, List<String> list) {
        System.out.println("--- " + msg + " ---");
        Collection<String> c = list;
        Collection<String> subList = list.subList(1, 8);
        // Copia de la sublista:
        Collection<String> c2 = new ArrayList<String>(subList);
        try { c.addAll(c2); } catch(Exception e) {
            System.out.println("addAll(): " + e);
        }
        try { c.removeAll(c2); } catch(Exception e) {
            System.out.println("removeAll(): " + e);
        }
        try { c.clear(); } catch(Exception e) {
            System.out.println("clear(): " + e);
        }
        try { c.add("X"); } catch(Exception e) {
            System.out.println("add(): " + e);
        }
        try { c.addAll(c2); } catch(Exception e) {
            System.out.println("addAll(): " + e);
        }
        try { c.remove("C"); } catch(Exception e) {
            System.out.println("remove(): " + e);
        }
        // El método List.set() modifica el valor pero no
        // cambia el tamaño de la estructura de datos:
        try {
            list.set(0, "X");
        } catch(Exception e) {
            System.out.println("List.set(): " + e);
        }
    }
    public static void main(String[] args) {
        List<String> list =
            Arrays.asList("A B C D E F G H I J K L".split(" "));
        test("Modifiable Copy", new ArrayList<String>(list));
        test("Arrays.asList()", list);
        test("unmodifiableList()", Collections.unmodifiableList(
            new ArrayList<String>(list)));
    }
} /* Output:
--- Modifiable Copy ---
--- Arrays.asList() ---
addAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
```

```

clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*///:-*

```

Como **Arrays.asList()** produce un contenedor **List** que está respaldado por una matriz de tamaño fijo, tiene bastante sentido que las únicas operaciones soportadas sean aquellas que no cambien el *tamaño* de la matriz. Cualquier método que provoque un cambio del tamaño de la estructura de datos subyacente generará una excepción **UnsupportedOperationException**, para indicar que se ha producido una llamada a un método no soportado (un error de programación).

Observe que siempre podemos pasar el resultado de **Arrays.asList()** como argumento de un constructor de cualquier colección (o utilizar el método **addAll()** o el método estático **Collections.addAll()**) para crear un contenedor normal que permita el uso de todos los métodos; esta técnica se muestra en la primera llamada a **test()** de **main()**. Dicha llamada produce una nueva estructura de datos subyacente de tamaño variable.

Los métodos “no modificables” de la clase **Collections** envuelven el contenedor en un *proxy* que genera una excepción **UnsupportedOperationException** si se realiza cualquier operación que modifique el contenedor de alguna forma. El objetivo de utilizar estos métodos es producir un objeto contenedor “constante”. Posteriormente, describiremos la lista completa de métodos “no modificables” de **Collections**.

El último bloque **try** de **test()** examina el método **set()** que forma parte de **List**. Este bloque es interesante, porque podemos ver cómo nos resulta útil la granularidad de la técnica de “operaciones no soportadas”: La “interfaz” resultante puede variar en un método entre el objeto devuelto por **Arrays.asList()** y el que devuelve **Collections.unmodifiableList()**. **Arrays.asList()** devuelve una lista de tamaño fijo, mientras que **Collections.unmodifiableList()** genera una lista que no se puede modificar. Como puede verse analizando la salida, resulta posible *modificar* los elementos de la lista devuelta por **Arrays.asList()**, porque esto no viola la característica de “tamaño fijo” de dicha lista. Pero es obvio que el resultado de **unmodifiableList()** no debe ser modificable de ninguna forma. Si usáramos interfaces, esto habría requerido dos interfaces adicionales: una con un método **set()** funcional y otra sin dicho método. Asimismo, podrían requerirse interfaces adicionales para diversos subtipos no modificables de **Collection**.

La documentación de un método que tome un contenedor como argumento deberá especificar cuáles de los métodos opcionales debe implementarse.

Ejercicio 6: (2) Observe que **List** tiene operaciones “opcionales” adicionales que no están incluidas en **Collection**. Escriba una versión de **Unsupported.java** que pruebe estas operaciones opcionales adicionales.

Funcionalidad de List

Como hemos visto, el contenedor **List** básico es bastante simple de usar: la mayor parte del tiempo nos limitaremos a invocar **add()** para insertar los objetos, o a utilizar **get()** para extraerlos de uno en uno o a llamar a **iterator()** para obtener un objeto **Iterator** para la secuencia.

Los métodos del siguiente ejemplo cubren, cada uno de ellos, un grupo distinto de actividades: las cosas que todas las listas pueden hacer (**basicTest()**); el desplazamiento por el contenedor con un iterador (**iterMotion()**) comparado con la modificación de valores mediante un iterador (**iterManipulation()**), la comprobación de los efectos de la manipulación de una lista (**testVisual()**); y las operaciones disponibles únicamente para contenedores de tipo **LinkedLists**:

```

//: containers/Lists.java
// Cosas que se pueden hacer con las listas.
import java.util.*;
import net.mindview.util.*;

```

```

import static net.mindview.util.Print.*;

public class Lists {
    private static boolean b;
    private static String s;
    private static int i;
    private static Iterator<String> it;
    private static ListIterator<String> lit;
    public static void basicTest(List<String> a) {
        a.add(1, "x"); // Agregar en la posición 1
        a.add("x"); // Agergar al final
        // Agregar una colección:
        a.addAll(Countries.names(25));
        // Agregar una colección comenzando en la posición 3:
        a.addAll(3, Countries.names(25));
        b = a.contains("1"); // Is it in there?
        // ¿Está toda la colección contenida?
        b = a.containsAll(Countries.names(25));
        // Las listas permiten el acceso aleatorio, que es poco
        // costoso para ArrayList, y caro para LinkedList:
        s = a.get(1); // Obtener objeto (con tipo) en la posición 1
        i = a.indexOf("1"); // Determinar índice del objeto
        b = a.isEmpty(); // ¿Hay algún elemento?
        it = a.iterator(); // Iterador normal
        lit = a.listIterator(); // ListIterator
        lit = a.listIterator(3); // Empezar en la posición 3
        i = a.lastIndexOf("1"); // Última correspondencia
        a.remove(1); // Eliminar posición 1
        a.remove("3"); // Eliminar este objeto
        a.set(1, "y"); // Asignar "y" a la posición 1
        // Conservar todo lo que forme parte del argumento
        // (la intersección de dos conjuntos):
        a.retainAll(Countries.names(25));
        // Eliminar todo lo que forme parte del argumento:
        a.removeAll(Countries.names(25));
        i = a.size(); // ¿Qué tamaño tiene?
        a.clear(); // Eliminar todos los elementos
    }
    public static void iterMotion(List<String> a) {
        ListIterator<String> it = a.listIterator();
        b = it.hasNext();
        b = it.hasPrevious();
        s = it.next();
        i = it.nextInt();
        s = it.previous();
        i = it.previousIndex();
    }
    public static void iterManipulation(List<String> a) {
        ListIterator<String> it = a.listIterator();
        it.add("47");
        // Hay que desplazarse a un elemento después de add():
        it.next();
        // Eliminar el elemento situado después del recién generado:
        it.remove();
        // Hay que desplazarse un elemento después de remove():
        it.next();
        // Cambiar el elemento situado después del borrado:
        it.set("47");
    }
}

```

```

public static void testVisual(List<String> a) {
    print(a);
    List<String> b = Countries.names(25);
    print("b = " + b);
    a.addAll(b);
    a.addAll(b);
    print(a);
    // Insertar, eliminar y reemplazar elementos
    // utilizando ListIterator:
    ListIterator<String> x = a.listIterator(a.size()/2);
    x.add("one");
    print(a);
    print(x.next());
    x.remove();
    print(x.next());
    x.set("47");
    print(a);
    // Recorrer la lista hacia atrás:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
        println(x.previous() + " ");
    print();
    print("testVisual finished");
}
// Hay algunas cosas que sólo los contenedores
// tipo LinkedLists pueden hacer:
public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<String>();
    ll.addAll(Countries.names(25));
    print(ll);
    // Tratarlo como una pila, insertando:
    ll.addFirst("one");
    ll.addFirst("two");
    print(ll);
    // Como si se "consultara" la cima de la pila:
    print(ll.getFirst());
    // Como si se extrajera de una pila:
    print(ll.removeFirst());
    print(ll.removeFirst());
    // Tratarlo como una cola, extrayendo elementos
    // del final:
    print(ll.removeLast());
    print(ll);
}
public static void main(String[] args) {
    // Crear y llenar una nueva lista cada vez:
    basicTest(
        new LinkedList<String>(Countries.names(25)));
    basicTest(
        new ArrayList<String>(Countries.names(25)));
    iterMotion(
        new LinkedList<String>(Countries.names(25)));
    iterMotion(
        new ArrayList<String>(Countries.names(25)));
    iterManipulation(
        new LinkedList<String>(Countries.names(25)));
    iterManipulation(
        new ArrayList<String>(Countries.names(25)));
    testVisual(

```

```

    new LinkedList<String>(Countries.names(25));
    testLinkedList();
}
/* (Execute to see output) */:-
```

En **basicTest()** e **iterMotion()**, las llamadas se realizan en orden para mostrar la sintaxis apropiada, y aunque se captura el valor de retorno, dicho valor no se utiliza. En algunos casos, el valor de retorno no se captura en absoluto. Consulte los detalles completos de utilización de cada uno de estos métodos en la documentación del JDK antes de utilizarlos.

Ejercicio 7: (4) Cree tanto un contenedor **ArrayList** como otro de tipo **LinkedList** y rellénelos utilizando el generador **Countries.names()**. Imprima cada lista utilizando un iterador normal y luego inserte una lista en la otra empleando un iterador **ListIterator**, realizando las inserciones en una de cada dos posiciones. Ahora realice la inserción comenzando por el final de la primera lista y desplazándose hacia atrás.

Ejercicio 8: (7) Cree una clase que represente una lista genérica simplemente enlazada denominada **SList**, la cual, para hacer las cosas simples, *no* implemente la interfaz **List**. Cada objeto **Link** (enlace) de la lista puede tener una referencia al siguiente elemento de la lista, pero no al anterior (**LinkedList**, por contraste, es una lista doblemente enlazada, lo que significa que mantiene enlaces en ambas direcciones). Cree su propio iterador **SListIterator** que, de nuevo por simplicidad, *no* implemente **ListIterator**. El único método de **SList** aparte de **toString()**, debe ser **iterator()**, que generará un elemento **SListIterator**. La única manera de insertar y eliminar elementos de un contenedor **SList** es mediante **SListIterator**. Escriba el código necesario para ilustrar el uso de **SList**.

Conjuntos y orden de almacenamiento

Los ejemplos del contenedor **Set** del Capítulo 11, *Almacenamiento de objetos*, proporcionan una buena introducción a las operaciones que pueden realizarse con los conjuntos básicos. Sin embargo, dichos ejemplos utilizan, por comodidad, tipos de Java predefinidos como **Integer** y **String**, que estaban diseñados para poderlos utilizar dentro de contenedores. A la hora de crear nuestros propios tipos, debemos tener en cuenta que un contenedor **Set** necesita una forma de mantener el orden de almacenamiento. El cómo se mantenga ese orden de almacenamiento varía de una implementación de **Set** a otra. Por tanto, las diferentes implementaciones de **Set** no sólo tienen diferentes comportamientos, sino también diferentes requisitos, adaptados al tipo de objeto que puede introducirse dentro de un contenedor **Set** concreto:

| | |
|-----------------------|---|
| Set (interfaz) | Cada elemento que se añada al conjunto debe ser diferente; en caso contrario, el objeto Set no añadirá el elemento duplicado. Los elementos añadidos a un conjunto deben al menos definir equals() con el fin de establecer la unicidad de los objetos. Set tiene exactamente la misma interfaz que Collection . La interfaz Set no garantiza que vaya a mantener sus elementos en ningún orden determinado. |
| HashSet* | Para los conjuntos en los que el tiempo de búsqueda sea importante. Los elementos deben definir también hashCode() . |
| TreeSet | Un conjunto ordenado respaldado por un árbol. De esta forma, se puede extraer una secuencia ordenada de un conjunto. Los elementos también deben implementar la interfaz Comparable . |
| LinkedHashSet | Tiene la velocidad de búsqueda de un contenedor HashSet , pero mantiene internamente el orden en que se añaden los elementos (el orden de inserción) utilizando una lista enlazada. Por tanto, cuando iteramos a través del conjunto, los resultados aparecen en orden de inserción. Los elementos también deben definir hashCode() . |

El asterisco en **HashSet** indica que, en ausencia de otras restricciones, ésta debe ser la opción preferida, porque está optimizada para conseguir la máxima velocidad.

La definición de **hashCode()** se describirá más adelante en el capítulo. Es necesario crear un método **equals()** para el almacenamiento tanto de tipo **hash** como de tipo árbol, pero el método **hashCode()** sólo es necesario si se va a almacenar la clase en un contenedor **HashSet** (lo cual es bastante probable, ya que esa debería ser nuestra primera elección como implementación del conjunto) o **LinkedHashSet**. Sin embargo, con el fin de mantener un buen estilo de programación, conviene sustituir siempre **hashCode()** cada vez que se sustituya **equals()**.

Este ejemplo ilustra los métodos que deben definirse para utilizar convenientemente un tipo de datos con una implementación concreta de Set:

```

//: containers/TypesForSets.java
// Métodos necesarios para almacenar nuestro propio
// tipo de datos en un conjunto.
// tipos de datos.
import java.util.*;

class SetType {
    int i;
    public SetType(int n) { i = n; }
    public boolean equals(Object o) {
        return o instanceof SetType && (i == ((SetType)o).i);
    }
    public String toString() { return Integer.toString(i); }
}

class HashType extends SetType {
    public HashType(int n) { super(n); }
    public int hashCode() { return i; }
}

class TreeType extends SetType
implements Comparable<TreeType> {
    public TreeType(int n) { super(n); }
    public int compareTo(TreeType arg) {
        return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
    }
}

public class TypesForSets {
    static <T> Set<T> fill(Set<T> set, Class<T> type) {
        try {
            for(int i = 0; i < 10; i++)
                set.add(
                    type.getConstructor(int.class).newInstance(i));
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        return set;
    }
    static <T> void test(Set<T> set, Class<T> type) {
        fill(set, type);
        fill(set, type); // Intentamos añadir duplicados
        fill(set, type);
        System.out.println(set);
    }
    public static void main(String[] args) {
        test(new HashSet<HashType>(), HashType.class);
        test(new LinkedHashSet<HashType>(), HashType.class);
        test(new TreeSet<TreeType>(), TreeType.class);
        // Cosas que no funcionan:
        test(new HashSet<SetType>(), SetType.class);
        test(new HashSet<TreeType>(), TreeType.class);
        test(new LinkedHashSet<SetType>(), SetType.class);
        test(new LinkedHashSet<TreeType>(), TreeType.class);
        try {
            test(new TreeSet<SetType>(), SetType.class);
        } catch(Exception e) {

```

```

        System.out.println(e.getMessage());
    }
    try {
        test(new TreeSet<HashType>(), HashType.class);
    } catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
/* Output: (Sample)
[2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 9, 7, 5, 1, 2, 6, 3, 0, 7, 2, 4, 4, 7, 9, 1, 3, 6, 2,
4, 3, 0, 5, 0, 8, 8, 8, 6, 5, 1]
[0, 5, 5, 6, 5, 0, 3, 1, 9, 8, 4, 2, 3, 9, 7, 3, 4, 4, 0,
7, 1, 9, 6, 2, 1, 8, 2, 8, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
java.lang.ClassCastException: SetType cannot be cast to java.lang.Comparable
java.lang.ClassCastException: HashType cannot be cast to java.lang.Comparable
*///:-
```

Para demostrar qué métodos son necesarios para un contenedor **Set** concreto y para evitar, al mismo tiempo, la duplicación de código, hemos creado tres clases. La clase base, **SetType**, simplemente almacena un valor **int** y lo imprime mediante **toString()**. Puesto que todas las clases almacenadas en conjuntos deben tener un método **equals()**, también incluimos dicho método en la clase base. La igualdad está basada en el valor de la variable **int i**.

HashType hereda de **SetType** y añade el método **hashCode()** necesario para insertar un objeto en una implementación **hash** de un conjunto.

La interfaz **Comparable**, implementada mediante **TreeType**, es necesaria si vamos a usar un objeto en algún tipo de contenedor ordenado, como por ejemplo **SortedSet** (del cual **TreeSet** es la única implementación). En **compareTo()**, observe que *no hemos usado* la forma “simple y obvia” **return i-i2**. Aunque se trata de un error de programación común, sólo funcionaría adecuadamente si **i** e **i2** fueran valores **int** “sin signo” (si Java *tuviera* una palabra clave “**unsigned**”, que no es el caso). La expresión no funciona para los valores **int** con signo de Java, que no son lo suficientemente grandes como para representar la diferencia de dos valores **int** con signo. Si **i** es un entero positivo de gran tamaño y **j** es un entero negativo de gran tamaño, **i-j** producirá un desbordamiento y devolverá un valor negativo, lo cual es incorrecto.

Normalmente, lo que queremos es que el método **compareTo()** permita obtener una ordenación natural que sea coherente con el método **equals()**. Si **equals()** devuelve **true** para una comparación concreta, entonces **compareTo()** debería devolver un resultado igual a cero para dicha comparación, y si **equals()** devuelve **false** para una comparación, entonces **compareTo()** debería dar un resultado distinto de cero para dicha comparación.

En **TypesForSets**, tanto **fill()** como **test()** se definen utilizando genéricos, con el fin de evitar la duplicación de código. Para verificar el comportamiento de un conjunto, **test()** invoca **fill()** sobre el conjunto de prueba **set** tres veces, tratando de introducir objetos duplicados. El método **fill()** toma un contenedor **Set** de cualquier tipo y un objeto **Class** del mismo tipo. Utiliza el objeto **Class** para descubrir el constructor que admite un argumento **int**, e invoca dicho constructor para añadir elementos al conjunto.

Analizando la salida, podemos ver que **HashSet** mantiene los elementos en alguna especie de orden misterioso (que entenderemos claramente más adelante en el capítulo), **LinkedHashSet** mantiene los elementos en el orden en que fueron insertados y **TreeSet** mantiene los elementos ordenados (debido a la forma en que se implementa **compareTo()**, dicho orden resulta ser descendente).

Si tratamos de utilizar tipos de datos que no soporten apropiadamente las operaciones necesarias con conjuntos que requieren dichas operaciones, el funcionamiento es incorrecto. Al insertar un objeto **SetType** o **TreeType**, que no incluye un método **hashCode()** redefinido, en una implementación **hash** se generan valores duplicados, con lo que se viola la característica principal de un conjunto. Este error es bastante molesto, porque ni siquiera se produce un error en tiempo de ejecución: sin

embargo, el método `hashCode()` predeterminado es legítimo, y por tanto es un comportamiento legal, aun cuando sea incorrecto. La única forma fiable de garantizar la corrección de ese programa consiste en incorporar código de pruebas en el sistema final de producción. (consulte el suplemento en <http://MindView.net/Books/BetterJava> para obtener más información).

Si tratamos de utilizar un tipo de datos que no implemente `Comparable` en un contenedor `TreeSet`, se obtiene un resultado más definido: se genera una excepción cuando el contenedor `TreeSet` trata de utilizar el objeto como si fuera de tipo `Comparable`.

SortedSet

Los contenedores `SortedSet` garantizan que sus elementos estén ordenados, lo que permite proporcionar funcionalidad adicional mediante los siguientes métodos, definidos en la interfaz `SortedSet`:

Comparator comparator(): genera el objeto `Comparator` utilizado para este contenedor `Set`, o `null` para el caso de una ordenación natural.

Object first(): devuelve el elemento más bajo.

Object last(): devuelve el elemento más alto.

SortedSet subSet(fromElement, toElement): genera una vista de este contenedor `Set` de los elementos comprendidos entre `fromElement`, incluido, y `toElement`, excluido.

SortedSet headSet(toElement): genera una vista de este contenedor `Set` con los elementos inferiores a `toElement`.

SortedSet tailSet(fromElement): genera una vista de este contenedor `Set` con los elementos superiores o iguales a `fromElement`.

He aquí un ejemplo simple:

```
//: containers/SortedSetDemo.java
// Lo que se puede hacer con un contenedor TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class SortedSetDemo {
    public static void main(String[] args) {
        SortedSet<String> sortedSet = new TreeSet<String>();
        Collections.addAll(sortedSet,
            "one two three four five six seven eight"
                .split(" "));
        print(sortedSet);
        String low = sortedSet.first();
        String high = sortedSet.last();
        print(low);
        print(high);
        Iterator<String> it = sortedSet.iterator();
        for(int i = 0; i <= 6; i++) {
            if(i == 3) low = it.next();
            if(i == 6) high = it.next();
            else it.next();
        }
        print(low);
        print(high);
        print(sortedSet.subSet(low, high));
        print(sortedSet.headSet(high));
        print(sortedSet.tailSet(low));
    }
} /* Output:
[eight, five, four, one, seven, six, three, two]
eight
two
```

```

one
two
[one, seven, six, three]
[eight, five, four, one, seven, six, three]
[one, seven, six, three, two]
*///:-
```

Observe que **SortedSet** quiere decir “ordenado de acuerdo con la función de comparación del objeto”, no “orden de inserción”. El orden de inserción puede conservarse utilizando **LinkedHashSet**.

Ejercicio 9: (2) Utilice **RandomGenerator.String** para llenar un contenedor **TreeSet**, pero empleando ordenación alfabética. Imprima el contenedor **TreeSet** para verificar la ordenación.

Ejercicio 10: (7) Utilizando un contenedor **LinkedList** como implementación subyacente, defina su propio contenedor **SortedSet**.

Colas

Dejando aparte las aplicaciones de concurrencia, las dos únicas implementaciones de colas en Java SE5 son **LinkedList** y **PriorityQueue**, que se diferencian por el comportamiento en lo que respecta a la ordenación más que por el rendimiento. He aquí un ejemplo básico donde se ilustran la mayoría de las implementaciones de **Queue** (no todas ellas funcionan en este ejemplo), incluyendo las colas basadas en concurrencia. Los elementos se insertan por un extremo y se extraen por el otro:

```

//: containers/QueueBehavior.java
// Compara el comportamiento de algunas de las colas
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

public class QueueBehavior {
    private static int count = 10;
    static <T> void test(Queue<T> queue, Generator<T> gen) {
        for(int i = 0; i < count; i++)
            queue.offer(gen.next());
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    static class Gen implements Generator<String> {
        String[] s = {"one two three four five six seven " +
                      "eight nine ten").split(" ");
        int i;
        public String next() { return s[i++]; }
    }
    public static void main(String[] args) {
        test(new LinkedList<String>(), new Gen());
        test(new PriorityQueue<String>(), new Gen());
        test(new ArrayBlockingQueue<String>(count), new Gen());
        test(new ConcurrentLinkedQueue<String>(), new Gen());
        test(new LinkedBlockingQueue<String>(), new Gen());
        test(new PriorityBlockingQueue<String>(), new Gen());
    }
} /* Output:
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
*///:-
```

Podemos ver que, con la excepción de las colas con prioridad, un contenedor `Queue` devuelve los elementos exactamente en el mismo orden en que fueron insertados en la cola.

Colas con prioridad

Ya hemos proporcionado una breve introducción a las colas con prioridad en el Capítulo 11, *Almacenamiento de objetos*. Un problema más interesante que los que allí analizamos sería el de una lista de tareas que hacer, en la que cada objeto contenga una cadena de caracteres y sendos valores de prioridad principal y secundaria. La ordenación de esta lista está, de nuevo, controlada por la implementación de `Comparable`:

```
//: containers/ToDoList.java
// Un uso más complejo de PriorityQueue.
import java.util.*;

class ToDoList extends PriorityQueue<ToDoList.ToDoItem> {
    static class ToDoItem implements Comparable<ToDoItem> {
        private char primary;
        private int secondary;
        private String item;
        public ToDoItem(String td, char pri, int sec) {
            primary = pri;
            secondary = sec;
            item = td;
        }
        public int compareTo(ToDoItem arg) {
            if(primary > arg.primary)
                return +1;
            if(primary == arg.primary)
                if(secondary > arg.secondary)
                    return +1;
                else if(secondary == arg.secondary)
                    return 0;
                return -1;
        }
        public String toString() {
            return Character.toString(primary) +
                secondary + ":" + item;
        }
    }
    public void add(String td, char pri, int sec) {
        super.add(new ToDoItem(td, pri, sec));
    }
    public static void main(String[] args) {
        ToDoList toDoList = new ToDoList();
        toDoList.add("Empty trash", 'C', 4);
        toDoList.add("Feed dog", 'A', 2);
        toDoList.add("Feed bird", 'B', 7);
        toDoList.add("Mow lawn", 'C', 3);
        toDoList.add("Water lawn", 'A', 1);
        toDoList.add("Feed cat", 'B', 1);
        while(!toDoList.isEmpty())
            System.out.println(toDoList.remove());
    }
} /* Output:
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
*///:-
```

Podemos ver cómo la ordenación de los elementos se realiza automáticamente gracias a la cola con prioridad.

Ejercicio 11: (2) Cree una clase que contenga un objeto **Integer** que se inicialice con un valor comprendido entre 0 y 100 utilizando **java.util.Random**. Implemente **Comparable** empleando este campo **Integer**. Rellene una cola de tipo **PriorityQueue** con objetos de dicha clase y extraiga los valores usando **poll()** para demostrar que se obtiene el orden deseado.

Colas dobles

Una *cola doble* es similar a una cola normal, pero se pueden añadir y eliminar elementos de cualquier extremo. Existen métodos en **LinkedList** que soportan las operaciones de doble cola, pero no existe ninguna interfaz explícita para una doble cola en las bibliotecas estándar de Java. Por tanto, **LinkedList** no puede implementar esta interfaz y no resulta posible efectuar una generalización a una interfaz **Deque** (cola doble), a diferencia de lo que podríamos hacer con **Queue** en el ejercicio anterior. Sin embargo, podemos crear una clase **Deque** empleando el mecanismo de composición y exponer, simplemente, los métodos relevantes de **LinkedList**:

```
//: net/mindview/util/Deque.java
// Creación de una cola doble a partir de LinkedList.
package net.mindview.util;
import java.util.*;

public class Deque<T> {
    private LinkedList<T> deque = new LinkedList<T>();
    public void addFirst(T e) { deque.addFirst(e); }
    public void addLast(T e) { deque.addLast(e); }
    public T getFirst() { return deque.getFirst(); }
    public T getLast() { return deque.getLast(); }
    public T removeFirst() { return deque.removeFirst(); }
    public T removeLast() { return deque.removeLast(); }
    public int size() { return deque.size(); }
    public String toString() { return deque.toString(); }
    // Y otros métodos según sean necesarios...
} ///:-
```

Si utilizamos esta clase **Deque** en nuestros propios programas, es posible que descubramos que necesitamos añadir otros métodos para que la clase resulte práctica.

He aquí un ejemplo simple de prueba de la clase **Deque**:

```
//: containers/DequeTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DequeTest {
    static void fillTest(Deque<Integer> deque) {
        for(int i = 20; i < 27; i++)
            deque.addFirst(i);
        for(int i = 50; i < 55; i++)
            deque.addLast(i);
    }
    public static void main(String[] args) {
        Deque<Integer> di = new Deque<Integer>();
        fillTest(di);
        print(di);
        while(di.size() != 0)
            printnb(di.removeFirst() + " ");
        print();
        fillTest(di);
        while(di.size() != 0)
```

```

        println(di.removeLast() + " ");
    }
} /* Output:
[26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
26 25 24 23 22 21 20 50 51 52 53 54
54 53 52 51 50 20 21 22 23 24 25 26
*//*:-

```

Resulta poco probable que tengamos que insertar y extraer elementos por ambos extremos, por lo que **Deque** no se emplea tan comúnmente como **Queue**.

Mapas

Como vimos en el Capítulo 11, *Almacenamiento de objetos*, la idea básica de un mapa (también denominada *matriz asociativa*) es la de almacenar asociaciones clave-valor (pares), de modo que se pueda buscar un valor utilizando una clave. La biblioteca estándar de Java contiene diferentes implementaciones básicas de mapas: **HashMap**, **TreeMap**, **LinkedHashMap**, **WeakHashMap**, **ConcurrentHashMap** y **IdentityHashMap**. Todas tienen la misma interfaz básica **Map**, pero difieren en cuanto a su comportamiento, incluyendo la eficiencia, el orden en el que se almacenan y se presentan los pares, el tiempo que el mapa conserva los objetos, el funcionamiento de los mapas en los programas multihebra y el modo de determinar la igualdad de claves. La gran cantidad de implementaciones de la interfaz **Map** nos indica la importancia de este tipo de contenedor.

Para comprender mejor los mapas resulta útil ver cómo se construye una matriz asociativa. He aquí una implementación extremadamente simple:

```

//: containers/AssociativeArray.java
// Asocia claves con valores.
import static net.mindview.util.Print.*;

public class AssociativeArray<K,V> {
    private Object[][] pairs;
    private int index;
    public AssociativeArray(int length) {
        pairs = new Object[length][2];
    }
    public void put(K key, V value) {
        if(index >= pairs.length)
            throw new ArrayIndexOutOfBoundsException();
        pairs[index++] = new Object[]{key, value};
    }
    @SuppressWarnings("unchecked")
    public V get(K key) {
        for(int i = 0; i < index; i++)
            if(key.equals(pairs[i][0]))
                return (V)pairs[i][1];
        return null; // Clave no encontrada
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < index; i++) {
            result.append(pairs[i][0].toString());
            result.append(" : ");
            result.append(pairs[i][1].toString());
            if(i < index - 1)
                result.append("\n");
        }
        return result.toString();
    }
}

```

```

public static void main(String[] args) {
    AssociativeArray<String, String> map =
        new AssociativeArray<String, String>(6);
    map.put("sky", "blue");
    map.put("grass", "green");
    map.put("ocean", "dancing");
    map.put("tree", "tall");
    map.put("earth", "brown");
    map.put("sun", "warm");
    try {
        map.put("extra", "object"); // Past the end
    } catch(ArrayIndexOutOfBoundsException e) {
        print("Too many objects!");
    }
    print(map);
    print(map.get("ocean"));
}
} /* Output:
Too many objects!
sky : blue
grass : green
ocean : dancing
tree : tall
earth : brown
sun : warm
dancing
*///:-

```

Los métodos esenciales en una matriz asociativa son `put()` y `get()`, pero para facilitar la visualización se ha sustituido `toString()` con el fin de imprimir los pares clave-valor. Para demostrar que funciona, `main()` carga una matriz `AssociativeArray` con pares de cadenas de caracteres e imprime el mapa resultante, extrayendo a continuación con `get()` uno de los valores.

Para utilizar el método `get()`, se pasa la clave (`key`) que queremos buscar y el método devuelve como resultado el valor asociado, o bien devuelve `null` si no se puede encontrar esa clave. El método `get()` utiliza la que posiblemente sea la técnica menos eficiente imaginable con el fin de localizar el valor: comienza por la parte superior de la matriz y usa `equals()` para comparar las claves. Pero el objetivo del ejemplo es la simplicidad no la eficiencia.

Por tanto, la versión anterior es instructiva, pero no resulta muy eficiente y tiene un tamaño fijo, lo que da como resultado una implementación poco flexible. Afortunadamente, los mapas de `java.util` no tienen estos problemas y pueden emplearse perfectamente en el ejemplo anterior.

Ejercicio 12: (1) Utilice mapas de tipo `HashMap`, `TreeMap` y `LinkedHashMap` en el método `main()` de `AssociativeArray.java`.

Ejercicio 13: (4) Utilice `AssociativeArray.java` para crear un contador de apariciones de palabras, que establezca la correspondencia entre un valor de tipo `String` y un valor de tipo `Integer`. Empleando la utilidad `net.mindview.util.TextFile` de este libro, abra un archivo de texto y extraiga las palabras de dicho archivo utilizando los espacios en blanco y los signos de puntuación como delimitadores. Cuente el número de veces que cada palabra aparece en dicho archivo.

Rendimiento

El rendimiento es una de las cuestiones fundamentales en los mapas, y resulta demasiado lento utilizar una búsqueda lineal en `get()` a la hora de intentar localizar una clave. Es en este aspecto donde `HashMap` permite acelerar las operaciones. En lugar de realizar una lenta búsqueda de la clave, este contenedor utiliza un valor especial denominado *código hash*. El *código hash* es una forma de tomar una cierta información contenida en el objeto en cuestión y transformarla en un valor `int` “relativamente unívoco” que se utilizará para representar dicho objeto. `hashCode()` es un método de la clase raíz

Object, por lo que todos los objetos Java pueden generar un código *hash*. Un contenedor **HashMap** toma el código *hash* devuelto por **hashCode()** y lo utiliza para localizar rápidamente la clave. Esto permite mejorar enormemente el rendimiento.⁶

He aquí las implementaciones básicas de **Map**. El asterisco situado junto a **HashMap** indica que, en ausencia de otras restricciones, ésta debería ser la opción preferida, ya que está optimizada para maximizar la velocidad. Las otras implementaciones enfatizan otras características, por lo que no resultan tan rápidas como **HashMap**.

| | |
|--------------------------|---|
| HashMap* | Implementación basada en una tabla <i>hash</i> (utilice esta clase en lugar de Hashtable). Proporciona un rendimiento de tiempo constante para la inserción y localización de pares. El rendimiento puede ajustarse mediante constructores que permiten fijar la <i>capacidad</i> y el <i>factor de carga</i> de la tabla <i>hash</i> . |
| LinkedHashMap | Como HashMap , pero cuando se realiza una iteración a su través, se extraen los pares en orden de inserción o en orden LRU (<i>least-recently-used</i> , menos recientemente utilizado). Es ligeramente más lento que HashMap , salvo cuando se está realizando una iteración, en cuyo caso es más rápido debido a que se emplea una lista enlazada para mantener la ordenación interna. |
| TreeMap | Implementación basada en un árbol rojo-negro. Cuando se examinen las claves o las parejas, estarán ordenadas (la ordenación está determinada por Comparable o Comparator). La ventaja de un contenedor TreeMap es que los resultados se obtienen ordenados. TreeMap es el único tipo de mapa con el método subMap() , que permite devolver una parte del árbol. |
| WeakHashMap | Un mapa de <i>claves débiles</i> que permite eliminar los objetos a los que hace referencia el mapa; está diseñado para resolver ciertos tipos especiales de problemas. Si no se conserva ninguna referencia a una clave concreta fuera del mapa, dicha clave puede ser depurada de la memoria. |
| ConcurrentHashMap | Un mapa preparado para hebras de programación que no incluye bloqueo de sincronización. Hablaremos de este tema en el Capítulo 21, <i>Concurrencia</i> . |
| IdentityHashMap | Un mapa <i>hash</i> que utiliza == en lugar de equals() para comparar las claves. Se utiliza para resolver ciertos tipos especiales de problemas, no para uso general. |

El almacenamiento *hash* es la forma que más comúnmente se utiliza para almacenar elementos en un mapa. Posteriormente veremos cómo funciona este tipo de almacenamiento.

Los requisitos para las claves utilizadas en un mapa son iguales que para los elementos de un conjunto. Ya hemos visto una ilustración de estos requisitos en **TypesForSets.java**. Todas las claves deben disponer de un método **equals()**. Si la clave se utiliza en un mapa *hash*, también debe disponer de un método **hashCode()** apropiado. Si la clave se utiliza en un mapa de tipo **TreeMap**, deberá implementar **Comparable**.

El siguiente ejemplo muestra las operaciones disponibles en la interfaz **Map**, utilizando el conjunto de datos de prueba **CountingMapData** anteriormente definido:

```
//: containers/Maps.java
// Cosas que se pueden hacer con mapas.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

⁶ Si este tipo de utilización sigue sin satisfacer sus requisitos de rendimiento, puede acelerar todavía más las búsquedas en tablas escribiendo su propio contenedor **Map** y personalizándolo para los tipos particulares de datos que esté utilizando, con el fin de evitar los retardos asociados con las proyecciones de tipo hacia y desde **Object**. Para obtener niveles todavía mejores de rendimiento, los entusiastas de la velocidad pueden consultar el libro de Donald Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Segunda edición, con el fin de sustituir las listas de segmentos con desbordamiento por matrices que tienen dos ventajas adicionales: pueden optimizarse de acuerdo con las características de almacenamiento del disco y permiten ahorrar la mayor parte del tiempo invertido en crear y depurar los registros individuales.

```

public static void printKeys(Map<Integer,String> map) {
    printnb("Size = " + map.size() + ", ");
    printnb("Keys: ");
    print(map.keySet()); // Generar un conjunto con las claves
}
public static void test(Map<Integer,String> map) {
    print(map.getClass().getSimpleName());
    map.putAll(new CountingMapData(25));
    // El mapa presenta el comportamiento de un conjunto para las claves:
    map.putAll(new CountingMapData(25));
    printKeys(map);
    // Generación de una colección con los valores:
    printnb("Values: ");
    print(map.values());
    print(map);
    print("map.containsKey(11): " + map.containsKey(11));
    print("map.get(11): " + map.get(11));
    print("map.containsValue(\"F0\"): "
        + map.containsValue("F0"));
    Integer key = map.keySet().iterator().next();
    print("First key in map: " + key);
    map.remove(key);
    printKeys(map);
    map.clear();
    print("map.isEmpty(): " + map.isEmpty());
    map.putAll(new CountingMapData(25));
    // Las operaciones efectuadas sobre el conjunto modifican el mapa:
    map.keySet().removeAll(map.keySet());
    print("map.isEmpty(): " + map.isEmpty());
}
public static void main(String[] args) {
    test(new HashMap<Integer,String>());
    test(new TreeMap<Integer,String>());
    test(new LinkedHashMap<Integer,String>());
    test(new IdentityHashMap<Integer,String>());
    test(new ConcurrentHashMap<Integer,String>());
    test(new WeakHashMap<Integer,String>());
}
} /* Output:
HashMap
Size = 25, Keys: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14,
24, 4, 19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
Values: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0,
E0, T0, L0, S0, D0, M0, R0, C0, N0, U0, K0, F0, A0]
{15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0,
1=B0, 14=O0, 24=Y0, 4=E0, 19=T0, 11=L0, 18=S0, 3=D0, 12=M0,
17=R0, 2=C0, 13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
First key in map: 15
Size = 24, Keys: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4,
19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
map.isEmpty(): true
map.isEmpty(): true
*/

```

El método **printKeys()** muestra cómo generar una vista de tipo **Collection** para un mapa. El método **keySet()** genera un conjunto con las claves del mapa. Debido a las mejoras en el soporte de impresión introducidas en Java SE5, podemos imprimir:

mir los resultados del método `values()`, que genera una colección con todos los valores del mapa (observe que las claves debe ser univocas, pero que los valores pueden contener duplicados). Puesto que estas colecciones están respaldadas por el propio mapa, cualquier cambio efectuado en una colección se reflejará en el mapa asociado.

El resto del programa proporciona ejemplos simples de cada operación efectuada con el mapa y prueba cada tipo básico de mapa.

Ejercicio 14: (3) Demuestre que `java.util.Properties` funciona en el programa anterior.

SortedMap

Si utilizamos un contenedor **SortedMap** (del cual la única implementación disponible es **TreeMap**), se garantiza que las claves estarán ordenadas, lo que permite proporcionar funcionalidad adicional con los siguientes métodos de la interfaz **SortedMap**:

Comparator comparator(): genera el comparador utilizado para este mapa o **null** si se utiliza la ordenación natural.

T firstKey(): devuelve la clave más baja.

T lastKey(): devuelve la clave más alta.

SortedMap subMap(fromKey, toKey): genera una vista de este mapa, con las claves comprendidas entre **fromKey**, incluido, y **toKey**, excluido.

SortedMap headMap(toKey): genera una vista de este mapa con las claves que sean inferiores a **toKey**.

SortedMap tailMap(fromKey): genera una vista de este mapa con las claves que sean iguales o superiores a **fromKey**.

He aquí un ejemplo similar a **SortedSetDemo.java** donde se ilustra este comportamiento adicional de los mapas **TreeMap**:

```
//: containers/SortedMapDemo.java
// Lo que se puede hacer con TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class SortedMapDemo {
    public static void main(String[] args) {
        TreeMap<Integer, String> sortedMap =
            new TreeMap<Integer, String>(new CountingMapData(10));
        print(sortedMap);
        Integer low = sortedMap.firstKey();
        Integer high = sortedMap.lastKey();
        print(low);
        print(high);
        Iterator<Integer> it = sortedMap.keySet().iterator();
        for(int i = 0; i <= 6; i++) {
            if(i == 3) low = it.next();
            if(i == 6) high = it.next();
            else it.next();
        }
        print(low);
        print(high);
        print(sortedMap.subMap(low, high));
        print(sortedMap.headMap(high));
        print(sortedMap.tailMap(low));
    }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
0
9
```

```

3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*///:-
```

Aquí, los pares se almacenan ordenados según la clave. Puesto que existe el concepto de orden en **TreeMap**, el concepto de “posición” también tiene sentido, así que se pueden tener submapas y también se puede determinar el primer elemento y el último.

LinkedHashMap

LinkedHashMap utiliza un almacenamiento *hash* para conseguir velocidad, pero también genera las parejas en orden de inserción cuando se recorre el mapa (`System.out.println()` itera a través del mapa, por lo que se pueden comprobar los resultados de ese recorrido). Además, **LinkedHashMap** puede configurarse mediante el constructor para utilizar un algoritmo LRU (*least-recently-used*) basado en el acceso a los elementos, de modo que los elementos a los que se haya accedido menos (y sean, por tanto, candidatos a la eliminación) aparezcan al principio de la lista. Esto permite crear fácilmente programas que realizan una limpieza periódica con el fin de ahorrar espacio. He aquí un ejemplo donde se ilustran ambas características:

```

//: containers/LinkedHashMapDemo.java
// Lo que se puede hacer con LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<Integer, String> linkedMap =
            new LinkedHashMap<Integer, String>(
                new CountingMapData(9));
        print(linkedMap);
        // Orden LRU:
        linkedMap =
            new LinkedHashMap<Integer, String>(16, 0.75f, true);
        linkedMap.putAll(new CountingMapData(9));
        print(linkedMap);
        for(int i = 0; i < 6; i++) // Provocar accesos:
            linkedMap.get(i);
        print(linkedMap);
        linkedMap.get(0);
        print(linkedMap);
    }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*///:-
```

Podemos ver, analizando la salida, que las parejas se recorren por orden de inserción, incluso para la versión LRU. Sin embargo, después de acceder exclusivamente a los primeros seis elementos en la versión LRU, los tres últimos elementos se desplazan al principio de la lista. Después, cuando se vuelve a acceder a “0”, dicho elemento se desplaza al final de la lista.

Almacenamiento y códigos hash

Los ejemplos del Capítulo 11, *Almacenamiento de objetos*, utilizan clases predefinidas como claves para **HashMap**. Dichos ejemplos funcionaban porque esas clases predefinidas contenían todos los elementos necesarios para poder comportarse correctamente como claves.

Uno de los problemas más comunes es el que se produce cuando creamos nuestras propias clases para utilizarlas como claves para mapas de tipo **HashMap**, y nos olvidamos de añadir los elementos necesarios. Por ejemplo, considere un sistema de predicción meteorológica basado en el estudio del comportamiento de las marmotas donde se hagan corresponder objetos **Groundhog** (marmota) con objetos **Prediction** (predicción meteorológica). La tarea parece sencilla: basta con crear las dos clases y usar **Groundhog** como clave y **Prediction** como valor:

```

//: containers/Groundhog.java
// Parece plausible, pero no funciona como clave para HashMap.

public class Groundhog {
    protected int number;
    public Groundhog(int n) { number = n; }
    public String toString() {
        return "Groundhog #" + number;
    }
} //:-

//: containers/Prediction.java
// Predicción del clima mediante marmotas.
import java.util.*;

public class Prediction {
    private static Random rand = new Random(47);
    private boolean shadow = rand.nextDouble() > 0.5;
    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
} //:-

//: containers/SpringDetector.java
// ¿Qué tiempo hará?
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SpringDetector {
    // Utiliza Groundhog o una clase derivada de Groundhog:
    public static <T extends Groundhog>
    void detectSpring(Class<T> type) throws Exception {
        Constructor<T> ghog = type.getConstructor(int.class);
        Map<Groundhog, Prediction> map =
            new HashMap<Groundhog, Prediction>();
        for(int i = 0; i < 10; i++)
            map.put(ghog.newInstance(i), new Prediction());
        print("map = " + map);
        Groundhog gh = ghog.newInstance(3);
        print("Looking up prediction for " + gh);
        if(map.containsKey(gh))
            print(map.get(gh));
        else
            print("Key not found: " + gh);
    }
    public static void main(String[] args) throws Exception {
        detectSpring(Groundhog.class);
    }
} /* Output:

```

```

map = [Groundhog #3=Early Spring!, Groundhog #7=Early Spring!,
Groundhog #5=Early Spring!, Groundhog #9=Six more weeks of Winter!,
Groundhog #8=Six more weeks of Winter!, Groundhog #0=Six more weeks
of Winter!, Groundhog #6=Early Spring!, Groundhog #4=Six more weeks
of Winter!, Groundhog #1=Six more weeks of Winter!, Groundhog #2=Early Spring!]
Looking up prediction for Groundhog #3
Key not found: Groundhog #3
*//:-_

```

A cada objeto **Groundhog** se le da un número identificador, de modo que se puede buscar un objeto **Prediction** en el contenedor **HashMap** diciendo: "Dame el objeto **Prediction** asociado con el objeto asociado **Groundhog #3**". La clase **Prediction** contiene un valor de tipo **boolean** que se inicializa utilizando **java.util.random()** y un método **toString()** que interpreta el resultado por nosotros. El método **detectSpring()** (detectar la primavera) se crea empleando el mecanismo de reflexión para instancias y usa la clase **Groundhog** o cualquier clase derivada de **Groundhog**. Esto nos será útil posteriormente, cuando heredemos una nueva clase a partir de **Groundhog** para resolver el problema ilustrado en este ejemplo.

Rellenamos un objeto **HashMap** como objetos **Groundhog** y sus objetos **Prediction** asociados. El mapa **HashMap** se imprime para poder ver que ha sido llenado. Después, se utiliza un objeto **Groundhog** con número identificador igual a 3 como clave para buscar la predicción para **Groundhog #3** (que, como vemos, debe encontrarse en el mapa).

El ejemplo parece lo suficientemente simple, pero no funciona; ya que no se puede encontrar la clave correspondiente a #3. El problema es que **Groundhog** hereda automáticamente de la clase raíz común **Object**, y se está utilizando el método **hashCode()** de **Object** para generar el código *hash* correspondiente a cada objeto. De manera predeterminada, dicho método se limita a utilizar la dirección de su objeto. Por tanto, la primera instancia de **Groundhog(3)** no produce un código *hash* igual al código *hash* de la segunda instancia de **Groundhog(3)** que hemos tratado de utilizar como clave de búsqueda.

Podriamos pensar que lo único que hace falta es escribir un método de sustitución apropiado para **hashCode()**. Sin embargo, esta solución seguirá sin funcionar hasta que hagamos una cosa más: sustituir el método **equals()** que también forma parte de **Object**. El método **equals()** es utilizado por **HashMap** a la hora de determinar si la clave es igual a cualquiera de las claves contenidas en la tabla.

Un método **equals()** apropiado deberá satisfacer las siguientes cinco condiciones:

1. Reflexiva: para cualquier **x**, **x.equals(x)** debe devolver **true**.
2. Simétrica: para cualesquiera **x** e **y**, **x.equals(y)** debe devolver **true** si y sólo si **y.equals(x)** devuelve **true**.
3. Transitiva: para cualesquiera **x**, **y** y **z**, si **x.equals(y)** devuelve **true** e **y.equals(z)** devuelve **true**, entonces **x.equals(z)** devolverá **true**.
4. Coherencia: para cualesquiera **x** e **y**, múltiples invocaciones de **x.equals(y)** deberán devolver continuamente **true** o continuamente **false**, en tanto que no se modifique ninguna información utilizada en las comparaciones de igualdad de los objetos.
5. Para cualquier **x** distinta de **null**, **x.equals(null)** debe devolver **false**.

De nuevo, el método predeterminado **Object.equals()** simplemente compara las direcciones de los objetos, por lo que una instancia **Groundhog(3)** no es igual a la otra instancia **Groundhog(3)**. Por tanto, para poder usar nuestras propias clases como claves en un contenedor **HashMap**, debemos sustituir tanto **hashCode()** como **equals()**, como se muestra en la siguiente solución al problema de las marmotas:

```

//: containers/Groundhog2.java
// Una clase utilizada como clave en un contenedor HashMap
// debe sustituir hashCode() y equals().
public class Groundhog2 extends Groundhog {
    public Groundhog2(int n) { super(n); }
    public int hashCode() { return number; }
    public boolean equals(Object o) {
        return o instanceof Groundhog2 &&
               (number == ((Groundhog2)o).number);
    }
} //:-

```

```
//: containers/SpringDetector2.java
// Una clave adecuada.

public class SpringDetector2 {
    public static void main(String[] args) throws Exception {
        SpringDetector.detectSpring(Groundhog2.class);
    }
} /* Output:
map = {Groundhog #2=Early Spring!, Groundhog #4=Six more weeks of Winter!, Groundhog
#9=Six more weeks of Winter!, Groundhog #8=Six more weeks of Winter!, Groundhog #6=Early
Spring!, Groundhog #1=Six more weeks of Winter!, Groundhog #3=Early Spring!, Groundhog
#7=Early Spring!, Groundhog #5=Early Spring!, Groundhog #0=Six more weeks of Winter!}
Looking up prediction for Groundhog #3
Early Spring!
*///:-
```

Groundhog2.hashCode() devuelve el número de marmota como valor de *hash*. En este ejemplo, el programador es responsable de garantizar que no existan dos marmotas con el mismo número identificador. El método **hashCode()** no tiene por qué devolver un identificador unívoco (esto es algo que explicaremos con más detalle más adelante en el capítulo), pero el método **equals()** debe determinar de manera estricta si dos objetos son equivalentes. Aquí, **equals()** se basa en el número de marmota, por lo que existen como claves dos objetos **Groundhog2** en el mapa **HashMap** que tengan el mismo número de marmota, el método fallará.

Aunque parece que el método **equals()** se limita a comprobar si el argumento es una instancia de **Groundhog2** (usando la palabra clave **instanceof**, de la que hemos hablado en el Capítulo 14, *Información de tipos*), **instanceof** realiza, en realidad, una segunda comprobación automática para ver si el objeto es **null**, ya que **instanceof** devuelve **false** si el argumento de la izquierda es **null**. Suponiendo que el objeto sea del tipo correcto y distinto de **null**, la comparación se basa en los valores **number** de cada objeto. Puede ver, analizando la salida, que ahora el comportamiento es correcto.

A la hora de crear su propia clase para emplearla en un contenedor **HashSet**, deberá prestar atención a los mismos problemas que cuando se utiliza como clave en un mapa de tipo **HashMap**.

Funcionamiento de **hashCode()**

El ejemplo anterior es sólo un primer paso en la resolución correcta del problema. Demuestra que si no sustituimos **hashCode()** y **equals()** para nuestra clave, la estructura de datos *hash* (**HashSet**, **HashMap**, **LinkedHashSet** o **LinkedHashMap**) probablemente no podrá gestionar nuestra clave apropiadamente. Sin embargo, para obtener una *buenasolución* del problema, necesitamos comprender qué es lo que sucede dentro de la estructura de datos *hash*.

En primer lugar, consideremos cuál es la motivación para utilizar almacenamiento *hash*: lo que queremos es buscar un objeto empleando otro objeto. Pero también podríamos hacer esto con un contenedor **TreeMap**, o incluso podríamos implementar nuestro propio contenedor **Map**. Por contraste con una implementación *hash*, el siguiente ejemplo implementa un mapa usando una pareja de contenedores **ArrayList**. A diferencia de **AssociativeArray.java**, esto incluye una implementación completa de la interfaz **Map**, para poder disponer del método **entrySet()**:

```
//: containers/SlowMap.java
// Un mapa implementado con ArrayList.
import java.util.*;
import net.mindview.util.*;

public class SlowMap<K,V> extends AbstractMap<K,V> {
    private List<K> keys = new ArrayList<K>();
    private List<V> values = new ArrayList<V>();
    public V put(K key, V value) {
        V oldValue = get(key); // El valor anterior o null
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
```

```

        values.set(keys.indexOf(key), value);
        return oldValue;
    }
    public V get(Object key) { // La clave es de tipo Object, no K
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
        Iterator<K> ki = keys.iterator();
        Iterator<V> vi = values.iterator();
        while(ki.hasNext())
            set.add(new MapEntry<K,V>(ki.next(), vi.next()));
        return set;
    }
    public static void main(String[] args) {
        SlowMap<String,String> m= new SlowMap<String,String>();
        m.putAll(Countries.capitals(15));
        System.out.println(m);
        System.out.println(m.get("BULGARIA"));
        System.out.println(m.entrySet());
    }
} /* Output:
{CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE
VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN
REPUBLIC=Bangui, BOTSWANA=Gaberone, BURUNDI=Bujumbura,
BENIN=Porto-Novo, BULGARIA=Sofia, EGYPT=Cairo,
ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Djibouti}
Sofia
[CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE
VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, CENTRAL
AFRICAN REPUBLIC=Bangui, BOTSWANA=Gaberone, BURUNDI=Bujumbura,
BENIN=Porto-Novo, BULGARIA=Sofia, EGYPT=Cairo, ANGOLA=Luanda,
BURKINA FASO=Ouagadougou, DJIBOUTI=Djibouti]
*///:-
```

El método **put()** simplemente coloca las claves y valores en sendos contenedores **ArrayList** relacionados. De acuerdo con la interfaz **Map**, tiene que devolver la clave anterior o **null** si no había clave anterior.

De acuerdo también con las especificaciones de **Map**, **get()** devuelve **null** si la clave no se encuentra en el mapa **SlowMap**. Si la clave existe, se utiliza para buscar el índice numérico que indica su posición dentro de la lista de claves **keys**, y este número se emplea como índice para generar el valor asociado a partir de la lista **values**. Observe que el tipo de key es **Object** en **get()**, en lugar de ser del tipo parametrizado **K** como cabría esperar (y que se utilizaba en **AssociativeArray.java**). Esto es a consecuencia de la introducción de los genéricos dentro del lenguaje Java en una etapa tan tardía: si los genéricos hubieran sido una de las características originales del lenguaje, **get()** podría haber especificado el tipo de su parámetro.

El método **Map.entrySet()** debe generar un conjunto de objetos **Map.Entry**. Sin embargo, **Map.Entry** es una interfaz que describe una estructura dependiente de la implementación, por lo que si queremos hacer nuestro propio tipo de mapa, deberemos también definir una implementación de **Map.Entry**:

```

//: containers/MapEntry.java
// Una definición simple de Map.Entry para implementaciones
// de ejemplo de mapas.
import java.util.*;

public class MapEntry<K,V> implements Map.Entry<K,V> {
    private K key;
    private V value;
    public MapEntry(K key, V value) {

```

```

        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public V setValue(V v) {
        V result = value;
        value = v;
        return result;
    }
    public int hashCode() {
        return (key==null ? 0 : key.hashCode()) *
            (value==null ? 0 : value.hashCode());
    }
    public boolean equals(Object o) {
        if(!(o instanceof MapEntry)) return false;
        MapEntry me = (MapEntry)o;
        return
            (key == null ?
            me.getKey() == null : key.equals(me.getKey())) &&
            (value == null ?
            me.getValue() == null : value.equals(me.getValue()));
    }
    public String toString() { return key + "=" + value; }
} //:-/

```

Aquí, una clase muy simple denominada **MapEntry** almacena y extrae las claves y valores. Ésta se usa en **entrySet()** para generar un conjunto de parejas clave-valor. Observe que **entrySet()** utiliza un conjunto **HashSet** para almacenar las parejas, y **MapEntry** adopta una solución sencilla consistente en limitarse a utilizar el método **hashCode()** de la clave **key**. Aunque esta solución es muy simple y parece funcionar en la prueba trivial realizada en **SlowMap.main()**, no es una implementación correcta porque se realiza una copia de las claves y valores. Una implementación correcta de **entrySet()** proporcionaría una *vista* del mapa en lugar de una copia, y esta vista permitiría la modificación del mapa original (lo que una copia no permite). El Ejercicio 16 proporciona la oportunidad de corregir este problema.

Observe que el método **equals()** en **MapEntry** debe comprobar tanto claves como valores. El significado del método **hashCode()** se describirá en breve.

La representación del tipo **String** del contenido del mapa **SlowMap** se genera automáticamente mediante el método **toString()** definido en **AbstractMap**.

En **SlowMap.main()**, se carga un mapa **SlowMap** y luego se muestran los contenidos. Una llamada a **get()** demuestra que la solución funciona.

Ejercicio 15: (1) Repita el Ejercicio 13 utilizando un mapa **SlowMap**.

Ejercicio 16: (7) Aplique las pruebas de **Maps.java** a **SlowMap** para verificar que funciona. Corrija cualquier cosa de **SlowMap** que no funcione correctamente.

Ejercicio 17: (2) Implemente el resto de la interfaz **Map** para **SlowMap**.

Ejercicio 18: (3) Utilizando como modelo **SlowMap.java**, cree un conjunto **SlowSet**.

Mejora de la velocidad con el almacenamiento *hash*

SlowMap.java muestra que no resulta tan difícil producir un nuevo tipo de mapa. Pero, como su nombre en inglés sugiere, **SlowMap** no es muy rápido, por lo que lo más normal es que no lo utilicemos si disponemos de alguna otra alternativa. El problema está en la búsqueda de la clave; las claves no se conservan en ningún orden concreto, así que no hay más remedio que usar una simple búsqueda lineal. La búsqueda lineal es la forma más lenta de encontrar algo.

El almacenamiento *hash* tiene como único objetivo la velocidad: este almacenamiento permite realizar las búsquedas rápidamente. Puesto que el cuello de botella se encuentra en la velocidad de búsqueda de las claves, una de las soluciones al

problema consiste en mantener las claves ordenadas y luego utilizar `Collections.binarySearch()` para realizar la búsqueda (analizaremos el proceso correspondiente en un ejercicio).

El almacenamiento *hash* va un paso más allá presuponiendo que en realidad lo único que queremos hacer es almacenar la clave *en algún lugar* de forma tal que pueda ser encontrada rápidamente. La estructura más rápida en la que se puede almacenar un grupo de elementos es una matriz, así que eso es lo que se utilizará para representar la información de claves (observe que decimos "información de claves", y no las claves mismas). Pero, como una matriz no puede cambiar de tamaño, tenemos un problema: queremos almacenar un número indeterminado de valores en el mapa, pero si el número de valores está fijado por el tamaño de la matriz, ¿cómo podemos solucionar el problema?

La respuesta es que la matriz no almacena las claves. A partir del objeto clave, determinaremos un número que servirá como índice dentro de la matriz. Este número es el *código hash* generado por el método `hashCode()` (en términos de la jerga informática, este método sería la *función de hash*) definido en `Object` y, normalmente, sustituido en la clase que estemos utilizando.

Para resolver el problema de la matriz de tamaño fijo, es perfectamente posible que haya más de una clave que genere el mismo índice. En otras palabras, puede haber *colisiones*. Debido a esto, no importa lo grande que sea la matriz; el *código hash* del objeto clave estará almacenado en alguna parte de la matriz.

De modo que el proceso de buscar el valor comienza calculando el *código hash* y utilizándolo como índice para acceder a la matriz. Si pudiéramos garantizar que no habrá colisiones (lo cual es posible si disponemos de un número fijo de valores), tendríamos una *función hash perfecta*, pero eso es un caso especial.⁷ En todos los demás casos, las colisiones se gestionan mediante un mecanismo de *encadenamiento externo*. La matriz no apunta directamente a un valor, sino a una lista de valores. Estos valores se exploran de forma lineal utilizando el método `equals()`. Por supuesto, este aspecto de la búsqueda es mucho más lento, pero si la función *hash* es buena, sólo habrá unos pocos valores en cada posición. De este modo, en lugar de buscar en la lista completa, saltamos rápidamente a una posición donde sólo tenemos que comparar unas pocas entradas para encontrar el valor. Esto es mucho más rápido, que es la razón de que `HashMap` sea tan veloz.

Ahora que conocemos los fundamentos del almacenamiento *hash*, podemos implementar un mapa *hash* simple:

```
//: containers/SimpleHashMap.java
// Un mapa hash de demostración.
import java.util.*;
import net.mindview.util.*;

public class SimpleHashMap<K,V> extends AbstractMap<K,V> {
    // Seleccione un número primo como tamaño de la tabla hash,
    // para conseguir una distribución uniforme:
    static final int SIZE = 997;
    // No se puede tener una matriz física de genéricos,
    // pero si que podemos generalizar para obtener una:
    @SuppressWarnings("unchecked")
    LinkedList<MapEntry<K,V>>[] buckets =
        new LinkedList[SIZE];
    public V put(K key, V value) {
        V oldValue = null;
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null)
            buckets[index] = new LinkedList<MapEntry<K,V>>();
        LinkedList<MapEntry<K,V>> bucket = buckets[index];
        MapEntry<K,V> pair = new MapEntry<K,V>(key, value);
        boolean found = false;
        ListIterator<MapEntry<K,V>> it = bucket.listIterator();
        while(it.hasNext()) {
            MapEntry<K,V> iPair = it.next();
            if(iPair.getKey().equals(key)) {
                oldValue = iPair.getValue();
                iPair.setValue(value);
            }
        }
        return oldValue;
    }
    public V get(K key) {
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null)
            return null;
        LinkedList<MapEntry<K,V>> bucket = buckets[index];
        MapEntry<K,V> pair = new MapEntry<K,V>(key, null);
        boolean found = false;
        ListIterator<MapEntry<K,V>> it = bucket.listIterator();
        while(it.hasNext()) {
            MapEntry<K,V> iPair = it.next();
            if(iPair.getKey().equals(key))
                found = true;
        }
        if(found)
            return iPair.getValue();
        else
            return null;
    }
}
```

⁷ El caso de una función *hash* perfecta está implementado en las estructuras `EnumMap` y `EnumSet` de Java SE5, porque las enumeraciones definen un número fijo de valores. Consulte el Capítulo 19, *Tipos enumerados*.

```

        it.set(pair); // Sustituir antiguo por nuevo
        found = true;
        break;
    }
}
if(!found)
    buckets[index].add(pair);
return oldValue;
}

public V get(Object key) {
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null) return null;
    for(MapEntry<K,V> iPair : buckets[index])
        if(iPair.getKey().equals(key))
            return iPair.getValue();
    return null;
}

public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
    for(LinkedList<MapEntry<K,V>> bucket : buckets) {
        if(bucket == null) continue;
        for(MapEntry<K,V> mpair : bucket)
            set.add(mpair);
    }
    return set;
}

public static void main(String[] args) {
    SimpleHashMap<String,String> m =
        new SimpleHashMap<String,String>();
    m.putAll(Countries.capitals(25));
    System.out.println(m);
    System.out.println(m.get("ERITREA"));
    System.out.println(m.entrySet());
}

/* Output:
{CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE
D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul,
KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
GHANA=Accra, DJIBOUTI=Djibouti, ETHIOPIA=Addis Ababa}
Asmara
{CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE
D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul,
KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
GHANA=Accra, DJIBOUTI=Djibouti, ETHIOPIA=Addis Ababa}
*///:-
```

Como las “posiciones” de una tabla *hash* se denominan a menudo *segmentos* (*buckets*), la matriz que representa a la tabla se denomina **buckets**. Para conseguir una distribución uniforme, el número de segmentos es, normalmente, un número

primo.⁸ Observe que se trata de una matriz de tipo **LinkedList**, que se encarga automáticamente de las colisiones. Cada nuevo elemento se añade simplemente al final de la lista correspondiente a un segmento concreto. Incluso aunque Java no nos permite crear una matriz de genéricos, si que es posible hacer una *referencia* a dicha matriz. Aquí resulta útil hacer una generalización de dicha matriz, para evitar las proyecciones adicionales de tipos posteriormente en el código.

Para el método **put()**, se invoca **hashCode()** utilizando la clave y el resultado se transforma en un número positivo. Para insertar el número resultante en la matriz **buckets**, se emplea el operador de módulo junto con el tamaño de la matriz. Si dicha posición es **null**, quiere decir que no hay ningún elemento cuyo código *hash* se corresponda con esa posición, por lo que se crea un nuevo objeto **LinkedList** para almacenar el objeto que acaba de ser asignado a esa posición. Sin embargo, el proceso normal consiste en examinar la lista para ver si hay duplicados y, en caso de que los haya, el valor antiguo se almacena en **oldValue** y el valor nuevo sustituye al antiguo. El indicador **found** nos dice si se ha encontrado una pareja clave-valor antigua y, en caso contrario, la nueva pareja se añade al final de la lista.

El método **get()** calcula el índice para la matriz **buckets** de la misma forma que **put()** (esto es importante con el fin de garantizar que terminemos en la misma posición). Si existe un objeto **LinkedList**, se le explora en busca de una correspondencia.

Observe que no pretendemos decir que esta implementación esté optimizada para obtener el mejor rendimiento posible; sólo tratamos de ilustrar las operaciones realizadas por un mapa *hash*. Si examinamos el código fuente de **java.util.HashMap**, podremos ver la implementación realmente optimizada. Asimismo, por simplicidad, **SimpleHashMap** usa la misma técnica para **entrySet()** que ya se utilizó en **SlowMap**, la cual es demasiado simplista y no funciona para los mapas de propósito general.

Ejercicio 19: (1) Repita el Ejercicio 13 utilizando un contenedor **SimpleHashMap**.

Ejercicio 20: (3) Modifique **SimpleHashMap** para que informe de las colisiones y pruebe el sistema añadiendo el mismo conjunto de datos dos veces, con el fin de que se produzcan colisiones.

Ejercicio 21: (2) Modifique **SimpleHashMap** para que informe del número de "consultas" necesarias cuando se producen colisiones. En otras palabras, informe del número de llamadas a **next()** que hay que realizar en los iteradores que recorren las listas enlazadas.

Ejercicio 22: (4) Implemente los métodos **clear()** y **remove()** para **SimpleHashMap**.

Ejercicio 23: (3) Implemente el resto de la interfaz **Map** para **SimpleHashMap**.

Ejercicio 24: (5) Siguiendo el ejemplo de **SimpleHashMap.java**, cree y pruebe un contenedor **SimpleHashSet**.

Ejercicio 25: (6) En lugar de usar un iterador **ListIterator** para cada segmento, modifique **MapEntry** para que sea una única lista enlazada autocontenido (cada objeto **MapEntry** debe tener un enlace directo al siguiente objeto **MapEntry**). Modifique el resto del código de **SimpleHashMap.java** para que funcione correctamente esta solución.

Sustitución de **hashCode()**

Ahora que comprendemos cómo funciona el almacenamiento *hash*, tiene más sentido escribir nuestro propio método **hashCode()**.

En primer lugar, nosotros no controlamos la creación del valor concreto que se usa para obtener el índice de la matriz de segmentos. Ese valor depende de la capacidad del objeto **HashMap** concreto y dicha capacidad varía dependiendo de lo lleno que esté el contenedor y de cuál sea el *factor de carga* (describiremos este término más adelante). Por tanto, el valor generado por nuestro método **hashCode()** se procesará ulteriormente para crear el índice de la matriz de segmentos (en **SimpleHashMap**, el cálculo consiste simplemente en hacer una operación de módulo según el tamaño de la matriz de segmentos).

⁸ En realidad, un número primo no es en la práctica el tamaño ideal para los segmentos *hash*, y las implementaciones *hash* más recientes en Java utilizan un tamaño igual a las potencias de dos (después de haber realizado pruebas exhaustivas). La división o el cálculo del resto es la operación más lenta en un procesador moderno. Con una longitud de la tabla *hash* igual a una potencia de dos, se puede utilizar una operación de enmascaramiento en lugar de la de división. Puesto que **get()** es, con mucho, la operación más común, % representa una gran parte del coste y la solución basada en una potencia de dos elimina este coste (aunque puede que también afecte a algunos métodos **hashCode()**).

El factor más importante a la hora de crear un método `hashCode()` es que, independientemente de cuándo se invoque `hashCode()`, éste debe producir el mismo valor para un objeto concreto cada vez que sea invocado. Si tuviéramos un objeto que produjera un valor `hashCode()` al insertarlo con `put()` en un contenedor `HashMap` y otro valor distinto al extraerlo con `get()`, no podríamos nunca extraer los objetos. Por tanto, si nuestro método `hashCode()` depende de datos del objeto que varían, es necesario informar al usuario de que al modificar los datos se generará una clave diferente, porque se tendrá un código `hashCode()` diferente.

Además, *normalmente no conviene* generar un valor `hashCode()` que esté basado en información de los objetos de carácter distintivo, en concreto, el valor de `this` genera un código `hashCode()` no muy bueno, porque entonces es imposible generar una nueva clave idéntica a la que se ha usado para insertar con `put()` la pareja original clave-valor. Éste era el problema que ya detectamos en `SpringDetector.java`, porque la implementación predeterminada de `hashCode()` *utiliza precisamente* la dirección del objeto. Por tanto, lo que conviene es utilizar información del objeto que le identifique de alguna manera significativa.

Podemos ver un buen ejemplo en la clase `String`. Las cadenas de caracteres tienen la característica especial de que si un programa dispone de varios objetos `String` que contienen secuencias de caracteres idénticas, entonces todos esos objetos `String` se corresponden con la misma zona de memoria. Por tanto, tiene bastante sentido que el código `hashCode()` producido por dos instancias diferentes de la cadena de caracteres “hello” deba ser idéntico. Podemos ver esto en el siguiente programa:

```
//: containers/StringHashCode.java

public class StringHashCode {
    public static void main(String[] args) {
        String[] hellos = "Hello Hello".split(" ");
        System.out.println(hellos[0].hashCode());
        System.out.println(hellos[1].hashCode());
    }
} /* Output: (Sample)
69609650
69609650
*///:~
```

El método `hashCode()` para `String` está claramente basado en el contenido de la cadena de caracteres.

Por tanto, para que un método `hashCode()` sea efectivo, debe ser rápido y además significativo; en otras palabras, debe generar un valor basado en el contenido del objeto. Recuerde que este valor no tiene por qué ser único (lo que nos interesa es la velocidad no la unicidad) pero entre `hashCode()` y `equals()`, la identidad del objeto debe ser completamente especificada.

Puesto que el código generado por `hashCode()` se procesa adicionalmente antes de generar el índice de la matriz, el rango de valores no es importante, basta con que el método genere un valor `int`.

Hay otro factor más a tener cuenta: un buen método `hashCode()` debe producir una distribución homogénea de los valores. Si los valores tienden a estar agrupados, entonces el contenedor `HashMap` o `HashSet` estará más cargado en unas áreas que en otras, y no será tan rápido como podría serlo si se dispusiera de una función `hash` uniformemente distribuida.

En el libro *Effective Java™ Programming Language Guide* (Addison-Wesley, 2001), Joshua Bloch nos da una receta básica para generar un método `hashCode()` aceptable:

- Almacene algún valor constante distinto de cero, como por ejemplo 17, en una variable `int` denominada `result`.
- Por cada campo significativo `f` del objeto (es decir, cada campo que sea tenido en cuenta por el método `equals()`), calcule un código `hash` `c` de tipo `int` correspondiente a ese campo:

| Tipo del campo | Cálculo |
|--------------------------------------|--|
| <code>boolean</code> | <code>c = (f ? 0 : 1)</code> |
| <code>byte, char, short o int</code> | <code>c = (int)f</code> |
| <code>long</code> | <code>c = (int)(f ^ (f >>>32))</code> |
| <code>float</code> | <code>c = Float.floatToIntBits(f);</code> |

| Tipo del campo | Cálculo |
|--|--|
| double | long l = Double.doubleToLongBits(f); c = (int)(l ^ (l >>> 32)) |
| Object, donde equals() invoca a equals() para este campo | c = f.hashCode() |
| Matriz | Aplicar las reglas anteriores a cada elemento |

3. Combine el código hash recién calculado: result = 37 * result + c;

4. Devuelva result.

5. Examine el código hashCode() resultante y asegúrese de que instancias iguales tengan códigos hash iguales.

He aquí un ejemplo construido sobre estas directrices:

```
//: containers/CountedString.java
// Creación de un método hashCode() adecuado.
import java.util.*;
import static net.mindview.util.Print.*;

public class CountedString {
    private static List<String> created =
        new ArrayList<String>();
    private String s;
    private int id = 0;
    public CountedString(String str) {
        s = str;
        created.add(s);
        // id es el número total de instancias
        // de esta cadena que CountedString está usando:
        for(String s2 : created)
            if(s2.equals(s))
                id++;
    }
    public String toString() {
        return "String: " + s + " id: " + id +
            " hashCode(): " + hashCode();
    }
    public int hashCode() {
        // La técnica simple:
        // return s.hashCode() * id;
        // Utilización de la receta de Joshua Bloch:
        int result = 17;
        result = 37 * result + s.hashCode();
        result = 37 * result + id;
        return result;
    }
    public boolean equals(Object o) {
        return o instanceof CountedString &&
            s.equals(((CountedString)o).s) &&
            id == ((CountedString)o).id;
    }
    public static void main(String[] args) {
        Map<CountedString, Integer> map =
            new HashMap<CountedString, Integer>();
        CountedString[] cs = new CountedString[5];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString("hi");
            map.put(cs[i], i); // Autobox int -> Integer
        }
    }
}
```

```

        print(map);
        for(CountedString cstring : cs) {
            print("Looking up " + cstring);
            print(map.get(cstring));
        }
    }
} /* Output: (Sample)
(String; hi id: 4 hashCode(): 146450=3, String: hi id: 1
hashCode(): 146447=0, String: hi id: 3 hashCode(): 146449=2,
String: hi id: 5 hashCode(): 146451=4, String: hi
id: 2 hashCode(): 146448=1}
Looking up String: hi id: 1 hashCode(): 146447
0
Looking up String: hi id: 2 hashCode(): 146448
1
Looking up String: hi id: 3 hashCode(): 146449
2
Looking up String: hi id: 4 hashCode(): 146450
3
Looking up String: hi id: 5 hashCode(): 146451
4
*///:-

```

CountedString incluye un objeto **String** y un **id** que representa el número de objetos **CountedString** que contienen un objeto **String** idéntico. El recuento se realiza en el constructor, iterando a través del contenedor **ArrayList** estático en el que están almacenadas todas las cadenas de caracteres.

Tanto **hashCode()** como **equals()** generan resultados basados en ambos campos; si estuvieran basados simplemente en el objeto **String** o en el valor **id**, habría correspondencias duplicadas para valores diferentes.

En **main()**, se crean varios objetos **CountedString** utilizando el mismo objeto **String**, con el fin de demostrar que los duplicados crean valores distintos, debido al campo **id** que se utiliza como recuento. El ejemplo muestra el contenedor **HashMap**, para que veamos cómo se almacenan los elementos internamente (no hay ningún orden discernible), y después se busca cada clave individualmente para demostrar que el mecanismo de búsqueda funciona correctamente.

Como segundo ejemplo, considere la clase **Individual** que ya usamos como clase base para la biblioteca **typeinfo.pet** definida en el Capítulo 14, *Información de tipos*. La clase **Individual** se usaba en dicho capítulo, pero habíamos retardado su definición hasta este capítulo con el fin de que el lector comprendiera la implementación:

```

//: typeinfo/pets/Individual.java
package typeinfo.pets;

public class Individual implements Comparable<Individual> {
    private static long counter = 0;
    private final long id = counter++;
    private String name;
    public Individual(String name) { this.name = name; }
    // 'name' es opcional:
    public Individual() {}
    public String toString() {
        return getClass().getSimpleName() +
            (name == null ? "" : " " + name);
    }
    public long id() { return id; }
    public boolean equals(Object o) {
        return o instanceof Individual &&
            id == ((Individual)o).id;
    }
    public int hashCode() {
        int result = 17;
        if(name != null)

```

```

        result = 37 * result + name.hashCode();
        result = 37 * result + (int)id;
        return result;
    }
    public int compareTo(Individual arg) {
        // Comparar primero por el nombre de la clase:
        String first = getClass().getSimpleName();
        String argFirst = arg.getClass().getSimpleName();
        int firstCompare = first.compareTo(argFirst);
        if(firstCompare != 0)
            return firstCompare;
        if(name != null && arg.name != null) {
            int secondCompare = name.compareTo(arg.name);
            if(secondCompare != 0)
                return secondCompare;
        }
        return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
    }
} //:-:

```

El método **compareTo()** tiene una jerarquía de comparaciones, de modo que producirá una secuencia que estará ordenada primero por el tipo real, y luego por **name** si es que existe y finalmente por orden de creación. He aquí un ejemplo que demuestra cómo funciona:

```

//: containers/IndividualTest.java
import holding.MapOfList;
import typeinfo.pets.*;
import java.util.*;

public class IndividualTest {
    public static void main(String[] args) {
        Set<Individual> pets = new TreeSet<Individual>();
        for(List<? extends Pet> lp :
            MapOfList.petPeople.values())
            for(Pet p : lp)
                pets.add(p);
        System.out.println(pets);
    }
} /* Output:
[Cat Elsie May, Cat Pinkola, Cat Shackleton, Cat Stanford
 aka Stinky el Negro, Cymric Molly, Dog Margrett, Mutt Spot,
 Pug Louis aka Louis Snorkelstein Dupree, Rat Fizzy,
 Rat Freckly, Rat Fuzzy]
*///:-:

```

Puesto que todas estas mascotas utilizadas en el ejemplo tienen nombres, se las ordena primero por tipo y luego, dentro de cada tipo, según su nombre.

Escribir sendos métodos **hashCode()** y **equals()** para una nueva clase puede resultar complicado. Puede encontrar herramientas que le ayudarán a hacerlo en el proyecto “Jakarta Commons” de Apache, en jakarta.apache.org/commons, en el subdirectorio “lang” (este proyecto dispone también de muchas otras bibliotecas potencialmente útiles, y parece ser la respuesta de la comunidad Java a la comunidad www.boost.org de C++).

Ejercicio 26: (2) Añade un campo **char** a **CountedString** que también se inicialice en el constructor, y modifique los métodos **hashCode()** y **equals()** para incluir el valor de este campo **char**.

Ejercicio 27: (3) Modifique el método **hashCode()** en **CountedString.java** eliminando la combinación con **id**, y demuestre que **CountedString** sigue funcionando como clave. ¿Cuál es el problema con esta técnica?

Ejercicio 28: (4) Modifique **net/mindview/util/Tuple.java** para convertirla en una clase de propósito general añadiendo **hashCode()**, **equals()**, e implementando **Comparable** para cada tipo de **Tuple**.

Selección de una implementación

A estas alturas, el lector debería entender, que aunque sólo hay cuatro tipos de contenedores (**Map**, **List**, **Set** y **Queue**) hay más de una implementación de cada interfaz. Si necesitamos utilizar la funcionalidad ofrecida por una interfaz concreta, ¿cómo podemos decidir qué implementación emplear?

Cada una de las diferentes implementaciones tiene sus propias características, ventajas e inconvenientes. Por ejemplo, puede ver en la imagen incluida al principio de este capítulo que la “ventaja” de **Hashtable**, **Vector** y **Stack** es que son clases heredadas, de modo que el código antiguo no dejará de funcionar (aunque lo mejor es que no utilice esos contenedores en los programas nuevos).

Los diferentes tipos de **Queue** en la biblioteca Java se diferencian sólo en la forma en que aceptan y devuelven los valores (veremos la importancia de esto en el Capítulo 21, *Concurrencia*).

La distinción entre unos contenedores y otros usualmente reside en el almacenamiento que se utiliza como respaldo; es decir, en las estructuras de datos que implementan físicamente la interfaz deseada. Por ejemplo, como **ArrayList** y **LinkedList** implementan la interfaz **List**, las operaciones *básicas* de **List** son iguales independientemente de cuál utilicemos. Sin embargo, **ArrayList** utiliza como respaldo una matriz mientras que **LinkedList** se implementa en la forma usual de las listas doblemente enlazadas, en forma de objetos individuales cada uno de los cuales contiene tanto los datos como sendas referencias a los elementos anterior y siguiente de la lista. Debido a esto, si queremos hacer muchas inserciones y eliminaciones en mitad de una lista, la elección apropiada será **LinkedList** (**LinkedList** dispone también de funcionalidad adicional que está definida en **AbstractSequentialList**). Si no es el caso, **ArrayList** suele ser más rápido.

Como ejemplo adicional, podemos implementar un conjunto mediante los contenedores **TreeSet**, **HashSet** o **LinkedHashSet**.⁹ Cada uno de estos contenedores tiene diferentes comportamientos: **HashSet** es para uso normal y proporciona una gran velocidad en las búsquedas, **LinkedHashSet** mantiene las parejas en orden de inserción y **TreeSet** está respaldado por un contenedor **TreeMap** y está diseñado para disponer de un conjunto constantemente ordenado. La implementación se elige basándose en el comportamiento que se necesite.

En ocasiones, las diferentes implementaciones de un contenedor concreto tendrán una serie de operaciones en común, pero el rendimiento de esas operaciones será diferente. En este caso, la selección entre unas implementaciones y otras se basa en la frecuencia con la que se utilice una operación concreta y lo veloz que necesitemos que sea esa operación. Para casos como estos, una de las maneras de evaluar las diferencias entre las distintas implementaciones de contenedores es mediante una prueba de rendimiento.

Un marco de trabajo para pruebas de rendimiento

Para evitar la duplicación de código y para que las pruebas sean coherentes, he incluido la funcionalidad básica del proceso de pruebas en un marco de trabajo que define la parte principal del programa. El código siguiente establece una clase base a partir de la cual se crea una lista de clases internas anónimas, una clase para cada una de las diferentes pruebas. Cada una de estas clases internas se invoca como parte del proceso de pruebas. Esta solución permite añadir y eliminar fácilmente distintos tipos de pruebas.

Se trata de otro ejemplo del patrón de diseño basado en el *Método de plantillas*. Aunque se sigue la solución típica del método de plantillas consistente en sustituir el método **Test.test()** para cada prueba concreta, en este caso la parte fundamental del código (que no cambia) se encuentra en una clase separada **Tester**.¹⁰ El tipo de contenedor que estamos probando es el parámetro genérico **C**:

```
//: containers/Test.java
// Marco de trabajo para realizar pruebas temporizadas de contenedores.

public abstract class Test<C> {
    String name;
    public Test(String name) { this.name = name; }
```

⁹ O como **EnumSet** o **CopyOnWriteArrayList**, que son casos especiales. Aunque somos conscientes de que puede haber muchas otras implementaciones adicionales especializadas de diversas interfaces de contenedores, en esta sección estamos tratando de examinar sólo los casos más generales.

¹⁰ Krzysztof Sobolewski me ayudó a diseñar los genéricos de este ejemplo.

```
// Sustituir este método para las diferentes pruebas.
// Devuelve el número real de repeticiones de la prueba.
abstract int test(C container, TestParam tpi;
} //:-
```

Cada objeto **Test** almacena el nombre de dicha prueba. Cuando se invoca el método **test()**, hay que pasarle el contenedor que hay que probar junto con un "elemento de transferencia de datos" o "mensajero" que almacene los diversos parámetros correspondientes a dicha prueba concreta. Los parámetros incluyen **size**, que indica el número de elementos del contenedor y **loops**, que controla el número de iteraciones de la prueba. Estos parámetros pueden o no utilizarse en todas las pruebas.

Para cada contenedor se realizará una secuencia de llamadas a **test()**, cada una con un objeto **TestParam** diferente, de modo que **TestParam** también contiene métodos **array()** estáticos que hacen que resulte fácil crear matrices de objetos **TestParam**. La primera versión de **array()** toma una lista de argumentos variables que contiene valores **size** y **loops** alternantes, mientras que la segunda versión toma el mismo tipo de lista, aunque con valores almacenados dentro de objetos **String**, de esta forma, puede utilizarse para analizar argumentos de la línea de comandos:

```
//: containers/TestParam.java
// Un "objeto de transferencia de datos".

public class TestParam {
    public final int size;
    public final int loops;
    public TestParam(int size, int loops) {
        this.size = size;
        this.loops = loops;
    }
    // Crear una matriz de TestParam a partir de una secuencia de varargs:
    public static TestParam[] array(int... values) {
        int size = values.length/2;
        TestParam[] result = new TestParam[size];
        int n = 0;
        for(int i = 0; i < size; i++)
            result[i] = new TestParam(values[n++], values[n++]);
        return result;
    }
    // Convertir una matriz de tipo String a una matriz TestParam:
    public static TestParam[] array(String[] values) {
        int[] vals = new int[values.length];
        for(int i = 0; i < vals.length; i++)
            vals[i] = Integer.decode(values[i]);
        return array(vals);
    }
} //:-
```

Para utilizar el marco de trabajo, lo que hacemos es pasar el contenedor que hay que probar junto con una lista de objetos **Test** a un método **Tester.run()** (se trata de métodos genéricos de utilidad sobrecargados, que reducen la cantidad de texto que hay que escribir para utilizarlos). **Tester.run()** invoca el constructor sobrecargado apropiado y luego llama a **timedTest()**, que ejecuta para ese contenedor cada una de las pruebas de la lista. **timedTest()** repite cada prueba para cada uno de los objetos **TestParam** contenidos en **paramList**. Puesto que **paramList** se inicializa a partir de la matriz estática **defaultParams**, podemos cambiar la lista **paramList** para todas las pruebas resiniendo **defaultParams**, o bien podemos modificar la lista **paramList** para una prueba determinada, pasándole para esa prueba una lista **paramList** personalizada:

```
//: containers/Tester.java
// Aplica objetos Test a listas de diferentes contenedores.
import java.util.*;

public class Tester<C> {
    public static int fieldWidth = 8;
    public static TestParam[] defaultParams= TestParam.array(
        10, 5000, 100, 5000, 1000, 5000, 10000, 500);
```

```

// Sustituir esto para modificar la inicialización anterior a la prueba:
protected C initialize(int size) { return container; }
protected C container;
private String headline = "";
private List<Test<C>> tests;
private static String stringField() {
    return "%" + fieldWidth + "s";
}
private static String numberField() {
    return "%" + fieldWidth + "d";
}
private static int sizeWidth = 5;
private static String sizeField = "%" + sizeWidth + "s";
private TestParam[] paramList = defaultParams;
public Tester(C container, List<Test<C>> tests) {
    this.container = container;
    this.tests = tests;
    if(container != null)
        headline = container.getClass().getSimpleName();
}
public Tester(C container, List<Test<C>> tests,
    TestParam[] paramList) {
    this(container, tests);
    this.paramList = paramList;
}
public void setHeadline(String newHeadline) {
    headline = newHeadline;
}
// Métodos genéricos de utilidad:
public static <C> void run(C cntnr, List<Test<C>> tests){
    new Tester<C>(cntnr, tests).timedTest();
}
public static <C> void run(C cntnr,
    List<Test<C>> tests, TestParam[] paramList) {
    new Tester<C>(cntnr, tests, paramList).timedTest();
}
private void displayHeader() {
    // Calcular anchura y rellenar con '-':
    int width = fieldWidth * tests.size() + sizeWidth;
    int dashLength = width - headline.length() - 1;
    StringBuilder head = new StringBuilder(width);
    for(int i = 0; i < dashLength/2; i++)
        head.append('-');
    head.append(' ');
    head.append(headline);
    head.append(' ');
    for(int i = 0; i < dashLength/2; i++)
        head.append('-');
    System.out.println(head);
    // Imprimir cabeceras de columnas:
    System.out.format(sizeField, "size");
    for(Test test : tests)
        System.out.format(stringField(), test.name);
    System.out.println();
}
// Ejecutar las pruebas para este contenedor:
public void timedTest() {
    displayHeader();
    for(TestParam param : paramList) {

```

```

        System.out.format(sizeField, param.size);
        for(Test<C> test : tests) {
            C kontainer = initialize(param.size);
            long start = System.nanoTime();
            // Invocar el método sustituido:
            int reps = test.test(kontainer, param);
            long duration = System.nanoTime() - start;
            long timePerRep = duration / reps; // Nanosegundos
            System.out.format(numberField(), timePerRep);
        }
        System.out.println();
    }
}
} //:~
```

Los métodos **stringField()** y **numberField()** producen cadenas de formateo para imprimir los resultados. La anchura estándar de formateo puede variarse modificando el valor estático **fieldWidth**. El método **displayHeader()** da formato e imprime la información de cabecera para cada prueba.

Si necesitamos realizar una inicialización especial, sustituimos el método **initialize()**. Esto produce un objeto contenedor inicializado con el tamaño apropiado; podemos modificar el objeto contenedor existente o crear uno nuevo. Como puede ver en **test()** el resultado se capture en una referencia local denominada **kontainer**, que permite sustituir el miembro almacenado **container** por un contenedor inicializado completamente diferente.

El valor de retorno de cada método **Test.test()** debe ser el número de operaciones realizado por dicha prueba, lo que se utiliza para calcular el número de nanosegundos requerido por cada operación. Hay que tener en cuenta que **System.nanoTime()** produce normalmente valores con una granularidad mayor que uno (y esta granularidad variará de una máquina a otra y de un sistema operativo a otro), lo que produce un cierto error estadístico en los resultados.

Los resultados pueden variar de una máquina a otra, estas pruebas sólo pretenden comparar el rendimiento de los diferentes contenedores.

Selección entre listas

He aquí una prueba de rendimiento para las operaciones de **List** más esenciales. Por comparación, también se muestran las operaciones de **Queue** más importantes. Se crean dos listas separadas de pruebas, con el fin de probar cada clase de contenedor. En este caso, las operaciones de **Queue** sólo se aplican a listas de tipo **LinkedList**.

```

//: containers/ListPerformance.java
// Ilustra las diferencias de rendimiento en las listas.
// (Args: 100 500) Pequeño para que las pruebas sean cortas
import java.util.*;
import net.mindview.util.*;

public class ListPerformance {
    static Random rand = new Random();
    static int reps = 1000;
    static List<Test<List<Integer>>> tests =
        new ArrayList<Test<List<Integer>>>();
    static List<Test<LinkedList<Integer>>> qTests =
        new ArrayList<Test<LinkedList<Integer>>>();
    static {
        tests.add(new Test<List<Integer>>("add") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops;
                int listSize = tp.size;
                for(int i = 0; i < loops; i++) {
                    list.clear();
                    for(int j = 0; j < listSize; j++)
                        list.add(j);
```

```

        }
        return loops * listSize;
    }
});
tests.add(new Test<List<Integer>>("get") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops * reps;
        int listSize = list.size();
        for(int i = 0; i < loops; i++)
            list.get(rand.nextInt(listSize));
        return loops;
    }
});
tests.add(new Test<List<Integer>>("set") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops * reps;
        int listSize = list.size();
        for(int i = 0; i < loops; i++)
            list.set(rand.nextInt(listSize), 47);
        return loops;
    }
});
tests.add(new Test<List<Integer>>("iteradd") {
    int test(List<Integer> list, TestParam tp) {
        final int LOOPS = 1000000;
        int half = list.size() / 2;
        ListIterator<Integer> it = list.listIterator(half);
        for(int i = 0; i < LOOPS; i++)
            it.add(47);
        return LOOPS;
    }
});
tests.add(new Test<List<Integer>>("insert") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        for(int i = 0; i < loops; i++)
            list.add(5, 47); // Minimizar el coste del acceso aleatorio
        return loops;
    }
});
tests.add(new Test<List<Integer>>("remove") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            list.addAll(new CountingIntegerList(size));
            while(list.size() > 5)
                list.remove(5); // Minimizar el coste del acceso aleatorio
        }
        return loops * size;
    }
});
// Pruebas de comportamiento de las colas:
qTests.add(new Test<LinkedList<Integer>>("addFirst") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();

```

```

        for(int j = 0; j < size; j++)
            list.addFirst(47);
    }
    return loops * size;
}
});
qTests.add(new Test<LinkedList<Integer>>("addLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            for(int j = 0; j < size; j++)
                list.addLast(47);
        }
        return loops * size;
    }
});
qTests.add(
    new Test<LinkedList<Integer>>("rmFirst") {
        int test(LinkedList<Integer> list, TestParam tp) {
            int loops = tp.loops;
            int size = tp.size;
            for(int i = 0; i < loops; i++) {
                list.clear();
                list.addAll(new CountingIntegerList(size));
                while(list.size() > 0)
                    list.removeFirst();
            }
            return loops * size;
        }
    });
qTests.add(new Test<LinkedList<Integer>>("rmLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            list.addAll(new CountingIntegerList(size));
            while(list.size() > 0)
                list.removeLast();
        }
        return loops * size;
    }
});
}
static class ListTester extends Tester<List<Integer>> {
    public ListTester(List<Integer> container,
                      List<Test<List<Integer>>> tests) {
        super(container, tests);
    }
    // Rellenar con el tamaño apropiado antes de cada prueba:
    @Override protected List<Integer> initialize(int size){
        container.clear();
        container.addAll(new CountingIntegerList(size));
        return container;
    }
    // Método de utilidad:
    public static void run(List<Integer> list,

```

```

        List<Test<List<Integer>>> tests) {
            new ListTester(list, tests).timedTest();
        }
    }
    public static void main(String[] args) {
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        // Sólo se pueden hacer estas dos pruebas en una matriz:
        Tester<List<Integer>> arrayTest =
            new Tester<List<Integer>>(null, tests.subList(1, 3)){
                // Esto se invocará antes de cada prueba.
                // Produce una lista de tamaño fijo respaldada por una matriz:
                @Override protected
                List<Integer> initialize(int size) {
                    Integer[] ia = Generated.array(Integer.class,
                        new CountingGenerator.Integer(), size);
                    return Arrays.asList(ia);
                }
            };
        arrayTest.setHeadline("Array as List");
        arrayTest.timedTest();
        Tester.defaultParams= TestParam.array(
            10, 5000, 100, 5000, 1000, 1000, 10000, 200);
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        ListTester.run(new ArrayList<Integer>(), tests);
        ListTester.run(new LinkedList<Integer>(), tests);
        ListTester.run(new Vector<Integer>(), tests);
        Tester.fieldWidth = 12;
        Tester<LinkedList<Integer>> qTest =
            new Tester<LinkedList<Integer>>(
                new LinkedList<Integer>(), qTests);
        qTest.setHeadline("Queue tests");
        qTest.timedTest();
    }
} /* Output: (Sample)
--- Array as List ---
size      get      set
  10      130      183
  100     130      164
  1000     129      165
 10000     129      165
----- ArrayList -----
size      add      get      set iteradd  insert  remove
   10      121      139      191     435     3952     446
   100     72       141      191     247     3934     296
   1000     98       141      194     839     2202     923
  10000     122      144      190     6880    14042    7333
----- LinkedList -----
size      add      get      set iteradd  insert  remove
   10      182      164      198     658     366     262
   100     106      202      230     457     108     201
   1000     133     1289     1353     430     136     239
  10000     172    13648    13187     435     255     239
----- Vector -----
size      add      get      set iteradd  insert  remove
   10      129      145      187     290     3635     253
   100     72       144      190     263     3691     292
   1000     99       145      193     846     2162     927
  10000     108      145      186     6871    14730    7135

```

| Queue tests | | | | |
|-------------|----------|---------|---------|--------|
| size | addFirst | addLast | rmFirst | rmLast |
| 10 | 199 | 163 | 251 | 253 |
| 100 | 98 | 92 | 180 | 179 |
| 1000 | 99 | 93 | 216 | 212 |
| 10000 | 111 | 109 | 262 | 384 |
| *///:- | | | | |

Cada una de las pruebas requiere una consideración cuidadosa para garantizar que estemos produciendo resultados significativos. Por ejemplo, la prueba “**add**” borra la lista y luego la rellena de acuerdo con el tamaño de lista especificado. La llamada **clear()** para borrar forma parte, por tanto, de la prueba y puede tener un impacto sobre el tiempo de ejecución, especialmente para las pruebas más pequeñas. Aunque los resultados parecen bastante razonables en este caso, podríamos perfectamente pensar en reescribir el marco de trabajo de pruebas para crear una llamada a un método de preparación, (que en este caso incluiría la llamada a **clear()**) fuera del bucle de cronometrado.

Observe que, para cada prueba, es necesario calcular con precisión el número de operaciones que tienen lugar y devolver dicho valor desde **test()**, para que la temporización sea correcta.

Las pruebas “**get**” y “**set**” utilizan el generador de número aleatorios para realizar accesos a la lista. Analizando la salida podemos ver que, para una lista respaldada por una matriz y para un contenedor **ArrayList**, estos accesos son rápidos y muy coherentes independientemente del tamaño de la lista, mientras que para un contenedor **LinkedList**, el tiempo de acceso aumenta de manera muy significativa para las listas de mayor tamaño. Claramente, las listas enlazadas no representan una buena elección si vamos a realizar muchos accesos aleatorios.

La prueba “**iteradd**” utiliza un iterador en mitad de la lista para insertar nuevos elementos. Para un contenedor **ArrayList**, esta operación resulta costosa a medida que el tamaño de la lista crece, pero para otro de tipo **LinkedList** es relativamente barata y ese coste es constante independientemente del tamaño. Esto tiene bastante sentido porque un contenedor **ArrayList** debe crear espacio y copiar todas sus referencias durante una inserción. Esta operación resulta muy costosa a medida que crece el tamaño del contenedor **ArrayList**. El contenedor **LinkedList**, por el contrario, sólo necesita enlazar un nuevo elemento, y no necesita modificar el resto de la lista, por lo que cabe esperar que el coste sea aproximadamente el mismo independientemente del tamaño de la lista.

Las pruebas “**insert**” y “**remove**” utilizan la posición número 5 como punto de inserción y de eliminación, en lugar de utilizar uno de los extremos de la lista. Un contenedor **LinkedList** trata los extremos de la lista de manera especial, lo que permite mejorar la velocidad cuando se usa el contenedor **LinkedList** como una cola. Sin embargo, si se añaden o eliminan en mitad de la lista, hay que incluir el coste del acceso aleatorio, que ya hemos visto que varía para las diferentes implementaciones de la lista. Realizando las inserciones y eliminaciones en la posición 5, el coste del acceso aleatorio debería ser despreciable y sólo deberíamos ver el coste de la inserción y la eliminación, pero no podremos observar los resultados de las optimizaciones especiales que se aplican a los extremos de un contenedor **LinkedList**. Podemos ver, analizando la salida, que el coste de adición y eliminación en un contenedor **LinkedList** es bastante bajo y que no varía con el tamaño de la lista, pero con un contenedor **ArrayList**, las inserciones son *especialmente* caras y el coste se incrementa con el tamaño de la lista.

A partir de los datos correspondientes a **Queue**, podemos ver la rapidez con que **LinkedList** permite insertar y eliminar elementos de los extremos de la lista, lo cual es el comportamiento óptimo para una cola.

Normalmente, podemos limitarnos a invocar **Tester.run()**, pasando el contenedor y la lista **tests**. Aquí, sin embargo, debemos sustituir el método **initialize()** para que la lista se borre y se rellene antes de cada prueba; en caso contrario, el control del tamaño de la lista se perdería durante las diversas pruebas. **ListTester** hereda de **Tester** y realiza esta inicialización empleando **CountingIntegerList**. El método de utilidad **run()** también se sustituye.

También queremos comparar el acceso a una matriz con el acceso a un contenedor (principalmente con el acceso a **ArrayList**). En la primera prueba de **main()**, se crea un objeto **Test** especial utilizando una clase interna anónima. El método **initialize()** se sustituye para crear un nuevo objeto cada vez que se le invoca (ignorando el objeto **container** almacenado, de modo que **null** es el argumento **container** para este constructor **Tester**). El nuevo objeto se crea utilizando **Generated.array()** (que se ha definido en el Capítulo 16, *Matrices*) y **Arrays.asList()**. Sólo dos de las pruebas pueden realizarse en este caso, porque no se pueden insertar o eliminar elementos cuando se usa una lista respaldada por una matriz, así que se emplea el método **List.subList()** para seleccionar las pruebas deseadas de entre la lista **tests**.

Para las operaciones `get()` y `set()` de acceso aleatorio, una lista respaldada por una matriz es ligeramente más rápida que `ArrayList`, pero esas mismas operaciones son muchísimo más caras para `LinkedList` porque este contenedor no está diseñado para operaciones de acceso aleatorio.

Es necesario evitar la utilización de `Vector`; sólo se ha incluido en la biblioteca para soporte del código heredado (la única razón de que funcione en este programa es porque fue adaptado para ser de tipo `List` por razones de compatibilidad con sucesivas versiones).

La mejor solución consiste probablemente en elegir `ArrayList` como contenedor predeterminado y cambiar a `LinkedList` si hace falta su funcionalidad adicional o si descubre problemas de rendimiento debido a que se hacen múltiples inserciones y eliminaciones en mitad de la lista. Si estamos trabajando con un grupo de elementos de tamaño fijo, deberemos emplear una lista respaldada por una matriz (como las que produce `Arrays.asList()`), o en caso necesario, una verdadera matriz.

`CopyOnWriteArrayList` es una implementación especial de `List` que se utiliza en programación concurrente y de la que hablaremos en el Capítulo 21, *Concurrencia*.

Ejercicio 29: (2) Modifique `ListPerformance.java` para que las listas almacenen objetos `String` en lugar de `Integer`. Utilice el objeto `Generator` del Capítulo 16, *Matrices*, para crear valores de prueba.

Ejercicio 30: (3) Compare el rendimiento de `Collections.sort()` en `ArrayList` y en `LinkedList`.

Ejercicio 31: (5) Cree un contenedor que encapsule una matriz de objetos `String` y que sólo permita añadir y extraer cadenas de caracteres, de modo que no surjan problemas de proyección de tipos durante la utilización del contenedor. Si la matriz no es lo suficientemente grande para la siguiente inserción, el contenedor debe redimensionar automáticamente la matriz. En `main()`, compare el rendimiento de este contenedor con el de `ArrayList<String>`.

Ejercicio 32: (2) Repita el ejercicio anterior para un contenedor de `int` y compare el rendimiento con el de `ArrayList<Integer>`. En la comparación de rendimiento, incluya el proceso de incrementar cada objeto en el contenedor.

Ejercicio 33: (5) Cree una lista `FastTraversalLinkedList` que utilice internamente un contenedor `LinkedList` para conseguir inserciones y eliminaciones rápidas de elementos y un contenedor `ArrayList` para realizar recorridos rápidos de los elementos y operaciones `get()` rápidas. Pruebe la solución modificando `ListPerformance.java`.

Peligros asociados a las micropruebas de rendimiento

A la hora de escribir las denominadas *micropruebas de rendimiento*, hay que tener cuidado de no dar demasiadas cosas por supuesto y de enfocar las pruebas lo más posible, de modo que sólo se cronometren los elementos de interés. También hay que garantizar que las pruebas se ejecuten durante una cantidad de tiempo lo suficientemente grande como para producir datos interesantes y hay que tener también en cuenta que algunas de las tecnologías HotSpot de Java sólo entran en acción cuando un programa se ha estado ejecutando durante un determinado periodo de tiempo (también es importante tener esto en cuenta para los programas de corta duración).

Los resultados serán distintos, dependiendo de la computadora y de la máquina JVM que estemos utilizando, por lo que conviene que ejecute estas pruebas por sí mismo con el fin de verificar que los resultados son similares a los que se muestran en este libro. No deben preocuparle tanto los valores absolutos como las comparaciones de rendimiento entre un tipo de contenedor y otro.

Asimismo, una herramienta de *perfilado* puede realizar un mejor análisis de rendimiento del que nosotros podemos llevar a cabo. Java incluye un perfilador (consulte el suplemento en <http://MindView.net/Books/BetterJava>) y también hay perfiladores de otros fabricantes, tanto gratuitos/código abierto como comerciales.

Un ejemplo relacionado es el que afecta a `Math.random()`. Este método produce un valor comprendido entre cero y uno, pero ¿eso incluye o excluye el valor "1"? En lenguaje matemático, ¿se trata del intervalo $(0,1)$, o $[0,1]$, o $(0,1]$ o $[0,1)$? (el corchete indica "inclusión", mientras el paréntesis indica "no inclusión"). Un programa de pruebas podría proporcionar la respuesta:

```
//: containers/RandomBounds.java
// ¿Permite Math.random() generar 0.0 y 1.0?
```

```
// {RunByHand}
import static net.mindview.util.Print.*;
public class RandomBounds {
    static void usage() {
        print("Usage:");
        print("\tRandomBounds lower");
        print("\tRandomBounds upper");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length != 1) usage();
        if(args[0].equals("lower")) {
            while(Math.random() != 0.0)
                ; // Continuar intentándolo
            print("Produced 0.0!");
        }
        else if(args[0].equals("upper")) {
            while(Math.random() != 1.0)
                ; // Continuar intentándolo
            print("Produced 1.0!");
        }
        else
            usage();
    }
} //:i~
```

Para ejecutar el programa, hay que escribir la linea de comandos:

```
java RandomBounds lower
0
java RandomBounds upper
```

En ambos casos, estamos obligados a interrumpir la ejecución del programa manualmente, por lo que podría parecer que `Math.random()` nunca genera ni 0.0 ni 1.0. Pero es precisamente aquí donde este tipo de experimentos pueden resultar engañosos. Si tenemos en cuenta que la cantidad de números fraccionarios comprendidos entre 0 y 1 son unas 262 fracciones diferentes, la probabilidad de obtener cualquiera de esos dos valores experimentalmente podría exceder el tiempo de vida de una computadora, o incluso del propio experimentador. En realidad, 0.0 *sí que está incluido* en el rango de salida de `Math.random()`. O, dicho en jerga matemática, el intervalo es [0,1]. Por tanto, debemos tener cuidado a la hora de realizar nuestros experimentos con el fin de asegurarnos de que entendemos sus limitaciones.

Selección de un tipo de conjunto

Dependiendo del comportamiento que deseemos, podemos elegir entre `TreeSet`, `HashSet` o `LinkedHashSet`. El siguiente programa de pruebas proporciona una indicación de los compromisos de rendimiento que afectan a estas implementaciones:

```
//: containers/SetPerformance.java
// Ilustra las diferencias de rendimiento entre conjuntos.
// {Args: 100 5000} pequeño para que las pruebas sean cortas
import java.util.*;

public class SetPerformance {
    static List<Test<Set<Integer>>> tests =
        new ArrayList<Test<Set<Integer>>>();
    static {
        tests.add(new Test<Set<Integer>>("add") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
```

```

        for(int i = 0; i < loops; i++) {
            set.clear();
            for(int j = 0; j < size; j++)
                set.add(j);
        }
        return loops * size;
    });
}
tests.add(new Test<Set<Integer>>("contains") {
    int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops;
        int span = tp.size * 2;
        for(int i = 0; i < loops; i++)
            for(int j = 0; j < span; j++)
                set.contains(j);
        return loops * span;
    }
});
tests.add(new Test<Set<Integer>>("iterate") {
    int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops * 10;
        for(int i = 0; i < loops; i++) {
            Iterator<Integer> it = set.iterator();
            while(it.hasNext())
                it.next();
        }
        return loops * set.size();
    }
});
}
public static void main(String[] args) {
    if(args.length > 0)
        Tester.defaultParams = TestParam.array(args);
    Tester.fieldWidth = 10;
    Tester.run(new TreeSet<Integer>(), tests);
    Tester.run(new HashSet<Integer>(), tests);
    Tester.run(new LinkedHashSet<Integer>(), tests);
}
/* Output: (Sample)
----- TreeSet -----
size      add  contains   iterate
  10       746     173       89
  100      501     264       68
  1000     714     410       69
10000     1975    552       69
----- HashSet -----
size      add  contains   iterate
  10       308     91        94
  100      178     75        73
  1000     216     110       72
10000     711     215       100
----- LinkedHashMap -----
size      add  contains   iterate
  10       350     65        83
  100      270     74        55
  1000     303     111       54
10000     1615    256       58
*///:-
```

El rendimiento de **HashSet** generalmente es superior al de **TreeSet**, pero especialmente a la hora de añadir elementos y de buscarlos, que son las dos operaciones más importantes. **TreeSet** existe porque mantiene sus elementos en orden, de modo que sólo se suele utilizar cuando hace falta un contenedor Set ordenado. Debido a la estructura interna necesaria para soportar las ordenaciones y debido a que la iteración suele ser una operación muy común, la iteración suele resultar más rápida con **TreeSet** que con **HashSet**.

Observe que **LinkedHashSet** es más caro respecto a las inserciones que **HashSet**; esto se debe al coste adicional de mantener la lista enlazada además del contenedor *hash*.

Ejercicio 34: (1) Modifique **SetPerformance.java** para que los conjuntos almacenen objetos **String** en lugar de objetos **Integer**. Utilice el objeto **Generator** del Capítulo 16, *Matrices* para crear valores de prueba.

Selección de un tipo de mapa

Este programa nos proporciona una indicación de los compromisos de rendimiento en las distintas implementaciones de **Map**:

```
//: containers/MapPerformance.java
// Ilustra la diferencias de rendimiento entre mapas.
// {Args: 100 5000} pequeño para mantener corto el tiempo de prueba
import java.util.*;

public class MapPerformance {
    static List<Test<Map<Integer, Integer>>> tests =
        new ArrayList<Test<Map<Integer, Integer>>>();
    static {
        tests.add(new Test<Map<Integer, Integer>>("put") {
            int test(Map<Integer, Integer> map, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    map.clear();
                    for(int j = 0; j < size; j++)
                        map.put(j, j);
                }
                return loops * size;
            }
        });
        tests.add(new Test<Map<Integer, Integer>>("get") {
            int test(Map<Integer, Integer> map, TestParam tp) {
                int loops = tp.loops;
                int span = tp.size * 2;
                for(int i = 0; i < loops; i++)
                    for(int j = 0; j < span; j++)
                        map.get(j);
                return loops * span;
            }
        });
        tests.add(new Test<Map<Integer, Integer>>("iterate") {
            int test(Map<Integer, Integer> map, TestParam tp) {
                int loops = tp.loops * 10;
                for(int i = 0; i < loops; i++) {
                    Iterator it = map.entrySet().iterator();
                    while(it.hasNext())
                        it.next();
                }
                return loops * map.size();
            }
        });
    }
}
```

```

}
public static void main(String[] args) {
    if(args.length > 0)
        Tester.defaultParams = TestParam.array(args);
    Tester.run(new TreeMap<Integer, Integer>(), tests);
    Tester.run(new HashMap<Integer, Integer>(), tests);
    Tester.run(new LinkedHashMap<Integer, Integer>(), tests);
    Tester.run(
        new IdentityHashMap<Integer, Integer>(), tests);
    Tester.run(new WeakHashMap<Integer, Integer>(), tests);
    Tester.run(new Hashtable<Integer, Integer>(), tests);
}
/* Output: (Sample)
----- TreeMap -----
size      put      get iterate
  10      748     168     100
  100     505     264      76
  1000    771     450      78
  10000   2962     561      83
----- HashMap -----
size      put      get iterate
  10      281      76      93
  100     179      70      73
  1000    267     102      72
  10000   1305     265      97
----- LinkedHashMap -----
size      put      get iterate
  10      354     100      72
  100     273      89      50
  1000    385     222      56
  10000   2787     341      56
----- IdentityHashMap -----
size      put      get iterate
  10      290     144     101
  100     204     287     132
  1000    508     336      77
  10000   767     266      56
----- WeakHashMap -----
size      put      get iterate
  10      484     146     151
  100     292     126     117
  1000    411     136     152
  10000   2165     138     555
----- Hashtable -----
size      put      get iterate
  10      264     113     113
  100     181     105      76
  1000    260     201      80
  10000   1245     134      77
*//*/-

```

Las inserciones en todas las implementaciones de **Map** excepto en **IdentityHashMap** van siendo significativamente más lentas a medida que el tamaño del mapa se incrementa. Sin embargo, en general, la búsqueda es mucho menos costosa que la inserción, lo cual resulta muy adecuado, porque lo normal es que busquemos elementos con mucha más frecuencia de la que los insertamos.

El rendimiento de **Hashtable** es aproximadamente igual al de **HashMap**. Puesto que **HashMap** pretende sustituir a **Hashtable**, y utiliza por tanto la misma estructura subyacente de almacenamiento y el mismo mecanismo de búsqueda (de lo que hablaremos posteriormente), no resulta demasiado sorprendente que tenga un rendimiento mejor.

TreeMap es generalmente más lento que **HashMap**. Al igual que sucede con **TreeSet**, **TreeMap** es una forma de crear una lista ordenada. El comportamiento de un árbol es tal que sus elementos siempre están en orden, sin que sea necesario ordenarlos especialmente. Una vez llenado un contenedor **TreeMap**, podemos invocar **keySet()** para obtener una vista de tipo **Set** de las claves y luego podemos llamar a **toArray()** para generar una matriz de dichas claves. Podemos a continuación emplear un método estático **Arrays.binarySearch()** para localizar rápidamente objetos en esa matriz ordenada. Por supuesto, esto sólo tiene sentido si el comportamiento de un contenedor **HashMap** no es aceptable, ya que **HashMap** está diseñado para poder localizar rápidamente las claves. Asimismo, podemos crear rápidamente un contenedor **HashMap** a partir de otro de tipo **TreeMap** mediante una única creación de objeto o una llamada a **putAll()**. En resumen, cuando estemos utilizando un mapa nuestra primera elección deberá ser **HashMap**, y sólo deberíamos recurrir a **TreeMap** si lo que necesitamos es un mapa que esté constantemente ordenado.

LinkedHashMap tiende a ser más lento que **HashMap** para las inserciones, porque mantiene la lista enlazada (con el fin de preservar el orden de inserción), además de mantener la estructura de datos *hash*. Sin embargo, debido a esta lista, la iteración es más rápida.

IdentityHashMap tiene un rendimiento distinto porque utiliza **==** en lugar de **equals()** para hacer las comparaciones. **WeakHashMap** se describe más adelante en el capítulo.

Ejercicio 35: (1) Modifique **MapPerformance.java** para incluir pruebas de **SlowMap**.

Ejercicio 36: (5) Modifique **SlowMap** de modo que, en lugar de dos contenedores **ArrayList**, almacene un único **ArrayList** de objetos **MapEntry**. Verifique que la versión modificada funciona correctamente. Utilizando **MapPerformance.java**, compruebe la velocidad de su nuevo mapa. Ahora cambie el método **put()** de modo que realice con **sort()** una ordenación después de introducir cada pareja y modifique **get()** para utilizar **Collections.binarySearch()** con el fin de buscar la clave. Compare el rendimiento de la nueva versión con el de las anteriores.

Ejercicio 37: (2) Modifique **SimpleHashMap** para utilizar contenedores **ArrayList** en lugar de **LinkedList**. Modifique **MapPerformance.java** para comparar el rendimiento de las dos implementaciones.

Factores de rendimiento que afectan a **HashMap**

Resulta posible optimizar manualmente un contenedor **HashMap** para incrementar su rendimiento de cara a una aplicación concreta. Pero para poder comprender las cuestiones de rendimiento a la hora de optimizar un contenedor **HashMap**, nos hace falta algo de terminología:

Capacidad: el número de segmentos de la tabla.

Capacidad inicial: el número de segmentos en el momento de crear la tabla. **HashMap** y **HashSet** tienen constructores que permiten especificar la capacidad inicial.

Tamaño: el número de entradas que hay actualmente en la tabla.

Factor de carga: Tamaño/capacidad. Un factor de carga de 0 representa una tabla vacía, 0.5 es una tabla medio llena, etc. Una tabla ligeramente cargada tendrá menos colisiones y por tanto resulta óptima de cara a las inserciones y búsquedas (aunque ralentizará el proceso de recorrer el contenedor con un iterador). **HashMap** y **HashSet** tienen constructores que permiten especificar el factor de carga, lo que significa que cuando se alcanza este factor de carga, el contenedor incrementará automáticamente la capacidad (el número de segmentos), por el procedimiento de aproximadamente duplicar dicho número y luego redistribuir los objetos existentes en el nuevo conjunto de segmentos (este proceso se denomina *rehashing*).

El factor de carga predeterminado utilizado por **HashMap** es 0.75 (no efectúa una redistribución hasta que la tabla está llena en sus tres cuartas partes). Éste parece ser un buen compromiso entre los costes de tiempo y de espacio de almacenamiento. Un factor de carga más alto reduce el espacio requerido por la tabla pero incrementa el coste de búsqueda, lo cual es importante, porque la mayor parte del tiempo es búsquedas (incluyendo tanto **get()** como **put()**).

Si sabemos de antemano que vamos a almacenar numerosas entradas en un contenedor **HashMap**, crearlo con una capacidad inicial suficientemente grande evitará el gasto adicional del cambio de tamaño automático.¹¹

¹¹ En un mensaje privado, Joshua Bloch escribió: "... Creo que hemos cometido un error al incluir detalles de implementación (como el factor de carga y el tamaño de las tablas *hash*) en nuestras API. El cliente debería, quizás, decírnos el tamaño máximo esperado de una colección y nosotros deberíamos actuar

Ejercicio 38: (3) Examine la clase **HashMap** de la documentación del JDK. Cree un contenedor **HashMap**, rellénelo con elementos y determine el factor de carga. Pruebe la velocidad de búsqueda con este mapa, luego trate de incrementar la velocidad creando un nuevo contenedor **HashMap** con una capacidad inicial mayor y copiando el mapa antiguo en el nuevo; después, ejecute otra vez la prueba de velocidad de búsqueda con el nuevo mapa.

Ejercicio 39: (6) Añada un método privado **rehash()** a **SimpleHashMap** que se invoque cuando el factor de carga excede de 0.75. Durante el cambio automático de tamaño, duplique el número de segmentos y luego busque el primer número primo superior a ese número, con el fin de determinar el nuevo número de segmentos.

Utilidades

Hay diversas utilidades autónomas para contenedores, expresadas como métodos estáticos dentro de la clase **java.util.Collections**. Ya hemos visto algunas de ellas, como **addAll()**, **reverseOrder()** y **binarySearch()**. He aquí las otras (las utilidades de tipo **synchronized** y **unmodifiable** se cubrirán en las secciones siguientes). En esta tabla, se usan genéricos allí donde son relevantes:

| | |
|--|---|
| <code>checkedCollection(Collection<T>, Class<T> type)</code> <code>checkedList(List<T>, Class<T> type)</code> <code>checkedMap(Map<K,V>, Class<K> keyType, Class<V> valueType)</code> <code>checkedSet(Set<T>, Class<T> type)</code> <code>checkedSortedMap(SortedMap<K,V>, Class<K> keyType, Class<V> valueType)</code> <code>checkedSortedSet(SortedSet<T>, Class<T> type)</code> | Produce una vista <i>dinámicamente segura</i> con respecto a tipos de Collection , o de un subtipo específico de Collection . Utilice este método cuando no sea posible emplear la versión con comprobación estática. Ya hemos visto estos métodos en el Capítulo 15, <i>Genéricos</i> , en la sección “Seguridad dinámica de tipos”. |
| <code>max(Collection)</code> <code>min(Collection)</code> | Devuelve el elemento máximo o mínimo contenido en el argumento utilizando el método de comparación natural de los objetos de la colección. |
| <code>max(Collection, Comparator)</code> <code>min(Collection, Comparator)</code> | Devuelve el elemento máximo o mínimo del objeto Collection utilizando el objeto Comparator . |
| <code>indexOfSubList(List source, List target)</code> | Devuelve el índice de inicio del <i>primer</i> lugar donde target aparece dentro de source , o 21 si no aparece. |
| <code>lastIndexOfSubList(List source, List target)</code> | Devuelve el índice de inicio del <i>último</i> lugar donde target aparece dentro de source , o 21 si no aparece. |
| <code>replaceAll(List<T>, T oldVal, T newVal)</code> | Reemplaza todos los valores oldVal por newVal . |
| <code>reverse(List)</code> | Invierte el orden de todos los elementos en la lista. |
| <code>reverseOrder()</code> <code>reverseOrder(Comparator<T>)</code> | Devuelve un objeto Comparator que invierte la ordenación natural de una colección de objetos que implemente Comparable<T> . La segunda versión invierte el orden del objeto Comparator suministrado. |
| <code>rotate(List, int distance)</code> | Mueve todos los elementos hacia adelante una distancia distance , extrayéndolos por el extremo y volviendo a colocarlos al principio. |

¹¹(continuación) a partir de ahí. Los clientes pueden, fácilmente, generar más problemas que beneficios al seleccionar los valores de estos parámetros. Como ejemplo exagerado, considera el valor **capacityIncrement** de **Vector**. Nadie debería configurar este valor y no deberíamos haberlo incluido. Si se le asigna un valor distinto de cero, el coste asintótico de una secuencia de adiciones pasa de lineal a cuadrático. En otras palabras, el rendimiento se viene abajo. Con el paso del tiempo, cada vez tenemos más experiencia con este tipo de cosa. Si examinas **IdentityHashMap**, verás que no dispone de ningún parámetro de optimización de bajo nivel”.

| | |
|---|--|
| <code>shuffle(List)</code> | Permuta de manera aleatoria la lista especificada. La primera forma proporciona su propio mecanismo de aleatorización, o bien podemos proporcionar nuestro propio mecanismo con la segunda forma. |
| <code>sort(List<T>)</code> <code>sort(List<T>, Comparator<? super T> c)</code> | Ordena la lista <code>List<T></code> utilizando una ordenación natural. La segunda forma permite proporcionar un objeto <code>Comparator</code> para la ordenación. |
| <code>copy(List<? super T> dest, List<? extends T> src)</code> | Copia elementos desde <code>src</code> a <code>dest</code> . |
| <code>swap(List, int i, int j)</code> | Intercambia los elementos en las posiciones <code>i</code> y <code>j</code> dentro de <code>List</code> . Probablemente más rápido que los métodos que pudieramos escribir nosotros. |
| <code>fill(List<? super T>, T x)</code> | Sustituye todos los elementos de la lista por <code>x</code> . |
| <code>nCopies(int n, T x)</code> | Devuelve una lista inmutable <code>List<T></code> de tamaño <code>n</code> cuyas referencias apuntan todas a <code>x</code> . |
| <code>disjoint(Collection, Collection)</code> | Devuelve <code>true</code> si las dos colecciones no tienen elementos en común. |
| <code>frequency(Collection, Object x)</code> | Devuelve el número de elementos de <code>Collection</code> que sean iguales a <code>x</code> . |
| <code>emptyList()</code> <code>emptyMap()</code> <code>emptySet()</code> | Devuelve un objeto <code>List</code> , <code>Map</code> o <code>Set</code> vacío inmutable. Son objetos genéricos, por lo que el objeto <code>Collection</code> resultante estará parametrizado con el tipo deseado. |
| <code>singleton(T x)</code> <code>singletonList(T x)</code> <code>singletonMap(K key, V value)</code> | Produce un objeto <code>Set<T></code> , <code>List<T></code> , o <code>Map<K,V></code> inmutable que contiene una única entrada basada en los argumentos proporcionados. |
| <code>list(Enumeration<T> e)</code> | Produce un objeto <code>ArrayList<T></code> que contiene los elementos en el orden en el que son devueltos por el (antiguo) objeto <code>Enumeration</code> (predecesor de <code>Iterator</code>). Para la conversión de código heredado. |
| <code>enumeration(Collection<T>)</code> | Genera un objeto <code>Enumeration<T></code> de estilo antiguo para el argumento. |

Observe que `min()` y `max()` funcionan con objetos `Collection` no con listas, por lo que no es necesario preocuparse acerca de si la colección ya está ordenada o no (como ya hemos mencionado anteriormente, *sí* que es necesario ordenar con `sort()` una lista o una matriz antes de realizar una búsqueda binaria con `binarySearch()`).

He aquí un ejemplo que muestra el uso básico de la mayoría de las utilidades de la tabla anterior:

```
//: containers/Utilities.java
// Ejemplo simple de las utilidades para colecciones.
import java.util.*;
import static net.mindview.util.Print.*;

public class Utilities {
    static List<String> list = Arrays.asList(
        "one Two three Four five six one".split(" "));
    public static void main(String[] args) {
        print(list);
        print("list disjoint (Four)?: " +
            Collections.disjoint(list,
                Collections.singletonList("Four")));
        print("max: " + Collections.max(list));
        print("min: " + Collections.min(list));
        print("max w/ comparator: " + Collections.max(list,
```

```

    String.CASE_INSENSITIVE_ORDER));
print("min w/ comparator: " + Collections.min(list,
    String.CASE_INSENSITIVE_ORDER));
List<String> sublist =
    Arrays.asList("Four five six".split(" "));
print("indexOfSubList: " +
    Collections.indexOfSubList(list, sublist));
print("lastIndexOfSubList: " +
    Collections.lastIndexOfSubList(list, sublist));
Collections.replaceAll(list, "one", "Yo");
print("replaceAll: " + list);
Collections.reverse(list);
print("reverse: " + list);
Collections.rotate(list, 3);
print("rotate: " + list);
List<String> source =
    Arrays.asList("in the matrix".split(" "));
Collections.copy(list, source);
print("copy: " + list);
Collections.swap(list, 0, list.size() - 1);
print("swap: " + list);
Collections.shuffle(list, new Random(47));
print("shuffled: " + list);
Collections.fill(list, "pop");
print("fill: " + list);
print("frequency of 'pop': " +
    Collections.frequency(list, "pop"));
List<String> dups = Collections.nCopies(3, "snap");
print("dups: " + dups);
print("'list' disjoint 'dups'? " +
    Collections.disjoint(list, dups));
// Obtención de un objeto Enumeration al estilo antiguo:
Enumeration<String> e = Collections.enumeration(dups);
Vector<String> v = new Vector<String>();
while(e.hasMoreElements())
    v.addElement(e.nextElement());
// Conversión de un vector de estilo antiguo
// en una lista a través de un objeto Enumeration:
ArrayList<String> arrayList =
    Collections.list(v.elements());
print("arrayList: " + arrayList);
}
} /* Output:
[one, Two, three, Four, five, six, one]
'list' disjoint (Four)?: false
max: three
min: Four
max w/ comparator: Two
min w/ comparator: five
indexOfSubList: 3
lastIndexOfSubList: 3
replaceAll: [Yo, Two, three, Four, five, six, Yo]
reverse: [Yo, six, five, Four, three, Two, Yo]
rotate: [three, Two, Yo, Yo, six, five, Four]
copy: [in, the, matrix, Yo, six, five, Four]
swap: [Four, the, matrix, Yo, six, five, in]
shuffled: [six, matrix, the, Four, Yo, five, in]
fill: [pop, pop, pop, pop, pop, pop, pop]
frequency of 'pop': 7

```

```

dups: [snap, snap, snap]
'list' disjoint 'dups'?: true
arraylist: [snap, snap, snap]
*///:-
```

La salida explica el comportamiento de cada método de utilidad. Obsérve la diferencia en `min()` y `max()` con el objeto `String.CASE_INSENSITIVE_ORDER Comparator` debido a la utilización de mayúsculas y minúsculas.

Ordenaciones y búsquedas en las listas

Las utilidades para realizar ordenaciones y búsquedas en las listas tienen los mismos nombres y signaturas que se emplean para ordenar matrices de objetos, pero se trata de métodos estáticos de `Collections` en lugar de ser métodos de `Arrays`. He aquí un ejemplo que emplea la lista de datos `list` de `Utilities.java`:

```

//: containers/ListSortSearch.java
// Ordenación y búsqueda en listas con las utilidades de Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class ListSortSearch {
    public static void main(String[] args) {
        List<String> list =
            new ArrayList<String>(Utilities.list);
        list.addAll(Utilities.list);
        print(list);
        Collections.shuffle(list, new Random(47));
        print("Shuffled: " + list);
        // Utilizar ListIterator para eliminar los últimos elementos:
        ListIterator<String> it = list.listIterator(10);
        while(it.hasNext()) {
            it.next();
            it.remove();
        }
        print("Trimmed: " + list);
        Collections.sort(list);
        print("Sorted: " + list);
        String key = list.get(7);
        int index = Collections.binarySearch(list, key);
        print("Location of " + key + " is " + index +
            ", list.get(" + index + ") = " + list.get(index));
        Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sorted: " + list);
        key = list.get(7);
        index = Collections.binarySearch(list, key,
            String.CASE_INSENSITIVE_ORDER);
        print("Location of " + key + " is " + index +
            ", list.get(" + index + ") = " + list.get(index));
    }
} /* Output:
[one, Two, three, Four, five, six, one, one, Two, three,
Four, five, six, one]
Shuffled: [Four, five, one, one, Two, six, six, three,
three, five, Four, Two, one, one]
Trimmed: [Four, five, one, one, Two, six, six, three,
three, five]
Sorted: [Four, Two, five, five, one, one, six, six, three,
three]
Location of six is 7, list.get(7) = six
Case-insensitive sorted: [five, five, Four, one, one, six,
```

```
six, three, three, Two]
Location of three is 7, list.get(7) = three
*///:-
```

Al igual que cuando se realizan búsquedas y ordenaciones con matrices, si ordenamos utilizando un objeto **Comparator**, debemos efectuar la búsqueda con **binarySearch()** usando el mismo objeto **Comparator**.

Este programa también ilustra el método **shuffle()** de **Collections**, que aleatoriza el orden de una lista. Se crea un objeto **ListIterator** en una posición concreta de la lista aleatorizada y se utiliza para eliminar los elementos comprendidos entre dicha posición y el final de la lista.

Ejercicio 40: (5) Cree una clase que contenga dos objetos **String** y haga que sea de tipo **Comparable** de modo que la comparación sólo tenga en cuenta el primer objeto **String**. Rellene una matriz y un contenedor **ArrayList** con objetos de esa clase, utilizando el generador **RandomGenerator**. Demuestre que la ordenación funciona apropiadamente. Ahora defina un objeto **Comparator** que sólo tenga en cuenta el segundo objeto **String** y demuestre que la ordenación funciona correctamente. Asimismo, realice una búsqueda binaria utilizando ese objeto **Comparator**.

Ejercicio 41: (3) Modifique la clase del ejercicio anterior para que funcione con contenedores **HashSet** y como clave en contenedores **HashMap**.

Ejercicio 42: (2) Modifique el Ejercicio 40 para que se utilice una ordenación alfabética.

Creación de colecciones o mapas no modificables

A menudo, resulta conveniente crear una versión de sólo lectura de una colección o mapa. La clase **Collections** permite hacer esto, pasando el contenedor original a un método que devuelve una versión de sólo lectura. Hay diversas variantes de este método, para colecciones (si no se puede tratar un objeto **Collection** como si fuera de tipo más específico), listas, conjuntos y mapas. El siguiente ejemplo muestra la forma de construir versiones de sólo lectura de cada uno de estos contenedores:

```
//: containers/ReadOnly.java
// Utilización de los métodos Collections.unmodifiable.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ReadOnly {
    static Collection<String> data =
        new ArrayList<String>(Countries.names(6));
    public static void main(String[] args) {
        Collection<String> c =
            Collections.unmodifiableCollection(
                new ArrayList<String>(data));
        print(c); // Se puede leer
        // c.add("one"); // No se puede modificar

        List<String> a = Collections.unmodifiableList(
            new ArrayList<String>(data));
        ListIterator<String> lit = a.listIterator();
        print(lit.next()); // Se puede leer
        // lit.add("one"); // No se puede modificar

        Set<String> s = Collections.unmodifiableSet(
            new HashSet<String>(data));
        print(s); // Se puede leer
        // s.add("one"); // No se puede modificar

        // Para un contenedor SortedSet:
        Set<String> ss = Collections.unmodifiableSortedSet(
```

```

    new TreeSet<String>(data));

Map<String, String> m = Collections.unmodifiableMap(
    new HashMap<String, String>(Countries.capitals(6)));
print(m); // Se puede leer
//! m.put("Ralph", "Howdy!");

// Para un contenedor SortedMap:
Map<String, String> sm =
    Collections.unmodifiableSortedMap(
        new TreeMap<String, String>(Countries.capitals(6)));
}

} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
ALGERIA
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
{BULGARIA=Sofia, BURKINA FASO=Ouagadougou, BOTSWANA=Gaberone,
BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
*///:-

```

La invocación del método “unmodifiable” (no modificable) para un tipo concreto no hace que se generen advertencias en tiempo de compilación, pero una vez que la transformación se ha producido, cualquier llamada a un método que modifique el contenido de un contenedor concreto generará una excepción **UnsupportedOperationException**.

En cada caso, debe llenar el contenedor con datos significativos *antes* de hacerlo de sólo lectura. Una vez cargado, lo mejor es sustituir la referencia existente por la referencia generada por la llamada al método “unmodifiable”. De esa forma, no corremos el riesgo de tratar de modificar accidentalmente el contenido una vez que hemos dicho que no es modificable. Por otro lado, esta herramienta también permite mantener el contenedor modificable como privado dentro de una clase y devolver una referencia de sólo lectura a dicho contenedor desde una llamada a método. De este modo, podemos cambiar el contenedor dentro de la clase, pero desde cualquier otro lugar sólo podrá leerse.

Sincronización de una colección o un mapa

La palabra clave **synchronized** es una parte importante del tema de la *programación multihebra*, un tema más complicado del que no hablaremos hasta el Capítulo 21, *Concurrencia*. Aquí, nos limitaremos a resaltar que la clase **Collections** contiene una forma de sincronizar automáticamente un contenedor completo. La sintaxis es similar a la de los métodos “unmodifiable”:

```

//: containers/Synchronization.java
// Utilización de los métodos Collections.synchronized.
import java.util.*;

public class Synchronization {
    public static void main(String[] args) {
        Collection<String> c =
            Collections.synchronizedCollection(
                new ArrayList<String>());
        List<String> list = Collections.synchronizedList(
            new ArrayList<String>());
        Set<String> s = Collections.synchronizedSet(
            new HashSet<String>());
        Set<String> ss = Collections.synchronizedSortedSet(
            new TreeSet<String>());
        Map<String, String> m = Collections.synchronizedMap(
            new HashMap<String, String>());
        Map<String, String> sm =
            Collections.synchronizedSortedMap(
                new TreeMap<String, String>());
    }
} //:-

```

Lo mejor es pasar inmediatamente el nuevo contenedor a través del apropiado método "synchronized", como se muestra en el ejemplo. De esa forma, no existe ninguna posibilidad de que la versión no sincronizada quede accidentalmente expuesta.

Fallo rápido

Los contenedores de Java también tienen un mecanismo para evitar que más de un proceso modifique el contenido de un contenedor. El problema se presenta si estamos en mitad de un proceso de iteración a través de un contenedor y entonces algún otro proceso interviene e inserta, modifica o elimina un objeto de dicho contenedor. Puede que ya hayamos pasado dicho elemento del contenedor, o puede que todavía no hayamos llegado a ese elemento, puede que el tamaño del contenedor se reduzca después de que hayamos invocado `size()`... existen muchas posibilidades de que se produzca un desastre. La biblioteca de contenedores de Java utiliza un mecanismo de *fallo rápido* que examina si se ha producido en el contenedor algún cambio distinto de aquellos de los que nuestro proceso es personalmente responsable. Si detecta que alguien más está modificando el contenedor, genera inmediatamente una excepción `ConcurrentModificationException`. A eso se refiere el término de fallo rápido: no trata de detectar un problema más adelante utilizando un algoritmo más complejo.

Resulta bastante fácil ver el mecanismo de fallo rápido en acción: lo único que hace falta es crear un iterador y luego añadir algo a la colección a la que el iterador esté apuntando, como el ejemplo siguiente:

```
//: containers/FailFast.java
// Ilustración del comportamiento del "fallo rápido".
import java.util.*;

public class FailFast {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        Iterator<String> it = c.iterator();
        c.add("An object");
        try {
            String s = it.next();
        } catch(ConcurrentModificationException e) {
            System.out.println(e);
        }
    }
} /* Output:
java.util.ConcurrentModificationException
*///:-
```

La excepción se produce porque se ha incluido algo en el contenedor *después* de que se haya adquirido el iterador para ese contenedor. La posibilidad de que dos partes del programa puedan modificar el mismo contenedor hace que acabemos en un estado incierto, por lo que la excepción nos notifica que es necesario modificar el código; en este caso, habrá que adquirir el iterador *después* de haber añadido todos los elementos al contenedor.

Los contenedores `ConcurrentHashMap`, `CopyOnWriteArrayList` y `CopyOnWriteArraySet` utilizan técnicas que impiden que se genere la excepción `ConcurrentModificationException`.

Almacenamiento de referencias

La biblioteca `java.lang.ref` contiene un conjunto de clases que aumentan la flexibilidad del mecanismo de depuración de memoria. Estas clases son especialmente útiles cuando tenemos objetos de gran tamaño que puedan hacer que la memoria se agote. Existen tres clases heredadas de la clase abstracta `Reference`: `SoftReference`, `WeakReference` y `PhantomReference`. Cada una de ellas proporciona un nivel distinto de indirección para el depurador de memoria si el objeto en cuestión sólo es alcanzable a través de uno de estos objetos `Reference`.

Si un objeto es *alcanzable*, quiere decir que en algún lugar del programa puede encontrarse el objeto. Esto puede querer decir que disponemos de una referencia normal en la pila que apunta directamente al objeto, pero también podemos tener una referencia a un objeto que tenga a su vez una referencia al objeto en cuestión; puede incluso haber muchos enlaces intermedios. Si un objeto es alcanzable, el depurador de memoria no puede ignorarlo, porque sigue siendo utilizado por el programa. Si el objeto no es alcanzable, no hay ninguna forma de que nuestro programa lo use, así que resulta seguro depurar dicho objeto de la memoria.

Utilizamos objetos **Reference** cuando queremos continuar almacenando una referencia al objeto (es decir, queremos poder alcanzar el objeto), pero también queremos permitir que el depurador de memoria libere el objeto. De este modo, tenemos una forma de utilizar el objeto, pero si está a punto de agotarse la memoria, dejamos que ese objeto sea eliminado.

Para hacer esto, utilizamos un objeto **Reference** como intermediario (*proxy*) entre nosotros y la referencia normal. Además, no debe haber referencias normales al objeto (es decir, referencias que no estén envueltas dentro de objetos de tipo **Reference**). Si el depurador de memoria descubre que un objeto es alcanzable a través de una referencia normal, no podrá liberar dicho objeto.

Ordenando los distintos tipos de referencias **SoftReference**, **WeakReference** y **PhantomReference**, cada uno de ellos es “más débil” que el anterior y se corresponde con un nivel distinto de alcanzabilidad. Las referencias blandas (*soft references*) son para implementar cachés sensibles a la memoria. Las referencias débiles (*weak references*) sirven para implementar “mapas canónicos” (con los que pueden usarse instancias de objetos simultáneamente en múltiples lugares de un programa, con el fin de ahorrar espacio de almacenamiento) que no impiden que sus claves o valores sean eliminados. Las referencias fantasma (*phantom references*) sirven para planificar acciones de limpieza pre-mortem de una manera más flexible de lo que puede conseguirse con el mecanismo de finalización de Java.

Con **SoftReference** y **WeakReference**, tenemos la opción de situar esas referencias en una cola de referencias de tipo **ReferenceQueue** (que se utiliza para acciones de limpieza pre-mortem), pero una referencia **PhantomReference** tiene obligatoriamente que construirse dentro de una cola **ReferenceQueue**. He aquí un ejemplo simple:

```
//: containers/References.java
// Ejemplo de objetos Reference
import java.lang.ref.*;
import java.util.*;

class VeryBig {
    private static final int SIZE = 10000;
    private long[] la = new long[SIZE];
    private String ident;
    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    protected void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    private static ReferenceQueue<VeryBig> rq =
        new ReferenceQueue<VeryBig>();
    public static void checkQueue() {
        Reference<? extends VeryBig> inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " + inq.get());
    }
    public static void main(String[] args) {
        int size = 10;
        // O bien elegir el tamaño a través de la línea de comandos:
        if(args.length > 0)
            size = new Integer(args[0]);
        LinkedList<SoftReference<VeryBig>> sa =
            new LinkedList<SoftReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            sa.add(new SoftReference<VeryBig>(
                new VeryBig("Soft " + i), rq));
            System.out.println("Just created: " + sa.getLast());
            checkQueue();
        }
        LinkedList<WeakReference<VeryBig>> wa =
            new LinkedList<WeakReference<VeryBig>>();
```

```

for(int i = 0; i < size; i++) {
    wa.add(new WeakReference<VeryBig>(
        new VeryBig("Weak " + i), rq));
    System.out.println("Just created: " + wa.getLast());
    checkQueue();
}
SoftReference<VeryBig> s =
    new SoftReference<VeryBig>(new VeryBig("Soft"));
WeakReference<VeryBig> w =
    new WeakReference<VeryBig>(new VeryBig("Weak"));
System.gc();
LinkedList<PhantomReference<VeryBig>> pa =
    new LinkedList<PhantomReference<VeryBig>>();
for(int i = 0; i < size; i++) {
    pa.add(new PhantomReference<VeryBig>(
        new VeryBig("Phantom " + i), rq));
    System.out.println("Just created: " + pa.getLast());
    checkQueue();
}
}
} /* (Execute to see output) *///:-

```

Cuando se ejecuta este programa (conviene redirigir la salida a un archivo de texto para poder ver la salida página a página), podemos ver que los objetos se depuran de la memoria, aún cuando seguimos teniendo acceso a ellos a través del objeto **Reference** (para obtener la referencia real al objeto se utiliza `get()`). También podemos ver que el objeto **ReferenceQueue** siempre genera un objeto **Reference** que contiene un objeto **null**. Para usar éste, herede de una clase **Reference** concreta y añada otros métodos más útiles a la nueva clase.

WeakHashMap

La biblioteca de contenedores dispone de un tipo especial de mapa para almacenar referencias débiles **WeakHashMap**. Esta clase está diseñada para facilitar la creación de mapas canónicos. En dicho tipo de mapas se ahorra espacio de almacenamiento creando una instancia de cada valor concreto. Cuando el programa necesita dicho valor, busca el objeto existente en el mapa y lo utiliza (en lugar de crear uno partiendo de cero). El mapa puede construir los valores como parte de su proceso de inicialización, pero lo más probable es que los valores se construyan a medida que son necesarios.

Puesto que se trata de una técnica de ahorro de espacio de almacenamiento, resulta bastante útil que **WeakHashMap** permita al depurador de memoria limpiar automáticamente las claves y los valores. No hace falta hacer nada especial con las claves y valores que se incluyan en el contenedor **WeakHashMap**; dichas claves y valores son envueltos automáticamente por el mapa en referencias de tipo **WeakReference**. Lo que hace que quede permitida la tarea de limpieza del depurador de memoria es que la clave ya no esté siendo utilizada, como se ilustra en el siguiente ejemplo:

```

//: containers/CanonicalMapping.java
// Ilustra el contenedor WeakHashMap.
import java.util.*;

class Element {
    private String ident;
    public Element(String id) { ident = id; }
    public String toString() { return ident; }
    public int hashCode() { return ident.hashCode(); }
    public boolean equals(Object r) {
        return r instanceof Element &&
            ident.equals(((Element)r).ident);
    }
    protected void finalize() {
        System.out.println("Finalizing " +
            getClass().getSimpleName() + " " + ident);
    }
}

```

```

}

class Key extends Element {
    public Key(String id) { super(id); }
}

class Value extends Element {
    public Value(String id) { super(id); }
}

public class CanonicalMapping {
    public static void main(String[] args) {
        int size = 1000;
        // O bien elegir tamaño a través de la línea de comandos:
        if(args.length > 0)
            size = new Integer(args[0]);
        Key[] keys = new Key[size];
        WeakHashMap<Key,Value> map =
            new WeakHashMap<Key,Value>();
        for(int i = 0; i < size; i++) {
            Key k = new Key(Integer.toString(i));
            Value v = new Value(Integer.toString(i));
            if(i % 3 == 0)
                keys[i] = k; // Guardar como referencias "reales"
            map.put(k, v);
        }
        System.gc();
    }
} /* (Execute to see output) */:-

```

La clase **Key** debe tener sendos métodos **hashCode()** y **equals()**, puesto que está siendo usada como clave en una estructura de datos *hash*. Ya hemos hablado anteriormente en el capítulo del método **hashCode()**.

Cuando se ejecuta el programa, vemos que el depurador de memoria se saltará una de cada tres claves, porque en la matriz **keys** se ha incluido también una referencia normal a dichas claves, por lo que esos objetos no pueden ser depurados de la memoria.

Contenedores Java 1.0/1.1

Lamentablemente, hay una gran cantidad de código que se escribió utilizando los contenedores de Java 1.0/1.1, incluso hoy día se sigue describiendo algo de código nuevo empleando dichas clases. De modo que aunque nunca vayamos a utilizar los contenedores antiguos a la hora de escribir nuevo código, sí que tenemos que ser conscientes de su existencia. De todos modos, los contenedores antiguos eran bastante limitados, así que es mucho lo que se puede contar acerca de ellos. Y, como son anacrónicos, trataremos de evitar poner demasiado énfasis en alguno de los detalles relativos a las decisiones de diseño que con esos contenedores se tomaron.

Vector y Enumeration

El único tipo de secuencia auto-expansiva en Java 1.0/1.1 era **Vector**, así que dicho contenedor se utilizaba con gran frecuencia. Sus defectos son demasiado numerosos como para describirlos aquí (véase la primera edición de este libro, disponible en inglés para descarga gratuita en www.MindView.net). Básicamente, podemos considerar este contenedor como un tipo **ArrayList** con nombres de métodos muy largos y complicados. En la biblioteca revisada de contenedores Java, se adaptó **Vector** para que pudiera funcionar como un contenedor de tipo a **Collection** y de tipo **List**. Esta solución no es excesivamente buena, ya que podría inducir a algunas personas a pensar que **Vector** ha mejorado, cuando en realidad sólo se ha incluido para poder soportar el código Java más antiguo.

Para la versión Java 1.0/1.1 del iterador se decidió inventar un nuevo nombre, “enumeration”, en lugar de usar el término con el que todo el mundo estaba familiarizado (“iterator”). La interfaz **Enumeration** es más simple que la de **Iterator**, con

sólo dos métodos y utiliza nombres de método más largos: `boolean hasMoreElements()` devuelve `true` si esta enumeración contiene más elementos y `Object nextElement()` devuelve el siguiente elemento de esta enumeración si es que existe (en caso contrario, genera una excepción).

Enumeration es sólo una interfaz, no una implementación, e incluso las nuevas bibliotecas utilizan todavía en ocasiones la interfaz **Enumeration**, lo que no resulta muy afortunado, aunque tampoco es que genere grandes problemas. En general, debemos usar **Iterator** siempre que podamos en nuestro propio código, pero aún así debemos estar preparados para encontrarnos con bibliotecas en las que nos veamos forzados a manejar contenedores de tipo **Enumeration**.

Además, podemos generar un contenedor de tipo **Enumeration** para cualquier contenedor de tipo **Collection** utilizando el método `Collections.enumeration()`, como se puede ver en el siguiente ejemplo:

```
//: containers/Enumerations.java
// Vector y Enumeration de Java 1.0/1.1.
import java.util.*;
import net.mindview.util.*;

public class Enumerations {
    public static void main(String[] args) {
        Vector<String> v =
            new Vector<String>(Countries.names(10));
        Enumeration<String> e = v.elements();
        while(e.hasMoreElements())
            System.out.print(e.nextElement() + ", ");
        // Generar un objeto Enumeration a partir de otro Collection:
        e = Collections.enumeration(new ArrayList<String>());
    }
} /* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, BURUNDI, CAMEROON, CAPE VERDE,
CENTRAL AFRICAN REPUBLIC,
*///:-
```

Para generar un objeto **Enumeration**, llamamos al método `elements()`, después de lo cual podemos usarlo para realizar una iteración en sentido directo.

La última línea crea un objeto **ArrayList** y utiliza `enumeration()` para adaptar un objeto **Enumeration** a partir del iterador de **ArrayList Iterator**. Así, si disponemos de código antiguo que necesite manejar un objeto **Enumeration**, podemos seguir usando los nuevos contenedores.

Hashtable

Como hemos visto en la comparativa de rendimiento contenida en este ejemplo, el contenedor básico **Hashtable** es muy similar a **HashMap**, incluso en lo que respecta a nombres de métodos. No hay ninguna razón para utilizar **Hashtable** en lugar de **HashMap** en el código nuevo que escribamos.

Stack

El concepto de pila ya ha sido explicado anteriormente al hablar de **LinkedList**. Lo que resulta más confuso acerca del contenedor **Stack** de Java 1.0/1.1 es que en lugar de utilizar un **Vector** empleando el mecanismo de composición, **Stack** hereda de **Vector**. Por tanto, tiene todas las características y comportamientos de **Vector** más alguna funcionalidad propia de **Stack**. Resulta difícil saber si los diseñadores decidieron conscientemente que ésta era una forma especialmente útil de hacer las cosas, o si se trata simplemente de un error absurdo; en cualquier caso, lo que está claro es que nadie revisó el diseño antes de lanzarlo a distribución, de modo que este diseño incorrecto continúa haciéndose notar todavía hoy en muchos programas (pero no debería utilizarse en ningún programa nuevo).

He aquí una ilustración simple de **Stack** en la que se inserta cada representación de tipo **String** de una enumeración **enum**. El ejemplo muestra también cómo resulta igual de fácil de utilizar un elemento **LinkedList** como pila, o bien la clase **Stack** creada en el Capítulo 11, *Almacenamiento de objetos*:

```

//: containers/Stacks.java
// Ilustración de la clase Stack.
import java.util.*;
import static net.mindview.util.Print.*;

enum Month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER }

public class Stacks {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(Month m : Month.values())
            stack.push(m.toString());
        print("stack = " + stack);
        // Tratar una pila como un Vector:
        stack.addElement("The last line");
        print("element 5 = " + stack.elementAt(5));
        print("popping elements:");
        while(!stack.isEmpty())
            println(stack.pop() + " ");
        // Utilizar un objeto LinkedList como una pila:
        LinkedList<String> lstack = new LinkedList<String>();
        for(Month m : Month.values())
            lstack.addFirst(m.toString());
        print("lstack = " + lstack);
        while(!lstack.isEmpty())
            println(lstack.removeFirst() + " ");
        // Uso de la clase Stack del Capítulo 11,
        // Almacenamiento de objetos:
        net.mindview.util.Stack<String> stack2 =
            new net.mindview.util.Stack<String>();
        for(Month m : Month.values())
            stack2.push(m.toString());
        print("stack2 = " + stack2);
        while(!stack2.isEmpty())
            println(stack2.pop() + " ");
    }
} /* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER]
element 5 = JUNE
popping elements:
The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY
JUNE MAY APRIL MARCH FEBRUARY JANUARY lstack = [NOVEMBER,
OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL, MARCH,
FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY stack2 = [NOVEMBER, OCTOBER, SEPTEMBER,
AUGUST, JULY, JUNE, MAY, APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY
*///:-
```

A partir de las constantes **enum Month** se genera una representación de **String**, que se inserta en el contenedor **Stack** mediante **push()**, y que luego se extrae de la pila mediante **pop()**. Para resaltar uno de los puntos que hemos comentado anteriormente, también se realizan operaciones de tipo, **Vector** sobre el objeto **Stack**. Esto es posible porque, debido a la

herencia, un objeto **Stack** es un **Vector**. Por tanto, todas las operaciones que puedan realizarse sobre un objeto **Vector** también podrán realizarse sobre un objeto **Stack**, como por ejemplo la operación **elementAt()**.

Como hemos mencionado anteriormente, debemos utilizar un contenedor **LinkedList** cuando queramos un contenedor con comportamiento de pila; o bien, la clase **net.mindview.util.Stack** creada a partir de la clase **LinkedList**.

BitSet

BitSet se utiliza si se quiere almacenar de manera eficiente una gran cantidad de información de tipo binario. Sólo resulta eficiente desde el punto de vista del tamaño; si lo que estamos buscando es eficiencia de acceso, este contenedor resulta ligeramente más lento que emplear una matriz nativa.

Además, el tamaño mínimo de un contenedor **BitSet** es el de los valores a **long**: 64 bits. Esto implica que si estamos almacenando algún conjunto de bits menor, como por ejemplo, 8 bits, con **BitSet** estaremos desperdiando buena parte del espacio; en este caso, sería simplemente mejor crear una clase, o una matriz, para almacenar los indicadores binarios, en caso de que el tamaño sea un problema (esto sólo será un problema si estamos creando una gran cantidad de objetos que contengan listas de información binaria; y sólo debería tomarse esta decisión después de realizar un perfilado del programa o de realizar algún otro tipo de métrica. Si tomamos la decisión basándonos simplemente en nuestra propia creencia de que algo es demasiado grande, terminaremos creando una complejidad innecesaria y desperdiando una gran cantidad de tiempo).

Un contenedor normal se expande a medida que añadimos más elementos y **BitSet** también actúa de esta misma manera. El siguiente ejemplo muestra cómo funciona **BitSet**:

```
//: containers/Bits.java
// Ilustración de BitSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bits {
    public static void printBitSet(BitSet b) {
        print("bits: " + b);
        StringBuilder bbits = new StringBuilder();
        for(int j = 0; j < b.size(); j++)
            bbits.append(b.get(j) ? "1" : "0");
        print("bit pattern: " + bbits);
    }
    public static void main(String[] args) {
        Random rand = new Random(47);
        // Tomar bit de menos peso de nextInt():
        byte bt = (byte)rand.nextInt();
        BitSet bb = new BitSet();
        for(int i = 7; i >= 0; i--)
            if((1 << i) & bt) != 0
                bb.set(i);
            else
                bb.clear(i);
        print("byte value: " + bt);
        printBitSet(bb);

        short st = (short)rand.nextInt();
        BitSet bs = new BitSet();
        for(int i = 15; i >= 0; i--)
            if((1 << i) & st) != 0
                bs.set(i);
            else
                bs.clear(i);
        print("short value: " + st);
        printBitSet(bs);
    }
}
```

Utilizamos el generador de números aleatorios para crear números aleatorios `byte`, `short` e `int`, transformando cada uno en su correspondiente patrón de bits que se almacena en un contenedor `BitSet`. Esto no causa ningún problema porque `BitSet` tiene 64 bits como mínimo, así que ninguno de estos valores hace que se tenga que incrementar el tamaño. A continuación, se crea contenedores `BitSet` de mayor tamaño. Como podemos ver en el ejemplo, cada contenedor `BitSet` se expande según es necesario.

Normalmente, resulta más conveniente utilizar un contenedor **EnumSet** (véase el Capítulo 19, *Tipos enumerados*) en lugar de **BitSet** cuando disponemos de un conjunto fijo de indicadores binarios a los que podemos asignar nombre, porque **EnumSet** nos permite manipular los nombres en lugar de las posiciones numéricas de cada bit, con lo que se reduce la posibilidad de cometer errores en el programa. **EnumSet** también nos impide añadir accidentalmente nuevas posiciones binarias, que es algo que podría causar algunos errores graves y difíciles de localizar. Las únicas razones por las que deberíamos utilizar **BitSet** en lugar de **EnumSet** son: que no sepamos hasta el momento de la ejecución cuántos indicadores binarios vamos a utilizar, o que resulte poco razonable asignar nombres a los indicadores, o que necesitemos algunas de las operaciones especiales incluidas en **BitSet** (consulte la documentación del JDK relativa a **BitSet** y **EnumSet**).

Resumen

La biblioteca de contenedores es, probablemente, la más importante de cualquier lenguaje orientado a objetos. En la mayoría de los programas se utilizarán contenedores más que cualquier otro componente de la biblioteca. Algunos lenguajes (Python, por ejemplo) incluyen incluso los componentes contenedores fundamentales (listas, mapas y conjuntos) como parte del propio lenguaje.

Como hemos visto en el Capítulo 11, *Almacenamiento de objetos*, podemos hacer varias cosas enormemente interesantes utilizando contenedores sin necesidad de mucho esfuerzo de programación. Sin embargo, llegados a un cierto punto, nos vemos obligados a conocer más detalles acerca de los contenedores para poder utilizarlos adecuadamente: en particular, debemos conocer los suficientes detalles acerca de las operaciones *hash* como para escribir nuestro propio método `hashCode()` (y debemos saber también cuándo es necesario hacer esto), y debemos conocer lo suficiente acerca de las distintas implementaciones de contenedores como para poder decidir cuál es la apropiada para nuestras necesidades. En este capítulo hemos cubierto estos conceptos y hemos proporcionado detalles útiles adicionales acerca de la biblioteca de contenedores. Llegados a este punto, el lector debería estar razonablemente bien preparado para utilizar los contenedores de Java como parte de sus tareas cotidianas de programación.

El diseño de una biblioteca de contenedores resulta complicado (como sucede con la mayoría de los problemas de diseño de bibliotecas). En C++, las clases de contenedores cubren todos los aspectos fundamentales empleando muchas clases diferentes. Evidentemente, esta solución era mejor que la que había disponible antes de que aparecieran las clases de contenedores de C++ (es decir, nada), pero no era una solución que pudiera traducirse demasiado bien a Java. En el otro extremo, yo he visto una biblioteca de contenedores que está compuesta por una única clase, “container” que actúa al mismo tiempo como secuencia lineal y como matriz asociativa. La biblioteca de contenedores de Java trata de conseguir un cierto equilibrio, la funcionalidad completa que esperaríamos obtener de una biblioteca de contenedores madura, pero con una facilidad de aprendizaje de uso superior a la de las clases contenedoras de C++ y a otras bibliotecas de contenedores similares. El resultado puede parecer algo extraño en algunos aspectos, pero, a diferencia de algunas de las decisiones tomadas en las primeras bibliotecas Java, esos aspectos extraños no son simples accidentes, sino decisiones de diseño cuidadosamente adoptadas y basadas en una serie de compromisos relativos a la complejidad de la solución adoptada.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

La creación de un buen sistema de entrada/salida (E/S) es una de las tareas más difíciles para el diseñador de lenguajes, cosa que queda de manifiesto sin más que ver la gran cantidad de técnicas distintas que se han utilizado.

Parece ser que el desafío radica en tratar de cubrir todas las posibilidades. No sólo hay diferentes fuentes y consumidores de datos de E/S con los que es necesario comunicarse: archivos, la consola, conexiones de red, etc., sino que también hay que hablar con esos distintos interlocutores de diferentes formas (secuencial, acceso aleatorio, con *buffer*, binaria, de caracteres, por líneas, por palabras, etc.).

Los diseñadores de la biblioteca Java abordaron el problema creando una gran cantidad de clases. De hecho, hay tantas clases para el sistema de E/S de Java que a primera vista uno se siente intimidado (irónicamente, el diseño del sistema de E/S en Java evita, de hecho, que se produzca una auténtica explosión de clases). Asimismo, hubo un significativo cambio de diseño en la biblioteca de E/S después de la versión Java 1.0, cuando la biblioteca original orientada a bytes fue suplementada con una serie de clases de E/S orientadas a caracteres y basadas en el conjunto de caracteres Unicode. Las clases **nio** (que quiere decir “new I/O”, nueva E/S, las cuales fueron introducidas en el JDK 1.4 y ya son, por tanto, bastante “anti-guas”) fueron añadidas para mejorar el rendimiento y la funcionalidad. Como resultado, hay un gran número de clases que es necesario aprender antes de comprender los suficientes detalles del sistema de E/S de Java como para poderlo utilizar apropiadamente. Además, es importante entender la evolución de la biblioteca de E/S, aún cuando nuestra primera reacción sea decir: “No me des la lata con cuestiones históricas, ¡limitate a enseñarme cómo utilizar las clases!”. El problema es que, sin la perspectiva histórica, podemos llegar rápidamente a sentirnos confundidos por algunas de las clases y a no tener claro cuándo deberían usarse y cuándo no.

En este capítulo proporcionaremos una introducción a las diversas clases de E/S existentes en la biblioteca estándar de Java y veremos también cómo utilizarlas.

La clase **File**

Antes de presentar las clases que se encargan propiamente de leer y de escribir flujos de datos, vamos a examinar una utilidad de la biblioteca que nos sirve de ayuda a la hora de tratar con aspectos relativos a los directorios de archivos.

La clase **File** tiene un nombre que se presta a confusión, podríamos pensar que hace referencia a un archivo, pero en realidad no es así. De hecho, “FilePath” (ruta de archivo) hubiera sido un nombre mucho mejor para esa clase. Puede representar o bien el *nombre* de un archivo concreto o los *nombres* de un conjunto de archivos en un directorio. Si se trata de un conjunto de archivos, podemos pedir que se nos devuelva dicho conjunto utilizando el método **list()**, que devuelve una matriz de objetos **String**. Tiene bastante sentido devolver una matriz en lugar de una de las clases contenedoras más flexibles, porque el número de elementos es fijo y, si queremos un listado de directorio distinto, basta con crear un objeto **File** diferente. Esta sección muestra un ejemplo de utilización de esta clase, incluyendo la interfaz asociada **FilenameFilter**.

Una utilidad para listados de directorio

Supongamos que deseamos obtener un listado de directorio. El objeto **File** puede utilizarse de dos maneras distintas. Si invocamos **list()** sin ningún argumento, obtendremos la lista completa contenida en el objeto **File**. Sin embargo, si queremos una

lista restringida, por ejemplo, todos los archivos que tengan la extensión .java, entonces debemos utilizar un “filtro de directorio”, que es una clase que especifica cómo seleccionar los objetos `File` que queramos visualizar.

He aquí el ejemplo. Observe que el resultado se ha ordenado muy fácilmente (de manera alfabética) mediante el método `java.util.Arrays.sort()` y el comparador `String.CASE_INSENSITIVE_ORDER`:

```
//: io/DirList.java
// Mostrar un listado de directorio utilizando expresiones regulares.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
}

class DirFilter implements FilenameFilter {
    private Pattern pattern;
    public DirFilter(String regex) {
        pattern = Pattern.compile(regex);
    }
    public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
    }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:-
```

La clase `DirFilter` implementa la interfaz `FilenameFilter`. Observe lo simple que es esta interfaz:

```
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

La única razón de existir de `DirFilter` es proporcionar el método `accept()` al método `list()`, de modo que éste pueda “retrollamar” a `accept()` con el fin de determinar qué nombres de archivos deben incluirse en la lista. Debido a esto, esta estructura se suele calificar como *retrollamada*. Más específicamente, éste es un ejemplo del patrón de diseño de *Estrategia*, porque `list()` implementa la funcionalidad básica y proporciona la Estrategia en la forma de un objeto `FilenameFilter`, con el fin de completar el algoritmo necesario para que `list()` proporcione su servicio. Puesto que `list()` toma un objeto `FilenameFilter` como argumento, quiere decir que podemos pasar a ese método un objeto de cualquier clase que implemente `FilenameFilter` con el fin de elegir (incluso en tiempo de ejecución) cómo debe comportarse el método `list()`. El propósito del patrón de diseño de *Estrategia* es proporcionar una dosis de flexibilidad en el comportamiento del código.

El método `accept()` debe aceptar un objeto `File` que represente el directorio en el que se encuentre un archivo concreto y un objeto `String` que contenga el nombre de dicho archivo. Recuerde que el método `list()` llama a `accept()` para cada uno

de los nombres de archivo del objeto directorio, para ver cuáles hay que incluir; esto se indica mediante el resultado de tipo **boolean** devuelto por **accept()**.

accept() utiliza un objeto **matcher** de expresiones regulares con el fin de ver si la expresión regular **regex** se corresponde con el nombre del archivo. Utilizando **accept()**, el método **list()** devuelve una matriz.

Clases internas anónimas

Este ejemplo es ideal para reescribirlo empleando una clase interna anónima (concepto descrito en el Capítulo 10, *Clases internas*). Como primera aproximación se crea un método **filter()** que devuelve una referencia a un objeto **FilenameFilter**:

```
//: io/DirList2.java
// Utilización de clases internas anónimas.
// {Args: "D.*\\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
    public static FilenameFilter filter(final String regex) {
        // Creación de la clase interna:
        return new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex);
            public boolean accept(File dir, String name) {
                return pattern.matcher(name).matches();
            }
        }; // Fin de la clase interna anónima
    }
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(filter(args[0]));
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:-
```

Observe que el argumento de **filter()** debe ser de tipo **final**. Esto es requerimiento de la clase interna anónima, para que ésta pueda emplear un objeto que está fuera de su ámbito.

Este diseño representa una mejora porque la clase **FilenameFilter** está ahora estrechamente acoplada a **DirList2**. Sin embargo, podemos llevar este enfoque un paso más allá y definir la clase interna anónima como un argumento de **list()**, en cuyo caso el ejemplo es todavía más sucinto:

```
//: io/DirList3.java
// Creación de la clase interna anónima "sobre el terreno".
// {Args: "D.*\\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
```

```

public static void main(final String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
        list = path.list();
    else
        list = path.list(new FilenameFilter() {
            private Pattern pattern = Pattern.compile(args[0]);
            public boolean accept(File dir, String name) {
                return pattern.matcher(name).matches();
            }
        });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
        System.out.println(dirItem);
}
/* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:-

```

El argumento de **main()** es ahora de tipo **final**, puesto que la clase interna anónima utiliza **args[0]** directamente.

Este ejemplo nos muestra cómo las clases internas anónimas permiten la creación de clases específicas de un solo uso para resolver ciertos problemas. Una de las ventajas de esta técnica es que mantiene aislado en un único lugar el código que resuelve un problema concreto. Por otro lado, no siempre resulta tan fácil de leer y entender dicho código, así que hay que utilizar juiciosamente esta técnica.

Ejercicio 1: (3) Modifique **DirList.java** (o una de sus variantes) para que el objeto **FilenameFilter** abra y lea cada archivo (utilizando la utilidad **net.mindview.util.TextFile**) y acepte el archivo basándose en si alguno de los argumentos finales de la línea de comandos existe en dicho archivo.

Ejercicio 2: (2) Cree una clase denominada **SortedDirList** con un constructor que tome un objeto **File** y construya una lista de directorio ordenada a partir de los archivos contenidos en dicho objeto **File**. Añada a esta clase dos métodos **list()** sobrecargados: el primero produce la lista completa y el segundo produce el subconjunto de la lista que se corresponda con su argumento (que será una expresión regular).

Ejercicio 3: (3) Modifique **DirList.java** (o una de sus variantes) para que calcule la suma de los tamaños de los archivos seleccionados.

Utilidades de directorio

Una tarea común en programación consiste en realizar operaciones sobre conjuntos de archivos, bien en el directorio local o bien recorriendo todo el árbol de directorios. Resulta útil disponer de una herramienta que produzca el conjunto de archivos para nosotros. La siguiente clase de utilidad produce una matriz de objetos **File** en el directorio local utilizando el método **local()** o un objeto **List<File>** que representa todo el árbol de directorios a partir del directorio indicado, empleando **walk()** (los objetos **File** son más útiles que los nombres de archivo porque dichos objetos contienen más información). Los archivos se eligen basándose en la expresión regular que proporcionemos:

```

//: net/mindview/util/Directory.java
// Generar una secuencia de objetos File que se correspondan
// con una expresión regular bien en el directorio local,
// o bien recorriendo un árbol de directorios.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

```

```

public final class Directory {
    public static File[] local(File dir, final String regex) {
        return dir.listFiles(new FilenameFilter() {
            private Pattern pattern = Pattern.compile(regex);
            public boolean accept(File dir, String name) {
                return pattern.matcher(
                    new File(name).getName()).matches();
            }
        });
    }
    public static File[] local(String path, final String regex) { // Sobrecargado
        return local(new File(path), regex);
    }
    // Una tupla de dos elementos para devolver una pareja de objetos:
    public static class TreeInfo implements Iterable<File> {
        public List<File> files = new ArrayList<File>();
        public List<File> dirs = new ArrayList<File>();
        // El elemento iterable predeterminado es la lista de archivos:
        public Iterator<File> iterator() {
            return files.iterator();
        }
        void addAll(TreeInfo other) {
            files.addAll(other.files);
            dirs.addAll(other.dirs);
        }
        public String toString() {
            return "dirs: " + PPrint.pformat(dirs) +
                "\n\nfiles: " + PPrint.pformat(files);
        }
    }
    public static TreeInfo walk(String start, String regex) { // Comenzar recursión
        return recurseDirs(new File(start), regex);
    }
    public static TreeInfo walk(File start, String regex) { // Sobrecargado
        return recurseDirs(start, regex);
    }
    public static TreeInfo walk(File start) { // Todo
        return recurseDirs(start, ".*");
    }
    public static TreeInfo walk(String start) {
        return recurseDirs(new File(start), ".*");
    }
    static TreeInfo recurseDirs(File startDir, String regex) {
        TreeInfo result = new TreeInfo();
        for(File item : startDir.listFiles()) {
            if(item.isDirectory()) {
                result.dirs.add(item);
                result.addAll(recurseDirs(item, regex));
            } else // Archivo normal
                if(item.getName().matches(regex))
                    result.files.add(item);
        }
        return result;
    }
    // Prueba simple de validación:
}

```

```

public static void main(String[] args) {
    if(args.length == 0)
        System.out.println(walk("."));
    else
        for(String arg : args)
            System.out.println(walk(arg));
}
} //:-

```

El método **local()** utiliza una variante de **File.list()** denominada **listFiles()** que genera una matriz de objetos **File**. Podemos ver también que utiliza un objeto **FilenameFilter**. Si hace falta una lista en lugar de una matriz, podemos convertir nosotros mismos el resultado utilizando **Arrays.asList()**.

El método **walk()** convierte el nombre del directorio de inicio en un objeto **File** e invoca **reurseDirs()**, que realiza un recorrido recursivo del directorio, recopilando más información con cada recursión. Para distinguir los archivos normales de los directorios, el valor de retorno es, de hecho, una “tupla” de objetos: un contenedor **List** que almacena archivos normales y otro que almacena los directorios. Los archivos están definidos aquí como **public** a propósito, porque el objeto **TreeInfo** es simplemente para recopilar los objetos; si estuviéramos simplemente devolviendo una lista, no lo definiríamos como **private**, así que el hecho de que estemos devolviendo un par de objetos no quiere decir que los tengamos que definir como **private**. Observe que **TreeInfo** implementa **Iterable<File>**, que genera los archivos, de modo que disponemos de una “iteración predeterminada” a través de la lista de archivos, mientras que para especificar directorios tenemos que escribir **".dirs"**.

El método **TreeInfo.toString()** utiliza una clase de “impresión avanzada”, para que la salida sea más fácil de visualizar. Los métodos predeterminados **toString()** de los contenedores imprimen todos los elementos de un contenedor en una misma línea. Para colecciones de gran tamaño, esto puede hacerse difícil de leer, así que podemos tratar de emplear un formato alternativo. He aquí la herramienta que añade avances de línea y sangrados a cada elemento:

```

//: net/mindview/util/PPrint.java
// Impresión avanzada de colecciones
package net.mindview.util;
import java.util.*;

public class PPrint {
    public static String pformat(Collection<?> c) {
        if(c.size() == 0) return "[]";
        StringBuilder result = new StringBuilder("[");
        for(Object elem : c) {
            if(c.size() != 1)
                result.append("\n  ");
            result.append(elem);
        }
        if(c.size() != 1)
            result.append("\n");
        result.append("]");
        return result.toString();
    }
    public static void pprint(Collection<?> c) {
        System.out.println(pformat(c));
    }
    public static void pprint(Object[] c) {
        System.out.println(pformat(Arrays.asList(c)));
    }
} //:-

```

El método **pformat()** produce un objeto **String** formateado a partir de un objeto **Collection**, y el método **pprint()** utiliza **pformat()** para llevar a cabo esa tarea. Observe que los casos especiales en los que no existe ningún elemento o sólo existe uno se gestionan de manera distinta. También hay una versión de **pprint()** para matrices.

La utilidad **Directory** está incluida en el paquete **net.mindview.util**, para que esté fácilmente disponible. He aquí un ejemplo de utilización:

```

//: io/DirectoryDemo.java
// Ejemplo de uso de las utilidades de directorio.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
    public static void main(String[] args) {
        // Todos los directorios:
        PPrint.pprint(Directory.walk(".").dirs);
        // Todos los archivos que comiencen con 'T'
        for(File file : Directory.local(".", "T.*"))
            print(file);
        print("-----");
        // Todos los archivos Java que comienzan con 'T':
        for(File file : Directory.walk(".", "T.*\\*.java"))
            print(file);
        print("=====");
        // Todos los archivos de clase que contengan "Z" o "z":
        for(File file : Directory.walk(".", ".*[Zz].*[\\*.class"]"))
            print(file);
    }
} /* Output: (ejemplo)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
-----
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
=====
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:-
```

Puede que necesite refrescar sus conocimientos sobre expresiones regulares en el Capítulo 13, *Cadenas de caracteres*, para comprender los argumentos situados en segunda posición en `local()` y `walk()`.

Podemos llevar esta idea un paso más allá y crear una herramienta que recorra directorios y procese los archivos contenidos en ellos de acuerdo con un objeto **Strategy** (se trata de otro ejemplo del patrón de diseño *Estrategia*):

```

//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
    public interface Strategy {
        void process(File file);
    }
    private Strategy strategy;
    private String ext;
    public ProcessFiles(Strategy strategy, String ext) {
        this.strategy = strategy;
        this.ext = ext;
    }
    public void start(String[] args) {
        try {
            if(args.length == 0)
```

```

        processDirectoryTree(new File("."));
    else
        for(String arg : args) {
            File fileArg = new File(arg);
            if(fileArg.isDirectory())
                processDirectoryTree(fileArg);
            else {
                // Permitir que el usuario no incluya la extensión:
                if(!arg.endsWith("." + ext))
                    arg += "." + ext;
                strategy.process(
                    new File(arg).getCanonicalFile());
            }
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
public void
processDirectoryTree(File root) throws IOException {
    for(File file : Directory.walk(
        root.getAbsolutePath(), ".*\\\\" + ext))
        strategy.process(file.getCanonicalFile());
}
// Ejemplo de utilización:
public static void main(String[] args) {
    new ProcessFiles(new ProcessFiles.Strategy() {
        public void process(File file) {
            System.out.println(file);
        }
    }, "java").start(args);
}
/* (Execute to see output) *///:-

```

La interfaz **Strategy** está anidada dentro de **ProcessFiles**, de modo que si queremos implementarla debemos implementar **ProcessFiles.Strategy**, que proporciona más información al lector. **ProcessFiles** realiza todo el trabajo de localizar los archivos que tengan una extensión concreta (el argumento **ext** del constructor), y cuando localiza un archivo que cumpla con el criterio simplemente se lo entrega al objeto **Strategy** (que también es un argumento del constructor).

Si no le damos ningún argumento, **ProcessFiles** asume que queremos recorrer todos los directorios a partir del directorio actual. También podemos especificar un archivo concreto, con o sin la extensión (el programa añadirá la extensión en caso necesario) o uno o más directorios.

En **main()** podemos ver un ejemplo básico de utilización de la herramienta; en el que se imprimen los nombres de todos los archivos fuente Java de acuerdo con la línea de comandos que proporcionemos.

Ejercicio 4: (2) Utilice **Directory.walk()** para sumar los tamaños de todos los archivos de un árbol de directorios cuyos nombres se correspondan con una expresión regular concreta.

Ejercicio 5: (1) Modifique **ProcessFiles.java** para que busque correspondencias con una expresión regular en lugar de con una extensión fija.

Búsqueda y creación de directorios

La clase **File** es algo más que una mera representación de un directorio o un archivo existente. También podemos utilizar un objeto **File** para crear un nuevo directorio o una ruta completa de directorios, si es que no existe. También podemos examinar las características de los archivos (tamaño, fecha de la última modificación, permisos de lectura/escritura), para ver si un objeto **File** representa a un archivo o a un directorio, y borrar un archivo. El ejemplo siguiente muestra algunos de los otros métodos disponibles en la clase **File** (véase la documentación del JDK disponible en <http://java.sun.com> para conocer el conjunto completo de métodos):

```

//: io/MakeDirectories.java
// Ejemplo de uso de la clase File para crear
// directorios y manipular archivos.
// (Args: MakeDirectoriesTest)
import java.io.*;

public class MakeDirectories {
    private static void usage() {
        System.err.println(
            "Usage:MakeDirectories path1 ...\\n" +
            "Creates each path\\n" +
            "Usage:MakeDirectories -d path1 ...\\n" +
            "Deletes each path\\n" +
            "Usage:MakeDirectories -r path1 path2\\n" +
            "Renames from path1 to path2");
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
            "\\n Can read: " + f.canRead() +
            "\\n Can write: " + f.canWrite() +
            "\\n getName: " + f.getName() +
            "\\n getParent: " + f.getParent() +
            "\\n getPath: " + f.getPath() +
            "\\n length: " + f.length() +
            "\\n lastModified: " + f.lastModified());
        if(f.isFile())
            System.out.println("It's a file");
        else if(f.isDirectory())
            System.out.println("It's a directory");
    }
    public static void main(String[] args) {
        if(args.length < 1) usage();
        if(args[0].equals("-r")) {
            if(args.length != 3) usage();
            File
                old = new File(args[1]),
                rname = new File(args[2]);
            old.renameTo(rname);
            fileData(old);
            fileData(rname);
            return; // Salir de main
        }
        int count = 0;
        boolean del = false;
        if(args[0].equals("-d")) {
            count++;
            del = true;
        }
        count--;
        while(++count < args.length) {
            File f = new File(args[count]);
            if(f.exists()) {
                System.out.println(f + " exists");
                if(del) {
                    System.out.println("deleting... " + f);
                    f.delete();
                }
            }
        }
    }
}

```

```

        else { // No existe
            if(!del) {
                f.mkdirs();
                System.out.println("created " + f);
            }
        }
        fileData(f);
    }
}
/* Output: (80% match)
created MakeDirectoriesTest
Absolute path: d:\aaa-TIJ4\code\io\MakeDirectoriesTest
Can read: true
Can write: true
getName: MakeDirectoriesTest
getParent: null
getPath: MakeDirectoriesTest
length: 0
lastModified: 1101690308831
It's a directory
*///:-
```

En `fileData()` podemos ver diversos métodos de consulta de archivos utilizados para mostrar información acerca del archivo o de la ruta de directorios.

El primer método utilizado por `main()` es `renameTo()`, que permite renombrar (o desplazar) un archivo a una ruta de directorios nueva, representada por el argumento, que es otro objeto `File`. Esto funciona también con directorios de cualquier longitud.

Si experimenta con el programa anterior, verá que puede construir una ruta de directorios todo lo compleja que quiera, porque `mkdirs()` se encarga de hacer el trabajo por nosotros.

Ejercicio 6: (5) Utilice `ProcessFiles` para encontrar todos los archivos de código fuente Java en un subárbol de directorios concreto que hayan sido modificados después de una fecha concreta.

Entrada y salida

Las bibliotecas de E/S de los lenguajes de programación utilizan a menudo la abstracción del *flujo de datos (stream)*, para representar cualquier origen o destino de datos como un objeto capaz de producir o recibir elementos de datos. El flujo de datos oculta los detalles de lo que ocurre con los datos dentro del dispositivo real de E/S.

Las clases de la biblioteca de Java para E/S están divididas en entrada y salida, como se puede ver en la jerarquía de clases al examinar la documentación del JDK. A través del mecanismo de herencia, todas las clases derivadas de las clases **InputStream** o **Reader** disponen de métodos básicos denominados `read()` para leer un único byte o una matriz de bytes. De la misma forma, todas las clases derivadas de las clases **OutputStream** o **Writer** disponen de métodos básicos denominados `write()` para escribir un único byte o una matriz de bytes. Sin embargo, generalmente no utilizaremos estos métodos; esos métodos existen para que otras clases puedan emplearlos, pero estas otras clases nos proporcionan una interfaz más útil. Por tanto, raramente crearemos nuestro objeto flujo de datos utilizando una única clase, sino que en su lugar apilaremos múltiples objetos para obtener la funcionalidad deseada (se trata del patrón de diseño *Decorador*, como veremos en esta sección). El hecho de que creemos más de un objeto para producir un único flujo de datos es la razón principal de que la biblioteca de E/S de Java sea tan confusa.

Resulta útil clasificar las clases según su funcionalidad. En Java 1.0, los diseñadores de bibliotecas comenzaron decidiendo que todas las clases que tuvieran algo que ver con la entrada de datos heredarian de **InputStream**, mientras que todas las clases que estuvieran asociadas con la salida heredarian de **OutputStream**.

Como suele ser habitual en este libro, trataré de proporcionar una panorámica de las clases, pero asumiendo que el lector va a emplear la documentación del JDK para conocer el resto de los detalles, como por ejemplo la lista exhaustiva de métodos de una clase concreta.

Tipos de InputStream

La tarea de **InputStream** consiste en representar clases que produzcan datos de entrada procedentes de diferentes fuentes. Estos orígenes de datos pueden ser:

1. Una matriz de bytes.
2. Un objeto **String**.
3. Un archivo.
4. Una “canalización (*pipe*)” que funciona como una canalización física: se introducen las cosas por un extremo y esas cosas salen por el otro.
5. Una secuencia de otros flujos de datos, de modo que se los pueda recopilar en un único flujo.
6. Otros orígenes de datos, como por ejemplo una conexión a Internet (este tema se cubre en *Thinking in Enterprise Java*, disponible en www.MindView.net).

Cada uno de estos orígenes tiene una subclase asociada de **InputStream**. Además, **FilterInputStream** también es un tipo de **InputStream**, para proporcionar una clase base para las clases “decoradoras” que asocian atributos o interfaces fáciles a los flujos de datos de entrada. Este tema se analiza más adelante.

Tabla E/S.1. Tipos de InputStream.

| Clase | Función | Argumentos del constructor |
|--------------------------------|---|--|
| | | Cómo utilizarlos |
| ByteArrayInputStream | Permite utilizar un <i>buffer</i> de memoria como un InputStream . | El <i>buffer</i> del cual hay que extraer los bytes. |
| | | Como origen de datos: conéctelo a un objeto FilterInputStream para proporcionar una interfaz útil. |
| StringBufferInputStream | Convierte un String en un InputStream . | Un objeto String . La implementación subyacente utiliza en realidad un objeto StringBuffer . |
| | | Como origen de datos: conéctelo a un objeto FilterInputStream para proporcionar una interfaz útil. |
| FileInputStream | Para leer información de un archivo. | Un objeto String que representa el nombre del archivo o un objeto File o FileDescriptor . |
| | | Como origen de datos: conéctelo a un objeto FilterInputStream para proporcionar una interfaz útil. |
| PipedInputStream | Genera los datos que estén siendo escritos en el objeto PipedOutputStream asociado. Implementa el concepto de “canalización” de flujos de datos. | PipedOutputStream |
| | | Como origen de datos en programación multihebra: conéctelo a un objeto FilterInputStream para proporcionar una interfaz útil. |
| SequenceInputStream | Convierte dos o más objetos InputStream en un único InputStream . | Dos objetos InputStream o un objeto Enumeration para un contenedor de objetos InputStream . |
| | | Como origen de datos: conéctelo a un objeto FilterInputStream para proporcionar una interfaz útil. |
| FilterInputStream | Clase abstracta que actúa como interfaz para las clases decoradoras que proporcionan funcionalidad útil a las otras clases InputStream . Véase la Tabla E/S.3. | Véase la Tabla E/S.3 |
| | | Véase la Tabla E/S.3 |

Tipos de OutputStream

Esta categoría incluye las clases que deciden a dónde debe ir la salida: a una matriz de bytes (pero no a un objeto `String`, aunque presumiblemente podemos crear uno usando la matriz de bytes), a un archivo o a una canalización.

Además, el objeto `FilterOutputStream` proporciona una clase base para las clases “decoradoras” que asocien atributos o interfaces útiles a los flujos de datos de salida. Este tema se analiza más adelante.

Tabla E/S.2. Tipos de OutputStream.

| Clase | Función | Argumentos del constructor |
|------------------------------------|---|--|
| | | Cómo utilizarlos |
| <code>ByteArrayOutputStream</code> | Crea un <i>buffer</i> en memoria. Todos los datos que se envíen al flujo de datos se colocan en este <i>buffer</i> . | Tamaño inicial opcional del <i>buffer</i> . |
| | | Para designar el destino de los datos: conéctelo a un objeto <code>FilterOutputStream</code> para proporcionar una interfaz útil. |
| <code>FileOutputStream</code> | Para enviar información a un archivo. | Un objeto <code>String</code> que representa el nombre del archivo o un objeto <code>File</code> o <code>FileDescriptor</code> . |
| | | Para designar el destino de los datos: conéctelo a un objeto <code>FilterOutputStream</code> para proporcionar una interfaz útil. |
| <code>PipedOutputStream</code> | Cualquier información que escribamos en este flujo de datos termina siendo automáticamente la entrada para el objeto <code>PipedInputStream</code> asociado. Implementa el concepto de “canalización” de flujos de datos. | <code>PipedInputStream</code> |
| | | Para designar el destino de los datos en programación multihebra: conéctelo a un objeto <code>FilterOutputStream</code> para proporcionar una interfaz útil. |
| <code>FilterOutputStream</code> | Clase abstracta que actúa como interfaz para las clases decoradoras que proporcionan funcionalidad útil a las otras clases <code>OutputStream</code> . Véase la Tabla E/S.4. | Véase la Tabla E/S.4. |
| | | Véase la Tabla E/S.4. |

Adición de atributos e interfaces útiles

Los decoradores ya han sido introducidos en el Capítulo 15, *Genéricos*. La biblioteca de E/S de Java requiere muchas combinaciones diferentes de características, y ésta es la justificación de utilizar el patrón de diseño Decorador.¹ La razón de la existencia de las clases “filtro” de la biblioteca de E/S de Java es que la clase abstracta de “filtro” es la clase base para todos los decoradores. Un decorador debe tener la misma interfaz que el objeto al que decore, pero el decorador también puede ampliar la interfaz, que es algo que ocurre en varias de las clases de “filtro”.

Existe, sin embargo, un problema con la estrategia Decorador. Los decoradores nos dan mucha más flexibilidad a la hora de escribir un programa, (ya que se pueden mezclar y ajustar fácilmente los atributos), pero aumentan la complejidad del código. La razón de que la biblioteca de E/S de Java sea tan complicada de utilizar es que es necesario crear muchas clases (la clase de E/S “básica” más todos los decoradores) para poder disponer de ese único objeto de E/S que deseamos.

Las clases que proporcionan la interfaz de decoración para controlar un flujo de datos `InputStream` o `OutputStream` concreto son `FilterInputStream` y `FilterOutputStream`, que no tienen nombres muy intuitivos. `FilterInputStream`

¹ No está claro que ésta haya sido una buena decisión de diseño, especialmente si la comparamos con la simplicidad de las bibliotecas de E/S en otros lenguajes. Pero es, sin ninguna duda, la justificación de esa decisión.

FilterOutputStream derivan de las clases base de la biblioteca de E/S **InputStream** y **OutputStream**, lo que es un requisito clave de los decoradores (para que puedan proporcionar la interfaz común para todos los objetos que están siendo decorados).

Lectura de un flujo InputStream con FilterInputStream

Las clases **FilterInputStream** realizan dos tareas significativamente distintas. **DataInputStream** permite leer diferentes tipos de datos primitivos, así como objetos **String** (todos los métodos empiezan con "read", como por ejemplo **readByte()**, **readFloat()**, etc). Esta clase, junto con su compañera **DataOutputStream**, permite desplazar datos primitivos de un lugar a otro a través de un flujo de datos. Esos "lugares" están determinados por las clases de la Tabla E/S.1.

Las clases **FilterInputStream** restantes modifican la forma en que se comporta internamente un flujo **InputStream**: indican si éste dispone de *buffer* o no, si llevan la cuenta de las líneas que están leyendo (lo que nos permite preguntar por los números de linea o asignar números de linea), y si se puede retroceder un único carácter. Las dos últimas clases parecen pensadas para permitir la escritura de un compilador (probablemente se añadieron para soportar el experimento de "construir un compilador de Java en Java"), por lo que probablemente no los use nunca en tareas de programación general.

La mayor parte de las veces será necesario almacenar la entrada en un *buffer*, independientemente del dispositivo de E/S con el que nos estemos conectando, así que habría tenido más sentido que la biblioteca de E/S dispusiera de un caso especial (o simplemente una llamada de método) para la entrada sin *buffer* en lugar de para la entrada con *buffer*.

Tabla E/S.3. Tipos de FilterInputStream

| Clase | Función | Argumentos del constructor |
|------------------------------|---|--|
| | | Cómo utilizarlos |
| DataInputStream | Se utiliza conjuntamente con DataOutputStream , para poder leer primitivas (int , char , long , etc.) de un flujo de datos de una manera portable. | InputStream |
| | | Contiene una interfaz completa con la que leer tipos primitivos. |
| BufferedInputStream | Utilice ésta para evitar una lectura física cada vez que desee más datos. Lo que estamos diciendo es: "Utiliza un <i>buffer</i> ". | InputStream , con un tamaño de <i>buffer</i> opcional. |
| | | No proporciona una interfaz per se. Sólo añade la capacidad de <i>buffer</i> al proceso. Asocie un objeto interfaz. |
| LineNumberInputStream | Lleva la cuenta de los números de linea en el flujo de datos de entrada; podemos invocar a getLineNumber() y setLineNumber(int) . | InputStream |
| | | Sólo añade la numeración de líneas, así que probablemente la utilicemos asociando un objeto interfaz. |
| PushbackInputStream | Dispone de un <i>buffer</i> de retroceso de un único byte para poder retroceder al último carácter leído. | InputStream |
| | | Generalmente, se utiliza en el análisis sintáctico realizado por un compilador. Probablemente no utilice nunca esta clase. |

Escritura de un flujo OutputStream con FilterOutputStream

El complemento a **DataInputStream** es **DataOutputStream**, que formatea cada uno de los tipos primitivos y objetos **String** en un flujo de datos de tal manera que cualquier objeto **DataInputStream**, en cualquier máquina, pueda leerlos. Todos los métodos comienzan con "write", como por ejemplo **writeByte()**, **writeFloat()**, etc.

La intención original de **PrintStream** era imprimir todos los tipos de datos primitivos y objetos **String** en una forma legible. Esto difiere de **DataOutputStream**, cuyo objetivo es insertar los elementos de datos en un flujo de datos de tal manera que **DataInputStream** pueda reconstruirlos de manera portable.

Los dos métodos más importantes de **PrintStream** son **print()** y **println()**, que están sobrecargados para imprimir todos los diversos tipos de datos. La diferencia entre **print()** y **println()** es que este último añade un avance de línea cuando ha acabado.

PrintStream puede ser problemático porque activa todas las excepciones de **IOException** (hay que comprobar explícitamente el estado de error con **checkError()**, que devuelve **true** si se ha producido un error). Asimismo, **PrintStream** no se internacionaliza adecuadamente y no gestiona los saltos de línea de manera independiente de la plataforma. Estos problemas están resueltos en **PrintWriter**, que se describe más adelante.

BufferedOutputStream es un modificador que le dice al flujo de datos que utilice un *buffer*, para que no se realicen escrituras físicas cada vez que escribamos en el flujo de datos. Normalmente, siempre conviene utilizar este modificador a la hora de llevar a cabo una salida de datos.

Tabla E/S.4. Tipos de **FilterOutputStream**.

| Clase | Función | Argumentos del constructor |
|-----------------------------|---|--|
| | | Cómo utilizarlos |
| DataOutputStream | Se utiliza en conjunción con DataInputStream para poder escribir primitivas (int , char , long , etc.) en un flujo de datos en una forma portable. | OutputStream Contiene una interfaz completa que permite escribir tipos primitivos. |
| PrintStream | Para generar salida formateada. Mientras que DataOutputStream se encarga del <i>almacenamiento</i> de los datos, PrintStream gestiona la <i>visualización</i> . | OutputStream , con un valor boolean opcional que indica que el <i>buffer</i> debe vaciarse con cada nueva linea. Debe ser el envoltorio “final” para el objeto OutputStream . Probablemente utilice esta clase muy a menudo. |
| BufferedOutputStream | Utilice esto para evitar que se realice una escritura física cada vez que se envíe un elemento de datos. Estamos diciendo: “Utiliza un <i>buffer</i> ”. Puede utilizar flush() para vaciar el <i>buffer</i> . | OutputStream , con un tamaño de <i>buffer</i> opcional. No proporciona una interfaz por se. Simplemente añade un <i>buffer</i> al proceso. Asócielo a un objeto interfaz. |

Lectores y escritores

Java 1.1 realizó significativas modificaciones en la biblioteca fundamental de flujos de E/S. Cuando examine las clases **Reader** (lector) y **Writer** (escritor), su primer pensamiento (al igual que me pasó a mí) puede ser que esas clases intentaban sustituir a **InputStream** y a **OutputStream**, pero en realidad no es así. Aunque algunos aspectos de la biblioteca original de la biblioteca de flujos de datos están obsoletos y ahora se desaconsejan (si los emplea, el compilador generará una advertencia), las clases **InputStream** y **OutputStream** siguen proporcionando una valiosa funcionalidad en la forma de mecanismos de E/S orientados a bytes, mientras que las clases **Reader** y **Writer** proporcionan mecanismos de E/S orientados a caracteres y compatibles con Unicode. Además:

1. Java 1.1 añadió nuevas clases a las jerarquías **InputStream** y **OutputStream**, así que resulta obvio que la intención no era sustituir esas jerarquías.
2. Existen ocasiones en las que es necesario utilizar clases de la jerarquía orientada a “bytes” *en combinación* con las clases de la jerarquía orientada a “caracteres”. Para hacer esto, existen clases “adaptadoras”: **InputStreamReader** convierte un objeto **InputStream** en un objeto **Reader**, y **OutputStreamWriter** convierte un objeto **OutputStream** en un objeto **Writer**.

La razón más importante para la existencia de las jerarquías **Reader** y **Writer** es la internalización. La antigua jerarquía de flujos de datos de E/S sólo soportaba flujos de datos con bytes de 8 bits y no gestionaba adecuadamente los caracteres Unicode de 16 bits. Dado que Unicode se utiliza para la internalización (y el tipo **char** nativo de Java es Unicode de 16-bits), las jerarquías **Reader** y **Writer** se añadieron para soportar Unicode en todas las operaciones de E/S. Además, las nuevas bibliotecas están diseñadas para que sus operaciones sean más rápidas que las de las bibliotecas antiguas.

Orígenes y destinos de los datos

Casi todas las clase originales para flujos de datos de E/S de Java disponen de clases **Reader** y **Writer** correspondientes con el fin de permitir la manipulación nativa de datos Unicode. Sin embargo, existen algunas ocasiones en las que los flujos **InputStream** y **OutputStream** orientados a bytes constituyen la solución correcta; en particular, las bibliotecas **java.util.zip** están orientadas a bytes, en lugar de a caracteres. Por tanto, el enfoque más apropiado consiste en *tratar* de utilizar las clases **Reader** y **Writer** siempre que se pueda y descubrir aquellas situaciones en las que haya que utilizar las bibliotecas orientadas a bytes; resulta fácil descubrir esas situaciones, porque los programas no podrán compilarse en caso contrario.

He aquí una tabla que muestra la correspondencia entre los orígenes y los destinos de información (es decir, de dónde vienen y a dónde van físicamente los datos) en las dos jerarquías.

| Orígenes y destinos: clase Java 1.0 | Clase Java 1.1 correspondiente |
|--|---|
| InputStream | Reader adaptador: InputStreamReader |
| OutputStream | Writer adaptador: OutputStreamWriter |
| FileInputStream | FileReader |
| FileOutputStream | FileWriter |
| StringBufferInputStream (desaconsejado) | StringReader |
| (no hay clase correspondiente) | StringWriter |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| PipedInputStream | PipedReader |
| PipedOutputStream | PipedWriter |

En general, encontrará que las interfaces para las dos jerarquías son similares, sino idénticas.

Modificación del comportamiento de los flujos de datos

Para los flujos de tipo **InputStream** y **OutputStream**, los flujos de datos se adaptaban para cada necesidad particular utilizando subclases “decoradoras” de **FilterInputStream** y **FilterOutputStream**. Las jerarquías de clases de **Reader** y **Writer** continúan utilizando esta idea, pero no exactamente.

En la siguiente tabla, la correspondencia es algo menos precisa que en la tabla anterior. La diferencia se debe a la organización de las clases; aunque **BufferedOutputStream** es una subclase de **FilterOutputStream**, **BufferedWriter** *no* es una subclase de **FilterWriter** (la cual, aunque es abstracta no tiene subclases y parece, por tanto, que se ha incluido simplemente para poder utilizarla en el futuro o para que no nos rompamos la cabeza preguntándonos dónde está). Sin embargo, las interfaces de las clases sí que se parecen bastante.

| Filtros: clase Java 1.0 | Clase Java 1.1 correspondiente |
|---------------------------------------|--|
| FilterInputStream | FilterReader |
| FilterOutputStream | FilterWriter (clase abstracta sin subclases) |
| BufferedInputStream | BufferedReader (también tiene readLine()) |
| BufferedOutputStream | BufferedWriter |
| DataInputStream | Utilice DataInputStream (excepto cuando necesite usar readLine(), en cuyo caso debe emplear BufferedReader) |
| PrintStream | PrintWriter |
| LineNumberInputStream (desaconsejado) | LineNumberReader |
| StreamTokenizer | StreamTokenizer (utilice el constructor que admite un objeto Reader) |
| PushbackInputStream | PushbackReader |

Existe una directriz bastante clara: cuando quiera utilizar `readLine()`, no debe hacerlo con un flujo **DataInputStream** (esto origina un mensaje de advertencia en tiempo de compilación donde se informa de que esa técnica está desaconsejada), sino que debe utilizar **BufferedReader**. Por lo demás, **DataInputStream** continúa siendo uno de los miembros “aconsejados” de la biblioteca de E/S.

Para facilitar la transición a soluciones que empleen **PrintWriter**, esta clase tiene constructores que admiten cualquier objeto **OutputStream** así como objetos **Writer**. La interfaz de formateo de **PrintWriter** es casi idéntica a la de **PrintStream**.

En Java SE5, se han añadido constructores a **PrintWriter** para simplificar la creación de archivos a la hora de escribir la salida, como veremos enseguida.

Un constructor **PrintWriter** también dispone de una opción para realizar un vaciado de *buffer* automático, lo que tiene lugar después de cada llamada a `println()` si se activa el correspondiente indicador en el constructor.

Clases no modificadas

Algunas clases no sufrieron modificaciones entre las versiones Java 1.0 y Java 1.1:

| Clases Java 1.0 sin clases Java 1.1 correspondientes |
|--|
| DataOutputStream |
| File |
| RandomAccessFile |
| SequenceInputStream |

DataOutputStream, en concreto, se utiliza sin modificación, por lo que para almacenar y extraer datos en un formato trans-portable, se utilizan las jerarquías **InputStream** y **OutputStream**.

RandomAccessFile

RandomAccessFile se utiliza para archivos que contengan registros de tamaño conocido, de modo que podamos desplazarnos de un registro a otro usando `seek()`, y luego leer o modificar los registros. Los registros no tienen que tener el mismo tamaño; simplemente tenemos que determinar el tamaño que tienen y el lugar del archivo donde se encuentran.

En principio, resulta bastante difícil de entender que **RandomAccessFile** no forme parte de la jerarquía **InputStream** o **OutputStream**. Sin embargo, no tiene ninguna asociación con dichas jerarquías, salvo el hecho de que implementa las interfaces **DataInput** y **DataOutput** (que también son implementadas por **DataInputStream** y **DataOutputStream**). Ni siquiera utiliza ninguna de las funcionalidades de las clases **InputStream** o **OutputStream** existentes; se trata de una clase completamente separada, que se ha escrito partiendo de cero, con sus propios métodos (en su mayor parte nativos). La razón puede ser que **RandomAccessFile** tiene un comportamiento esencialmente distinto del de los otros tipos de E/S, ya que nos podemos mover hacia adelante y hacia atrás dentro de un archivo. En cualquier caso, se trata de una clase aislada, descendiente directa de **Object**.

Esencialmente, un objeto **RandomAccessFile** funciona como un flujo **DataInputStream** que se hubiera conectado con un flujo **DataOutputStream**, junto con los métodos de **getFilePointer()** para averiguar en qué lugar del archivo nos encontramos, **seek()** para desplazarse a un nuevo punto en el archivo y **length()** para determinar el tamaño máximo del archivo. Además, los constructores requieren un segundo argumento (idéntico a **fopen()** en C) que indica si estamos simplemente leyendo de manera aleatoria ("r") o leyendo y escribiendo ("rw"). No existe soporte para archivos de sólo escritura, lo que podría sugerir que **RandomAccessFile** podría también haberse diseñado como clase heredada de **DataInputStream**.

Los métodos de búsqueda sólo están disponibles en **RandomAccessFile**, que solamente puede aplicarse a archivos. **BufferedInputStream** permite marcar con **mark()** una posición (cuyo valor se almacena en una única variable interna) y efectuar un reposicionamiento con **reset()** a dicha posición, pero esta funcionalidad es muy limitada y no resulta muy útil.

La mayor parte de la funcionalidad de **RandomAccessFile**, si es que no toda ella, ha sido sustituida en el JDK 1.4 por los *archivos mapeados en memoria nio*, que describiremos más adelante en el capítulo.

Utilización típica de los flujos de E/S

Aunque podemos combinar las clases de flujos de E/S de muchas maneras distintas, lo más probable es que en nuestros programas utilicemos sólo unas cuantas combinaciones. Los siguientes ejemplos pueden usarse como referencia básica de lo que constituye una utilización típica de los mecanismos de E/S.

En estos ejemplos, simplificaremos el tratamiento de excepciones pasando las excepciones a la consola, pero esta forma de proceder sólo resulta apropiada en utilidades y ejemplos de pequeño tamaño. En el código de los programas reales, conviene utilizar técnicas de tratamiento de errores más sofisticadas.

Archivo de entrada con *buffer*

Para abrir un archivo con entrada orientada a caracteres, utilizamos un objeto **FileInputStream** con un objeto **String** o **File** como nombre de archivo. Para aumentar la velocidad, conviene asociar con el archivo un *buffer*, para lo cual se proporciona al constructor la referencia a un objeto **BufferedReader**. Puesto que **BufferedReader** también proporciona el método **readLine()**, éste será nuestro objeto final y la interfaz a través de la cual efectuaremos las lecturas. Cuando **readLine()** devuelva **null**, habremos alcanzado el final del archivo.

```
//: io/BufferedInputStream.java
import java.io.*;

public class BufferedInputStream {
    // Pasar excepciones a la consola:
    public static String
    read(String filename) throws IOException {
        // Leer la entrada línea a línea:
        BufferedReader in = new BufferedReader(
            new FileReader(filename));
        String s;
        StringBuilder sb = new StringBuilder();
        while((s = in.readLine())!= null)
            sb.append(s + "\n");
        in.close();
        return sb.toString();
    }
}
```

```

    }
    public static void main(String[] args)
    throws IOException {
        System.out.print(read("BufferedInputStream.java"));
    }
} /* (Ejecutar para ver la salida) *///:-

```

El objeto **StringBuilder** **sb** se utiliza para acumular el contenido completo del archivo (incluyendo los avances de línea que haya que añadir, ya que **readLine()** los elimina). Finalmente, se invoca **close()** para cerrar el archivo.²

- Ejercicio 7:** (2) Abra un archivo de texto para poder leer el contenido de línea en línea. Lea cada línea en forma de una cadena de caracteres y sitúe dicho objeto **String** dentro de un contenedor **LinkedList**. Imprima todas las líneas de **LinkedList** en orden inverso.
- Ejercicio 8:** (1) Modifique el Ejercicio 7 para proporcionar como argumento de la línea de comandos el nombre del archivo que haya que leer.
- Ejercicio 9:** (1) Modifique el Ejercicio 8 para pasar a mayúsculas todas las líneas del contenedor **LinkedList** y envíe los resultados a **System.out**.
- Ejercicio 10:** (2) Modifique el Ejercicio 8 para admitir argumentos adicionales en la linea de comandos que especifiquen palabras que haya que encontrar en el archivo. Imprima todas las líneas que contengan alguna de las palabras.
- Ejercicio 11:** (2) En el ejemplo **innerclasses/GreenhouseController.java**, **GreenhouseController** contiene un conjunto precodificado de sucesos. Modifique el programa para que lea dos sucesos y sus instantes relativos de un archivo de texto (nivel de dificultad 8); utilice un patrón de diseño basado en el *Método de factoría* para construir los sucesos; consulte *Thinking in Patterns (with Java)* en www.MindView.net.

Entrada desde memoria

Aquí, el objeto **String** resultante de **BufferedInputStream.read()** se utiliza para crear un objeto **StringReader**. A continuación, se utiliza **read()** para leer de carácter en carácter y enviar los datos a la consola:

```

//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
    public static void main(String[] args)
    throws IOException {
        StringReader in = new StringReader(
            BufferedInputStream.read("MemoryInput.java"));
        int c;
        while((c = in.read()) != -1)
            System.out.print((char)c);
    }
} /* (Ejecutar para ver la salida) *///:-

```

Observe que **read()** devuelve el siguiente carácter como un valor **int** y debe, por tanto, proyectarse el resultado sobre un valor **char** para imprimirllo adecuadamente.

Entrada de memoria formateada

Para leer datos “formateados”, utilizamos un flujo **DataInputStream**, que es una clase de E/S orientada a bytes (en lugar de a caracteres). Por tanto, debemos utilizar todas las clases **InputStream** en lugar de clases **Reader**. Por supuesto, pode-

² En el diseño original, se suponía que **close()** debía ser invocado cuando se ejecutara **finalize()**, y tendrá ocasión de ver en algunos textos que **finalize()** se define de esta forma para las clases de E/S. Sin embargo, como hemos comentado anteriormente, la funcionalidad **finalize()** no ha llegado a funcionar de la forma en que los diseñadores de Java habían previsto originalmente (en otras palabras, no va a llegar a funcionar nunca), por lo que la técnica segura consiste en invocar **close()** explícitamente para los archivos.

mos leer cualquier cosa (por ejemplo un archivo) de byte en byte utilizando clases **InputStream**, pero lo que aquí se utiliza es una cadena de caracteres:

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
    public static void main(String[] args)
        throws IOException {
        try {
            DataInputStream in = new DataInputStream(
                new ByteArrayInputStream(
                    BufferedInputStream.read(
                        "FormattedMemoryInput.java").getBytes()));
            while(true)
                System.out.print((char)in.readByte());
        } catch(EOFException e) {
            System.err.println("End of stream");
        }
    }
} /* (Execute to see output) */:-
```

A un flujo **ByteArrayInputStream** hay que proporcionarle una matriz de bytes. Para generarla, **String** dispone de un método **getBytes()**. El flujo **ByteArrayInputStream** resultante es un objeto **InputStream** apropiado para entregárselo a un flujo **DataInputStream**.

Si leemos los caracteres de un flujo **DataInputStream** de byte en byte usando **readByte()**, todo valor de tipo **byte** es un resultado legítimo, por lo que el valor de retorno no puede utilizarse para detectar el final de la entrada. En lugar de ello, puede emplearse el método **available()** para determinar cuántos caracteres más hay disponibles. He aquí un ejemplo que muestra cómo leer un archivo de byte en byte:

```
//: io/TestEOF.java
// Comprobación del final del archivo mientras se lee de byte en byte.
import java.io.*;

public class TestEOF {
    public static void main(String[] args)
        throws IOException {
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("TestEOF.java")));
        while(in.available() != 0)
            System.out.print((char)in.readByte());
    }
} /* (Ejecutar para ver la salida) */:-
```

Observe que **available()** funciona de manera distinta dependiendo del tipo de medio del que estemos leyendo; literalmente, ese método nos da “el número de bytes que pueden leerse *sin que se produzca un bloqueo*”. Con un archivo, esto significa todo el archivo, pero con otro flujo de datos distinto podríamos obtener alguna otra cosa, así que utilice el método con cuidado.

También podemos detectar el final de la entrada en casos como éste tratando de capturar una excepción. Sin embargo, el uso de excepciones para control de flujo se considera una técnica poco recomendable.

Salida básica a archivo

Un objeto **FileWriter** escribe datos en un archivo. Prácticamente en todas las ocasiones nos convendrá añadir un *buffer* a la salida, envolviendo el objeto en otro objeto **BufferedWriter** (trate de eliminar este objeto envoltorio para ver el impacto sobre el rendimiento: la utilización de *buffers* ayuda a incrementar enormemente el rendimiento de las operaciones de E/S). En este ejemplo, se utiliza **PrintWriter** como decorador para que se encargue de las tareas de formateo. El archivo de datos creado de esta forma se puede leer como un archivo de texto normal.

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
    static String file = "BasicFileOutput.out";
    public static void main(String[] args)
        throws IOException {
        BufferedReader in = new BufferedReader(
            new StringReader(
                BufferedInputStream.read("BasicFileOutput.java")));
        PrintWriter out = new PrintWriter(
            new BufferedWriter(new FileWriter(file)));
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + ":" + s);
        out.close();
        // Mostrar el archivo almacenado:
        System.out.println(BufferedInputStream.read(file));
    }
} /* (Ejecutar para ver las salida) */:-
```

A medida que se escriben líneas en el archivo, se añaden los números de línea. Observe que *no se utiliza LineNumberReader*, porque es una clase muy simple y no resulta necesaria. Como podemos ver en este ejemplo, resulta trivial llevar la cuenta de nuestros propios números de línea.

Una vez que se han terminado los datos del flujo de entrada, `readLine()` devuelve `null`. El ejemplo muestra una llamada explícita a `close()` para `out`, porque si no invocamos a `close()` para todos los archivos de salida, podríamos encontrarnos con que los *buffers* no se vaciarán, con lo que el archivo estaría incompleto.

Un atajo para realizar la salida correspondiente a un archivo de texto

Java SE5 ha añadido un constructor a `PrintWriter` para que no tengamos que encargarnos de realizar a mano todas las tareas de decoración cada vez que queramos crear un archivo texto y escribir en él. El ejemplo siguiente muestra el archivo `BasicFileOutput.java` reescrito con esta nueva técnica:

```
//: io/FileOutputShortcut.java
import java.io.*;

public class FileOutputShortcut {
    static String file = "FileOutputShortcut.out";
    public static void main(String[] args)
        throws IOException {
        BufferedReader in = new BufferedReader(
            new StringReader(
                BufferedInputStream.read("FileOutputShortcut.java")));
        // He aquí la técnica más simple:
        PrintWriter out = new PrintWriter(file);
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + ":" + s);
        out.close();
        // Mostrar el archivo almacenado:
        System.out.println(BufferedInputStream.read(file));
    }
} /* (Ejecutar para ver la salida) */:-
```

Seguimos disponiendo de un *buffer*, pero no tenemos que encargarnos nosotros del mecanismo del mismo. Lamentablemente, no existen atajos similares para otras tareas muy comunes, por lo que un programa típico de E/S sigue

requiriendo una gran cantidad de texto redundante. Sin embargo, la utilidad **TextFile** que se usa en este libro, que definiremos más adelante en este capítulo, permite simplificar estas tareas comunes.

- Ejercicio 12:** (3) Modifique el Ejercicio 8 para abrir también un archivo de texto de modo que podamos escribir texto en él. Escriba en el archivo las líneas del contenedor **LinkedList**, junto con sus correspondientes números de línea (no trate de utilizar las clases “**LineNumber**”).
- Ejercicio 13:** (3) Modifique **BasicFileOutput.java** para que utilice **LineNumberReader** con el fin de controlar el número de líneas. Observe que resulta mucho más sencillo llevar la cuenta mediante programa.
- Ejercicio 14:** (2) Comenzando con **BasicFileOutput.java**, escriba un programa que compare la velocidad de escritura en un archivo al utilizar mecanismos de E/S con *buffer* y sin *buffer*.

Almacenamiento y recuperación de datos

Un objeto **PrintWriter** formatea los datos para que resulten legibles. Sin embargo, para sacar los datos de manera que puedan ser recuperados por otro flujo de datos, se utiliza **DataOutputStream** para escribir los datos y **DataInputStream** para recuperarlos. Por supuesto, estos flujos de datos pueden ser cualquier cosa, pero en el siguiente ejemplo se utiliza un archivo, usándose *buffers* tanto para lectura como para escritura. **DataOutputStream** y **DataInputStream** están orientados a bytes y requieren por tanto, flujos de datos **InputStream** y **OutputStream**:

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
    public static void main(String[] args)
        throws IOException {
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
        out.writeDouble(3.14159);
        out.writeUTF("That was pi");
        out.writeDouble(1.41413);
        out.writeUTF("Square root of 2");
        out.close();
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
        System.out.println(in.readDouble());
        // Sólo readUTF() permite recuperar la cadena
        // de caracteres Java-UTF apropiadamente:
        System.out.println(in.readUTF());
        System.out.println(in.readDouble());
        System.out.println(in.readUTF());
    }
} /* Output:
3.14159
That was pi
1.41413
Square root of 2
*///:-
```

Si utilizamos **DataOutputStream** para escribir los datos, Java garantiza que podemos recuperar perfectamente los datos con **DataInputStream**, independientemente de las plataformas que se empleen para leer y escribir los datos. Esto resulta enormemente útil, como puede atestiguar cualquiera que haya dedicado algo de tiempo a resolver problemas relacionados con el tratamiento específico de los datos en cada plataforma. Dichos problemas desaparecen si disponemos de Java en ambas plataformas.³

³ XML es otra forma de resolver el problema de transmitir datos de una plataforma a otra, y no depende de si se dispone de Java en todas las plataformas. Hablaremos de XML más adelante en este capítulo.

Cuando estamos usando **DataOutputStream**, la única forma fiable de escribir una cadena de caracteres para que pueda recuperarse mediante un flujo **DataInputStream** consiste en utilizar codificación UTF-8, la cual se consigue en este ejemplo mediante **writeUTF()** y **readUTF()**. UTF-8 es un formato multibyte, y la longitud de codificación varía de acuerdo con el conjunto de caracteres que se esté empleando. Si estamos trabajando con ASCII o caracteres que sean ASCII en su mayor parte (los cuales sólo ocupan siete bits), Unicode representa un tremendo desperdicio de espacio y/o espacio de banda, por lo que se emplea UTF-8 para codificar los caracteres ASCII en un único byte, y los caracteres no-ASCII en dos o tres bytes. Además, la longitud de la cadena de caracteres se almacena en los dos primeros bytes de la cadena UTF-8. Sin embargo, **writeUTF()** y **readUTF()** utilizan una variante de UTF-8 especial para Java (que está completamente descrita en la documentación del JDK correspondiente a estos métodos), por lo que si leemos una cadena escrita con **writeUTF()** utilizando un programa no-Java, deberemos escribir un código especial para poder leer esa cadena apropiadamente.

Con **writeUTF()** y **readUTF()**, podemos mezclar cadenas de caracteres con otros tipos de datos empleando un flujo de datos **DataOutputStream**, en la seguridad de que las cadenas de caracteres serán apropiadamente almacenadas como datos Unicode y podrán recuperarse fácilmente mediante **DataInputStream**.

El método **writeDouble()** almacena el número **double** en el flujo de datos, y el método complementario **readDouble()** permite recuperarlo (existen métodos similares para poder leer y escribir los otros tipos de datos). Pero para que cualquiera de los métodos de lectura funcionen correctamente, es necesario conocer la posición exacta de los elementos de datos dentro del flujo de datos, ya que sería igualmente posible el valor **double** almacenado como una simple secuencia de bytes, o como un valor **char**, etc. Por tanto, tenemos que tener un formato fijo para los datos en el archivo o, alternativamente, deberemos almacenar en el archivo información adicional que será necesario analizar para determinar dónde están ubicados los datos. Observe que otras técnicas, como la de serialización de los objetos o XML (describiremos ambas más adelante en el capítulo), pueden ser más sencillas a la hora de almacenar y recuperar estructuras de datos más complejas.

Ejercicio 15: (4) Consulte **DataOutputStream** y **DataInputStream** en la documentación del JDK. Comenzando con **StoringAndRecoveringData.java**, cree un programa que almacene y luego extraiga todos los diferentes tipos posibles proporcionados por las clases **DataOutputStream** y **DataInputStream**. Verifique que los valores se almacenan y extraen adecuadamente.

Lectura y escritura de archivos de acceso aleatorio

Utilizar **RandomAccessFile** es como emplear sendos flujos **DataInputStream** y **DataOutputStream** compilados (porque implementa las mismas interfaces: **DataInput** y **DataOutput**). Además, podemos utilizar **seek()** para desplazarnos por el archivo y cambiar los valores.

Para poder usar **RandomAccessFile**, debemos conocer la disposición del archivo para poder manipularlo adecuadamente. **RandomAccessFile** tiene métodos específicos para leer y escribir primitivas y cadenas de caracteres UTF-8. He aquí un ejemplo:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
    static String file = "rtest.dat";
    static void display() throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, "r");
        for(int i = 0; i < 7; i++)
            System.out.println(
                "Value " + i + ": " + rf.readDouble());
        System.out.println(rf.readUTF());
        rf.close();
    }
    public static void main(String[] args)
        throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, "rw");
        for(int i = 0; i < 7; i++)
            rf.writeDouble(i*1.414);
        rf.writeUTF("The end of the file");
    }
}
```

```

rf.close();
display();
rf = new RandomAccessFile(file, "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();
display();
}
} /* Output:
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 47.0001
Value 6: 8.484
The end of the file
*///:-
```

El método **display()** abre un archivo y muestra siete elementos contenidos en él en forma de valores **double**. En **main()**, se crea el archivo y luego se abre y modifica. Puesto que, un valor **double** siempre tiene ocho bytes de longitud, para desplazarlos con **seek()** al número situado en la posición cinco basta con multiplicar **5*8** con el fin de obtener el valor de búsqueda.

Como hemos indicado anteriormente, **RandomAccessFile** está aislado, de hecho, del resto de la jerarquía de E/S, salvo por la circunstancia de que implementa las interfaces **DataInput** y **DataOutput**. Esta clase no soporta el mecanismo de decoración, así que no se la puede combinar con ninguno de los aspectos de las subclases **InputStream** y **OutputStream**. Tenemos que asumir, por tanto, que **RandomAccessFile** dispondrá de los mecanismos de *buffer* apropiados, ya que no tenemos manera de especificar que se emplee.

La única opción de la que disponemos se encuentra en el segundo argumento del constructor: podemos abrir un archivo **RandomAccessFile** para lectura ("r") o lectura y escritura ("rw").

También merece la pena considerar la utilización de archivos mapeados en memoria **nio** en lugar de **RandomAccessFile**.

Ejercicio 16: (2) Consulte **RandomAccessFile** en la documentación del JDK. Tomando como punto de partida **UsingRandomAccessFile.java**, cree un programa que almacene y luego extraiga todos los diferentes tipos posibles soportados por la clase **RandomAccessFile**. Verifique que los valores se almacenan y se extraen adecuadamente.

Flujos de datos canalizados

En este capítulo sólo hemos mencionado brevemente las clases **PipedOutputStream**, **PipedReader** y **PipedWriter**. Esto no quiere decir que dichas clases no sean útiles, pero la ventaja que proporcionan no se comprende adecuadamente hasta que no se comienza a analizar el tema de la concurrencia, ya que los flujos de datos canalizados se emplean para la comunicación entre tareas. Este tema se tratará junto con un ejemplo en el Capítulo 21, *Concurrencia*.

Utilidades de lectura y escritura de archivos

Una tarea de programación muy común consiste en leer un archivo de memoria, modificarlo y luego volverlo a escribir. Uno de los problemas con la biblioteca de E/S de Java es que nos obliga a escribir bastante código para poder realizar estas opéraciones.

raciones comunes: no existen funciones básicas de utilidad que se encarguen de realizar el trabajo por nosotros. Todavía peor: los decoradores hacen que resulte relativamente difícil acordarse de qué hay que hacer para abrir archivos. Por tanto, resulta bastante conveniente añadir clases de utilidad a nuestra biblioteca que se encarguen de realizar estas tareas por nosotros. Java SE5 ha añadido un constructor muy útil a **PrintWriter** para poder abrir fácilmente un archivo de texto con el fin de escribir en él. Sin embargo, existen muchas otras tareas comunes que tendremos que realizar una y otra vez y conviene tratar de eliminar el código redundante asociado con dichas tareas.

He aquí la clase **TextFile** que hemos utilizado en ejemplos anteriores de este libro para simplificar la lectura y escritura en archivos. Contiene métodos estáticos para leer y escribir archivos de texto en una única cadena de caracteres y también podemos crear un objeto **TextFile** que almacene las líneas del archivo en un contenedor **ArrayList** (con lo que tendremos toda la funcionalidad de **ArrayList** a la hora de manipular el contenido del archivo):

```
//: net/mindview/util/TextFile.java
// Funciones estáticas para leer y escribir archivos de texto en forma de
// una única cadena de caracteres y para tratar el archivo como un
// contenedor ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
    // Leer un archivo como una única cadena de caracteres:
    public static String read(String fileName) {
        StringBuilder sb = new StringBuilder();
        try {
            BufferedReader in= new BufferedReader(new FileReader(
                new File(fileName).getAbsoluteFile()));
            try {
                String s;
                while((s = in.readLine()) != null) {
                    sb.append(s);
                    sb.append("\n");
                }
            } finally {
                in.close();
            }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return sb.toString();
    }
    // Escribir un archivo en una llamada a método:
    public static void write(String fileName, String text) {
        try {
            PrintWriter out = new PrintWriter(
                new File(fileName).getAbsoluteFile());
            try {
                out.print(text);
            } finally {
                out.close();
            }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    // Leer un archivo, dividir según cualquier expresión regular:
    public TextFile(String fileName, String splitter) {
        super(Arrays.asList(read(fileName).split(splitter)));
        // El método split() con expresiones regulares suele dejar una
```

```

// cadena de caracteres vacía en la primera posición:
if(get(0).equals("")) remove(0);
}
// Leer normalmente línea a línea:
public TextFile(String fileName) {
    this(fileName, "\n");
}
public void write(String fileName) {
    try {
        PrintWriter out = new PrintWriter(
            new File(fileName).getAbsoluteFile());
        try {
            for(String item : this)
                out.println(item);
        } finally {
            out.close();
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
// Prueba simple:
public static void main(String[] args) {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
    // Descomponer en una lista ordenada de palabras distintas:
    TreeSet<String> words = new TreeSet<String>(
        new TextFile("TextFile.java", "\W+"));
    // Mostrar las palabras en mayúsculas:
    System.out.println(words.headSet("a"));
}
} /* Output:
[0, ArrayList, Arrays, Break, BufferedReader,
BufferedWriter, Clean, Display, File, FileReader,
FileWriter, IOException, Normally, Output, PrintWriter,
Read, Regular, RuntimeException, Simple, Static, String,
StringBuilder, System, TextFile, Tools, TreeSet, W, Write]
*///:-
```

read() añade cada línea a un objeto **StringBuilder**, seguida de un avance de línea, ya que los caracteres de avance de línea se eliminan durante la lectura. A continuación, devuelve un objeto **String** que contiene el archivo completo. **write()** abre el archivo y escribe en él el texto contenido en **String**.

Observe que todos los elementos de código en los que se abre un archivo protegen la llamada a **close()** dentro de una cláusula **finally** para garantizar que el archivo se cierre correctamente.

El constructor utiliza el método **read()** para transformar el archivo en un objeto **String**, y luego emplea **String.split()** para dividir el resultado en líneas, según estén dispuestos los avances de línea (si utiliza esta clase muy a menudo, trate de reescribir este constructor para mejorar el rendimiento). Como no existe ningún método para combinar las líneas es necesario utilizar el método **write()** no estático para escribir las líneas de forma manual.

Puesto que esta clase pretende simplificar el proceso de lectura y escritura de archivos, todas las excepciones **IOException** se convierten en excepciones **RuntimeException**, para que el usuario no tenga que emplear bloques **try-catch**. Sin embargo, en los programas reales puede que sea necesario crear otra versión que pase las excepciones **IOException** al llamante.

En **main()**, se realiza una prueba sencilla para verificar que el método funciona.

Aunque esta utilidad no requiere de una cantidad excesiva de código, sí que permite ahorrar una gran cantidad de tiempo de programación, como veremos posteriormente en algunos de los ejemplos de este capítulo.

Otra forma de resolver el problema de leer archivos de texto consiste en utilizar la clase `java.util.Scanner` introducida en Java SE5. Sin embargo, esa clase sólo sirve para leer archivos, no para escribirlos, y dicha herramienta (que *no se encuentra en java.io*) está diseñada principalmente para crear analizadores de lenguajes de programación o “pequeños lenguajes”.

Ejercicio 17: (4) Utilizando `TextFile` y un contenedor `Map<Character, Integer>`, cree un programa que cuente el número de apariciones de los diferentes caracteres dentro de un archivo (en otras palabras, si la letra ‘a’ aparece 12 veces en el archivo, el valor `Integer` asociado con el valor `Character` que contenga ‘a’ en el mapa será ‘12’).

Ejercicio 18: (!) Modifique `TextFile.java` para que pase las excepciones `IOException` al llamante.

Lectura de archivos binarios

Esta utilidad es similar a `TextFile.java`, en el sentido de que permite simplificar el proceso de lectura de archivos binarios:

```
//: net/mindview/util/BinaryFile.java
// Utilidad para leer archivos en forma binaria.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
    public static byte[] read(File bFile) throws IOException{
        BufferedInputStream bf = new BufferedInputStream(
            new FileInputStream(bFile));
        try {
            byte[] data = new byte[bf.available()];
            bf.read(data);
            return data;
        } finally {
            bf.close();
        }
    }
    public static byte[]
    read(String bFile) throws IOException {
        return read(new File(bFile).getAbsoluteFile());
    }
} //:-
```

Un método sobrecargado admite un argumento `File`; el segundo admite un argumento `String`, que representa el nombre del archivo. Ambos devuelven la matriz de tipo `byte` resultante.

Se utiliza el método `available()` para obtener el tamaño apropiado de la matriz y esta versión concreta del método `read()` sobrecargado se encarga de llenar la matriz.

Ejercicio 19: (2) Utilizando `BinaryFile` y un contenedor `Map<Byte, Integer>`, cree un programa que cuente el número de apariciones de los diferentes bytes dentro de un archivo.

Ejercicio 20: (4) Utilizando `Directory.walk()` y `BinaryFile`, verifique que todos los archivos `.class` en un árbol de directorios comienzan con los caracteres hexadecimales ‘CAFEBAE’.

E/S estándar

El término *E/S estándar* hace referencia al concepto Unix de un único flujo de información que es utilizado por un programa (esta idea se reproduce en cierta manera en Windows y muchos otros sistemas operativos). Toda la entrada de un programa puede provenir de la *entrada estándar*, toda su salida puede ir a la *salida estándar* y todos sus mensajes de error pueden enviarse a la *salida de error estándar*. El valor de la E/S estándar es que los programas se pueden encadenar fácilmente entre sí, y la salida de un programa puede ser la entrada estándar de otro programa. Se trata de una herramienta de gran potencia.

Lectura de la entrada estándar

Siguiendo el modelo de la E/S estándar, Java dispone de `System.in`, `System.out` y `System.err`. A lo largo de este libro hemos visto cómo escribir en la salida estándar utilizando `System.out`, que está ya pre-envuelta en un objeto `PrintStream`. `System.err` es también un objeto `PrintStream`, pero `System.in` es un objeto `InputStream` simple sin ningún tipo de envoltorio. Esto significa que aunque podemos utilizar `System.out` y `System.err` de manera directa, es necesario envolver `System.in` antes de leer el mismo.

Normalmente, leceremos la entrada de linea en linea empleando `readLine()`. Para hacer esto, envuelva `System.in` en un objeto `BufferedReader`, lo que requiere que convierta `System.in` en un objeto `Reader` mediante `InputStreamReader`. He aquí un ejemplo que se limita a devolver como un eco cada linea que escribamos:

```
//: io/Echo.java
// Cómo leer de la entrada estándar.
// {RunByHand}
import java.io.*;

public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = stdin.readLine()) != null && s.length() != 0)
            System.out.println(s);
        // Una linea vacía o Ctrl-Z hace que se termine el programa
    }
} /*:-
```

La razón de la especificación de excepciones es que `readLine()` puede generar una excepción `IOException`. Observe que `System.in` debería normalmente emplearse con un *buffer*, al igual que la mayoría de los flujos de datos.

Ejercicio 21: (1) Escriba un programa que tome datos de la entrada estándar y pase a mayúsculas todos los caracteres, y que luego inserte los resultados en la salida estándar. Redirija los contenidos de un archivo hacia este programa (el proceso de redirección variará dependiendo de su sistema operativo).

Cambio de `System.out` a un objeto `PrintWriter`

`System.out` es un objeto `PrintStream`, que a su vez es de tipo `OutputStream`. `PrintWriter` tienen un constructor que toma un objeto `OutputStream` como argumento. Por tanto, si queremos convertir `System.out` en un objeto `PrintWriter` utilizando dicho constructor:

```
//: io/ChangeSystemOut.java
// Transformación de System.out en un objeto PrintWriter.
import java.io.*;

public class ChangeSystemOut {
    public static void main(String[] args) {
        PrintWriter out = new PrintWriter(System.out, true);
        out.println("Hello, world");
    }
} /* Output:
Hello, world
*///:-
```

Es importante utilizar la versión de dos argumentos del constructor de `PrintWriter` y asignar al segundo argumento el valor `true`, para permitir el vaciado automático de *buffer*; en caso contrario, podríamos no llegar a ver la salida.

Redireccionamiento de la E/S estándar

La clase `System` de Java permite redirigir los flujos de E/S estándar de entrada, de salida y de error utilizando simples llamadas a los métodos estáticos:

```
setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)
```

La redirección de la salida resulta especialmente útil si comenzamos de repente a generar una gran cantidad de salida en la pantalla y ésta empieza a desplazarse más rápido de lo que la podamos leer.⁴ El redireccionamiento de la entrada resulta muy útil para un programa de línea de comandos en el que queramos probar repetidamente una secuencia concreta de entrada de datos de usuario. He aquí un ejemplo simple que muestra el uso de estos métodos:

```
//: io/Redirecting.java
// Ilustra la redirección de la E/S estàndar.
import java.io.*;

public class Redirecting {
    public static void main(String[] args)
        throws IOException {
        PrintStream console = System.out;
        BufferedInputStream in = new BufferedInputStream(
            new FileInputStream("Redirecting.java"));
        PrintStream out = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
            System.out.println(s);
        out.close(); // Remember this!
        System.setOut(console);
    }
} ///:-
```

Este programa asocia la entrada estándar con un archivo y redirige la salida estándar y la salida de error estándar a otro archivo. Observe que almacena al principio del programa una referencia al objeto `System.out` original y que restaura la salida del sistema hacia dicho objeto al final del programa.

El redireccionamiento de E/S manipula flujos de bytes, no flujos de caracteres; es por eso que se utilizan objetos `InputStream` y `OutputStream` en lugar de `Reader` y `Writer`.

Control de procesos

A menudo es necesario ejecutar otros programas del sistema operativo desde dentro de Java y controlar la entrada y la salida de tales programas. La biblioteca Java proporciona clases para realizar tales operaciones.

Una tarea común consiste en ejecutar un programa y enviar la salida resultante a la consola. En esta sección vamos a presentar una utilidad que permite simplificar dicha tarea.

Con esta utilidad pueden producirse dos tipos de errores: los errores normales que generan excepciones (para los que nos limitaremos a regenerar una excepción de tiempo de ejecución) y los errores debidos a la ejecución del propio proceso. Informaremos de dichos errores mediante una excepción diferente:

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;
```

⁴ En el Capítulo 22, *Interfaces gráficas de usuario*, se muestra una solución todavía más cómoda para este problema: un programa GUI con un área de texto desplazable.

```
public class OSExecuteException extends RuntimeException {
    public OSExecuteException(String why) { super(why); }
} //:-.
```

Para ejecutar un programa, hay que pasar a **OSExecute.command()** una cadena de caracteres **command**, que es el mismo comando que escribiríamos para ejecutar el programa en la consola. Este comando se pasa al constructor **java.lang.ProcessBuilder** (que requiere que se le suministre en forma de una secuencia de objetos **String**), después de lo cual se inicia un objeto **ProcessBuilder**:

```
//: net/mindview/util/OSExecute.java
// Ejecutar un comando del sistema operativo
// y enviar la salida a la consola.
package net.mindview.util;
import java.io.*;

public class OSExecute {
    public static void command(String command) {
        boolean err = false;
        try {
            Process process =
                new ProcessBuilder(command.split(" ")).start();
            BufferedReader results = new BufferedReader(
                new InputStreamReader(process.getInputStream()));
            String s;
            while((s = results.readLine())!= null)
                System.out.println(s);
            BufferedReader errors = new BufferedReader(
                new InputStreamReader(process.getErrorStream()));
            // Informar de los errores y devolver un valor distinto de
            // cero al proceso llamante si existen problemas:
            while((s = errors.readLine())!= null) {
                System.err.println(s);
                err = true;
            }
        } catch(Exception e) {
            // Corrección para Windows 2000, que genera una
            // excepción para la línea de comandos predeterminada:
            if(!command.startsWith("CMD /C"))
                command("CMD /C " + command);
            else
                throw new RuntimeException(e);
        }
        if(err)
            throw new OSExecuteException("Errors executing " +
                command);
    }
} //:-.
```

Para capturar el flujo de salida estándar del programa a medida que éste se ejecuta, hay que invocar **getInputStream()**. La razón es que un objeto **InputStream** es algo de lo cual podemos leer.

Los resultados del programa llegan línea a línea, así que los leemos mediante **readLine()**. Aquí nos limitamos a imprimir las líneas, pero también podríamos capturarlas y devolverlas desde **command()**.

Los errores de programa se envían al flujo estándar de error y se capturan invocando **getErrorStream()**. Si existen errores, se imprimen y se genera una excepción **OSExecuteException**, de modo que el programa llamante pueda gestionar el problema.

He aquí un ejemplo que muestra cómo utilizar **OSExecute**:

```
//: io/OSExecuteDemo.java
// Ilustra el redireccionamiento de la E/S estándar.
```

```

import net.mindview.util.*;

public class OSExecuteDemo {
    public static void main(String[] args) {
        OSExecute.command("javap OSExecuteDemo");
    }
} /* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo extends java.lang.Object{
    public OSExecuteDemo();
    public static void main(java.lang.String[]);
}
*///:-
```

Este ejemplo utiliza el descompilador **javap** (incluido en el JDK) para descompilar el programa.

Ejercicio 22: (5) Modifique **OSExecute.java** para que, en lugar de imprimir el flujo estándar de salida, devuelva los resultados de la ejecución del programa como una lista de cadenas de caracteres. Ilustre con un ejemplo el empleo de la nueva versión de esta utilidad.

Los paquetes new

La “nueva” biblioteca de E/S de Java introducida en los paquetes **java.nio.*** en el JDK 1.4 tiene como principal objetivo aumentar la velocidad. De hecho, los “antiguos” paquetes de E/S se han reimplementado utilizando **nio** para poder aprovechar este incremento de la velocidad, así que nos podremos beneficiar de esa mayor velocidad incluso aunque no escribamos código con **nio**. El incremento de velocidad se hace patente tanto en la E/S de archivos, que es la que vamos a explorar aquí, como en la E/S de red, de la que se trata en *Thinking in Enterprise Java*.

La mayor velocidad se obtiene utilizando estructuras que están más próximas a la forma en que se realiza la E/S en el sistema operativo: *canales* y *buffers*. Podríamos utilizar el símil de una mina de carbón: el canal sería la mina que contiene la veta del carbón (los datos) y el *buffer* sería la vagoneta que introducimos en la mina. La vagoneta sale de la mina llena de carbón y nosotros extraemos el carbón de la vagoneta. En otras palabras, nosotros no interactuamos directamente con el canal sino que lo hacemos con el *buffer*, enviando el *buffer* al canal. El canal extrae datos del *buffer* o pone datos en el *buffer*.

El único tipo de *buffer* que se comunica directamente con el canal es **ByteBuffer**, un *buffer* que almacena bytes sin ningún tipo de formato. Si examinamos la documentación del JDK para **java.nio.ByteBuffer**, podremos ver que se trata de una clase muy básica: creamos uno de estos *buffers* diciéndole cuánto espacio de almacenamiento hay que asignar y existen métodos para insertar y extraer datos, bien como bytes sin formato o como tipos de datos primitivos. Pero no existe manera de insertar o de extraer un objeto, ni siquiera de tipo **String**. Es una clase de nivel bastante bajo, precisamente porque esto hace que se pueda implementar de una forma más eficiente la mayoría de los sistemas operativos.

Tres de las clases del “antiguo” esquema de E/S han sido modificadas para poder generar un objeto **FileChannel**: **FileInputStream**, **FileOutputStream** y, tanto en lectura como en escritura, **RandomAccessFile**. Observe que se trata de los flujos de datos para manipulación de bytes, en consonancia con la naturaleza de bajo nivel de **nio**. Las clases **Reader** y **Writer** en modo carácter no generan canales, aunque la clase **java.nio.channels.Channels** dispone de métodos de utilidad para generar objetos **Reader** y **Writer** a partir de canales.

He aquí un ejemplo simple donde se prueban los tres tipos de flujo de datos con el fin de generar canales de escritura, de lectura/escritura y de lectura:

```

//: io/GetChannel.java
// Obtención de canales a partir de flujos de datos
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
    private static final int BSIZE = 1024;
```

```

public static void main(String[] args) throws Exception {
    // Escribir un archivo:
    FileChannel fc =
        new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text ".getBytes()));
    fc.close();
    // Añadir al final del archivo:
    fc =
        new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Desplazarse al final
    fc.write(ByteBuffer.wrap("Some more".getBytes()));
    fc.close();
    // Leer el archivo:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    while(buff.hasRemaining())
        System.out.print((char)buff.get());
}
} /* Output:
Some text Some more
*///:-

```

Para cualquiera de las clases de flujos de datos mostradas aquí, **getChannel()** generará un objeto **FileChannel**. Los canales son bastante básicos. Podemos entregarlos un objeto **ByteBuffer** para lectura o escritura, y podemos bloquear regiones del archivo con el fin de obtener acceso exclusivo (hablaremos más sobre esto posteriormente).

Una forma de insertar bytes en un objeto **ByteBuffer** consiste en introducirlos directamente utilizando uno de los métodos **put**, con el fin de insertar uno o más bytes, o valores de tipos primitivos. Sin embargo, como puede verse en el ejemplo, también podemos envolver una matriz de tipo **byte** en un objeto **ByteBuffer** utilizando el método **wrap()**. Cuando hacemos esto, no se copia la matriz subyacente, sino que se utiliza como almacenamiento para el objeto **ByteBuffer** generado. En este caso, decimos que el objeto **ByteBuffer** está “respaldado” por la matriz.

El archivo **data.txt** se vuelve a abrir utilizando un objeto **RandomAccessFile**. Observe que debemos desplazar el objeto **FileChannel** por el archivo; en el ejemplo, se le desplaza hasta al final para poder añadir nueva información mediante escrituras adicionales.

Para acceso de sólo lectura, es necesario asignar explícitamente un objeto **ByteBuffer** utilizando el método estático **allocate()**. El objetivo de **nio** consiste en transferir rápidamente grandes cantidades de datos, por lo que el tamaño del objeto **ByteBuffer** tiene su importancia: de hecho, el valor de 1K utilizado aquí resulta, probablemente, más pequeño de lo que normalmente conviene utilizar (tendrá que experimentar con cada aplicación para encontrar el tamaño adecuado).

Resulta posible obtener una velocidad aún mayor usando **allocateDirect()** en lugar de **allocate()**, con el fin de generar un buffer “directo” que pueda estar acoplado de forma aún más estrecha con el sistema operativo. Sin embargo, el gasto adicional de procesamiento de dicho tipo de asignación es mayor, y la implementación varía de un sistema operativo a otro, así que, de nuevo, será necesario experimentar con cada aplicación para determinar si un buffer directo permite obtener una ventaja de velocidad.

Después de invocar **read()** para decirle al objeto **FileChannel** que almacene bytes en el objeto **ByteBuffer**, es necesario invocar **flip()** en el *buffer* con el fin de que éste se prepare para la extracción de los bytes que contiene (como puede ver, el mecanismo parece un poco rudimentario, pero recuerde que es un mecanismo de muy bajo nivel y que está pensado para obtener la máxima velocidad). Y si fuéramos a utilizar el *buffer* para operaciones **read()** adicionales, tendríamos también que invocar **clear()** para preparar el *buffer* para cada **read()**. Podemos ilustrar esto mediante un sencillo programa de copia de archivos:

```

//: io/ChannelCopy.java
// Copia en un archivo utilizando canales y buffers
// {Args: ChannelCopy.java test.txt}
import java.nio.*;

```

```

import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
        while(in.read(buffer) != -1) {
            buffer.flip(); // Preparación para la escritura
            out.write(buffer);
            buffer.clear(); // Preparación para la lectura
        }
    }
} //:-/

```

Como puede ver, se abre un objeto **FileChannel** para lectura y otro para escritura. Se asigna un objeto **ByteBuffer**, y cuando **FileChannel.read()** devuelve **-1** (una reminiscencia, sin lugar a duda, de Unix y C), querrá decir que hemos alcanzado el final del flujo de datos de entrada. Después de cada **read()**, que inserta datos en el **buffer**, **flip()** prepara el **buffer** para poder extraer la información con una llamada a **write()**. Después de la ejecución **write()**, la información continuará estando en el **buffer**, y **clear()** permitirá reinicializar todos los punteros internos para que el **buffer** quede listo para aceptar datos durante otras llamadas a **read()**.

Sin embargo, el programa anterior no es la forma ideal de gestionar este tipo de operación. Dos métodos especiales **transferTo()** y **transferFrom()** permiten conectar directamente un canal con otro:

```

//: io/TransferTo.java
// Utilización de transferTo() entre canales
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        in.transferTo(0, in.size(), out);
        // O bien:
        // out.transferFrom(in, 0, in.size());
    }
} //:-/

```

No vamos a tener que hacer este tipo de cosas muy a menudo en nuestras tareas de programación, pero resulta conveniente conocerlas.

Conversión de datos

Si volvemos a examinar **GetChannel.java**, observaremos que para imprimir la información del archivo, estamos extrayendo los datos de byte en byte y proyectando cada **byte** sobre un valor **char**. Esto parece un tanto primitivo: si examinamos la clase **java.nio.CharBuffer**, veremos que dispone de un método **toString()** que dice: "Quiero que me devuelvas una cade-

na de caracteres que contenga los caracteres de este *buffer*". Puesto que un objeto **ByteBuffer** puede manipularse como un *buffer* de tipo **CharBuffer** mediante el método **asCharBuffer()**, ¿por qué no usar dicho método? Como puede ver analizando la primera línea de la salida del siguiente ejemplo, dicha solución no funciona:

```
//: io/BufferToText.java
// Conversión de texto para objetos ByteBuffer
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        FileChannel fc =
            new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text".getBytes()));
        fc.close();
        fc = new FileInputStream("data2.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        // No funciona:
        System.out.println(buff.asCharBuffer());
        // Decodificar usando el conjunto de caracteres
        // predeterminado de este sistema:
        buff.rewind();
        String encoding = System.getProperty("file.encoding");
        System.out.println("Decoded using " + encoding + ": "
            + Charset.forName(encoding).decode(buff));
        // O bien, podemos codificar con algo que permita imprimir:
        fc = new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap(
            "Some text".getBytes("UTF-16BE")));
        fc.close();
        // Ahora intentamos leer de nuevo:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
        // Utilizar un objeto CharBuffer a través del cual escribir:
        fc = new FileOutputStream("data2.txt").getChannel();
        buff = ByteBuffer.allocate(24); // Más que suficiente
        buff.asCharBuffer().put("Some text");
        fc.write(buff);
        fc.close();
        // Leer y visualizar:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
    }
} /* Output:
?????
Decoded using Cp1252: Some text
Some text
Some text
*///:-
```

El *buffer* contiene bytes sin formato, y para transformarlos en caracteres debemos *codificarlos* a medida que los introducimos (para que tengan significado cuando se los extraiga) o *decodificarlos* a medida que salen del *buffer*. Esto se puede conseguir utilizando la clase **java.nio.charset.Charset**, que proporciona herramientas para la codificación en muchos tipos distintos de conjuntos de caracteres:

```
//: io/AvailableCharsets.java
// Muestra los conjuntos de caracteres y sus alias
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class AvailableCharsets {
    public static void main(String[] args) {
        SortedMap<String, Charset> charSets =
            Charset.availableCharsets();
        Iterator<String> it = charSets.keySet().iterator();
        while(it.hasNext()) {
            String csName = it.next();
            printnb(csName);
            Iterator aliases =
                charSets.get(csName).aliases().iterator();
            if(aliases.hasNext())
                printnb(": ");
            while(aliases.hasNext()) {
                printnb(aliases.next());
                if(aliases.hasNext())
                    printnb(", ");
            }
            print();
        }
    }
} /* Output:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0,
big5hkscs, Big5_HKSCS
EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp,
Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp,
euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-
1987, euc_kr, ks_c_5601-1987, euckr, csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn, eucnn, x-EUC-CN
GBK: windows-936, CP936
*** */
*///:-
```

Por tanto, volviendo a **BufferToText.java**, si rebobinamos el *buffer* con *rewind()* (para retroceder al principio de los datos) y a continuación utilizamos el conjunto de caracteres predeterminado de esa plataforma para decodificar los datos con **decode()**, el objeto de *buffer* resultante **CharBuffer** podrá imprimirse sin problemas en la consola. Para averiguar el conjunto de caracteres predeterminado, use **System.getProperty("file.encoding")**, que genera la cadena de caracteres que da nombre al conjunto de caracteres. Pasando este nombre a **Charset.forName()** se genera el objeto **Charset** (conjunto de caracteres) que puede utilizarse para decodificar la cadena.

Otra alternativa consiste en codificar (con **encode()**) utilizando un conjunto de caracteres que permita obtener algo que pueda imprimirse en el momento de leer el archivo, como podemos ver en la tercera parte del archivo **BufferToText.java**. Aquí, se utiliza UTF-16BE para escribir el texto en el archivo, y en el momento de leerlo, todo lo que hay que hacer es convertirlo a un objeto **CharBuffer**, el cual generará el texto esperado.

Por último, podemos ver lo que sucede si *escribimos* en el objeto **ByteBuffer** a través de un objeto **CharBuffer** (analizaremos este tema con más detalle posteriormente). Observe que asignamos 24 bytes para el objeto **ByteBuffer**. Puesto que cada

valor **char** requiere dos bytes, esto es suficiente para 12 caracteres, pero "Some text" sólo tiene 9. Los restantes bytes, con valor cero, continuarán apareciendo en la representación del objeto **CharBuffer** generada por su método **toString()**, como puede verse en la salida.

Ejercicio 23: (6) Cree y pruebe un método de utilidad para imprimir el contenido de un objeto **CharBuffer** hasta la posición en que los caracteres dejen de ser imprimibles.

Extracción de primitivas

Aunque un objeto **ByteBuffer** sólo almacena bytes, contiene métodos para generar cada uno de los diferentes tipos de valores primitivos a partir de los bytes que contiene. Este ejemplo ilustra la inserción y la extracción de varios valores empleando dichos métodos:

```
//: io/GetData.java
// Obtención de diferentes representaciones
// a partir de un objeto ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;

public class GetData {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        // El proceso de asignación pone a cero
        // automáticamente el objeto ByteBuffer:
        int i = 0;
        while(i++ < bb.limit())
            if(bb.get() != 0)
                print("nonzero");
        print("i = " + i);
        bb.rewind();
        // Almacenar y leer una matriz char:
        bb.asCharBuffer().put("Howdy!");
        char c;
        while((c = bb.getChar()) != 0)
            printnb(c + " ");
        print();
        bb.rewind();
        // Almacenar y leer un valor short:
        bb.asShortBuffer().put((short)471142);
        print(bb.getShort());
        bb.rewind();
        // Almacenar y leer un valor int:
        bb.asIntBuffer().put(99471142);
        print(bb.getInt());
        bb.rewind();
        // Almacenar y leer un valor long:
        bb.asLongBuffer().put(99471142);
        print(bb.getLong());
        bb.rewind();
        // Almacenar y leer un valor float:
        bb.asFloatBuffer().put(99471142);
        print(bb.getFloat());
        bb.rewind();
        // Almacenar y leer un valor double:
        bb.asDoubleBuffer().put(99471142);
        print(bb.getDouble());
        bb.rewind();
    }
}
```

```

} /* Output:
i = 1025
H o w d y !
12390
99471142
99471142
9.9471144E7
9.9471142E7
*///:-
```

Después de asignar un objeto **ByteBuffer**, se comprueban sus valores para ver si el proceso de asignación del *buffer* pone a cero automáticamente el contenido; como podemos ver, así sucede. Se comprueban los 1.024 valores (hasta el límite de *buffer* obtenido con **limit()**), y veremos que todos ellos son cero.

La forma más fácil de insertar valores primitivos en un objeto **ByteBuffer** es obtener la “vista” apropiada de dicho *buffer* utilizando **asCharBuffer()**, **asShortBuffer()**, etc., y luego empleando el método **put()** correspondiente a dicha lista. Podemos ver en el ejemplo que éste es el proceso que se ha utilizado para cada uno de los tipos de datos primitivos. El único de estos casos que resulta un poco más extraño es el método **put()** para el objeto **ShortBuffer**, que requiere una proyección de datos (observe que la proyección trunca y modifica la proyección resultante). Todos los demás *buffers* utilizados como vistas no requieren que se efectúe ninguna proyección de datos en sus métodos **put()**.

Buffers utilizados como vistas

Los “*buffers* utilizados como vistas” permiten examinar un objeto **ByteBuffer** subyacente a través de la ventana que proporciona cada tipo primitivo concreto. El objeto **ByteBuffer** seguirá siendo el almacenamiento real que está “respaldando” a esa vista, por lo que cualquier cambio que se realice en la vista se verá reflejado en las correspondientes modificaciones de los datos contenidos en el objeto **ByteBuffer**. Como podemos ver en el ejemplo anterior, esto nos permite insertar cómodamente tipos primitivos en un *buffer* de tipo **ByteBuffer**. Una vista permite también leer tipos primitivos a partir de un objeto **ByteBuffer**, bien de uno en uno (tal como lo permite **ByteBuffer**) o por lotes (almacenando en matrices). He aquí un ejemplo en el que se manipulan objetos **int** en un objeto **ByteBuffer** a través de una vista **IntBuffer**:

```

//: io/IntBufferDemo.java
// Manipulación de valores int en un objeto ByteBuffer mediante IntBuffer
import java.nio.*;

public class IntBufferDemo {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        IntBuffer ib = bb.asIntBuffer();
        // Almacenar una matriz de valores int:
        ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
        // Lectura y escritura en posiciones absolutas:
        System.out.println(ib.get(3));
        ib.put(3, 1811);
        // Establecimiento de un nuevo límite antes rebobinar el buffer.
        ib.flip();
        while(ib.hasRemaining()) {
            int i = ib.get();
            System.out.println(i);
        }
    }
} /* Output:
99
11
42
47
1811
143
```

```
811
1016
*///:-
```

Se utiliza primero el método sobrecargado **put()** para almacenar una matriz de valores **int**. Las siguientes llamadas a los métodos **get()** y **put()** acceden directamente a una posición **int** dentro del objeto **ByteBuffer** subyacente. Observe que estos accesos mediante la posición absoluta están también disponibles para los tipos primitivos si manipulamos directamente el objeto **ByteBuffer**.

Una vez rellenado el objeto **ByteBuffer** con objetos **int** o algún otro tipo primitivo a través de un *buffer* de vista, podemos escribir el objeto **ByteBuffer** directamente en un canal. También podemos, con igual facilidad, leer de un canal y usar un *buffer* de vista para convertir todo a un tipo concreto de primitiva. He aquí un ejemplo que interpreta la secuencia de bytes como valores **short**, **int**, **float**, **long** y **double** generando diferentes *buffers* de vista para el mismo objeto **ByteBuffer**:

```
//: io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class ViewBuffers {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(
            new byte[]{ 0, 0, 0, 0, 0, 0, 0, 0, 'a' });
        bb.rewind();
        println("Byte Buffer ");
        while(bb.hasRemaining())
            println(bb.position() + " -> " + bb.get() + ", ");
        print();
        CharBuffer cb =
            ((ByteBuffer)bb.rewind()).asCharBuffer();
        println("Char Buffer ");
        while(cb.hasRemaining())
            println(cb.position() + " -> " + cb.get() + ", ");
        print();
        FloatBuffer fb =
            ((ByteBuffer)bb.rewind()).asFloatBuffer();
        println("Float Buffer ");
        while(fb.hasRemaining())
            println(fb.position() + " -> " + fb.get() + ", ");
        print();
        IntBuffer ib =
            ((ByteBuffer)bb.rewind()).asIntBuffer();
        println("Int Buffer ");
        while(ib.hasRemaining())
            println(ib.position() + " -> " + ib.get() + ", ");
        print();
        LongBuffer lb =
            ((ByteBuffer)bb.rewind()).asLongBuffer();
        println("Long Buffer ");
        while(lb.hasRemaining())
            println(lb.position() + " -> " + lb.get() + ", ");
        print();
        ShortBuffer sb =
            ((ByteBuffer)bb.rewind()).asShortBuffer();
        println("Short Buffer ");
        while(sb.hasRemaining())
            println(sb.position() + " -> " + sb.get() + ", ");
        print();
        DoubleBuffer db =
            ((ByteBuffer)bb.rewind()).asDoubleBuffer();
        println("Double Buffer ");
```

```

        while(db.hasRemaining())
            printnb(db.position() + " -> " + db.get() + ", ");
    }
} /* Output:
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0, 6 -> 0, 7 -> 97,
Char Buffer 0 -> , 1 -> , 2 -> , 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
*///:-
```

El objeto **ByteBuffer** se genera “envolviendo” una matriz de ocho bytes que a continuación se visualiza a través de *buffers* de vista apropiados para todos los diferentes tipos primitivos. Podemos ver en el siguiente diagrama las distintas formas en que los datos aparecen cuando se los lee desde los diferentes tipos de *buffers*:

| | | | | | | | | |
|-----|----|----------|----------|----|---|----|----|--------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 97 | byte |
| | | | | | | a | | char |
| 0 | | 0 | | 0 | | 97 | | short |
| | 0 | | | 97 | | | | int |
| 0.0 | | | 1.36E-43 | | | | | float |
| | 97 | | | | | | | long |
| | | 4.8E-322 | | | | | | double |

Estos valores se corresponderían con la salida del programa.

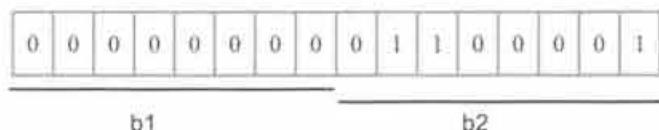
Ejercicio 24: (1) Modifique **IntBufferDemo.java** para utilizar valores **double**.

Terminaciones

Las diferentes máquinas pueden utilizar diferentes esquemas de ordenación de los bytes a la hora de almacenar los datos. Las máquinas con “terminación alta” (*big endian*) colocan el byte más significativo en la dirección de memoria más baja, mientras que las máquinas con “terminación baja” (*little endian*) colocan el byte más significativo en la dirección de memoria más alta.

A la hora de almacenar una magnitud con un tamaño superior a un **byte**, como por ejemplo **int**, **float**, etc., puede ser necesario tener en cuenta la ordenación de los bytes. Un objeto **ByteBuffer** almacena los datos en formato de terminación alta y los datos enviados a través de una red siempre utilizan terminación alta. Podemos cambiar el tipo de terminación de un objeto **ByteBuffer** utilizando **order()** y pasando a dicho método el argumento **ByteOrder.BIG_ENDIAN** o **ByteOrder.LITTLE_ENDIAN**.

Considere un objeto **ByteBuffer** que contenga los siguientes dos bytes:



Si leemos los datos como un valor **short** (**ByteBuffer.asShortBuffer()**), obtendremos el número 97 (00000000 01100001), pero si cambiamos a terminación baja, obtendremos el número 24832 (01100001 00000000).

He aquí un ejemplo que muestra cómo se modifica la ordenación de los bytes en los caracteres dependiendo de la terminación elegida:

```
//: io/Endians.java
// Diferencias de terminación y almacenamiento de datos.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endians {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.BIG_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.LITTLE_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
    }
} /* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:-
```

Al objeto **ByteBuffer** se le asigna suficiente espacio para almacenar todos los bytes de una matriz de caracteres, de modo que podemos invocar el método **array()** para visualizar los bytes subyacentes. El método **array()** es “opcional” y sólo se puede invocar sobre un *buffer* que esté respaldado por una matriz; en caso contrario, se generará una excepción **UnsupportedOperationException**.

Al visualizar los bytes subyacentes, podemos ver que la ordenación predeterminada coincide con la impuesta por el sistema de terminación alta, mientras que el sistema de terminación baja invierte los bytes.

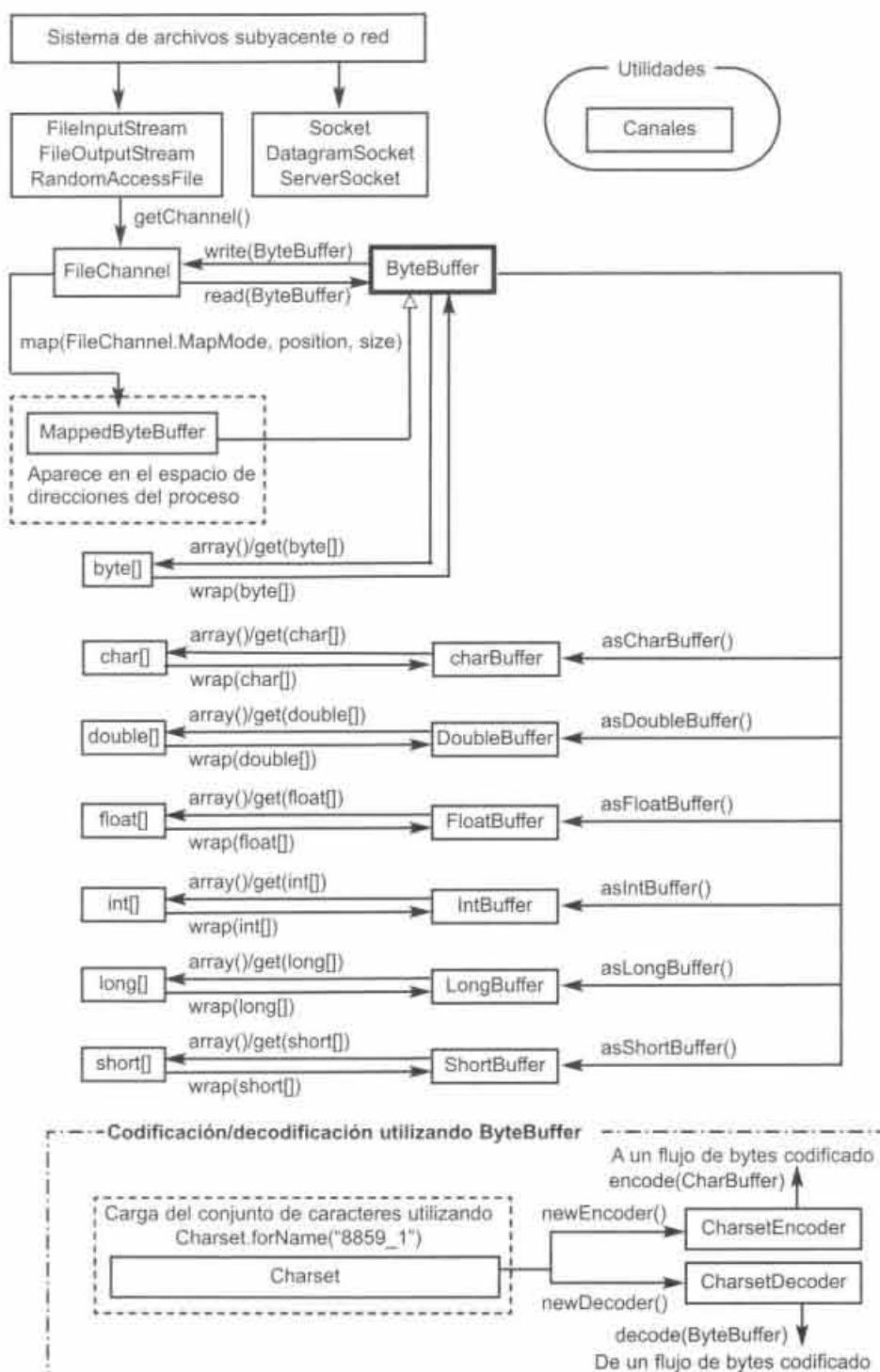
Manipulación de datos con *buffers*

El diagrama de la página siguiente ilustra las relaciones entre las clases **nio**, para poder entender mejor cómo se transfieren y se convierten los datos. Por ejemplo, si queremos escribir una matriz de tipo **byte** en un archivo, tenemos que envolver la matriz **byte** utilizando el método **ByteBuffer.wrap()**, abrir un canal en el flujo **FileOutputStream** usando el método **getChannel()** y luego escribir los datos en el canal **FileChannel** a partir de objeto **ByteBuffer**.

Observe que **ByteBuffer** es la única forma de transferir datos hacia y desde los canales y que nosotros sólo podemos crear un *buffer* autónomo con un tipo de datos primitivo, u obtener uno a partir de un objeto **ByteBuffer** empleando un método “**as**”. En otras palabras, no se puede convertir un *buffer* con tipo de datos primitivo *en* un objeto **ByteBuffer**. Sin embargo, puesto que podemos transferir datos primitivos hacia y desde un objeto **ByteBuffer** a través de un *buffer* de vista, esta restricción realmente no es tal.

Detalles acerca de los *buffers*

Un objeto **Buffer** está compuesto por datos y por cuatro índices que permiten acceder a estos datos y manipularlos eficientemente: *marca*, *posición*, *límite* y *capacidad*. Existen métodos para asignar valores a estos índices, para reiniciarlos y para consultar su valor (véase la tabla de las Páginas 626-627).



| | |
|-------------------------|--|
| <code>capacity()</code> | Devuelve la <i>capacidad</i> del <i>buffer</i> . |
| <code>clear()</code> | Borra el <i>buffer</i> , establece la <i>posición</i> en cero y asigna al <i>límite</i> el valor de la <i>capacidad</i> . Invocamos este método para sobreescribir un <i>buffer</i> existente. |
| <code>flip()</code> | Asigna al <i>límite</i> el valor de <i>posición</i> y asigna a <i>posición</i> el valor cero. Este método se utiliza para preparar el <i>buffer</i> para una lectura después de haber escrito datos en él. |
| <code>limit()</code> | Devuelve el valor del <i>límite</i> . |

| | |
|--------------------------|---|
| limit(int lim) | Establece el valor del <i>límite</i> . |
| mark() | Establece la <i>marca</i> en el valor correspondiente a <i>posición</i> . |
| position() | Devuelve el valor de <i>posición</i> . |
| position(int pos) | Establece el valor de <i>posición</i> . |
| remaining() | Devuelve (<i>límite - posición</i>). |
| hasRemaining() | Devuelve true si existe algún elemento entre <i>posición</i> y <i>límite</i> . |

Los métodos que insertan y extraen datos del *buffer* actualizan estos índices para reflejar los cambios.

Este ejemplo utiliza un algoritmo muy simple (intercambio de los caracteres adyacentes) para cifrar y descifrar los caracteres contenidos en un objeto **CharBuffer**:

```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
    private static void symmetricScramble(CharBuffer buffer) {
        while(buffer.hasRemaining()) {
            buffer.mark();
            char c1 = buffer.get();
            char c2 = buffer.get();
            buffer.reset();
            buffer.put(c2).put(c1);
        }
    }
    public static void main(String[] args) {
        char[] data = "UsingBuffers".toCharArray();
        ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
        CharBuffer cb = bb.asCharBuffer();
        cb.put(data);
        print(cb.rewind());
        symmetricScramble(cb);
        print(cb.rewind());
        symmetricScramble(cb);
        print(cb.rewind());
    }
} /* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*///:-
```

Aunque podríamos generar un *buffer* de tipo **CharBuffer** directamente invocando **wrap()** con una matriz de tipo **char**, asignamos en su lugar un objeto **ByteBuffer** subyacente, generándose el objeto **CharBuffer** como una vista del **ByteBuffer**. Este método enfatiza el hecho de que nuestro objetivo es siempre manipulado como un objeto **ByteBuffer**, ya que es éste el objeto el que interactúa con un canal.

He aquí el aspecto del *buffer* a la entrada del método **the symmetricScramble()**:



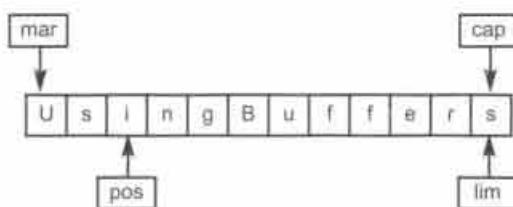
La *posición* apunta al primer elemento del *buffer*, y la *capacidad* y el *límite* apuntan al último elemento.

En `symmetricScramble()`, el bucle `while` realiza una serie de iteraciones hasta que *posición* es equivalente a *límite*. La *posición* del *buffer* varía cada vez que se invoca una función `get()` o `put()` relativa. También se pueden invocar métodos `get()` y `put()` absolutos, que incluyen un argumento de índice que especifica la ubicación en la que la operación `get()` o `put()` tienen lugar. Estos métodos no modifican el valor de la *posición* del *buffer*.

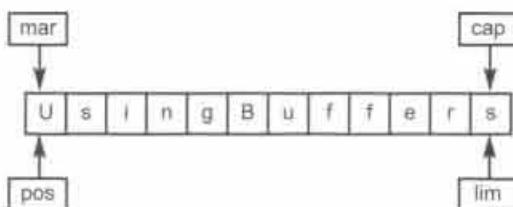
Cuando el control entra en el bucle `while`, el valor de *marca* se fija utilizando una llamada a `mark()`. El estado del *buffer* es entonces:



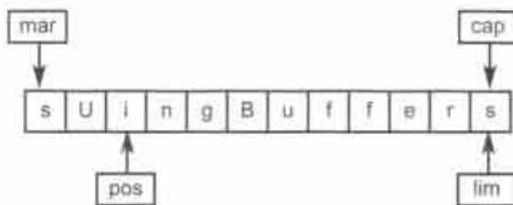
Las dos llamadas a `get()` relativas guardan el valor de los dos primeros caracteres en las variables `c1` y `c2`. Después de estas dos llamadas, el *buffer* tendrá el siguiente aspecto:



Para realizar el intercambio, necesitamos escribir `c2` en *posición* = 0 y `c1` en *posición* = 1. Podemos emplear el método absoluto de inserción para conseguir esto, o asignar a *posición* el valor de *marca*, que es precisamente lo que hace el método `reset()`:



Los dos métodos `put()` escriben `c2` y luego `c1`:



Durante la siguiente iteración del bucle, se asigna a *marca* el valor actual de *posición*:



El proceso continúa hasta que se ha recorrido todo el *buffer*. Al final del bucle **while**, *posición* apuntará al final del *buffer*. Si imprimimos el *buffer*, sólo se imprimirán los caracteres entre *posición* y *límite*. Por tanto, si queremos mostrar el contenido completo del *buffer*, deberemos fijar la *posición* al principio del *buffer* utilizando **rewind()**. He aquí el estado del *buffer* después de la llamada a **rewind()** (el valor de *marca* pasa a ser indefinido):



Cuando se invoca de nuevo la función **symmetricScramble()**, el *buffer CharBuffer* pasa por el mismo proceso y se restaura a su estado original.

Archivos mapeados en memoria

Los archivos mapeados en memoria permiten crear y modificar archivos que sean demasiado grandes como para cargarlos en memoria. Con un archivo mapeado en memoria, podemos actuar como si todo el archivo se encontrara en la memoria y podemos acceder a él tratándolo simplemente como si fuera una matriz de muy gran tamaño. Esta técnica simplifica enormemente el código que es necesario escribir para poder modificar el archivo. He aquí un ejemplo simple:

```

//: io/LargeMappedFiles.java
// Creación de un archivo de muy gran tamaño
// utilizando el mapeado de memoria.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class LargeMappedFiles {
    static int length = 0xFFFFFFFF; // 128 MB
    public static void main(String[] args) throws Exception {
        MappedByteBuffer out =
            new RandomAccessFile("test.dat", "rw").getChannel()
                .map(FileChannel.MapMode.READ_WRITE, 0, length);
        for(int i = 0; i < length; i++)
            out.put((byte)'x');
        print("Finished writing");
        for(int i = length/2; i < length/2 + 6; i++)
            printnb((char)out.get(i));
    }
} //:-
  
```

Para efectuar tanto lecturas como escrituras, comenzamos con un objeto **RandomAccessFile**, obtenemos un canal para dicho archivo y luego invocamos **map()** para generar un *buffer MappedByteBuffer*, que es un tipo particular de *buffer* directo. Observe que es necesario especificar el punto de inicio y la longitud de la región en la que queremos mapear el archivo; esto implica que tenemos la posibilidad de mapear regiones pequeñas de un archivo de gran tamaño.

MappedByteBuffer hereda de **ByteBuffer**, así que dispone de todos los métodos de dicha clase. Aquí sólo mostramos los usos más simples de **put()** y **get()**, pero también podemos emplear métodos como **asCharBuffer()**, etc.

El archivo creado en el programa anterior tiene 128 MB de longitud, lo cual es un tamaño probablemente mayor de lo que el sistema operativo permitirá residir en memoria en cualquier momento determinado. El archivo parece estar completamente accesible, aunque en realidad sólo se cargan en memoria partes del mismo, intercambiándose por otras partes a medida que es necesario. De esta forma, puede modificarse fácilmente un archivo de muy gran tamaño (hasta dos 2 GB). Observe que se utiliza la funcionalidad de mapeo de archivos del sistema operativo subyacente para maximizar el rendimiento.

Rendimiento

Aunque el rendimiento del “antiguo” mecanismo de flujos de datos de E/S se ha mejorado al implementarlo con **nio**, el acceso a archivos mapeados tiende a ser muchísimo más rápido. Este programa hace una comparativa simple de rendimiento:

```
//: io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
    private static int numOfInts = 4000000;
    private static int numOfUbuffInts = 200000;
    private abstract static class Tester {
        private String name;
        public Tester(String name) { this.name = name; }
        public void runTest() {
            System.out.print(name + ": ");
            try {
                long start = System.nanoTime();
                test();
                double duration = System.nanoTime() - start;
                System.out.format("%.2f\n", duration/1.0e9);
            } catch(IOException e) {
                throw new RuntimeException(e);
            }
        }
        public abstract void test() throws IOException;
    }
    private static Tester[] tests = {
        new Tester("Stream Write") {
            public void test() throws IOException {
                DataOutputStream dos = new DataOutputStream(
                    new BufferedOutputStream(
                        new FileOutputStream(new File("temp.tmp"))));
                for(int i = 0; i < numOfInts; i++)
                    dos.writeInt(i);
                dos.close();
            }
        },
        new Tester("Mapped Write") {
            public void test() throws IOException {
                FileChannel fc =
                    new RandomAccessFile("temp.tmp", "rw")
                    .getChannel();
                IntBuffer ib = fc.map(
                    FileChannel.MapMode.READ_WRITE, 0, fc.size())
                    .asIntBuffer();
                for(int i = 0; i < numOfInts; i++)
                    ib.putInt(i);
                fc.close();
            }
        },
        new Tester("Stream Read") {
            public void test() throws IOException {
                DataInputStream dis = new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream("temp.tmp")));
                for(int i = 0; i < numOfInts; i++)
                    dis.readInt();
            }
        }
    };
}
```

```

        dis.readInt();
        dis.close();
    },
    new Tester("Mapped Read") {
        public void test() throws IOException {
            FileChannel fc = new FileInputStream(
                new File("temp.tmp")).getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_ONLY, 0, fc.size())
                .asIntBuffer();
            while(ib.hasRemaining())
                ib.get();
            fc.close();
        }
    },
    new Tester("Stream Read/Write") {
        public void test() throws IOException {
            RandomAccessFile raf = new RandomAccessFile(
                new File("temp.tmp"), "rw");
            raf.writeInt(1);
            for(int i = 0; i < numOfUbuffInts; i++) {
                raf.seek(raf.length() - 4);
                raf.writeInt(raf.readInt());
            }
            raf.close();
        }
    },
    new Tester("Mapped Read/Write") {
        public void test() throws IOException {
            FileChannel fc = new RandomAccessFile(
                new File("temp.tmp"), "rw").getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_WRITE, 0, fc.size())
                .asIntBuffer();
            ib.put(0);
            for(int i = 1; i < numOfUbuffInts; i++)
                ib.put(ib.get(i - 1));
            fc.close();
        }
    }
};
public static void main(String[] args) {
    for(Tester test : tests)
        test.runTest();
}
} /* Output: (90% match)
Stream Writer: 0.56
Mapped Write: 0.12
Stream Read: 0.80
Mapped Read: 0.07
Stream Read/Write: 5.32
Mapped Read/Write: 0.02
*///:-
```

Como hemos visto en ejemplos anteriores de este libro, `runTest()` se utiliza con el *método de plantillas* para crear un marco de pruebas para diversas implementaciones de `test()` definidas en subclases internas anónimas. Cada una de estas subclases realiza un tipo de prueba, por lo que los métodos `test()` también nos proporcionan un prototipo para realizar las distintas actividades de E/S.

Aunque podría parecer que una escritura mapeada debería utilizar un flujo **FileOutputStream**, todas las operaciones de salida en el mecanismo de mapeo de archivos deben utilizar un objeto **RandomAccessFile**, al igual que se hace con las operaciones de lectura/escritura en el programa anterior.

Observe que los métodos **test()** incluyen el tiempo necesario para inicializar los distintos objetos de E/S, de modo que aunque la configuración de los archivos mapeados puede requerir un gasto de procesamiento considerable, la ganancia global de velocidad, por comparación con la E/S basada en flujos de datos resulta significativa.

Ejercicio 25: (6) Experimente cambiando las instrucciones **ByteBuffer.allocate()** de los ejemplos de este capítulo por **ByteBuffer.allocateDirect()**. Demuestre las diferencias de rendimiento que existen, pero observe también si el tiempo de arranque de los programas se modifica de manera perceptible.

Ejercicio 26: (3) Modifique **strings/JGrep.java** para utilizar archivos mapeados en memoria al estilo **nio** de Java.

Bloqueo de archivos

El bloqueo de archivos permite sincronizar el acceso a un archivo utilizado como recurso compartido. Sin embargo, las dos hebras que compiten por el mismo archivo pueden estar en diferentes máquinas virtuales Java, o bien una de ellas puede ser una hebra de programación Java y la otra puede ser alguna hebra nativa del sistema operativo. Los bloqueos de archivo son visibles para otros procesos del sistema operativo, porque el mecanismo de bloqueo de archivos de Java se mapea directamente sobre la funcionalidad de bloqueo nativa del sistema operativo.

He aquí un ejemplo simple de bloqueo de archivos.

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
    public static void main(String[] args) throws Exception {
        FileOutputStream fos= new FileOutputStream("file.txt");
        FileLock fl = fos.getChannel().tryLock();
        if(fl != null) {
            System.out.println("Locked File");
            TimeUnit.MILLISECONDS.sleep(100);
            fl.release();
            System.out.println("Released Lock");
        }
        fos.close();
    }
} /* Output:
Locked File
Released Lock
*///:-
```

Podemos obtener un bloqueo (**FileLock**) sobre el archivo completo invocando **tryLock()** o **lock()** sobre un objeto **FileChannel**. (**SocketChannel**, **DatagramChannel** y **ServerSocketChannel** no necesitan bloqueos, ya que son inherentemente entidades de un único proceso, generalmente un *socket* de red no se comparte entre dos procesos). **tryLock()** es no bloqueante: este método trata de establecer el bloqueo, pero si no puede (porque algún otro proceso ya ha establecido el mismo bloqueo y éste no es de tipo compartido), simplemente se limita a volver del método terminando así la llamada. **lock()** se bloquea hasta que adquiere el bloqueo indicado, o hasta que se interrumpe la hebra que ha invocado **lock()**, o hasta que se cierra el canal para el cual se ha invocado el método **lock()**. Un bloqueo se libera utilizando **FileLock.release()**.

También es posible bloquear una parte del archivo con:

```
tryLock(long posición, long tamaño, boolean compartido)
0
lock(long posición, long tamaño, boolean compartido)
```

que bloquea la región (**tamaño – posición**). El tercer argumento especifica si este bloqueo es compartido.

Aunque los métodos de bloqueo que no utilizan argumentos pueden adaptarse a los cambios en el tamaño de un archivo, los bloqueos con un tamaño fijo no se modifican cuando cambia el tamaño del archivo. Si se establece un bloqueo para una región comprendida entre **posición** y **posición + tamaño** y el archivo se incrementa más allá de **posición + tamaño**, entonces la sección situada después de **posición + tamaño** no estará bloqueada. Los métodos de bloqueo que no utilizan argumentos bloquean el archivo completo, incluso si éste aumenta de tamaño.

El soporte para los bloqueos exclusivos o compartidos debe ser proporcionado por el sistema operativo subyacente. Si el sistema operativo no soporta los bloqueos compartidos y se solicita uno de estos bloqueos, en su lugar se emplea un bloqueo exclusivo. El tipo de bloqueo (compartido o exclusivo) puede consultarse utilizando `FileLock.isShared()`.

Bloqueo de partes de un archivo mapeado

Como hemos mencionado anteriormente, el mecanismo de mapeo de archivos se utiliza principalmente para archivos de muy gran tamaño. Puede que necesitemos bloquear partes de dicho archivo de gran tamaño, de modo que se permita a otros procesos modificar partes del archivo no bloqueadas. Esto es lo que sucede, por ejemplo, con una base de datos, de tal manera que ésta pueda ser utilizada por muchos usuarios a la vez.

He aquí un ejemplo con dos hebras de programación, cada una de las cuales bloquea una parte distinta de un archivo:

```
//: io/LockingMappedFiles.java
// Bloqueo de partes de un archivo mapeado.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
    static final int LENGTH = 0x8FFFFFFF; // 128 MB
    static FileChannel fc;
    public static void main(String[] args) throws Exception {
        fc =
            new RandomAccessFile("test.dat", "rw").getChannel();
        MappedByteBuffer out =
            fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
        for(int i = 0; i < LENGTH; i++)
            out.put((byte)'x');
        new LockAndModify(out, 0, 0 + LENGTH/3);
        new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
    }
    private static class LockAndModify extends Thread {
        private ByteBuffer buff;
        private int start, end;
        LockAndModify(ByteBuffer mbb, int start, int end) {
            this.start = start;
            this.end = end;
            mbb.limit(end);
            mbb.position(start);
            buff = mbb.slice();
            start();
        }
        public void run() {
            try {
                // Bloqueo exclusivo sin solapamiento:
                FileLock fl = fc.lock(start, end, false);
                System.out.println("Locked: " + start + " to " + end);
                // Realizar modificación:
                while(buff.position() < buff.limit() - 1)
                    buff.put((byte)(buff.get() + 1));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        f1.release();
        System.out.println("Released: "+start+" to "+ end);
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
} //:-)

```

La clase de hebra **LockAndModify** configura la región del *buffer* y crea con **slice()** un fragmento para modificarlo. En **run()**, se establece el bloqueo sobre el canal del archivo (no se puede establecer un bloqueo sobre el *buffer*, sólo sobre el canal). La llamada a **lock()** es muy similar a establecer un bloqueo de un objeto en una hebra de programación: después de la llamada dispondremos de una sección crítica con acceso exclusivo a dicha parte del archivo.⁵

Los bloqueos se liberan automáticamente cuando termina la ejecución de la máquina JVM o cuando se cierra el canal para el que se hayan establecido los bloqueos, pero también se puede invocar explícitamente **release()** sobre el objeto **FileLock** como se muestra en el ejemplo.

Compresión

La biblioteca de E/S de Java contiene clases que permiten manejar flujos de lectura y escritura en formato comprimido. Estas clases se envuelven en otras clases de E/S para proporcionar la funcionalidad de compresión.

Estas clases no derivan de las clases **Reader** y **Writer**, sino que forman parte de las jerarquías **InputStream** y **OutputStream**. Esto se debe a que la biblioteca de compresión trabaja con bytes, no con caracteres. Sin embargo, a veces podemos vernos forzados a mezclar los dos tipos de flujos (recuerde que puede utilizar **InputStreamReader** y **OutputStreamWriter** para proporcionar un mecanismo sencillo de conversión entre un tipo y otro).

| Clase de compresión | Función |
|-----------------------------|---|
| CheckedInputStream | GetCheckSum() proporciona la suma de comprobación para cualquier objeto InputStream (no simplemente descompresión). |
| CheckedOutputStream | GetCheckSum() proporciona la suma de comprobación para cualquier objeto OutputStream (no simplemente compresión). |
| DeflaterOutputStream | Clase base para clases de compresión. |
| ZipOutputStream | Un flujo DeflaterOutputStream que comprime datos en el formato de archivos Zip. |
| GZIPOutputStream | Un flujo DeflaterOutputStream que comprime datos en el formato de archivos GZIP. |
| InflaterInputStream | Clase base para clases de descompresión. |
| ZipInputStream | Un flujo InflaterInputStream que descomprime los datos que hayan sido almacenados en el formato de archivos Zip. |
| GZIPInputStream | Un flujo InflaterInputStream que descomprime los datos que hayan sido almacenados en el formato de archivos GZIP. |

Aunque existen muchos algoritmos de compresión, Zip y GZIP son posiblemente los más comúnmente utilizados. De este modo, podemos manipular fácilmente los datos comprimidos con muchas de las herramientas disponibles para la lectura y escritura de estos formatos de archivo.

⁵ Puede encontrar más detalles acerca de las hebras de programación en el Capítulo 21, *Concurrencia*.

Compresión simple con GZIP

La interfaz GZIP es simple y resulta, por tanto, apropiada cuando disponemos de un único flujo de datos que queramos comprimir (en lugar de un contenedor de fragmentos de datos poco similares). He aquí un ejemplo en el que se comprime un único archivo:

```
//: io/GZIPcompress.java
// (Args: GZIPcompress.java)
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
    public static void main(String[] args)
        throws IOException {
        if(args.length == 0) {
            System.out.println(
                "Usage: \nGZIPcompress file\n" +
                "\tUses GZIP compression to compress " +
                "the file to test.gz");
            System.exit(1);
        }
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        BufferedOutputStream out = new BufferedOutputStream(
            new GZIPOutputStream(
                new FileOutputStream("test.gz")));
        System.out.println("Writing file");
        int c;
        while((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
        System.out.println("Reading file");
        BufferedReader in2 = new BufferedReader(
            new InputStreamReader(new GZIPInputStream(
                new FileInputStream("test.gz"))));
        String s;
        while((s = in2.readLine()) != null)
            System.out.println(s);
    }
} /* (Execute to see output) */:-
```

La utilización de las clases de compresión resulta sencilla; basta con envolver el flujo de salida en un objeto **GZIPOutputStream** o **ZipOutputStream**, y el flujo de datos de entrada en un objeto **GZIPInputStream** o **ZipInputStream**. Todo lo demás son lecturas y escrituras de E/S normales. Éste es un ejemplo de mezcla de los flujos de datos orientados a caracteres con los flujos de datos orientados a bytes; **in** utiliza las clases **Reader**, mientras que el constructor de **GZIPOutputStream** sólo puede aceptar un objeto **OutputStream**, no un objeto **Writer**. Cuando se abre el archivo, el objeto **GZIPInputStream** se convierte en un objeto **Reader**.

Almacenamiento de múltiples archivos con Zip

La biblioteca que soporta el formato Zip es más amplia. Con este formato, podemos almacenar fácilmente múltiples archivos y existe incluso una clase separada para facilitar el proceso de lectura de un archivo Zip. La biblioteca utiliza el formato Zip estándar, así que funciona de forma transparente con todas las herramientas Zip que podemos descargar actualmente a través de Internet. El siguiente ejemplo tiene la misma forma que el ejemplo anterior, pero permite tratar tantos argumentos de la línea de comandos como queramos. Además, ilustra el uso de las clases **Checksum** para calcular y verificar la suma de comprobación del archivo. Existen dos tipos de clases **Checksum**: **Adler32** (que es la más rápida) y **CRC32** (que es más lenta, pero ligeramente más precisa).

```

//: io/ZipCompress.java
// Utiliza compresión Zip para comprimir cualquier
// número de archivos que se indique en la línea de comandos.
// (Args: ZipCompress.java)
import java.util.zip.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;
import java.util.zip.*;

public class ZipCompress {
    public static void main(String[] args)
        throws IOException {
        FileOutputStream f = new FileOutputStream("test.zip");
        CheckedOutputStream csum =
            new CheckedOutputStream(f, new Adler32());
        ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out =
            new BufferedOutputStream(zos);
        zos.setComment("A test of Java Zipping");
        // Sin embargo, no hay un método getComment() correspondiente.
        for(String arg : args) {
            print("Writing file " + arg);
            BufferedReader in =
                new BufferedReader(new FileReader(arg));
            zos.putNextEntry(new ZipEntry(arg));
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
            out.flush();
        }
        out.close();
        // La suma de comprobación sólo es válida
        // después de cerrar el archivo!
        print("Checksum: " + csum.getChecksum().getValue());
        // Ahora extraer los archivos:
        print("Reading file");
        FileInputStream fi = new FileInputStream("test.zip");
        CheckedInputStream csumi =
            new CheckedInputStream(fi, new Adler32());
        ZipInputStream in2 = new ZipInputStream(csumi);
        BufferedInputStream bis = new BufferedInputStream(in2);
        ZipEntry ze;
        while((ze = in2.getNextEntry()) != null) {
            print("Reading file " + ze);
            int x;
            while((x = bis.read()) != -1)
                System.out.write(x);
        }
        if(args.length == 1)
            print("Checksum: " + csumi.getChecksum().getValue());
        bis.close();
        // Forma alternativa de abrir y leer archivos Zip:
        ZipFile zf = new ZipFile("test.zip");
        Enumeration e = zf.entries();
        while(e.hasMoreElements()) {
            ZipEntry ze2 = (ZipEntry)e.nextElement();

```

```

        print("File: " + ze2);
        // ... y extraer los datos como antes.
    }
    /* if(args.length == 1) */
}
/* (Execute to see output) */:-
```

Para cada archivo que haya que añadir al archivo comprimido, es preciso invocar **putNextEntry()** y pasarle al método un objeto **ZipEntry**. El objeto **ZipEntry** contiene una amplia interfaz que permite consultar y configurar todos los datos disponibles en esa entrada concreta del archivo Zip: nombre, tamaños comprimido y sin comprimir, fecha, suma de comprobación CRC, campo adicional de datos, comentario, método de compresión e indicador de si se trata de un directorio. Sin embargo, aún cuando el formato Zip dispone de una forma para establecer una contraseña, esta característica no está soportada en la biblioteca Zip de Java. Y aunque **CheckedInputStream** y **CheckedOutputStream** soportan las sumas de comprobación Adler32 y CRC32, la clase **ZipEntry** sólo proporciona una interfaz para CRC. Ésta es una restricción del formato Zip subyacente, pero se trata de una restricción que puede impedirnos utilizar la clase **Adler32** que es más rápida.

Para extraer los archivos, **ZipInputStream** dispone de un método **getNextEntry()** que devuelve la siguiente entrada **ZipEntry**, si es que existe alguna. Como alternativa más sucinta, podemos leer el archivo utilizando un objeto **ZipFile**, que dispone de un método **entries()** para devolver un objeto de tipo **Enumeration** con las entradas del archivo Zip.

Para leer la suma de comprobación, debemos conseguir acceder de alguna forma al objeto **Checksum** asociado. En el ejemplo, retenemos una referencia a los objetos **CheckedOutputStream** y **CheckedInputStream**, pero también podríamos habernos limitado a conservar una referencia al objeto **Checksum**.

Un método bastante absurdo dentro de la biblioteca Zip es **setComment()**. Como se muestra en **ZipCompress.java**, podemos fijar un comentario a la hora de escribir un archivo, pero no existe forma de recuperar el comentario en el objeto **ZipInputStream**. Parece que los comentarios sólo se soportan de manera completa accediendo entrada por entrada mediante **ZipEntry**.

Por supuesto, no tenemos por qué limitarnos a emplear archivos a la hora de utilizar las bibliotecas **GZIP** o **Zip**; podemos comprimir cualquier cosa, incluyendo datos que vayan a enviarse a través de una conexión de red.

Archivos Java (JAR)

El formato Zip también se utiliza en el formato de archivo JAR (Java ARchive), que es una forma de recopilar un grupo de archivos en un único archivo comprimido, igual que Zip. Sin embargo, como todos los demás componentes de Java, los archivos JAR son archivos interplataforma, así que no hay necesidad de preocuparse acerca de los problemas de portabilidad. Pueden incluirse también archivos de audio y de imagen, además de los archivos de clases.

Los archivos JAR resultan particularmente útiles a la hora de trabajar con Internet. Antes de que aparecieran los archivos JAR, el explorador web tenía que realizar solicitudes repetidas a un servidor web para poder descargar todos los archivos que conformaban un *applet*. Además, estos archivos no estaban comprimidos, al combinar todos los archivos de un *applet* concreto en un único archivo JAR, sólo es necesaria una solicitud al servidor y la transferencia se realiza más rápidamente, gracias a la compresión. Además, la entrada de un archivo JAR puede estar firmada digitalmente para aumentar la seguridad.

Un archivo JAR está compuesto por un único archivo que contiene una colección de archivos comprimidos en formato Zip, junto con un “manifiesto” que los describe (podemos crear nuestro propio manifiesto, pero si no lo hacemos, el programa **jar** lo hará por nosotros). Puede encontrar más información acerca de los manifiestos JAR en la documentación del JDK.

La utilidad **jar** incluida en el JDK de Sun comprime automáticamente los archivos que elijamos. Esta utilidad se invoca mediante la línea de comandos:

```
jar [opciones] destino [manifiesto] archivo(s) de entrada
```

Las opciones son simplemente un conjunto de letras (no hace falta ningún guion ni ningún otro símbolo indicador). Los usuarios de Unix/Linux se percatarán de la similitud que existe con las opciones de **tar**. Las opciones disponibles son:

| | |
|-----------|--|
| c | Crea un archivo nuevo o vacío. |
| t | Muestra la tabla de contenido. |
| x | Extrae todos los archivos. |
| x archivo | Extrae el archivo indicado. |
| f | Comunica al programa: "Voy a proporcionarte el nombre del archivo". Si no se usa esta opción, Jar presupone que su entrada procede de la entrada estándar, o, si está creando un archivo, que su salida irá a la salida estándar. |
| m | Especifica que el primer argumento va a ser el nombre del archivo de manifiesto creado por el usuario. |
| v | Genera una salida más prolífica que describe lo que Jar está haciendo. |
| 0 | Sólo almacena los archivos, sin comprimirlos (utilice esta opción para crear un archivo JAR que pueda incluir en su ruta de clases). |
| M | No crea automáticamente un archivo de manifiesto. |

Si se incluye un subdirectorio dentro de los archivos que hay que insertar en el archivo JAR, dicho subdirectorio se añade automáticamente incluyendo todos sus subdirectorios, etc. La información de ruta también se preserva.

He aquí algunas formas típicas de invocar **Jar**. El siguiente comando crea un archivo JAR denominado **myJarFile.jar** que contiene todos los archivos de clases del directorio actual, junto con un archivo de manifiesto generado automáticamente:

```
jar cf myJarFile.jar *.class
```

El siguiente comando es como el del ejemplo anterior, pero añade un archivo de manifiesto creado por el usuario que se denomina **myManifestFile.mf**:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Este otro comando genera una tabla de contenidos de los archivos en **myJarFile.jar**:

```
jar tf myJarFile.jar
```

En el siguiente comando se añade la opción de salida "verbosa", para proporcionar información más detallada acerca de los archivos **myJarFile.jar**:

```
jar tvf myJarFile.jar
```

Suponiendo que **audio**, **classes** e **Image** sean subdirectorios, el siguiente comando combina todos los subdirectorios dentro del archivo **myApp.jar**. También se incluye la opción "verbosa" para obtener información adicional sobre un proceso mientras que está trabajando el programa **Jar**:

```
jar cvf myApp.jar audio classes image
```

Si se crea un archivo JAR utilizando la opción **0** (cero), dicho archivo puede incluirse en la variable **CLASSPATH**:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Con esto, Java podrá explorar **lib1.jar** y **lib2.jar** en busca de archivos de clase.

La herramienta **Jar** no es de propósito tan general como una utilidad **Zip**. Por ejemplo, no podemos añadir archivos a un archivo JAR ni actualizar los archivos existentes; los archivos JAR sólo pueden crearse partiendo de cero. Asimismo, tampoco se pueden desplazar archivos a un archivo JAR, borrando los originales a medida que se los desplaza. Sin embargo, un archivo JAR creado en una plataforma podrá ser leído transparentemente por la herramienta **Jar** en cualquier otra plataforma (evitándose así uno de los problemas que en ocasiones afecta a las utilidades **Zip**).

Como veremos en el Capítulo 22, *Interfaces gráficas de usuario*, los archivos JAR se utilizan para empaquetar componentes JavaBeans.

Serialización de objetos

Cuando se crea un objeto, éste existe durante todo el tiempo que se le necesite, pero siempre deja de existir en cuanto el programa termina. Aunque esto tiene bastante sentido a primera vista, existen situaciones en las que sería enormemente útil que un programa pudiera existir y almacenar su información incluso cuando el programa no se estuviera ejecutando. Si esto fuera así, la siguiente vez que iniciáramos el programa, el objeto ya se encontraría allí y contendría la misma información que tuviera la vez anterior que se ejecutó el programa. Por supuesto, podemos conseguir un efecto similar escribiendo la información en un archivo o en una base de datos, pero si tratamos de mantener el espíritu de que todo sea un objeto, resultaría bastante conveniente poder declarar un objeto como “persistente”, y que el sistema se encargara de resolver todos los detalles por nosotros.

El mecanismo de *serialización de objetos* de Java nos permite tomar cualquier objeto que implemente la interfaz **Serializable** y transformarlo en una secuencia de bytes que pueda restaurarse posteriormente de modo completo, para regenerar el objeto original. Esto es así incluso si estamos trabajando a través de una red, lo que significa que el mecanismo de serialización trata de compensar automáticamente las diferencias existentes en los distintos sistemas operativos. En otras palabras, podemos crear un objeto en una máquina Windows, serializarlo y enviarlo a través de la red a un máquina Unix, donde podrá ser correctamente reconstruido. No es necesario preocuparse acerca de las representaciones de los datos en las distintas máquinas, de la ordenación de bytes, ni de cualquier otro detalle.

En sí misma, la serialización de objetos resulta interesante porque nos permite implementar lo que se denomina *persistencia ligera*. El concepto de persistencia quiere decir que el tiempo de vida de un objeto no está determinado por si un programa se esté ejecutando; el objeto continúa existiendo *entre sucesivas invocaciones* del programa. Tomando un objeto serializable, escribiéndolo en disco para posteriormente restaurar dicho objeto cuando se vuelva a invocar el programa, podemos obtener el efecto de persistencia. La razón por la que a esa persistencia se le denomina “ligera” es que no podemos limitarnos simplemente a definir un objeto utilizando algún tipo de palabra clave “**persistent**” y dejar que el sistema se ocupe de los detalles (aunque quizás pueda hacerse esto en el futuro). En lugar de ello, podemos serializar y des-serializar explícitamente los objetos en nuestro programa. Si necesitamos un mecanismo de persistencia más serio, considere la utilización de alguna herramienta como (<http://hibernate.sourceforge.net>). Para obtener más detalles, consulte *Thinking in Enterprise Java*, que se puede descargar en la dirección www.MindView.net.

La serialización de objetos se añadió al lenguaje para soportar dos características principales. El mecanismo RMI (*Remote Method Invocation*, invocación remota de métodos) de Java permite que los objetos que residen en otras máquinas se comporten como si estuvieran en nuestra propia máquina. Cuando se envían mensajes a los objetos remotos, la serialización de objetos es necesaria para transportar los argumentos y los valores de retorno. El mecanismo de RMI se analiza en *Thinking in Enterprise Java*.

La serialización de objetos también es necesaria para el sistema de componentes JavaBeans, descrito en el Capítulo 22, *Interfaces gráficas de usuario*. Cuando se utiliza un componente Bean, su información de estado suele configurarse, generalmente, en tiempo de diseño. Esta información de estado debe almacenarse, para poder recuperarse posteriormente cuando se inicie el programa; el mecanismo de serialización de objetos se encarga de esta tarea.

La serialización de un objeto es una tarea bastante simple, siempre y cuando el objeto implemente la interfaz **Serializable** (ésta es una interfaz marcadora que no incluye ningún método). Cuando se añadió la serialización al lenguaje, se modificaron muchas clases de la biblioteca estándar para hacerlas serializables, incluyendo todos los envoltorios de los tipos primitivos, todas las clases contenedoras y muchas otras. Incluso los objetos **Class** pueden serializarse.

Para serializar un objeto, creamos algún tipo de objeto **OutputStream** y luego lo envolvemos dentro de un objeto **ObjectOutputStream**. Con esto, lo único que necesitamos es invocar **writeObject()**, y el objeto se serializará y se enviará al flujo de salida **OutputStream** (la serialización de objetos está orientada a bytes, por lo que utiliza las jerarquías **InputStream** y **OutputStream**). Para invertir el proceso, envolvemos un objeto **InputStream** dentro de un objeto **ObjectInputStream** e invocamos **readObject()**. Lo que este método nos devuelve es, como de costumbre, una referencia a un objeto generalizado de tipo **Object**, con lo que es necesario realizar una especialización para que todo funcione correctamente.

Un aspecto particularmente inteligente del mecanismo de serialización de objetos es que éste no sólo guarda una imagen de nuestro objeto, sino que también sigue todas las referencias contenidas en nuestro objeto y guarda *esos* objetos, siguiendo a su vez todas las referencias de cada uno de esos objetos, etc. Esto se denomina en ocasiones la “red de objetos” a la que un único objeto puede estar conectado, e incluye matrices de referencias a objetos, además de objetos miembros. Si tuviéramos

que mantener nuestro propio esquema de serialización de objetos, mantener el código para poder seguir todos nuestros vínculos sería enormemente difícil. Sin embargo, el mecanismo de serialización de objetos de Java parece poder realizar esta tarea de manera muy precisa, utilizando sin ninguna duda algún algoritmo optimizado que recorre la red de objetos. El siguiente ejemplo permite probar el mecanismo de serialización utilizando una cadena de objetos vinculados, cada uno de los cuales tiene un enlace al siguiente segmento de la cadena, así como una matriz de referencias a objetos de una clase distinta, **Data**:

```
//: io/Worm.java
// Ilustra el mecanismo de serialización de objetos.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
    private int n;
    public Data(int n) { this.n = n; }
    public String toString() { return Integer.toString(n); }
}

public class Worm implements Serializable {
    private static Random rand = new Random(47);
    private Data[] d = {
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10))
    };
    private Worm next;
    private char c;
    // Valor de i == número de segmentos
    public Worm(int i, char x) {
        print("Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    public Worm() {
        print("Default constructor");
    }
    public String toString() {
        StringBuilder result = new StringBuilder(":");
        result.append(c);
        result.append("(");
        for(Data dat : d)
            result.append(dat);
        result.append(")");
        if(next != null)
            result.append(next);
        return result.toString();
    }
    public static void main(String[] args)
        throws ClassNotFoundException, IOException {
        Worm w = new Worm(6, 'a');
        print("w = " + w);
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("worm.out"));
        out.writeObject("Worm storage\n");
        out.writeObject(w);
        out.close(); // También vacía la salida
    }
}
```

```

ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("worm.out"));
String s = (String)in.readObject();
Worm w2 = (Worm)in.readObject();
print(s + "w2 = " + w2);
ByteArrayOutputStream bout =
    new ByteArrayOutputStream();
ObjectOutputStream out2 = new ObjectOutputStream(bout);
out2.writeObject("Worm storage\n");
out2.writeObject(w);
out2.flush();
ObjectInputStream in2 = new ObjectInputStream(
    new ByteArrayInputStream(bout.toByteArray()));
s = (String)in2.readObject();
Worm w3 = (Worm)in2.readObject();
print(s + "w3 = " + w3);
}
} /* Output:
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*//:-

```

Para que las cosas sean interesantes, la matriz de objetos **Data** contenida en la cadena **Worm** se inicializa con números aleatorios (de esta forma, eliminamos las sospechas de que el compilador esté conservando algún tipo de meta-information). Cada segmento de **Worm** se etiqueta con un valor **char** que se genera automáticamente como parte del proceso de generación recursiva de la lista enlazada de objetos **Worm**. Cuando se crea un **Worm**, se le dice al constructor la longitud que queremos que tenga. Para construir la referencia **next**, invoca al constructor de **Worm** con una longitud inferior en una unidad, etc. La referencia **next** final se deja con el valor **null**, lo que indica el final de la cadena **Worm**.

El objetivo de esto es construir una estructura razonablemente compleja que no pueda serializarse fácilmente. Sin embargo, el acto de serializar es bastante simple. Una vez que se crea el objeto **ObjectOutputStream** a partir de algún otro flujo de datos, el método **writeObject()** permite serializar el objeto. Observe que también se ha incluido una llamada a **writeObject()** para un objeto **String**. Se pueden también escribir todos los tipos de datos primitivos utilizando los mismos métodos que **DataOutputStream** (comparten la misma interfaz).

Existen dos secciones de código separadas que tienen un aspecto similar. La primera escribe y lee un archivo, mientras que la segunda, para tener un ejemplo más variado, escribe y lee una matriz **ByteArray**. Podemos leer y escribir un objeto, utilizando el mecanismo de serialización, en cualquier flujo **DataInputStream** o **DataOutputStream**, incluyendo (como puede verse en *Thinking in Enterprise Java*) una red.

Podemos ver, examinando la salida, que el objeto des-serializado contiene todos los enlaces que estaban en el objeto original.

Observe que no se invoca ningún constructor, ni siquiera el constructor predeterminado, en el proceso de des-serialización de un objeto de tipo **Serializable**. El objeto completo se restaura recuperando los datos desde el flujo de entrada **InputStream**.

Ejercicio 27: (1) Cree una clase **Serializable** que contenga una referencia a un objeto de una segunda clase **Serializable**. Cree una instancia de esa clase, serialicela en disco, restáurela a continuación y verifique que el proceso ha funcionado correctamente.

Localización de la clase

Podriamos preguntarnos qué es lo que hace falta para poder recuperar un objeto a partir de su estado serializado. Por ejemplo, suponga que serializamos un objeto y lo enviamos como un archivo o lo mandamos a través de una red hacia otra máquina. ¿Podría un programa en la otra máquina reconstruir el objeto utilizando únicamente los contenidos del archivo?

La mejor forma de responder a esta cuestión es (como siempre) realizando un experimento. El siguiente archivo está contenido en el subdirectorio de este capítulo:

```
//: io/Alien.java
// Una clase serializable.
import java.io.*;
public class Alien implements Serializable {} //:-
```

El archivo que crea y serializa un objeto **Alien** está incluido en el mismo directorio:

```
//: io/FreezeAlien.java
// Crear un archivo de salida serializable.
import java.io.*;

public class FreezeAlien {
    public static void main(String[] args) throws Exception {
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("X.file"));
        Alien quellek = new Alien();
        out.writeObject(quellek);
    }
} //:-
```

En lugar de capturar y tratar las excepciones, este programa adopta la poco elegante solución de pasar las excepciones hacia fuera de **main()**, con lo que se informará de su existencia a través de la consola.

Una vez compilado y ejecutado, el programa genera un archivo denominado **X.file** en el directorio **io**. El siguiente código está incluido en un subdirectorio denominado **xfiles**:

```
//: io/xfiles/ThawAlien.java
// Tratar de recuperar un archivo serializado sin la
// clase del objeto que está almacenado en dicho archivo.
// {RunByHand}
import java.io.*;

public class ThawAlien {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(new File("../", "X.file")));
        Object mystery = in.readObject();
        System.out.println(mystery.getClass());
    }
} /* Output:
class Alien
*//:-
```

La simple operación consistente en abrir el archivo y leer el objeto **mystery** requiere disponer del objeto **Class** correspondiente a **Alien**; la máquina JVM no puede localizar **Alien.class** (a menos que se encuentre en la ruta de clases, lo que no debería ser el caso en este ejemplo). Por ello, obtenemos una excepción **ClassNotFoundException**. La máquina JVM debe ser capaz de encontrar el archivo **.class** asociado.

Control en la serialización

Como podemos ver, el mecanismo predeterminado de serialización es bastante sencillo de usar. Pero ¿qué sucede si tenemos necesidades especiales? Quizá, haya problemas especiales de seguridad y no queramos serializar ciertas partes del objeto.

to, o quizás no tiene sentido que uno de los subobjetos sea serializado si de todos modos hay que crear de nuevo ese subobjeto cuando recuperemos el objeto completo.

Podemos controlar el proceso de serialización implementando la interfaz **Externalizable** en lugar de la interfaz **Serializable**. La interfaz **Externalizable** amplía la interfaz **Serializable** y añade dos métodos, **writeExternal()** y **readExternal()**, que se invocan automáticamente para el objeto durante la serialización y la des-serialización, para poder realizar esas operaciones especiales que necesitamos.

El siguiente ejemplo muestra implementaciones simples de los métodos de la interfaz **Externalizable**. Observe que **Blip1** y **Blip2** son casi idénticos salvo por una sutil diferencia (trate de descubrirla examinando el código):

```
//: io/Blips.java
// Uso simple de Externalizable, junto con un problema.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
    public Blip1() {
        print("Blip1 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip1.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip1.readExternal");
    }
}

class Blip2 implements Externalizable {
    Blip2() {
        print("Blip2 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip2.readExternal");
    }
}

public class Blips {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        print("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Blips.out"));
        print("Saving objects:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        // Ahora obtenerlos de nuevo:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Blips.out"));
        print("Recovering b1:");
    }
}
```

```

        bi = (Blip1)in.readObject();
        // ;Ha fallado! Genera una excepción:
//! print("Recovering b2:");
//! b2 = (Blip2)in.readObject();
    }
} /* Output:
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
*///:-
```

La razón de que el objeto **Blip2** no se recupere es que, al tratar de hacerlo, se genera una excepción. ¿Puede ver la diferencia entre **Blip1** y **Blip2**? El constructor de **Blip1** es público, mientras que el constructor de **Blip2** no lo es, y eso es lo que provoca la excepción al intentar efectuar la recuperación. Pruebe a definir como público el constructor de **Blip2** y elimine los comentarios `//!` para ver los resultados correctos.

Cuando se recupera **b1**, se invoca el constructor predeterminado de **Blip1**. Esto difiere del proceso normal de recuperación del objeto **Serializable**, durante el cual el objeto se reconstruye enteramente a partir de los bits almacenados, sin efectuar ninguna llamada a un constructor. Con un objeto **Externalizable**, tienen lugar todas las tareas normales predeterminadas de construcción (incluyendo las inicializaciones en los puntos donde se definen los campos), *después de lo cual* se invoca **readExternal()**. Es necesario tener esto en cuenta (en particular el hecho de que tienen lugar todas las tareas predeterminadas de construcción), para poder obtener el comportamiento correcto de los objetos **Externalizable**.

He aquí un ejemplo que muestra qué es lo que hay que hacer para almacenar y recuperar un objeto **Externalizable**:

```

//: io/Blip3.java
// Reconstrucción de un objeto externalizable.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
    private int i;
    private String s; // Sin inicialización
    public Blip3() {
        print("Blip3 Constructor");
        // s, i no inicializados
    }
    public Blip3(String x, int a) {
        print("Blip3(String x, int a)");
        s = x;
        i = a;
        // s & i inicializados sólo en el constructor no predeterminado.
    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
    throws IOException {
        print("Blip3.writeExternal");
        // Hay que hacer esto:
        out.writeObject(s);
        out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
        print("Blip3.readExternal");
```

```

// Hay que hacer esto:
s = (String)in.readObject();
i = in.readInt();
}
public static void main(String[] args)
throws IOException, ClassNotFoundException {
print("Constructing objects:");
Blip3 b3 = new Blip3("A String ", 47);
print(b3);
ObjectOutputStream o = new ObjectOutputStream(
    new FileOutputStream("Blip3.out"));
print("Saving object:");
o.writeObject(b3);
o.close();
// Ahora extraer los datos:
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("Blip3.out"));
print("Recovering b3:");
b3 = (Blip3)in.readObject();
print(b3);
}
} /* Output:
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
*///:~

```

Los campos `s` e `i` sólo se inicializan en el segundo constructor, pero no en el constructor predeterminado. Esto quiere decir que si no inicializamos `s` e `i` en `readExternal()`, `s` será `null` e `i` será cero (ya que el espacio de almacenamiento del objeto se pone a cero en el primer paso de la creación del objeto). Si desactivamos mediante comentarios las dos líneas de código a continuación de las frases: "Hay que hacer esto:" y ejecutamos el programa, podremos ver que al recuperar el objeto `s` es `null` e `i` es cero.

Si estamos heredando de un objeto `Externalizable`, lo que haremos normalmente será invocar las versiones de la clase base de `writeExternal()` y `readExternal()`, para almacenar y recuperar apropiadamente los componentes de la clase base.

Para hacer que las cosas funcionen correctamente, no sólo hay que escribir los datos importantes de los datos del objeto durante el método `writeExternal()` (no hay ningún comportamiento predeterminado que escriba ninguno de los objetos miembro de un objeto `Externalizable`), sino que también hay que recuperar esos datos en el método `readExternal()`. Esto puede resultar confuso a primera vista, porque la realización de las tareas de construcción predeterminadas para un objeto `Externalizable` podrían hacer parecer que se está produciendo automáticamente algún tipo de operación de almacenamiento y recuperación, pero en realidad no es así.

Ejercicio 28: (2) En `Blips.java`, copie el archivo y renómbrelo como `BlipCheck.java`. Renombre también la clase `Blip2` como `BlipCheck` (haciéndola pública y eliminando el ámbito público de la clase `Blips` en el proceso). Elimine las marcas de comentario `!!` del archivo y ejecute el programa, incluyendo las líneas problemáticas. A continuación, desactive con un comentario el constructor predeterminado de `BlipCheck`. Ejecute el programa y explique por qué funciona. Observe que, después de la compilación, es necesario ejecutar el programa con "java `Blips`" porque el método `main()` sigue estando en la clase `Blips`.

Ejercicio 29: (2) En `Blip3.java`, desactive con comentarios las dos líneas situadas después de las frases: "Hay que hacer esto:" y ejecute el programa. Explique el resultado y las razones de que éste difiera de lo que sucede cuando las dos líneas forman parte del programa.

La palabra clave transient

Cuando estamos controlando la serialización, puede que exista un subobjeto concreto que no queramos que sea automáticamente guardado y restaurado por el mecanismo de serialización de Java. Esto suele suceder cuando dicho subobjeto representa información confidencial que no queramos serializar, como por ejemplo una contraseña. Incluso si esa información es de tipo **private** dentro del objeto, una vez que ha sido serializada resulta posible que alguien acceda a ella leyendo un archivo o interceptando una transmisión de red.

Una forma de evitar que las partes confidenciales del objeto sean serializadas consiste, como hemos visto previamente, en implementar la clase **Externalizable**. En ese caso, no hay nada que se serialice automáticamente y podemos serializar explícitamente sólo aquellas partes que sean necesarias dentro de **writeExternal()**.

Sin embargo, si estamos trabajando con un objeto de tipo **Serializable**, toda la tarea de serialización tiene lugar automáticamente. Para controlar esto, podemos desactivar la serialización campo a campo utilizando la palabra clave **transient**, que lo que viene a decir es: "No te preocupes de guardar o restaurar esto, yo me haré cargo de ello".

Por ejemplo, considere un objeto **Logon** que mantenga información acerca de un inicio de sesión concreto. Suponga que, una vez verificados los datos de inicio de sesión, queremos almacenar los datos, pero sin la contraseña. La forma más fácil de hacer esto es implementando **Serializable** y marcando el campo **password** como **transient**. He aquí un ejemplo:

```
//: io/Logon.java
// Ilustra la palabra clave "transient".
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    public Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
    public String toString() {
        return "logon info: \n  username: " + username +
            "\n  date: " + date + "\n  password: " + password;
    }
    public static void main(String[] args) throws Exception {
        Logon a = new Logon("Hulk", "myLittlePony");
        print("logon a = " + a);
        ObjectOutputStream o = new ObjectOutputStream(
            new FileOutputStream("Logon.out"));
        o.writeObject(a);
        o.close();
        TimeUnit.SECONDS.sleep(1); // Retardo
        // Ahora recuperar los datos:
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("Logon.out"));
        print("Recovering object at " + new Date());
        a = (Logon)in.readObject();
        print("logon a = " + a);
    }
} /* Output: (Sample)
logon a = logon info:
username: Hulk
date: Sat Nov 19 15:03:26 MST 2005
password: myLittlePony
```

```

Recovering object at Sat Nov 19 15:03:28 MST 2005
logon a = logon info:
username: Hulk
date: Sat Nov 19 15:03:26 MST 2005
password: null
*///:-
```

Podemos ver que los campos **date** y **username** son normales (no de tipo **transient**), por lo que se los serializa automáticamente. Sin embargo, el campo **password** es de tipo **transient**, así que no se almacena en disco; asimismo, el mecanismo de serialización no hace nada por intentar recuperarlo. Cuando se recupera el objeto, el campo **password** contiene el valor **null**. Observe que, mientras **toString()** está construyendo un objeto **String** utilizando el operador sobrecargado '+', las referencias **null** se convierten automáticamente en la cadena "null".

También puede ver que el campo **date** se almacena en disco y se recupera desde allí, no siendo generado de nuevo.

Puesto que los objetos **Externalizable** no almacenan ninguno de sus campos de manera predeterminada, la palabra clave **transient** es para ser usada únicamente por los objetos **Serializable**.

Una alternativa a Externalizable

Si no desea implementar la interfaz **Externalizable**, existe otra técnica alternativa. Puede implementar la interfaz **Serializable** y *añadir* (observe que decimos "añadir" y no "sustituir" o "implementar") sendos métodos denominados **writeObject()** y **readObject()** que se invocarán automáticamente cuando el objeto se serialice o des-serialice, respectivamente. En otras palabras, si proporcionamos estos otros métodos se usarán esos métodos en lugar del mecanismo predeterminado de serialización.

Los métodos deben tener estas signaturas exactas:

```

private void writeObject(ObjectOutputStream stream)
throws IOException;

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

Desde un punto de vista de diseño, las cosas pueden ser bastante complicadas si recurrimos a esta solución. En primer lugar, podemos pensar que como estos métodos no forman parte de una clase base ni de la interfaz **Serializable**, deberían ser definidos en sus propias interfaces. Pero observe que esos métodos están definidos como **private**, lo que significa que sólo los pueden invocar otros miembros de esta clase. Sin embargo, en realidad no se invocan desde otros miembros de esta clase, sino que son los métodos **writeObject()** y **readObject()** de los objetos **ObjectOutputStream** y **ObjectInputStream** los que se encargan de invocar a los métodos **writeObject()** y **readObject()** de nuestro objeto (observe cómo estoy conteniéndome para no entrar en una larga discusión acerca de lo inapropiado que resulta utilizar aquí los mismos nombres de métodos; por decirlo en pocas palabras: resulta enormemente confuso). Puede estar preguntándose cómo es posible que los objetos **ObjectOutputStream** y **ObjectInputStream** tengan acceso privado a métodos de nuestra clase. La única respuesta en la que podemos pensar es que esto forma parte de la magia de la serialización.⁶

Cualquier cosa que definamos en una interfaz es automáticamente de tipo **public**, por lo que si **writeObject()** y **readObject()** deben ser privados, eso quiere decir que no pueden formar parte de una interfaz. Puesto que queremos ajustarnos a las signaturas exactamente, el efecto es el mismo que si estuviéramos implementando una interfaz.

Cabe imaginar que, cuando invocamos **ObjectOutputStream.writeObject()**, el objeto de tipo **Serializable** que pasamos a ese método es interrogado (utilizando, sin duda, el mecanismo de reflexión) para ver si implementa su propio método **writeObject()**. En caso afirmativo, se omite el proceso normal de serialización y se invoca el método **writeObject()** personalizado. La misma situación se produce para el método **readObject()**.

Existe otra consideración adicional que debemos tener en cuenta. Dentro de nuestro método **writeObject()**, podemos decidir llevar a cabo la acción **writeObject()** predeterminada invocando **defaultWriteObject()**. De la misma forma, dentro de

⁶ La sección "Interfaces e información de tipos" al final del Capítulo 14, *Información de tipos*, muestra cómo es posible acceder a métodos privados desde fuera de la clase.

`readObject()` podemos invocar `defaultReadObject()`. He aquí un ejemplo simple en el que se ilustra cómo puede controlarse el almacenamiento y la recuperación de un objeto **Serializable**:

```
//: io/SerialCtl.java
// Control de la serialización añadiendo nuestros propios
// métodos writeObject() y readObject().
import java.io.*;

public class SerialCtl implements Serializable {
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    public String toString() { return a + "\n" + b; }
    private void writeObject(ObjectOutputStream stream)
        throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
    private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {
        stream.defaultReadObject();
        b = (String)stream.readObject();
    }
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        SerialCtl sc = new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        ByteArrayOutputStream buf= new ByteArrayOutputStream();
        ObjectOutputStream o = new ObjectOutputStream(buf);
        o.writeObject(sc);
        // Ahora recuperar los datos:
        ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(buf.toByteArray()));
        SerialCtl sc2 = (SerialCtl)in.readObject();
        System.out.println("After:\n" + sc2);
    }
} /* Output:
Before:
Not Transient: Test1
Transient: Test2
After:
Not Transient: Test1
Transient: Test2
*///:-
```

En este ejemplo, uno de los campos **String** es de tipo normal y el otro está definido como **transient**, para demostrar que el campo que no es de tipo **transient** es guardado por el método `defaultWriteObject()` mientras que el campo **transient** se guarda y restaura explícitamente. Los campos se inicializan dentro del constructor en lugar de en el punto de definición, para demostrar que no están siendo inicializados por ningún mecanismo de tipo automático durante la des-serialización.

Si utilizamos un mecanismo predeterminado para escribir las partes no transitorias (no marcadas como **transient**) del objeto, debemos invocar `defaultWriteObject()` como primera operación en `writeObject()`, y `defaultReadObject()` como primera operación en `readObject()`. Se trata de sendas llamadas a métodos que resultan un tanto extrañas. Podría parecer, por ejemplo, que estamos invocando `defaultWriteObject()` para un objeto **ObjectOutputStream** sin pasarle ningún argumento, a pesar de lo cual, ese método es capaz de averiguar la referencia a nuestro objeto y cómo escribir todas las partes no transitorias. Verdaderamente asombroso.

El almacenamiento y recuperación de los objetos **transient** utiliza un código más familiar. Y, sin embargo, examine atentamente lo que sucede: en **main()**, se crea un objeto **SerialCtl** y luego se serializa en un flujo **ObjectOutputStream** (observe en este caso que se utiliza un *buffer* en lugar de un archivo; para el objeto **ObjectOutputStream** no hay ninguna diferencia). La serialización tiene lugar en la linea:

```
o.writeObject(sc);
```

El método **writeObject()** debe examinar **se** para ver si dispone de su propio método **writeObject()** (no comprobando la interfaz, ya que no existe ninguna, ni el tipo de la clase, sino buscando realmente el método mediante el mecanismo de reflexión). Si el objeto dispone de ese método, lo utilizará. Para **readObject()** se utiliza una técnica similar. Quizá ésta fuera la única forma práctica con la que se podía resolver el problema, pero hay que reconocer que resulta un tanto extraña.

Versionado

Es posible que queramos modificar la versión de una clase serializable (por ejemplo, los objetos de la clase original podrían estar almacenados en una base de datos). Este tipo de mecanismo está soportado en el lenguaje, aunque lo más probable es que no tengamos que recurrir a él más que en casos especiales; el mecanismo requiere un análisis más en profundidad que no vamos a realizar aquí. Los documentos del JDK descargables en la dirección <http://java.sun.com> analizan este tema de forma bastante exhaustiva.

También podrá observar en la documentación del JDK muchos comentarios que comienzan con la advertencia:

los objetos serializados de una determinada clase no serán compatibles con las futuras versiones de Swing y que el soporte actual de serialización resulta apropiado para el almacenamiento a corto plazo o para la invocación RMI entre aplicaciones...

Esto se debe a que el mecanismo de versionado es demasiado sencillo como para funcionar de manera fiable en todas las situaciones, especialmente con JavaBeans. Los diseñadores del lenguaje están trabajando para corregir el diseño, y a eso es a lo que hace referencia la advertencia.

Utilización de la persistencia

Resulta bastante atractiva la posibilidad de utilizar la tecnología de serialización para almacenar parte del estado del programa, de modo que se pueda posteriormente restaurar con sencillez el programa y devolverlo a su estado actual. Pero, antes de poder hacer esto, es necesario que respondamos algunas cuestiones. ¿Qué sucede si serializamos dos objetos que tienen una referencia a un tercer objeto? Cuando se restauren esos dos objetos a partir de su estado serializado, ¿obtenemos una única instancia de un tercer objeto? ¿Qué sucede si serializamos los dos objetos en archivos separados y los des-serializamos en diferentes partes del programa?

He aquí un ejemplo que ilustra el problema:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
    private String name;
    private House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    public String toString() {
        return name + "[" + super.toString() +
            "], " + preferredHouse + "\n";
    }
}
```

```

public class MyWorld {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        House house = new House();
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("Bosco the dog", house));
        animals.add(new Animal("Ralph the hamster", house));
        animals.add(new Animal("Molly the cat", house));
        print("animals: " + animals);
        ByteArrayOutputStream buf1 =
            new ByteArrayOutputStream();
        ObjectOutputStream o1 = new ObjectOutputStream(buf1);
        o1.writeObject(animals);
        o1.writeObject(animals); // Escribir un segundo conjunto
        // Escribir en un flujo de datos diferente:
        ByteArrayOutputStream buf2 =
            new ByteArrayOutputStream();
        ObjectOutputStream o2 = new ObjectOutputStream(buf2);
        o2.writeObject(animals);
        // Ahora recuperar los datos:
        ObjectInputStream in1 = new ObjectInputStream(
            new ByteArrayInputStream(buf1.toByteArray()));
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(buf2.toByteArray()));
        List
        animals1 = (List) in1.readObject(),
        animals2 = (List) in1.readObject(),
        animals3 = (List) in2.readObject();
        print("animals1: " + animals1);
        print("animals2: " + animals2);
        print("animals3: " + animals3);
    }
} /* Output: (Sample)
animals: [Bosco the dog[Animal@addbf1], House@42e816
, Ralph the hamster[Animal@9304b1], House@42e816
, Molly the cat[Animal@190d11], House@42e816
]
animals1: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals2: [Bosco the dog[Animal@de6f34], House@156ee8e
, Ralph the hamster[Animal@47b480], House@156ee8e
, Molly the cat[Animal@19b49e6], House@156ee8e
]
animals3: [Bosco the dog[Animal@10d448], House@e0e1c6
, Ralph the hamster[Animal@6ca1d], House@e0e1c6
, Molly the cat[Animal@1bf216a], House@e0e1c6
]
*/

```

Un aspecto interesante del ejemplo es que resulta posible utilizar el mecanismo de serialización de objetos con una matriz de tipo **byte**, como forma de obtener una “copia profunda” de cualquier objeto de tipo **Serializable** (una copia profunda quiere decir que estamos duplicando la red completa de objetos, en lugar de sólo los objetos básicos y sus referencias). La copia de objetos se cubre en detalle en los suplementos en linea del libro.

Los objetos de tipo **Animal** contienen campos de tipo **House**. En **main()**, se crea una lista de estos objetos **Animal** y se la serializa dos veces en sendos flujos de datos. Cuando se des-serializan e imprimen esos flujos de datos, podemos ver un ejemplo de la salida que se obtendría (en cada ejecución las posiciones de memoria correspondientes a los objetos serán diferentes).

Por supuesto, lo que cabría esperar es que los objetos des-serializados tuvieran direcciones diferentes de las de sus originales. Pero observe que en **animals1** y **animals2** aparecen las mismas direcciones, incluyendo las referencias al objeto **House** que ambos comparten. Por otro lado, cuando se recupera **animals3**, el sistema no tiene forma de saber que los objetos de este otro flujo de datos son alias de los objetos del primer flujo de datos, así que construye una red de objetos completamente distinta.

Mientras estamos serializando todo en un único flujo de datos, recuperaremos la misma red de objetos que hayamos escrito, sin que se pueda producir ninguna duplicación accidental de los objetos. Por supuesto, podemos modificar el estado de los objetos en el tiempo que transcurre entre la escritura del primer objeto y del último, pero eso es nuestra responsabilidad; los objetos se escribirán en el estado en que se encuentren (y con cualesquiera conexiones que tengan con otros objetos) en el momento de serializarlos.

Lo más seguro, si queremos guardar el estado de un sistema, es hacer la serialización en forma de operación “atómica”. Si serializamos algunos objetos, realizamos otras tareas y luego serializamos más objetos, etc., no estaremos guardando el estado del sistema de una forma segura. En lugar de ello, incluya todos los objetos que forman parte del estado de su sistema en un único contenedor y escriba simplemente dicho contenedor como parte de una única operación. Entonces podrá restaurarlo también con una única llamada a método.

El siguiente ejemplo es un sistema imaginario de diseño asistido por computadora (CAD, *computer-aided design*) que ilustra la técnica descrita. Además, el ejemplo plantea la cuestión de los campos estáticos; si examina la documentación del JDK, podrá ver que **Class** es **Serializable**, así que debe ser sencillo almacenar los campos de tipo **static** serializando simplemente el objeto **Class**. En cualquier caso, parece una solución llena de sentido común.

```
//: io/StoreCADState.java
// Almacenamiento del estado de un supuesto sistema CAD.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
    public static final int RED = 1, BLUE = 2, GREEN = 3;
    private int xPos, yPos, dimension;
    private static Random rand = new Random(47);
    private static int counter = 0;
    public abstract void setColor(int newColor);
    public abstract int getColor();
    public Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass() +
            "color[" + getColor() + "] xPos[" + xPos +
            "] yPos[" + yPos + "] dim[" + dimension + "]\n";
    }
    public static Shape randomFactory() {
        int xVal = rand.nextInt(100);
        int yVal = rand.nextInt(100);
        int dim = rand.nextInt(100);
        switch(counter++ % 3) {
            default:
            case 0: return new Circle(xVal, yVal, dim);
            case 1: return new Square(xVal, yVal, dim);
            case 2: return new Line(xVal, yVal, dim);
        }
    }
}
class Circle extends Shape {
```

```

private static int color = RED;
public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
}
public void setColor(int newColor) { color = newColor; }
public int getColor() { return color; }
}

class Square extends Shape {
private static int color;
public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
}
public void setColor(int newColor) { color = newColor; }
public int getColor() { return color; }
}

class Line extends Shape {
private static int color = RED;
public static void
serializeStaticState(ObjectOutputStream os)
throws IOException { os.writeInt(color); }
public static void
deserializeStaticState(ObjectInputStream os)
throws IOException { color = os.readInt(); }
public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
}
public void setColor(int newColor) { color = newColor; }
public int getColor() { return color; }
}

public class StoreCADState {
public static void main(String[] args) throws Exception {
List<Class<? extends Shape>> shapeTypes =
    new ArrayList<Class<? extends Shape>>();
// Añadir referencias a los objetos class:
shapeTypes.add(Circle.class);
shapeTypes.add(Square.class);
shapeTypes.add(Line.class);
List<Shape> shapes = new ArrayList<Shape>();
// Construir algunas forma geométricas:
for(int i = 0; i < 10; i++)
    shapes.add(Shape.randomFactory());
// Configurar todos los colores estáticos como GREEN:
for(int i = 0; i < 10; i++)
    ((Shape)shapes.get(i)).setColor(Shape.GREEN);
// Guardar el vector de estado:
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("CADState.out"));
out.writeObject(shapeTypes);
Line.serializeStaticState(out);
out.writeObject(shapes);
// Mostrar las formas geométricas:
System.out.println(shapes);
}
} /* Output:

```

```
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
, class Squarecolor[3] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[3] xPos[7] yPos[88] dim[28]
, class Squarecolor[3] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[3] xPos[20] yPos[58] dim[16]
, class Squarecolor[3] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[3] xPos[75] yPos[10] dim[42]
]
*///:-
```

La clase **Shape** implementa **Serializable**, por lo que cualquier cosa que herede de **Shape** será también automáticamente de tipo **Serializable**. Cada objeto **Shape** contiene datos y cada clase derivada de **Shape** contiene un campo **static** que determina el color de todos esos tipos de objetos **Shape** (si insertáramos un campo estático en la clase base sólo tendríamos un campo, ya que los campos estáticos no se duplican en las clases derivadas). Los métodos de la clase base pueden ser sustituidos para configurar el color de los diferentes tipos (los métodos estáticos no se acoplan dinámicamente, así que son métodos normales). El método **randomFactory()** crea un objeto **Shape** diferente cada vez que se lo invoca, utilizando valores aleatorios como datos para el objeto **Shape**.

Circle y **Square** son extensiones sencillas de **Shape**; la única diferencia es que **Circle** inicializa **color** en el punto de definición, mientras que **Square** lo inicializa en el constructor. Dejaremos el análisis de **Line** para más adelante.

En **main()**, se utiliza un contenedor **ArrayList** para almacenar los objetos **Class** y el otro para almacenar las formas geométricas.

La recuperación de los objetos es bastante sencilla:

```
//: io/RecoverCADState.java
// Restauración del estado del supuesto sistema CAD.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("CADState.out"));
        // Leer en el mismo orden en que fueron escritos:
        List<Class<? extends Shape>> shapeTypes =
            (List<Class<? extends Shape>>)in.readObject();
        Line.deserializeStaticState(in);
        List<Shape> shapes = (List<Shape>)in.readObject();
        System.out.println(shapes);
    }
} /* Output:
[class Circlecolor[1] xPos[58] yPos[55] dim[93]
, class Squarecolor[0] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[1] xPos[7] yPos[88] dim[28]
, class Squarecolor[0] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[1] xPos[20] yPos[58] dim[16]
, class Squarecolor[0] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[1] xPos[75] yPos[10] dim[42]
]
*///:-
```

Puede ver que los valores de `xPos`, `yPos` y `dim` son almacenados y recuperados satisfactoriamente, pero existe algún problema con la recuperación de la información de tipo `static`. Todos los valores son “3” al entrar, pero al salir son distintos. Los objetos `Circle` tienen un valor de 1 (**RED**, que es la definición) y los objetos `Square` tienen un valor de 0 (recuerde que se inicializaban en el constructor). ¡Es como si los datos de tipo `static` no se hubieran serializado en absoluto! En efecto, así es: aún cuando la clase `Class` es **Serializable**, no hace lo que cabría esperar. Por tanto, si queremos serializar valores de tipo `static`, debemos hacerlo nosotros mismos.

Para esto es para lo que se utilizan los métodos `serializeStaticState()` y `deserializeStaticState()` de tipo `static` en `Line`. Como podemos ver, se los invoca explícitamente como parte del proceso de almacenamiento y de recuperación (observe que es necesario mantener el orden de escritura y lectura en el archivo de serialización). Por tanto, para hacer que estos programas funcionen correctamente es necesario:

1. Añadir sendos métodos `serializeStaticState()` y `deserializeStaticState()` a las clases.
2. Eliminar el contenedor `ArrayList shapeTypes` y todo el código relacionado con él.
3. Añadir llamadas a los nuevos métodos estáticos de serialización y des-serialización en las clases que representan a las distintas formas geométricas.

Otra cuestión en la que hay que pensar es la de la seguridad, ya que el mecanismo de serialización también guarda los datos de tipo `private`. Si tenemos problemas de seguridad, dichos campos deben marcarse como `transient`. Pero entonces, será necesario diseñar alguna forma segura de almacenar dicha información, para que cuando efectuemos una restauración, podamos reinicializar dichas variables de tipo `private`.

Ejercicio 30: (1) Corrija el programa `CADState.java` como se ha descrito en los párrafos anteriores.

XML

Una importante limitación de la serialización de objetos es que es una solución válida sólo para Java: sólo los programas Java pueden des-serializar sus objetos. Una solución más interoperable consiste en convertir los datos a formato XML, lo que permite que sean consumidos por una amplia variedad de plataformas y de lenguajes.

Debido a su popularidad, existe un número enormemente grande y confuso de opciones para programar con XML, incluyendo las bibliotecas `javax.xml.*` distribuidas con el JDK. Aquí, he decidido utilizar la biblioteca XOM de código abierto de Elliotte Rusty Harold (puede descargar los archivos y la documentación en www.xom.nu) porque parece ser la forma más simple y directa de generar y modificar código XML utilizando Java. Además, XOM pone un gran énfasis en garantizar la corrección del código XML.

Como ejemplo, suponga que tenemos objetos `Person` que contienen campos para representar el nombre y el apellido, los cuales queremos serializar mediante código XML. La siguiente clase `Person` tiene un método `getXML()` que utiliza XOM para convertir los datos `Person` en un objeto `Element` XML y un constructor que toma un objeto `Element` y extrae los datos `Person` apropiados (observe que los ejemplos XML están en su propio subdirectorío):

```
//: xml/Person.java
// Utilizar la biblioteca XOM para escribir y leer XML
// {Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
    private String first, last;
    public Person(String first, String last) {
        this.first = first;
        this.last = last;
    }
    // Generar un objeto Element XML a partir de este objeto Person.
    public Element getXML() {

```

```

Element person = new Element("person");
Element firstName = new Element("first");
firstName.appendChild(first);
Element lastName = new Element("last");
lastName.appendChild(last);
person.appendChild(firstName);
person.appendChild(lastName);
return person;
}
// Constructor para restaurar un objeto Person
// a partir de un objeto Element XML:
public Person(Element person) {
    first= person.getFirstChildElement("first").getValue();
    last = person.getFirstChildElement("last").getValue();
}
public String toString() { return first + " " + last; }
// Hacer que sea legible:
public static void
format(OutputStream os, Document doc) throws Exception {
    Serializer serializer= new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
}
public static void main(String[] args) throws Exception {
    List<Person> people = Arrays.asList(
        new Person("Dr. Bunsen", "Honeydew"),
        new Person("Gonzo", "The Great"),
        new Person("Phillip J.", "Fry"));
    System.out.println(people);
    Element root = new Element("people");
    for(Person p : people)
        root.appendChild(p.getXML());
    Document doc = new Document(root);
    format(System.out, doc);
    format(new BufferedOutputStream(new FileOutputStream(
        "People.xml")), doc);
}
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
<person>
<first>Dr. Bunsen</first>
<last>Honeydew</last>
</person>
<person>
<first>Gonzo</first>
<last>The Great</last>
</person>
<person>
<first>Phillip J.</first>
<last>Fry</last>
</person>
</people>
*///:~

```

Los métodos XOM son bastante auto-explicativos y puede encontrarlos en la documentación de XOM.

XOM también contiene una clase **Serializer** que, como podemos ver, se utiliza en el método **format()** para transformar el código XML a un formato más legible. Con invocar simplemente **toXML()** todo el sistema funciona, así que **Serializer** es una herramienta bastante útil.

Des-serializar los objetos **Person** a partir de un archivo XML también resulta sencillo:

```
//: xml/People.java
// (Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu )
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
    public People(String fileName) throws Exception {
        Document doc = new Builder().build(fileName);
        Elements elements =
            doc.getRootElement().getChildElements();
        for(int i = 0; i < elements.size(); i++)
            add(new Person(elements.get(i)));
    }
    public static void main(String[] args) throws Exception {
        People p = new People("People.xml");
        System.out.println(p);
    }
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*///:-
```

El constructor **People** abre y lee un archivo usando el método **Builder.build()** de XOM, y el método **getChildElements()** genera una lista **Elements** (no es un objeto **List** estándar de Java, sino un objeto que sólo tiene un método **size()** y un método **get()**, Harold no quería obligar a los programadores a utilizar Java SE5, pero seguía queriendo disponer de un contenedor que fuera seguro en lo que respecta a tipos). Cada objeto **Element** de esta lista representa un objeto **Person**, por lo que se lo entrega al segundo constructor de **Person**. Observe que esto requiere que conozcamos por adelantado la estructura exacta del archivo XML, pero ésta suele ser la norma en este tipo de problemas. Si la estructura no se ajusta a lo que esperamos, XOM generará una excepción. También podríamos escribir código más complejo que analizará el documento XML en lugar de hacer suposiciones acerca del mismo, para aquellos casos en los que tengamos una información menos concreta acerca de la estructura XML entrante.

Para que estos ejemplos puedan compilarse, es necesario incluir los archivos JAR de la distribución XOM dentro de nuestra ruta de clases.

Esto sólo es una breve introducción a la programación XML con Java y a la biblioteca XOM; para obtener más información, consulte www.xom.nu.

Ejercicio 31: (2) Añada una información de dirección postal a **Person.java** y **People.java**.

Ejercicio 32: (4) Utilizando un contenedor **Map<String, Integer>** y la utilidad **net.mindview.util.TextFile**, escriba un programa que cuente el número de apariciones de las distintas palabras en un archivo (utilice "**\W+**" como segundo argumento para el constructor **TextFile**). Almacene los resultados como un archivo XML.

Preferencias

La API *Preferences* está mucho más próxima a lo que son los mecanismos de persistencia que a los de serialización de objetos, porque se encarga de almacenar y recuperar automáticamente información. Sin embargo, su uso está restringido a conjuntos de datos limitados de pequeño tamaño: sólo se pueden almacenar primitivas de objetos **String**, y la longitud de cada objeto **String** no puede ser superior a 8K (no es un tamaño pequeño, pero tampoco nos permite construir ninguna aplicación seria). Como su propio nombre sugiere, la API *Preferences* está diseñada para almacenar y extraer preferencias de usuario y opciones de configuración de los programas.

Las preferencias son conjuntos de clave-valor (como los contenedores **Map**) que se almacenan en una jerarquía de nodos. Aunque la jerarquía de nodos puede utilizarse para crear estructuras complicadas, lo normal es crear un único nodo con el mismo nombre que nuestra clase y almacenar allí la información. He aquí un ejemplo simple:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PreferencesDemo {
    public static void main(String[] args) throws Exception {
        Preferences prefs = Preferences
            .userNodeForPackage(PreferencesDemo.class);
        prefs.put("Location", "Oz");
        prefs.put("Footwear", "Ruby Slippers");
        prefs.putInt("Companions", 4);
        prefs.putBoolean("Are there witches?", true);
        int usageCount = prefs.getInt("UsageCount", 0);
        usageCount++;
        prefs.putInt("UsageCount", usageCount);
        for(String key : prefs.keys())
            print(key + ": " + prefs.get(key, null));
        // Siempre hay que proporcionar un valor predeterminado:
        print("How many companions does Dorothy have? " +
            prefs.getInt("Companions", 0));
    }
} /* Output: (Sample)
Location: Oz
Footwear: Ruby Slippers
Companions: 4
Are there witches?: true
UsageCount: 53
How many companions does Dorothy have? 4
*///:-
```

Aquí, se utiliza **userNodeForPackage()**, pero también podríamos elegir **systemNodeForPackage()**; la elección es hasta cierto punto arbitraria; pero la idea es que “user” es para preferencias de los usuarios individuales, mientras que “system” es para las opciones generales de configuración de una instalación. Puesto que **main()** es de tipo **static**, se utiliza **PreferencesDemo.class** para utilizar el nodo, pero dentro de un método no estático, probablemente utilizariamos **getClass()**. No es necesario utilizar la clase actual como identificador del nodo, aunque esa es la práctica habitual.

Una vez creado el nodo, estará disponible para cargar o leer los datos. Este ejemplo carga el nodo con varios tipos de elementos y luego obtiene las claves con **keys()**. Las claves se devuelven como un objeto **String[]**, lo que puede resultar un tanto sorprendente si estamos acostumbrados a utilizar el método **keys()** de la biblioteca de colecciones. Observe el segundo argumento de **get()**: se trata del valor predeterminado que se genera si no existe ninguna entrada para dicho valor de clave. Mientras estamos realizando una iteración a través de un conjunto de claves, siempre sabemos si existe una entrada, así que resulta seguro utilizar **null** como valor predeterminado, pero lo normal es que estemos extrayendo una clave nombrada, como en:

```
prefs.getInt("Companions", 0);
```

En el caso normal, lo que conviene es proporcionar un valor predeterminado razonable. De hecho, podemos ver una estructura bastante típica en las líneas:

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

De esta forma, la primera vez que ejecutemos el programa, **UsageCount** tendrá el valor cero, pero en las subsiguientes invocaciones será distinto de cero.

Al ejecutar `PreferencesDemo.java`, podemos ver que, en efecto, `UsageCount` se incrementa cada vez que se ejecuta el programa, pero ¿dónde se almacenan los datos? No aparece ningún archivo local después de ejecutar el programa por primera vez. La API Preferences utiliza los recursos apropiados del sistema para llevar a cabo su tarea y estos recursos variarán dependiendo del sistema operativo. En Windows, se utiliza el Registro (puesto que éste es ya de por sí una jerarquía de nodos con parejas clave-valor). Pero lo importante es que la información se almacena de alguna manera mágica y transparente, de forma que no tenemos que preocuparnos de cómo funciona el mecanismo en un sistema o en otro.

Habrá mucho más que comentar acerca de la API Preferences, por lo que puede consultar la documentación del JDK, que resulta bastante comprensible para obtener más detalles.

Ejercicio 33: (2) Escriba un programa que muestre el valor actual de un directorio denominado “directorio base” y que pida que introduzcamos un nuevo valor. Utilice la API Preferences para almacenar el valor.

Resumen

La biblioteca de flujos de datos de E/S de Java satisface los requisitos básicos. Podemos efectuar lecturas y escrituras a través de la consola, un archivo, un bloque de memoria o incluso a través de Internet. Mediante el mecanismo de herencia podemos crear nuevos tipos de objetos de entrada y de salida. E incluso podemos añadir un mecanismo simple de ampliabilidad a los tipos de objetos que un flujo de datos puede aceptar, redefiniendo el método `toString()` que se invoca automáticamente cada vez que pasamos un objeto a un método que esté esperando un argumento de tipo `String` (la limitada “conversión de tipos automática” de Java).

Existen diversas cuestiones que la documentación y el diseño de la biblioteca de flujos de E/S dejan sin resolver. Por ejemplo, hubiera resultado muy conveniente que pudierámos especificar que se generara una excepción cada vez que tratáramos de sobreescibir un archivo a la hora de abrirlo para llevar a cabo una salida de datos, algunos sistemas de programación permiten especificar qué queremos abrir un archivo de salida, pero sólo si éste no existía anteriormente. En Java, parece que debemos utilizar un objeto `File` para determinar si existe un archivo, porque si lo abrimos como `OutputStream` o `Writer`, siempre será sobreescrito.

La biblioteca de flujos de E/S tiene sus ventajas y sus inconvenientes; se encarga de realizar una parte de la tarea y es una biblioteca portable. Pero si no estamos familiarizados con el patrón de diseño Decorador, el diseño de la biblioteca no resulta intuitivo, por lo que existe una cierta curva de aprendizaje y también requiere más esfuerzo a la hora de explicar el funcionamiento de la biblioteca a los que estén aprendiendo el lenguaje. Asimismo, se trata de una biblioteca incompleta; por ejemplo, no tendríamos por qué tener necesidad de escribir utilidades como `TextFile` (la nueva utilidad `PrintWriter` de Java SE5 representa un paso en la solución correcta, pero sólo se trata de una solución parcial). Se han efectuado grandes mejoras en Java SE5, añadiendo por ejemplo los mecanismos de formateo de salida que siempre han estado soportados en prácticamente todos los demás lenguajes.

Una vez que *comprendamos* el patrón de diseño Decorador y que *comencemos* a utilizar la biblioteca en aquellas situaciones donde haga falta la flexibilidad que ésta proporciona, comenzaremos a sacar provecho de su diseño, siendo esa ventaja suficiente para compensar las líneas de código adicionales que se requieren para incorporar esa funcionalidad.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Tipos enumerados

19

La palabra clave **enum** nos permite crear un nuevo tipo con un conjunto restringido de valores nominados, y tratar dichos valores como componentes normales del programa. Esta característica resulta ser enormemente útil.¹

Las enumeraciones se han introducido brevemente al final del Capítulo 5, *Inicialización y limpieza*. Sin embargo, ahora que comprendemos los temas más avanzados de Java, podemos realizar un análisis más detallado de la funcionalidad de la enumeración incluida en Java SE5. Como veremos, las enumeraciones nos permiten realizar cosas enormemente interesantes, aunque este capítulo también nos permitirá comprender mejor otras características del lenguaje que ya hemos presentado antes, como los genéricos y el mecanismo de reflexión. También hablaremos de unos cuantos patrones de diseño adicionales.

Características básicas de las enumeraciones

Como hemos visto en el Capítulo 5, *Inicialización y limpieza*, podemos recorrer la lista de constantes **enum** invocando el método **values()** para dicha enumeración. El método **values()** genera una matriz con las constantes **enum** en el orden en que fueran declaradas, de modo que podemos utilizar la matriz resultante en, por ejemplo, un bucle *foreach*.

Cuando se crea una enumeración, el compilador genera por nosotros una clase asociada. Esta clase hereda automáticamente de **java.lang.Enum**, lo que proporciona ciertas capacidades que se ilustran en el siguiente ejemplo:

```
//: enumerated/EnumClass.java
// Capacidades de la clase Enum
import static net.mindview.util.Print.*;

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
    public static void main(String[] args) {
        for(Shrubbery s : Shrubbery.values()) {
            print(s + " ordinal: " + s.ordinal());
            println(s.compareTo(Shrubbery.CRAWLING) + " ");
            println(s.equals(Shrubbery.CRAWLING) + " ");
            print(s == Shrubbery.CRAWLING);
            print(s.getDeclaringClass());
            print(s.name());
            print("-----");
        }
        // Generar un valor enum a partir de una cadena de caracteres:
        for(String s : "HANGING CRAWLING GROUND".split(" "))
            Shrubbery shrub = Enum.valueOf(Shrubbery.class, s);
            print(shrub);
    }
}
```

¹ Joshua Bloch me ha ayudado enormemente en el desarrollo de este capítulo.

```

        }
    } /* Output:
GROUND ordinal: 0
-1 false false
class Shrubbery
GROUND
-----
CRAWLING ordinal: 1
0 true true
class Shrubbery
CRAWLING
-----
HANGING ordinal: 2
1 false false
class Shrubbery
HANGING
-----
HANGING
CRAWLING
GROUND
*///:-
```

El método `ordinal()` genera un valor `int` que indica el orden de declaración de cada instancia `enum`, comenzando en cero. Siempre podemos comparar con seguridad instancias `enum` utilizando `==`, y los métodos `equals()` y `hashCode()` se crean de manera automática y transparente. La clase `Enum` es de tipo `Comparable`, por lo que existe un método `compareTo()`, que también es de tipo `Serializable`.

Si invocamos `getDeclaringClass()` para una instancia `enum`, podemos averiguar cuál es la clase `enum` que se utiliza como envoltorio.

El método `name()` devuelve el nombre tal como está declarado, y esto es lo que devuelve también el método `toString()`. El método `valueOf()` es un miembro estático de `Enum` y devuelve la instancia `enum` correspondiente al nombre (en forma de cadena de caracteres) que se le pase; si no se localiza ninguna correspondencia, se genera una excepción.

Utilización de importaciones estáticas con las enumeraciones

Analizamos una variante del programa `Burrito.java` del Capítulo 5, *Inicialización y limpieza*:

```

//: enumerated/Spiciness.java
package enumerated;

public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} ///:-

//: enumerated/Burrito.java
package enumerated;
import static enumerated.Spiciness.*;

public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree; }
    public String toString() { return "Burrito is " + degree; }
    public static void main(String[] args) {
        System.out.println(new Burrito(NOT));
        System.out.println(new Burrito(MEDIUM));
        System.out.println(new Burrito(HOT));
    }
} /* Output:
```

```
Burrito is NOT
Burrito is MEDIUM
Burrito is HOT
*///:-
```

La importación estática trae todos los identificadores de instancias **enum** al espacio de nombres local, así que no es necesario cualificarlos. ¿Se trata de una buena idea o sería mejor ser explícito y cualificar todas las instancias **enum**? La respuesta dependerá, probablemente, de nuestro código. El compilador no nos permitirá en ningún caso utilizar el tipo incorrecto, por lo que lo único que nos debe preocupar es si el código resultará confuso para el lector. En muchas situaciones, probablemente resulte adecuado eliminar las cualificaciones, pero es algo que habrá que evaluar caso por caso.

Observe que no es posible utilizar esta técnica si la enumeración está definida en el mismo archivo o en el paquete predefinido (al parecer, se produjeron algunas discusiones dentro de Sun sobre si debía permitir hacer esto).

Adición de métodos a una enumeración

Salvo por el hecho de que no podemos heredar de ella, una enumeración puede tratarse de forma bastante similar a las clases normales. Esto quiere decir que podemos añadir métodos a una enumeración. Resulta incluso posible que una enumeración disponga de un método **main()**.

Podemos, por ejemplo, generar para una enumeración, descripciones que difieran de la proporcionada por el método **toString()** predeterminado, que simplemente proporcione el nombre de esa instancia **enum**. Para hacer esto, debemos proporcionar el constructor con el fin de capturar información adicional y métodos adicionales que proporcionen una descripción ampliada, como en el ejemplo siguiente:

```
//: enumerated/OzWitch.java
// Las brujas en la tierra de Oz.
import static net.mindview.util.Print.*;
public enum OzWitch {
    // Las instancias deben definirse primero, antes de los métodos:
    WEST("Miss Gulch, aka the Wicked Witch of the West"),
    NORTH("Glinda, the Good Witch of the North"),
    EAST("Wicked Witch of the East, wearer of the Ruby " +
        "Slippers, crushed by Dorothy's house"),
    SOUTH("Good by inference, but missing");
    private String description;
    // El constructor debe tener acceso de paquete o privado:
    private OzWitch(String description) {
        this.description = description;
    }
    public String getDescription() { return description; }
    public static void main(String[] args) {
        for(OzWitch witch : OzWitch.values())
            print(witch + ": " + witch.getDescription());
    }
} /* Output:
WEST: Miss Gulch, aka the Wicked Witch of the West
NORTH: Glinda, the Good Witch of the North
EAST: Wicked Witch of the East, wearer of the Ruby
Slippers, crushed by Dorothy's house
SOUTH: Good by inference, but missing
*///:-
```

Observe que si vamos a definir métodos, tenemos que finalizar la secuencia de instancias **enum** con un carácter de punto y coma. Asimismo, Java nos obliga a definir las instancias en primer lugar dentro de la enumeración. Si tratamos de definirlas después de algunos de los métodos o campos, obtendremos un error en tiempo de compilación.

El constructor y los métodos tienen la misma forma que las de las clases normales, porque *se trata de una clase normal*, sólo que con algunas restricciones. Así que podemos hacer prácticamente todo lo que queramos con las enumeraciones (si bien lo más normal es que empleemos enumeraciones simples).

Aunque el constructor se ha definido en nuestro ejemplo como privado, no tiene demasiada importancia el tipo de acceso que usemos: el constructor sólo puede utilizarse para crear las instancias `enum` que se declaren dentro de la definición de la enumeración; el compilador no nos permitirá utilizarlo para crear una nueva instancia una vez que esté completada la definición de la enumeración.

Sustitución de los métodos de una enumeración

He aquí otra técnica para generar diferentes valores de cadena para las enumeraciones. En este caso, los métodos de las instancias son correctos, pero queremos reformatearlas de cara a su visualización. La sustitución del método `toString()` en una enumeración es igual que la sustitución en una clase normal:

```
//: enumerated/SpaceShip.java
public enum SpaceShip {
    SCOUT, CARGO, TRANSPORT, CRUISER, BATTLESHIP, MOTHERSHIP;
    public String toString() {
        String id = name();
        String lower = id.substring(1).toLowerCase();
        return id.charAt(0) + lower;
    }
    public static void main(String[] args) {
        for(SpaceShip s : values()) {
            System.out.println(s);
        }
    }
} /* Output:
Scout
Cargo
Transport
Cruiser
Battleship
Mothership
*///:-
```

El método `toString()` obtiene el nombre de la instancia `SpaceShip` invocando `name()`, y modificando el resultado de modo que sólo la primera letra esté en mayúscula.

Enumeraciones en las instrucciones switch

Una funcionalidad muy útil de las enumeraciones es la forma en que pueden utilizarse en las instrucciones `switch`. Normalmente, una instrucción `switch` sólo funciona con valores enteros, pero como las enumeraciones tienen un orden entero asociado y la posición de una instancia puede obtenerse mediante el método `ordinal()` (aparentemente esto es lo que hace el compilador), las enumeraciones pueden emplearse también dentro de las instrucciones `switch`.

Aunque normalmente es preciso cualificar cada instancia `enum` con su tipo, esto no es necesario dentro de una instrucción `case`. He aquí un ejemplo que emplea una enumeración para crear un pequeña máquina de estados:

```
//: enumerated/TrafficLight.java
// Enumeraciones en instrucciones switch.
import static net.mindview.util.Print.*;

// Define un tipo enum:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
    Signal color = Signal.RED;
    public void change() {
        switch(color) {
            // Observe que no hay por qué escribir Signal.RED
```

```

// dentro de la instrucción case:
case RED:    color = Signal.GREEN;
              break;
case GREEN:   color = Signal.YELLOW;
              break;
case YELLOW:  color = Signal.RED;
              break;
}
}

public String toString() {
    return "The traffic light is " + color;
}

public static void main(String[] args) {
    TrafficLight t = new TrafficLight();
    for(int i = 0; i < 7; i++) {
        print(t);
        t.change();
    }
}

} /* Output:
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
*///:-
```

El compilador no se queja de que no haya ninguna instrucción **default** dentro de la estructura **switch**, pero eso no se debe a que detecte que hay instrucciones **case** para cada instancia **Signal**. Si desactivamos una de las instrucciones **case** mediante un comentario, el compilador seguirá sin quejarse. Esto quiere decir que es necesario tener cuidado y comprobar explícitamente que todos los casos están cubiertos. Por otro lado, si estamos ejecutando la instrucción **return** dentro de las instrucciones **case**, el compilador *si se quejará* si no incluimos una opción **default**, incluso aunque hayamos cubierto todos los valores de la enumeración.

Ejercicio 1: (2) Utilice la importación estática para modificar **TrafficLight.java** de modo que no haya que cualificar las instancias **enum**.

El misterio de **values()**

Como hemos indicado anteriormente, el compilador se encarga de crear automáticamente todas las clases **enum** y esas clases heredan de la clase **Enum**. Sin embargo, si examinamos **Enum**, veremos que no hay ningún método **values()**, a pesar de que nosotros sí que lo hemos estado utilizando. ¿Existe algún otro método oculto? Podemos escribir un pequeño programa basado en el mecanismo de reflexión para averiguarlo:

```

//: enumerated/Reflection.java
// Análisis de enumeraciones utilizando el mecanismo de reflexión.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Explore { HERE, THERE }

public class Reflection {
    public static Set<String> analyze(Class<?> enumClass) {
        print("----- Analyzing " + enumClass + " -----");
        print("Interfaces:");

```

```

        for(Type t : enumClass.getGenericInterfaces())
            print(t);
        print("Base: " + enumClass.getSuperclass());
        print("Methods: ");
        Set<String> methods = new TreeSet<String>();
        for(Method m : enumClass.getMethods())
            methods.add(m.getName());
        print(methods);
        return methods;
    }
    public static void main(String[] args) {
        Set<String> exploreMethods = analyze(Explore.class);
        Set<String> enumMethods = analyze(Enum.class);
        print("Explore.containsAll(Enum)? " +
            exploreMethods.containsAll(enumMethods));
        println("Explore.removeAll(Enum): ");
        exploreMethods.removeAll(enumMethods);
        print(exploreMethods);
        // Descompilar el código para la enumeración:
        OSExecute.command("javap Explore");
    }
} /* Output:
----- Analyzing class Explore -----
Interfaces:
Base: class java.lang.Enum
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf, values, wait]
----- Analyzing class java.lang.Enum -----
Interfaces:
java.lang.Comparable<E>
interface java.io.Serializable
Base: class java.lang.Object
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum{
    public static final Explore HERE;
    public static final Explore THERE;
    public static final Explore[] values();
    public static Explore valueOf(java.lang.String);
    static {};
}
*/

```

Así que la respuesta es que `values()` es un método estático añadido por el compilador. Debemos ver que también se añade a `Explore` el método `valueOf()` dentro del proceso de creación de la enumeración. Esto resulta un tanto confuso, porque también existe un método `valueOf()` que forma parte de la clase `Enum`, pero dicho método tiene dos argumentos y el método añadido sólo dispone de uno. Sin embargo, la utilización del método `Set` sólo comprueba los nombres de los métodos y no las signaturas, por lo que después de invocar `Explore.removeAll(Enum)`, lo único que queda es `[values]`.

A la salida, podemos ver que `Explore` ha sido definido como `final` por el compilador, por lo que no podemos heredar de una enumeración. También existe una cláusula de inicialización estática, la cual podemos redefinir como veremos más tarde.

Gracias al mecanismo de borrado de tipos (descrito en el Capítulo 15, *Genéricos*), el descompilador no dispone de información completa acerca de `Enum`, por lo que muestra la clase base de `Explore` como una clase `Enum` simple, en lugar de `Enum<Explore>`.

Puesto que `values()` es un método estático insertado dentro de la definición de `enum` por el compilador, si generalizamos un tipo `enum` a `Enum`, el método `values()` no estará disponible. Observe, sin embargo, que existe un método `getEnumConstants()` en `Class`, por lo que incluso `values()` no forma parte de la interfaz `Enum`, podemos seguir obteniendo las instancias `enum` a través del objeto `Class`:

```
//: enumerated/UpcastEnum.java
// No hay método values() si generalizamos la enumeración

enum Search { HITHER, YON }

public class UpcastEnum {
    public static void main(String[] args) {
        Search[] vals = Search.values();
        Enum e = Search.HITHER; // Upcast
        // e.values(); // No hay método values() en Enum
        for(Enum en : e.getClass().getEnumConstants())
            System.out.println(en);
    }
} /* Output:
HITHER
YON
*///:-
```

Como `getEnumConstants()` es un método `Class`, podemos invocarlo para una clase que no tenga enumeraciones:

```
//: enumerated/NonEnum.java

public class NonEnum {
    public static void main(String[] args) {
        Class<Integer> intClass = Integer.class;
        try {
            for(Object en : intClass.getEnumConstants())
                System.out.println(en);
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
java.lang.NullPointerException
*///:-
```

Sin embargo, el método devuelve `null`, así que se generará una excepción si tratamos de utilizar el resultado.

Implementa, no hereda

Ya hemos dicho que todas las enumeraciones amplian `java.lang.Enum`. Puesto que Java no soporta la herencia múltiple, esto quiere decir que no se puede crear una enumeración mediante herencia:

```
enum NotPossible extends Pet { ... // No funciona}
```

Sin embargo, *si es posible* crear una enumeración que implemente una o más interfaces:

```
//: enumerated/cartoons/EnumImplementation.java
// Una enumeración puede implementar una interfaz
package enumerated.cartoons;
import java.util.*;
import net.mindview.util.*;

enum CartoonCharacter
    implements Generator<CartoonCharacter> {
    SLAPPY, SPANKY, PUNCHY, SILLY, BOUNCY, NUTTY, BOB;
```

```

private Random rand = new Random(47);
public CartoonCharacter next() {
    return values()[rand.nextInt(values().length)];
}
}

public class EnumImplementation {
    public static <T> void printNext(Generator<T> rg) {
        System.out.print(rg.next() + ", ");
    }
    public static void main(String[] args) {
        // Elegir cualquier instancia:
        CartoonCharacter cc = CartoonCharacter.BOB;
        for(int i = 0; i < 10; i++)
            printNext(cc);
    }
} /* Output:
BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY,
NUTTY, SLAPPY,
*///:-
```

El resultado es algo extraño, porque para llamar a un método es necesario tener una instancia de la enumeración para la cual invocarlo. Sin embargo, cualquier método que admite un objeto **Generator** podrá ahora aceptar una instancia **CartoonCharacter**; por ejemplo, **printNext()**.

Ejercicio 2: (2) En lugar de implementar una interfaz, defina **next()** como un método estático. ¿Cuáles son las ventajas y desventajas de esta solución?

Selección aleatoria

Muchos de los ejemplos de este capítulo requieren efectuar una selección aleatoria entre varias instancias **enum**, como vimos en **CartoonCharacter.next()**. Es posible generalizar esta tarea utilizando genéricos e incluir el resultado en la biblioteca común:

```

//: net/mindview/util/Enums.java
package net.mindview.util;
import java.util.*;

public class Enums {
    private static Random rand = new Random(47);
    public static <T extends Enum<T>> T random(Class<T> ec) {
        return random(ec.getEnumConstants());
    }
    public static <T> T random(T[] values) {
        return values[rand.nextInt(values.length)];
    }
} /*:-
```

La extraña sintaxis **<T extends Enum<T>>** describe **T** como una instancia **enum**. Pasando **Class<T>**, hacemos que esté disponible el objeto clase, pudiéndose así generar la matriz de instancia **enum**. El método **random()** sobrecargado sólo necesita conocer que se le está pasando un objeto **T[]**, porque no necesita realizar operaciones de la clase **Enum**; sólo necesita seleccionar aleatoriamente un elemento de una matriz. El tipo de retorno es el tipo exacto de la enumeración.

He aquí una prueba simple del método **random()**:

```

//: enumerated/RandomTest.java
import net.mindview.util.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
    RUNNING, DODGING, JUMPING, FALLING, FLYING }
```

```

public class RandomTest {
    public static void main(String[] args) {
        for(int i = 0; i < 20; i++)
            System.out.print(Enum.random(Activity.class) + " ");
    }
} /* Output:
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING
DODGING SITTING RUNNING HOPPING HOPPING HOPPING RUNNING
STANDING LYING FALLING RUNNING FLYING LYING
*///:-

```

Aunque **Enum** es una clase no demasiado compleja, ya veremos en este capítulo que permite ahorrarse muchas duplicaciones. Las duplicaciones tienden a generar errores, así que eliminar esas duplicaciones es un objetivo bastante importante.

Utilización de interfaces con propósitos de organización

La imposibilidad de heredar de una enumeración puede resultar un tanto frustrante en algunas ocasiones. La razón para tratar de heredar de una enumeración proviene en parte del deseo de aumentar el número de elementos de la enumeración original, y por otro lado del deseo de crear subcategorías empleando subtipos.

Podemos realizar la categorización agrupando los elementos dentro de una interfaz y creando una enumeración basada en esa interfaz. Por ejemplo, supongamos que tenemos diferentes clases de alimentos y queremos definirlas como enumeraciones, pero sin que por ello las distintas clases de alimentos dejen de ser un tipo de una clase denominada **Food**. He aquí un ejemplo:

```

//: enumerated/menu/Food.java
// Subcategorización de enumeraciones dentro de interfaces.
package enumerated.menu;

public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
        LASAGNE, BURRITO, PAD_THAI,
        LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
        TIRAMISU, GELATO, BLACK_FOREST_CAKE,
        FRUIT, CREME_Caramel;
    }
    enum Coffee implements Food {
        BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
        LATTE, CAPPUCCINO, TEA, HERB_TEAS;
    }
} //://:-

```

Puesto que el único mecanismo de subtipos disponible para una enumeración está basado en la implementación de interfaz, cada enumeración anidada implementa la interfaz envoltorio **Food**. Ahora sí que podemos decir que “todo es un tipo de **Food**”, como podemos ver aquí:

```

//: enumerated/menu/TypeOfFood.java
package enumerated.menu;
import static enumerated.menu.Food.*;

public class TypeOfFood {
    public static void main(String[] args) {
        Food food = Appetizer.SALAD;
        food = MainCourse.LASAGNE;
    }
}

```

```

        food = Dessert.GELATO;
        food = Coffee.CAPPUCCINO;
    }
} //:-~

```

La generalización a **Food** funciona para cada tipo **enum** que implementa **Food**, así que todos ellos son tipos de **Food**.

Sin embargo, una interfaz no resulta tan útil como una enumeración cuando queremos tratar con un conjunto de tipos. Si deseamos disponer de una “enumeración de enumeraciones” podemos crear una enumeración de nivel superior con una instancia para cada enumeración de **Food**:

```

//: enumerated/menu/Course.java
package enumerated.menu;
import net.mindview.util.*;

public enum Course {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Course(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
} //:-~

```

Cada una de las enumeraciones anteriores toma el correspondiente objeto **Class** como argumento del constructor, pudiendo extraer de él todas las instancias **enum** utilizando **getEnumConstants()** y almacenarlos. Estas instancias se utilizan posteriormente en **randomSelection()**, por lo que ahora podemos crear un menú generado aleatoriamente seleccionando el elemento de **Food** de cada plato (**Course**):

```

//: enumerated/menu/Meal.java
package enumerated.menu;

public class Meal {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Course course : Course.values()) {
                Food food = course.randomSelection();
                System.out.println(food);
            }
            System.out.println("----");
        }
    }
} /* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
---
SOUP
VINDALOO
FRUIT
TEA
---
SALAD
BURRITO
FRUIT

```

```

TEA
---
SALAD
BURRITO
CREME_CARAMEL
LATTE
---
SOUP
BURRITO
TIRAMISU
ESPRESSO
---
*///:-
```

En este caso, la ventaja de crear una enumeración de enumeraciones es que con ello podemos iterar a través de cada objeto **Course**. Posteriormente, en el ejemplo **VendingMachine.java**, veremos otra técnica de categorización que está basada en un conjunto diferente de restricciones.

Otra solución más compacta al problema de la categorización consiste en anidar enumeraciones dentro de otras enumeraciones, como en el ejemplo siguiente:

```

//: enumerated/SecurityCategory.java
// Una subcategorización más sucinta.
import net.mindview.util.*;

enum SecurityCategory {
    STOCK(Security.Stock.class), BOND(Security.Bond.class);
    Security[] values;
    SecurityCategory(Class<? extends Security> kind) {
        values = kind.getEnumConstants();
    }
    interface Security {
        enum Stock implements Security { SHORT, LONG, MARGIN }
        enum Bond implements Security { MUNICIPAL, JUNK }
    }
    public Security randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            SecurityCategory category =
                Enums.random(SecurityCategory.class);
            System.out.println(category + ": " +
                category.randomSelection());
        }
    }
} /* Output:
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
*///:-
```

La interfaz **Security** es necesaria para recopilar todas las enumeraciones dentro de un tipo común. Entonces, se realiza la categorización dentro de **SecurityCategory**.

Si utilizamos esta solución con el ejemplo **Food**, el resultado sería:

```
//: enumerated/menu/Meal2.java
package enumerated.menu;
import net.mindview.util.*;

public enum Meal2 {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Meal2(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public interface Food {
        enum Appetizer implements Food {
            SALAD, SOUP, SPRING_ROLLS;
        }
        enum MainCourse implements Food {
            LASAGNE, BURRITO, PAD_THAI,
            LENTILS, HUMMOUS, VINDALOO;
        }
        enum Dessert implements Food {
            TIRAMISU, GELATO, BLACK_FOREST_CAKE,
            FRUIT, CREME_CARAMEL;
        }
        enum Coffee implements Food {
            BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
            LATTE, CAPPUCCINO, TEA, HERB_TEAS;
        }
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Meal2 meal : Meal2.values()) {
                Food food = meal.randomSelection();
                System.out.println(food);
            }
            System.out.println("----");
        }
    }
} /* Misma salida que en Meal.java */
```

Al final, se trata simplemente de una reorganización del código, pero permite obtener una estructura más clara en algunos casos.

Ejercicio 3: (1) Añada un nuevo objeto **Course** a **Course.java** y demuestre que funciona en **Meal.java**.

Ejercicio 4: (1) Repita el ejemplo anterior para **Meal2.java**.

Ejercicio 5: (4) Modifique **control/VowelsAndConsonants.java** para que utilice tres tipos de **enum**: **VOWEL**, **SOMETIMES_A_VOWEL** y **CONSONANT**. El constructor **enum** debe admitir las distintas letras que describen cada categoría concreta de vocales y consonantes. *Consejo:* utilice **varargs** y recuerde que éstos crean automáticamente una matriz.

Ejercicio 6: (3) ¿Existe alguna ventaja especial en anidar **Appetizer**, **MainCourse**, **Dessert** y **Coffee** dentro de **Food** en lugar de definirlos como enumeraciones independientes que se limiten a implementar **Food**?

Utilización de EnumSet en lugar de indicadores

Un contenedor **Set** es una especie de colección que sólo permite añadir un ejemplar de cada tipo de objeto. Por supuesto, una enumeración requiere que todos sus miembros sean diferentes, por lo que podría parecer que tiene un comportamiento similar al de los conjuntos, pero como se puede añadir o eliminar elementos, las enumeraciones no resultan demasiado útiles como conjuntos. La clase **EnumSet** se ha añadido a Java SE5 para funcionar de manera conjunta con las enumeraciones, con el fin de crear un sustituto para los tradicionales "bits indicadores" basados en valores enteros. Dichos indicadores se emplean para reflejar algún tipo de información de activado-desactivado, pero al final terminamos manipulando bits en lugar de conceptos, por lo que es bastante común que escribamos código bastante confuso.

EnumSet está diseñada para maximizar la velocidad, ya que debe competir de manera efectiva con los bits indicadores (las operaciones de esta clase serán normalmente mucho más rápidas que las de **HashSet**). Internamente, esta clase está representada (en caso de que sea posible) por un único valor **long** que se trata como un vector de bits, así que resulta extremadamente rápida y eficiente. La ventaja es que con ella disponemos de una forma mucho más expresiva para indicar la presencia o ausencia de una característica binaria, sin necesidad de preocuparnos acerca del rendimiento del programa.

Los elementos de un conjunto **EnumSet** deben provenir de una única enumeración **enum**. Veamos un posible ejemplo donde se utiliza una enumeración de los lugares de un edificio donde hay instalado un sensor de alarma:

```
//: enumerated/AlarmPoints.java
package enumerated;

public enum AlarmPoints {
    STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
    OFFICE4, BATHROOM, UTILITY, KITCHEN
} /*:-
```

Queremos utilizar el conjunto **EnumSet** para controlar el estado de alarma de los sensores:

```
//: enumerated/EnumSets.java
// Operaciones con conjuntos EnumSet
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

public class EnumSets {
    public static void main(String[] args) {
        EnumSet<AlarmPoints> points =
            EnumSet.noneOf(AlarmPoints.class); // Conjunto vacío
        points.add(BATHROOM);
        print(points);
        points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points = EnumSet.allOf(AlarmPoints.class);
        points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points.removeAll(EnumSet.range(OFFICE1, OFFICE4));
        print(points);
        points = EnumSet.complementOf(points);
        print(points);
    }
} /* Output:
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM, UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4, KITCHEN]
*/*:-
```

Se utiliza una cláusula de importación estática para simplificar el uso de las constantes **enum**. Los nombres de los métodos son bastantes auto-explicativos, y puede encontrar detalles completos de los mismos en la documentación del JDK. Cuando examine esta documentación, podrá ver un detalle interesante: el método **of()** ha sido sobrecargado tanto con **varargs** como con métodos individuales que admiten entre dos y cinco argumentos explícitos. Esto es un indicio de la preocupación por el rendimiento que imperaba a la hora de diseñar la clase **EnumSet**, porque un único método **of()** usando **varargs** podría haber resuelto el problema, pero es ligeramente menos eficiente que si se dispone de argumentos explícitos. Así, si invocamos **of()** con entre dos y cinco argumentos se obtienen las llamadas a método explícitas (ligeramente más rápidas), pero si lo invocamos con un argumento o con más cinco, se obtiene la versión **varargs** de **of()**. Observe que, si lo invocamos con un argumento, el compilador no construye la matriz **varargs**, así que no existe ningún gasto de procesamiento adicional si se invoca dicha versión con un único argumento.

Los conjuntos **EnumSet** se construyen a partir de valores **long**, cada valor **long** tiene 64 bits y cada instancia **enum** requiere un bit para indicar la presencia o ausencia. Esto significa que podemos tener un conjunto **EnumSet** para una enumeración de hasta 64 elementos sin utilizar más que un único valor **long**. ¿Qué sucede si tenemos más de 64 elementos en la enumeración?

```
//: enumerated/BigEnumSet.java
import java.util.*;

public class BigEnumSet {
    enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
               A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
               A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
               A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
               A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
               A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
               A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
    public static void main(String[] args) {
        EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
        System.out.println(bigEnumSet);
    }
} /* Output:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12,
A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, A23, A24,
A25, A26, A27, A28, A29, A30, A31, A32, A33, A34, A35, A36,
A37, A38, A39, A40, A41, A42, A43, A44, A45, A46, A47, A48,
A49, A50, A51, A52, A53, A54, A55, A56, A57, A58, A59, A60,
A61, A62, A63, A64, A65, A66, A67, A68, A69, A70, A71, A72,
A73, A74, A75]
*///:-
```

La clase **EnumSet**, como podemos ver, no tiene ningún problema con las enumeraciones que tengan más de 64 elementos, por lo que cabe presumir que se añade otro valor **long** adicional cada vez que es necesario.

Ejercicio 7: (3) Localice el código fuente correspondiente a **EnumSet** y explique cómo funciona.

Utilización de **EnumMap**

Un mapa **EnumMap** es un tipo de mapa especializado que requiere que sus claves formen parte de la misma enumeración. Debido a las restricciones impuestas en las enumeraciones, un mapa **EnumMap** puede implementarse internamente como una matriz. Por tanto, son extremadamente rápidos, así que podemos utilizar libremente los mapas **EnumMap** para búsquedas basadas en enumeraciones.

Únicamente podemos invocar el método **put()** para aquellas claves que formen parte de nuestra enumeración, pero por lo demás la utilización es similar a la de los mapas ordinarios.

He aquí un ejemplo que ilustra el uso del patrón de diseño *basado en comandos*. Este patrón comienza con una interfaz que contiene (normalmente) un único método y crea múltiples implementaciones de dicho método, cada una con un comportamiento distinto. Basta con instalar los objetos **Command** y el programa se encargará de llamarlos cuando sea necesario:

```

//: enumerated/EnumMaps.java
// Fundamentos de los mapas EnumMap.
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class EnumMaps {
    public static void main(String[] args) {
        EnumMap<AlarmPoints,Command> em =
            new EnumMap<AlarmPoints,Command>(AlarmPoints.class);
        em.put(KITCHEN, new Command() {
            public void action() { print("Kitchen fire!"); }
        });
        em.put(BATHROOM, new Command() {
            public void action() { print("Bathroom alert!"); }
        });
        for(Map.Entry<AlarmPoints,Command> e : em.entrySet()) {
            printnb(e.getKey() + ": ");
            e.getValue().action();
        }
        try { // Si no hay ningún valor para una clave concreta:
            em.get(UTILITY).action();
        } catch(Exception e) {
            print(e);
        }
    }
} /* Output:
BATHROOM: Bathroom alert!
KITCHEN: Kitchen fire!
java.lang.NullPointerException
*///:-
```

Al igual que con `EnumSet`, el orden de los elementos en `EnumMap` está determinado por su orden de definición dentro de la enumeración `enum`.

La última parte de `main()` muestra que siempre hay una entrada de clave para cada una de las instancias de la enumeración, pero el valor será `null` a menos que hayamos invocado `put()` para dicha clave.

Una ventaja de un mapa `EnumMap` con respecto a los *métodos específicos de constante* (que se describen a continuación) es que el mapa `EnumMap` permite modificar los objetos que representan los valores, mientras que, como veremos, los métodos específicos de constante se fijan en tiempo de compilación.

Como veremos posteriormente en el capítulo, los mapas `EnumMaps` pueden utilizarse para tareas de *despacho múltiple* en aquellas situaciones en las que se disponga de múltiples tipos de enumeraciones que interaccionen entre sí.

Métodos específicos de constante

Las enumeraciones Java tienen una característica muy interesante que nos permite asignar a cada instancia `enum` un comportamiento distinto, creando métodos para cada una de ellas. Para hacer esto, definimos uno o más métodos abstractos como parte de la enumeración, y luego definimos los métodos para cada instancia `enum`. Por ejemplo:

```

//: enumerated/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
    DATE_TIME {
```

```

        String getInfo() {
            return
                DateFormat.getDateInstance().format(new Date());
        }
    },
CLASSPATH {
    String getInfo() {
        return System.getenv("CLASSPATH");
    }
},
VERSION {
    String getInfo() {
        return System.getProperty("java.version");
    }
};
abstract String getInfo();
public static void main(String[] args) {
    for(ConstantSpecificMethod csm : values())
        System.out.println(csm.getInfo());
}
/* (Execute to see output) *///:-
```

Podemos buscar e invocar los métodos a través de su instancia **enum** asociada. Esto se denomina a menudo *código conducido por tablas* (observe, en especial, la similitud con el patrón de diseño Comando mencionado anteriormente).

En la programación orientada a objetos, se asocia un comportamiento distinto con las diferentes clases. Dado que cada instancia de una enumeración puede tener su propio comportamiento mediante métodos específicos de constante, esto sugiere que cada instancia es un tipo de datos distinto. En el ejemplo anterior, cada instancia **enum** se trata como el “tipo base” **ConstantSpecificMethod**, pero obtenemos un comportamiento polimórfico con la llamada a método **getInfo()**.

Sin embargo, esa similitud no puede llevarse más lejos. No podemos tratar las instancias **enum** como si fueran tipos de clases:

```

//: enumerated/NotClasses.java
// {Exec: javap -c LikeClasses}
import static net.mindview.util.Print.*;

enum LikeClasses {
    WINKEN { void behavior() { print("Behavior1"); } },
    BLINKEN { void behavior() { print("Behavior2"); } },
    NOD { void behavior() { print("Behavior3"); } };
    abstract void behavior();
}

public class NotClasses {
    // void f1(LikeClasses.WINKEN instance) {} // No se puede
} /* Output:
Compiled from "NotClasses.java"
abstract class LikeClasses extends java.lang.Enum{
public static final LikeClasses WINKEN;

public static final LikeClasses BLINKEN;

public static final LikeClasses NOD;
*/
*///:-
```

En **f1()**, podemos ver que el compilador no permite utilizar una instancia **enum** como un tipo de clase, lo que tiene bastante sentido si consideramos que el código generado por el compilador: cada elemento **enum** es una instancia de tipo **static final** de **LikeClasses**.

Asimismo, como son estáticas, las instancias `enum` de las enumeraciones internas no se comportan como clases internas normales, no podemos acceder a los campos o métodos no estáticos de la clase externa.

Veamos un ejemplo más interesante, en el que se intenta representar un sistema de lavado de coches. A cada cliente se le da un menú de opciones para su lavado y cada opción lleva a cabo una acción diferente. Podemos asociar cada opción con un método específico de constante y emplear un conjunto `EnumSet` para almacenar las selecciones del cliente:

```
//: enumerated/CarWash.java
import java.util.*;
import static net.mindview.util.Print.*;

public class CarWash {
    public enum Cycle {
        UNDERBODY {
            void action() { print("Spraying the underbody"); }
        },
        WHEELWASH {
            void action() { print("Washing the wheels"); }
        },
        PREWASH {
            void action() { print("Loosening the dirt"); }
        },
        BASIC {
            void action() { print("The basic wash"); }
        },
        HOTWAX {
            void action() { print("Applying hot wax"); }
        },
        RINSE {
            void action() { print("Rinsing"); }
        },
        BLOWDRY {
            void action() { print("Blowing dry"); }
        };
        abstract void action();
    }
    EnumSet<Cycle> cycles =
        EnumSet.of(Cycle.BASIC, Cycle.RINSE);
    public void add(Cycle cycle) { cycles.add(cycle); }
    public void washCar() {
        for(Cycle c : cycles)
            c.action();
    }
    public String toString() { return cycles.toString(); }
    public static void main(String[] args) {
        CarWash wash = new CarWash();
        print(wash);
        wash.washCar();
        // El orden de adición no es importante:
        wash.add(Cycle.BLOWDRY);
        wash.add(Cycle.BLOWDRY); // Se ignoran los duplicados
        wash.add(Cycle.RINSE);
        wash.add(Cycle.HOTWAX);
        print(wash);
        wash.washCar();
    }
} /* Output:
[BASIC, RINSE]
The basic wash
Rinsing
```

```
[BASIC, HOTWAX, RINSE, BLOWDRY]
The basic wash
Applying hot wax
Rinsing
Blowing dry
*///:-
```

La sintaxis para definir un método específico constante es, en la práctica, la de una clase interna anónima, pero más succincta.

Este ejemplo muestra también otras características adicionales de los conjuntos `EnumSet`. Puesto que se trata de un conjunto, sólo permitirá almacenar un ejemplar de cada elemento, así que las llamadas duplicadas con `add()` con el mismo argumento serán ignoradas (esto tiene bastante sentido, ya que un bit sólo se puede “activar” una vez). Asimismo, el orden en el que añadimos las instancias `enum` no tiene importancia: el orden de salida está determinado por el orden de declaración dentro de la enumeración.

¿Es posible sustituir los métodos específicos de constante, en lugar de implementar un método abstracto? Si que es posible, como podemos ver aquí:

```
//: enumerated/OverrideConstantSpecific.java
import static net.mindview.util.Print.*;

public enum OverrideConstantSpecific {
    NUT, BOLT,
    WASHER {
        void f() { print("Overridden method"); }
    };
    void f() { print("default behavior"); }
    public static void main(String[] args) {
        for(OverrideConstantSpecific ocs : values()) {
            printnb(ocs + ": ");
            ocs.f();
        }
    }
} /* Output:
NUT: default behavior
BOLT: default behavior
WASHER: Overridden method
*///:-
```

Aunque las enumeraciones impiden utilizar ciertos tipos de estructuras sintácticas, en general lo que deberá hacer es experimentar con ellas como si tratara de clases normales.

Cadena de responsabilidad en las enumeraciones

En el patrón de diseño *Cadena de responsabilidad*, creamos una serie de diferentes formas de resolver un problema y las encadenamos. Cuando tiene lugar una solicitud se la pasa a través de la cadena hasta que se encuentra una de las soluciones que pueda gestionarla.

Podemos implementar fácilmente una Cadena de responsabilidad simple utilizando métodos específicos de constante. Considere un modelo de una oficina de correos, que trate de gestionar cada correo de la forma más general posible, y que tiene que continuar intentando gestionar cada envío postal hasta conseguirlo o hasta llegar a la conclusión de que no es posible entregarlo. Cada uno de los intentos puede considerarse como un tipo de *Estrategia* (otro patrón de diseño) y la lista completa forma una Cadena de responsabilidad.

Comenzaremos describiendo lo que es un envío postal. Todas las diferentes características de interés pueden expresarse utilizando enumeraciones. Puesto que los objetos `Mail` (que representan los envíos postales) se generarán aleatoriamente, la forma más fácil de reducir la probabilidad de que, por ejemplo, a un envío postal le corresponda un valor `YES` para `GeneralDelivery` (entrega de carácter general) consiste en entregar más instancias que no correspondan con el valor `YES`, así que las definiciones `enum` parecen un poco extrañas al principio.

Dentro de **Mail**, vemos el método **randomMail()**, que crea ejemplos aleatorios de envíos postales de prueba. El método **generator()** produce un objeto **Iterable** que utiliza **randomMail()** para generar una serie de objetos que representan envíos postales, generándose un objeto cada vez que se invoca **next()** a través del iterador. Esta estructura permite crear de manera sencilla un bucle *foreach* invocando **Mail.generator()**:

```
//: enumerated/PostOffice.java
// Modelado de una oficina de correos.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Mail {
    // Los valores NO disminuyen la probabilidad de la selección aleatoria:
    enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
    enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
    enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
    enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
    enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
    GeneralDelivery generalDelivery;
    Scannability scannability;
    Readability readability;
    Address address;
    ReturnAddress returnAddress;
    static long counter = 0;
    long id = counter++;
    public String toString() { return "Mail " + id; }
    public String details() {
        return toString() +
            ", General Delivery: " + generalDelivery +
            ", Address Scanability: " + scannability +
            ", Address Readability: " + readability +
            ", Address Address: " + address +
            ", Return address: " + returnAddress;
    }
    // Generar objeto Mail de prueba:
    public static Mail randomMail() {
        Mail m = new Mail();
        m.generalDelivery= Enums.random(GeneralDelivery.class);
        m.scannability = Enums.random(Scannability.class);
        m.readability = Enums.random(Readability.class);
        m.address = Enums.random(Address.class);
        m.returnAddress = Enums.random(ReturnAddress.class);
        return m;
    }
    public static Iterable<Mail> generator(final int count) {
        return new Iterable<Mail>() {
            int n = count;
            public Iterator<Mail> iterator() {
                return new Iterator<Mail>() {
                    public boolean hasNext() { return n-- > 0; }
                    public Mail next() { return randomMail(); }
                    public void remove() { // No implementado
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}
```

```

public class PostOffice {
    enum MailHandler {
        GENERAL_DELIVERY {
            boolean handle(Mail m) {
                switch(m.generalDelivery) {
                    case YES:
                        print("Using general delivery for " + m);
                        return true;
                    default: return false;
                }
            }
        },
        MACHINE_SCAN {
            boolean handle(Mail m) {
                switch(m.scannability) {
                    case UNSCANNABLE: return false;
                    default:
                        switch(m.address) {
                            case INCORRECT: return false;
                            default:
                                print("Delivering " + m + " automatically");
                                return true;
                        }
                }
            }
        },
        VISUAL_INSPECTION {
            boolean handle(Mail m) {
                switch(m.readability) {
                    case ILLEGIBLE: return false;
                    default:
                        switch(m.address) {
                            case INCORRECT: return false;
                            default:
                                print("Delivering " + m + " normally");
                                return true;
                        }
                }
            }
        },
        RETURN_TO_SENDER {
            boolean handle(Mail m) {
                switch(m.returnAddress) {
                    case MISSING: return false;
                    default:
                        print("Returning " + m + " to sender");
                        return true;
                }
            }
        };
        abstract boolean handle(Mail m);
    }
    static void handle(Mail m) {
        for(MailHandler handler : MailHandler.values())
            if(handler.handle(m))
                return;
        print(m + " is a dead letter");
    }
    public static void main(String[] args) {

```

```

for(Mail mail : Mail.generator(10)) {
    print(mail.details());
    handle(mail);
    print("*****");
}
} /* Output:
Mail 0, General Delivery: NO2, Address Scanability:
UNSCANNABLE, Address Readability: YES3, Address Address: OK1,
Return address: OK1
Delivering Mail 0 normally
*****
Mail 1, General Delivery: NO5, Address Scanability: YES3,
Address Readability: ILLEGIBLE, Address Address: OK5,
Return address: OK1
Delivering Mail 1 automatically
*****
Mail 2, General Delivery: YES, Address Scanability: YES3,
Address Readability: YES1, Address Address: OK1,
Return address: OK5
Using general delivery for Mail 2
*****
Mail 3, General Delivery: NO4, Address Scanability: YES3,
Address Readability: YES1, Address Address: INCORRECT,
Return address: OK4
Returning Mail 3 to sender
*****
Mail 4, General Delivery: NO4, Address Scanability:
UNSCANNABLE, Address Readability: YES1, Address Address: INCORRECT,
Return address: OK2
Returning Mail 4 to sender
*****
Mail 5, General Delivery: NO3, Address Scanability: YES1,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK2
Delivering Mail 5 automatically
*****
Mail 6, General Delivery: YES, Address Scanability: YES4,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK4
Using general delivery for Mail 6
*****
Mail 7, General Delivery: YES, Address Scanability: YES3,
Address Readability: YES4, Address Address: OK2,
Return address: MISSING
Using general delivery for Mail 7
*****
Mail 8, General Delivery: NO3, Address Scanability: YES1,
Address Readability: YES3, Address Address: INCORRECT,
Return address: MISSING
Mail 8 is a dead letter
*****
Mail 9, General Delivery: NO1, Address Scanability:
UNSCANNABLE, Address Readability: YES2, Address Address: OK1,
Return address: OK4
Delivering Mail 9 normally
*****
*///:-
```

La Cadena de responsabilidad se expresa en la enumeración `enum MailHandler`, y el orden de las definiciones `enum` determina el orden en el que se intentarán aplicar las diferentes estrategias para cada envío postal. Se intenta aplicar cada una de las estrategias por turnos hasta que una de ellas tiene éxito, o todas ellas fallan, en cuyo caso tendremos un envío postal que no podrá ser entregado.

Ejercicio 8: (6) Modifique `PostOffice.java` para incluir la capacidad de reenviar correo.

Ejercicio 9: (5) Modifique la clase `PostOffice` para que utilice un mapa `EnumMap`.

Proyecto:² Los lenguajes especializados como Prolog utilizan el *encadenamiento inverso* para resolver problemas como éste. Utilizando `PostOffice.java` como base, haga una investigación acerca de dichos lenguajes y desarrolle un programa que permita añadir fácilmente nuevas “reglas” al sistema.

Máquinas de estado con enumeraciones

Los tipos enumerados pueden ser ideales para crear *máquinas de estado*. Una máquina de estado puede encontrarse en un número finito de estados específicos. Normalmente, la máquina pasa de un estado al siguiente basándose en una entrada, pero también existen *estados transitorios*: la máquina sale de estos estados en cuanto se ha realizado la correspondiente tarea.

Existen ciertas entradas permitidas para cada estado y las diferentes entradas cambian el estado de la máquina a diferentes nuevos estados. Puesto que las enumeraciones reducen el conjunto de posibles casos, resultan muy útiles para enumerar los diferentes estados y entradas.

Cada estado tiene también normalmente algún tipo de salida asociada.

Una máquina expendedora es un buen ejemplo de máquina de estados. En primer lugar, definimos las entradas dentro de una enumeración:

```
//: enumerated/Input.java
package enumerated;
import java.util.*;

public enum Input {
    NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
    TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
    ABORT_TRANSACTION {
        public int amount() { // No permitir
            throw new RuntimeException("ABORT.amount()"); }
    },
    STOP { // Esta debe ser la última instancia.
        public int amount() { // No permitir
            throw new RuntimeException("SHUT_DOWN.amount()"); }
    };
    int value; // En centavos
    Input(int value) { this.value = value; }
    Input() {}
    int amount() { return value; } // En centavos
    static Random rand = new Random(47);
    public static Input randomSelection() {
        // No incluir STOP:
        return values()[rand.nextInt(values().length - 1)];
    }
} ///:-
```

² Los proyectos son sugerencias que pueden utilizarse, por ejemplo, como proyectos de fin de curso. Las soluciones a los proyectos no se incluyen en la Guía de soluciones.

Observe que dos de las entradas **Input** tienen una cantidad asociada, así que definimos el método **amount()** para representar la cantidad dentro de la interfaz. Sin embargo, resulta inapropiado para los otros dos tipos de **Input**, así que se generará una excepción si invocamos ese método. Aunque se trata de un diseño un poco extraño (definir un método en una interfaz y luego generar una excepción si se lo invoca para ciertas implementaciones), nos vemos obligados a utilizarlo debido a las restricciones de la enumeraciones.

El objeto **VendingMachine** (máquina expendedora) reaccionará a estas entradas categorizándolas primero mediante la enumeración **Category**, para poder conmutar mediante **switch** entre las diferentes categorías. Este ejemplo muestra cómo las enumeraciones consiguen que el código sea más claro y más fácil de gestionar:

```
//: enumerated/VendingMachine.java
// {Args: VendingMachineInput.txt}
package enumerated;
import java.util.*;
import net.mindview.util.*;
import static enumerated.Input.*;
import static net.mindview.util.Print.*;

enum Category {
    MONEY(NICEL, DIME, QUARTER, DOLLAR),
    ITEM_SELECTION(TOOTHPASTE, CHIPS, SODA, SOAP),
    QUIT_TRANSACTION(ABORT_TRANSACTION),
    SHUT_DOWN(STOP);
    private Input[] values;
    Category(Input... types) { values = types; }
    private static EnumMap<Input,Category> categories =
        new EnumMap<Input,Category>(Input.class);
    static {
        for(Category c : Category.class.getEnumConstants())
            for(Input type : c.values)
                categories.put(type, c);
    }
    public static Category categorize(Input input) {
        return categories.get(input);
    }
}

public class VendingMachine {
    private static State state = State.RESTING;
    private static int amount = 0;
    private static Input selection = null;
    enum StateDuration { TRANSIENT } // Enumeración de marcación
    enum State {
        RESTING {
            void next(Input input) {
                switch(Category.categorize(input)) {
                    case MONEY:
                        amount += input.amount();
                        state = ADDING_MONEY;
                        break;
                    case SHUT_DOWN:
                        state = TERMINAL;
                        default:
                }
            }
        },
        ADDING_MONEY {
            void next(Input input) {
                switch(Category.categorize(input)) {
```

```

        case MONEY:
            amount += input.amount();
            break;
        case ITEM_SELECTION:
            selection = input;
            if(amount < selection.amount())
                print("Insufficient money for " + selection);
            else state = DISPENSING;
            break;
        case QUIT_TRANSACTION:
            state = GIVING_CHANGE;
            break;
        case SHUT_DOWN:
            state = TERMINAL;
        default:
    }
},
DISPENSING(StateDuration.TRANSIENT) {
    void next() {
        print("here is your " + selection);
        amount -= selection.amount();
        state = GIVING_CHANGE;
    }
},
GIVING_CHANGE(StateDuration.TRANSIENT) {
    void next() {
        if(amount > 0) {
            print("Your change: " + amount);
            amount = 0;
        }
        state = RESTING;
    }
},
TERMINAL { void output() { print("Halted"); } };
private boolean isTransient = false;
State() {}
State(StateDuration trans) { isTransient = true; }
void next(Input input) {
    throw new RuntimeException("Only call " +
        "next(Input input) for non-transient states");
}
void next() {
    throw new RuntimeException("Only call next() for " +
        "StateDuration.TRANSIENT states");
}
void output() { print(amount); }
}
static void run(Generator<Input> gen) {
    while(state != State.TERMINAL) {
        state.next(gen.next());
        while(state.isTransient)
            state.next();
        state.output();
    }
}
public static void main(String[] args) {
    Generator<Input> gen = new RandomInputGenerator();
    if(args.length == 1)
}

```

```

        gen = new FileInputStreamGenerator(args[0]);
        run(gen);
    }

    // Comprobación básica de que todo está en orden:
    class RandomInputGenerator implements Generator<Input> {
        public Input next() { return Input.randomSelection(); }
    }

    // Crear objetos Input a partir de un archivo de cadenas separadas por ":":
    class FileInputStreamGenerator implements Generator<Input> {
        private Iterator<String> input;
        public FileInputStreamGenerator(String fileName) {
            input = new TextFile(fileName, ":").iterator();
        }
        public Input next() {
            if(!input.hasNext())
                return null;
            return Enum.valueOf(Input.class, input.next().trim());
        }
    } /* Output:
25
50
75
here is your CHIPS
0
100
200
here is your TOOTHPASTE
0
25
35
Your change: 35
0
25
35
Insufficient money for SODA
35
60
70
75
Insufficient money for SODA
75
Your change: 75
0
Halted
*///:-
```

Puesto que la selección entre las distintas instancias **enum** se suele realizar con una instrucción a **switch** (observe el esfuerzo adicional que ha hecho el lenguaje para que se pueda aplicar fácilmente una instrucción **switch** a las enumeraciones), una de las cuestiones más comunes que podemos preguntarnos a la hora de organizar múltiples enumeraciones es: “¿Qué es lo que quiero utilizar para la instrucción **switch**?”. Aquí, lo más fácil es proceder en sentido inverso a partir del objeto **VendingMachine** observando que en cada estado **State**, necesitamos comutar con **switch** según las categorías básicas de acciones de entrada: si se ha insertado dinero, si se ha seleccionado un elemento, si se ha abortado la transacción y si se ha apagado la máquina. Sin embargo, dentro de estas categorías tenemos diferentes tipos de monedas que pueden insertarse y diferentes alimentos que se pueden seleccionar. La enumeración **Category** agrupa los diferentes tipos de objetos **Input** de modo que el método **categorize()** puede producir el elemento apropiado **Category** dentro de una instrucción **switch**. Este método utiliza un mapa **EnumMap** para realizar las búsquedas de manera eficiente y segura.

Si estudiamos la clase **VendingMachine**, podemos ver cómo cada estado es diferente y responde de forma distinta a las entradas. Observe también los dos estados transitorios. En **run()** la máquina espera una entrada **Input** y no deja de pasar a través de los estados hasta que deja de estar en un estado transitorio.

El sistema **VendingMachine** puede probarse de dos formas, utilizando dos objetos **Generator** diferentes. El objeto **RandomInputGenerator** simplemente genera de forma continua una serie de entradas, exceptuando **SHUT_DOWN** que hace que se pare la máquina. Ejecutando este procedimiento durante un tiempo lo suficientemente largo, podemos comprobar que todo está en orden y verificar que la máquina no entrará en un estado incorrecto. El objeto **FileInputGenerator** toma un archivo que describe las entradas en forma textual, las convierte en instancias **enum** y crea objetos **Input**. He aquí un archivo de texto utilizado para producir la salida mostrada en el ejemplo:

```
// :1 enumerated/VendingMachineInput.txt
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
// :-
```

Una limitación de este diseño es que los campos de **VendingMachine** a los que acceden las instancias de la enumeración **enum** deben ser estáticos, lo que significa que sólo podemos tener una única instancia **VendingMachine**. Esto no tiene por qué ser un problema si pensamos en una implementación real (Java embebido), ya que lo normal es que sólo tengamos una aplicación por cada máquina.

Ejercicio 10: (7) Modifique la clase **VendingMachine** (únicamente) utilizando **EnumMap** de modo que un programa pueda disponer de múltiples instancias de **VendingMachine**.

Ejercicio 11: (7) En una máquina expendedora real, conviene poder añadir y modificar fácilmente el tipo de elementos expedidos, porque los límites impuestos a **Input** por una enumeración resultan poco prácticos (recuerde que las enumeraciones son para un conjunto restringido de tipos). Modifique **VendingMachine.java** para que los elementos expedidos estén representados por una clase en lugar de ser parte de **Input** e inicialice un contenedor **ArrayList** de estos objetos a partir de un archivo de texto (utilizando **net.mindview.util.TextFile**).

Proyecto:³ Diseñe la maquina expendedora utilizando funcionalidades de internacionalización, de tal modo que una máquina pueda fácilmente ser adoptada en todos los países.

Despacho múltiple

Cuando estamos tratando con múltiples tipos que interactúan entre sí, los programas pueden llegar a ser especialmente complejos. Por ejemplo, consideremos un sistema que analice sintácticamente y luego ejecute expresiones matemáticas. Nos interesaría poder decir **Number.plus(Number)**, **Number.multiply(Number)**, etc., donde **Number** es la clase base de una familia de objetos numéricos. Pero cuando decimos **a.plus(b)**, y no conocemos el tipo de exacto de **a** ó **b**, ¿cómo podemos hacer que interactúen apropiadamente?

La respuesta comienza con algo en lo que probablemente no haya pensado hasta el momento: Java únicamente realiza lo que se denomina *despacho simple*. En otras palabras, si estamos revisando una operación sobre más de un objeto cuyos tipos sean desconocidos, Java sólo puede invocar el mecanismo de acoplamiento dinámico para uno de esos tipos. Esto no resuelve el problema descrito aquí, por lo que terminamos detectando unos tipos manualmente e implementando, en la práctica, nuestro propio comportamiento de acoplamiento dinámico.

La solución se denomina *despacho múltiple* (en este caso, sólo hay dos despachos, por lo que el mecanismo se denomina *despacho doble*). El polimorfismo sólo puede tener lugar desde llamadas a métodos, por lo que si queremos realizar un despacho doble, deben existir dos llamadas a métodos: la primera para determinar el primer tipo desconocido y la segunda para

³ Los proyectos son sugerencias que pueden utilizarse, por ejemplo, como proyectos de fin de curso. Las soluciones a los proyectos no se incluyen en la Guía de soluciones.

determinar el segundo tipo desconocido. Con el despacho múltiple, debemos tener una llamada virtual para cada uno de los tipos: estamos trabajando con dos jerarquías de tipos diferentes que están interactuando, necesitaremos una llamada virtual en cada jerarquía. Generalmente, lo que hacemos es establecer una configuración tal que una única llamada a método produzca una llamada a método virtual, permitiendo determinar así más de un tipo a lo largo del proceso. Para obtener este efecto, necesitamos trabajar con más de un método. Hará falta una llamada a método para cada operación de despacho. Los métodos del siguiente ejemplo (que implementa el juego “piedra, papel, tijera”) se denominan `compete()` y `eval()` y ambos son miembros de un mismo tipo. Estos métodos producen uno de tres posibles resultados:⁴

```
//: enumerated/Outcome.java
package enumerated;
public enum Outcome { WIN, LOSE, DRAW } //:->

//: enumerated/RoShamBo1.java
// Ilustración del mecanismo de despacho múltiple.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

interface Item {
    Outcome compete(Item it);
    Outcome eval(Paper p);
    Outcome eval(Scissors s);
    Outcome eval(Rock r);
}

class Paper implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return DRAW; }
    public Outcome eval(Scissors s) { return WIN; }
    public Outcome eval(Rock r) { return LOSE; }
    public String toString() { return "Paper"; }
}

class Scissors implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return LOSE; }
    public Outcome eval(Scissors s) { return DRAW; }
    public Outcome eval(Rock r) { return WIN; }
    public String toString() { return "Scissors"; }
}

class Rock implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return WIN; }
    public Outcome eval(Scissors s) { return LOSE; }
    public Outcome eval(Rock r) { return DRAW; }
    public String toString() { return "Rock"; }
}

public class RoShamBo1 {
    static final int SIZE = 20;
    private static Random rand = new Random(47);
    public static Item newItem() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Scissors();
            case 1: return new Paper();
            case 2: return new Rock();
        }
    }
}
```

⁴ Este ejemplo ha estado utilizándose desde hace muchos años tanto en C++ como en Java (en *Thinking in Patterns*) en www.MindView.net, antes de aparecer, sin citar la fuente en un libro escrito por otros autores.

```

        case 1: return new Paper();
        case 2: return new Rock();
    }
}

public static void match(Item a, Item b) {
    System.out.println(
        a + " vs. " + b + ": " + a.compete(b));
}

public static void main(String[] args) {
    for(int i = 0; i < SIZE; i++)
        match(newItem(), newItem());
}
} /* Output:
Rock vs. Rock: DRAW
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Scissors vs. Scissors: DRAW
Scissors vs. Paper: WIN
Rock vs. Paper: LOSE
Paper vs. Paper: DRAW
Rock vs. Paper: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Rock vs. Scissors: WIN
Rock vs. Paper: LOSE
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
*///:-

```

Item es la interfaz para los tipos con los que se va a realizar el despacho múltiple. **RoShamBo1.match()** toma dos objetos **Item** e inicia el proceso de doble despacho llamando a la función **Item.compete()**. El mecanismo virtual determina el tipo de **a**, por lo que se activa dentro de la función **competete()** correspondiente al tipo concreto de **a**. La función **competete()** realiza el segundo despacho llamando a **eval()** con el tipo restante. Pasándose a sí mismo (**this**) como un argumento a **eval()** se genera una llamada a la función **eval()** sobrecargada, preservándose así la información de tipo correspondiente al primer despacho. Al completarse el segundo despacho, conocemos el tipo exacto de ambos objetos **Item**.

Preparar el mecanismo de despacho múltiple requiere de un montón de ceremonia, pero recuerde que la ventaja que se obtiene es la elegancia sintáctica que se consigue al realizar la llamada: en lugar de escribir un código muy complicado para determinar el tipo de uno o más objetos durante una llamada, simplemente decimos: “¡Vosotros dos, no me importa de qué tipo sois, pero interactuar apropiadamente entre vosotros!”. Sin embargo, asegúrese de que este tipo de elegancia sea importante para usted antes de embarcarse en programas que impliquen un despacho múltiple.

Cómo despachar con enumeraciones

Realizar una traducción directa de **RoShamBo1.java** a una solución basada en enumeraciones resulta problemático, porque las instancias **enum** no son tipos, así que los métodos **eval()** sobrecargados no funcionan: no se pueden utilizar instancias **enum** como argumentos de tipo. Sin embargo, existen distintas técnicas para implementar técnicas de despacho múltiple que permiten sacar provecho de las enumeraciones.

Una de sus técnicas utiliza un constructor para inicializar una instancia **enum** con una “fila” de resultados; contemplado en su conjunto esto produce una especie de tabla de búsqueda:

```
//: enumerated/RoShamBo2.java
```

```

// Comutación de una enumeración basándose en otra.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
    PAPER(DRAW, LOSE, WIN),
    SCISSORS(WIN, DRAW, LOSE),
    ROCK(LOSE, WIN, DRAW);
    private Outcome vPAPER, vSCISSORS, vROCK;
    RoShamBo2(Outcome paper, Outcome scissors, Outcome rock) {
        this.vPAPER = paper;
        this.vSCISSORS = scissors;
        this.vROCK = rock;
    }
    public Outcome compete(RoShamBo2 it) {
        switch(it) {
            default:
            case PAPER: return vPAPER;
            case SCISSORS: return vSCISSORS;
            case ROCK: return vROCK;
        }
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo2.class, 20);
    }
} /* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*///:-

```

Una vez que se han determinado ambos tipos en `compete()`, la única acción consiste en devolver el resultado con `Outcome`. Sin embargo, también podríamos invocar otro método, incluso (por ejemplo, a través de un objeto `Comando` que hubiera sido asignado en el constructor).

`RoShamBo2.java` es más pequeño y más sencillo que el ejemplo original por lo que también resulta más fácil de controlar. Observe que estamos todavía utilizando dos despachos para determinar el tipo de ambos objetos. En `RoShamBo1.java`, ambos despachos se realizaban mediante llamadas virtuales a métodos pero aquí, sólo se utiliza una llamada virtual a método en el primer despacho. El segundo despacho emplea una instrucción `switch`, pero esta solución es perfectamente válida porque la enumeración limita las opciones disponibles en la instrucción `switch`.

El código relativo a la enumeración ha sido separado para que pueda utilizarse en otros ejemplos. En primer lugar, la interfaz `Competitor` define un tipo que compite con otro `Competitor`:

```
//: enumerated/Competitor.java
// Comutación de una enumeración basándose en otra.
package enumerated;

public interface Competitor<T extends Competitor<T>> {
    Outcome compete(T competitor);
} //:-
```

A continuación, definimos dos métodos estáticos (se hacen estáticos para evitar tener que especificar el tipo del parámetro explícitamente). En primer lugar, `match()` invoca a `compete()` para uno de los objetos `Competitor` comparándolo con otro, y podemos ver que en este caso el parámetro de tipo sólo necesita ser `Competitor<T>`. Pero en `play()`, el parámetro de tipo tiene que ser tanto `Enum<T>` porque se lo utiliza en `Enums.random()` como `Competitor<T>` porque se le pasa a `match()`:

```
//: enumerated/RoShamBo.java
// Herramientas comunes para los ejemplos RoShamBo.
package enumerated;
import net.mindview.util.*;

public class RoShamBo {
    public static <T extends Competitor<T>>
    void match(T a, T b) {
        System.out.println(
            a + " vs. " + b + ": " + a.compete(b));
    }
    public static <T extends Enum<T> & Competitor<T>>
    void play(Class<T> rsbClass, int size) {
        for(int i = 0; i < size; i++)
            match(
                Enums.random(rsbClass), Enums.random(rsbClass));
    }
} //:-
```

El método `play()` no tiene un valor de retorno que implique el parámetro de tipo `T`, por lo que parece que podríamos utilizar comodines dentro del tipo `Class<T>` en lugar de emplear la descripción proporcionada en el ejemplo. Sin embargo, los comodines no pueden abarcar más de un tipo base, así que estamos obligados a usar la expresión anterior.

Utilización de métodos específicos de constante

Puesto que los métodos específicos de constante nos permiten proporcionar diferentes implementaciones de método para cada instancia `enum`, parece una solución perfecta para configurar un sistema de despacho múltiple. Pero aunque se las puede proporcionar de este modo diferentes comportamientos, las instancias `enum` no son tipos, por lo que no se las puede emplear como argumentos de tipo en las signaturas de los métodos. Lo mejor que podemos hacer en este caso es definir una instrucción `switch`:

```
//: enumerated/RoShamBo3.java
// Utilización de métodos específicos de constante.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {
    PAPER {
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default: // Para aplacar al compilador
                case PAPER: return DRAW;
                case SCISSORS: return LOSE;
                case ROCK: return WIN;
            }
        }
    }
```

```

},
SCISSORS {
    public Outcome compete(RoShamBo3 it) {
        switch(it) {
            default:
            case PAPER: return WIN;
            case SCISSORS: return DRAW;
            case ROCK: return LOSE;
        }
    }
},
ROCK {
    public Outcome compete(RoShamBo3 it) {
        switch(it) {
            default:
            case PAPER: return LOSE;
            case SCISSORS: return WIN;
            case ROCK: return DRAW;
        }
    }
};
public abstract Outcome compete(RoShamBo3 it);
public static void main(String[] args) {
    RoShamBo.play(RoShamBo3.class, 20);
}
} /* Misma salida que RoShamBo2.java */:-
```

Aunque este programa funciona y no resulta irrazonable, la solución de `RoShamBo2.java` parece requerir menos código a la hora de añadir un nuevo tipo, por lo que resulta más sencilla.

Sin embargo, `RoShamBo3.java` puede simplificarse y comprimirse:

```

//: enumerated/RoShamBo4.java
package enumerated;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
    ROCK {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(SCISSORS, opponent);
        }
    },
    SCISSORS {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(PAPER, opponent);
        }
    },
    PAPER {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(ROCK, opponent);
        }
    };
    Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
        return ((opponent == this) ? Outcome.DRAW
            : ((opponent == loser) ? Outcome.WIN
                : Outcome.LOSE));
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo4.class, 20);
    }
} /* Misma salida que RoShamBo2.java */:-
```

Aquí, el segundo despacho es realizado por la versión de dos argumentos de `compete()`, que realiza una secuencia de comparaciones y es, por tanto, similar a la acción de una instrucción `switch`. Este ejemplo es más pequeño, pero resulta un poco confuso. Para un sistema de mayor envergadura, esta confusión puede ser una desventaja.

Cómo despachar con mapas `EnumMap`

Es posible realizar un “verdadero” despacho doble utilizando la clase `EnumMap`, que está diseñada específicamente para trabajar con las enumeraciones. Puesto que el objetivo es conmutar entre dos tipos desconocidos, podemos usar un mapa `EnumMap` de mapas `EnumMap` para realizar el doble despacho:

```
//: enumerated/RoShamBo5.java
// Despacho múltiple usando un mapa EnumMap de mapas EnumMaps.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
    PAPER, SCISSORS, ROCK;
    static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>>
        table = new EnumMap<RoShamBo5,
            EnumMap<RoShamBo5,Outcome>>(RoShamBo5.class);
    static {
        for(RoShamBo5 it : RoShamBo5.values())
            table.put(it,
                new EnumMap<RoShamBo5,Outcome>(RoShamBo5.class));
        initRow(PAPER, DRAW, LOSE, WIN);
        initRow(SCISSORS, WIN, DRAW, LOSE);
        initRow(ROCK, LOSE, WIN, DRAW);
    }
    static void initRow(RoShamBo5 it,
        Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
        EnumMap<RoShamBo5,Outcome> row =
            RoShamBo5.table.get(it);
        row.put(RoShamBo5.PAPER, vPAPER);
        row.put(RoShamBo5.SCISSORS, vSCISSORS);
        row.put(RoShamBo5.ROCK, vROCK);
    }
    public Outcome compete(RoShamBo5 it) {
        return table.get(this).get(it);
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo5.class, 20);
    }
} /* Misma salida que RoShamBo2.java */://:~
```

El mapa `EnumMap` se inicializa mediante una cláusula `static`; podemos ver la estructura con forma de tabla de llamadas a `initRow()`. Observe el método `compete()`, donde puede verse que ambos despachos tienen lugar en una única instrucción.

Utilización de una matriz 2-D

Podemos simplificar la solución aún más dándonos cuenta de que cada instancia `enum` tiene un valor fijo (basado en su orden de declaración) y que el método `ordinal()` genera este valor. Una matriz bidimensional que asigne los competidores a los distintos resultados, permite obtener la solución más pequeña y directa (y también posiblemente la más rápida aunque recuerde que `EnumMap` utiliza una matriz interna):

```
//: enumerated/RoShamBo6.java
// Enumeraciones utilizando "tablas" en lugar de despacho múltiple.
package enumerated;
```

```

import static enumerated.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
    PAPER, SCISSORS, ROCK;
    private static Outcome[][] table = {
        { DRAW, LOSE, WIN }, // PAPER
        { WIN, DRAW, LOSE }, // SCISSORS
        { LOSE, WIN, DRAW } // ROCK
    };
    public Outcome compete(RoShamBo6 other) {
        return table[this.ordinal()][other.ordinal()];
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo6.class, 20);
    }
} //:-)

```

La tabla `table` tiene exactamente el mismo orden que las llamadas a `initRow()` en el ejemplo anterior.

El tamaño pequeño de este código resulta muy atractivo si lo comparamos con los ejemplos anteriores, en parte porque parece mucho más fácil de entender y de modificar pero también porque parece una solución mucho más directa. Sin embargo, no es una solución tan “segura” como los ejemplos anteriores, porque utiliza una matriz. Con una matriz de mayor tamaño, podríamos obtener el tamaño inapropiado y, si las pruebas no cubrieran todas las posibilidades, algún error podría terminar por deslizarse.

Todas estas soluciones representan diferentes tipos de tablas, pero merece la pena explorar la forma de expresar cada una de las tablas para localizar la que mejor se ajuste a nuestras necesidades. Observe que, aunque la solución anterior es la más compacta, también resulta bastante rígida, porque sólo permite producir una salida constante a partir de unas entradas constantes. Sin embargo, no hay nada que nos impida tener una tabla que genera un objeto de función. Para ciertos tipos de problemas, el concepto de “código conducido por tablas” puede ser muy potente.

Resumen

Aún cuando los tipos enumerados no son terriblemente complejos por sí mismos, hemos pospuesto este capítulo hasta este momento debido a que queríamos analizar lo que se puede hacer con las enumeraciones al combinarlas con características tales como el polimorfismo, los genéricos y el mecanismo de reflexión.

Aunque son significativamente más sofisticadas que las enumeraciones de C o C++, las enumeraciones Java siguen siendo una característica menor, algo sin lo que el lenguaje ha sobrevivido (aunque a costa de una gran complejidad) durante muchos años. A pesar de ello, este capítulo muestra el valor que una característica “menor” puede tener. En ocasiones nos proporciona el mecanismo adecuado para resolver un problema de manera elegante y clara y, como hemos dicho en diversas ocasiones a lo largo del libro, la elegancia es importante y la claridad puede marcar la diferencia entre una solución adecuada y otra que fracasa porque las demás personas son incapaces de entenderla.

Hablando de claridad, una fuente muy lamentable de confusión proviene de la mala decisión que se tomó en Java 1.0, consistente en emplear el término “enumeration” en lugar del término más común y ampliamente aceptado de “iterator” para referirse a un objeto que selecciona cada elemento de una secuencia (como hemos mencionado al hablar de las colecciones). Algunos lenguajes hacen incluso referencias a los tipos de datos enumerados utilizando la palabra “enumerators”. Este error se ha rectificado desde entonces en Java, pero la interfaz `Enumeration` no pudo, por supuesto, eliminarse de manera directa, por lo que sigue estando presente en el código antiguo (¡y a veces en el código nuevo!), en la biblioteca y en la documentación.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Anotaciones

20

Las anotaciones (también conocidas como *metadatos*) proporcionan una manera formalizada de añadir información a nuestro código que nos permita utilizar fácilmente dichos datos en algún momento posterior.¹

Las anotaciones están en parte motivadas por la tendencia general existente hacia combinar metadatos con los archivos de código fuente en lugar de mantenerlos en documentos externos. También son una respuesta a las presiones provenientes de otros lenguajes como C# en el sentido de añadir más características al lenguaje.

Las anotaciones son uno de los cambios fundamentales del lenguaje introducidos en Java SE5. Proporcionan información que hace falta para describir completamente el programa, pero que no puede expresarse en Java. De este modo, las anotaciones nos permiten almacenar información adicional en un formato que es probado y verificado por el compilador. Pueden utilizarse anotaciones para generar archivos descriptores o incluso nuevas definiciones de clases y para ayudar a facilitar la tarea de escribir plantillas de código. Utilizando anotaciones podemos mantener estos metadatos en el código fuente Java y conseguir con ello un código de aspecto más limpio, un mecanismo de comprobación de tipos de compilación y la API anotaciones como ayuda para construir herramientas de procesamiento de las anotaciones. Aunque hay unos cuantos tipos de metadatos predefinidos en Java SE5, en general, el tipo de anotaciones que se añaden y lo que hagamos con ellas son responsabilidad completamente nuestra.

La sintaxis de las anotaciones es razonablemente simple y consiste principalmente en la adición del símbolo @ al lenguaje. Java SE5 contiene tres anotaciones predefinidas de propósito general, que están definidas en `java.lang`:

- **@Override**, para indicar que la definición de un método pretende sustituir otro método de la clase base. Esta anotación genera un error de compilación si se escribe mal accidentalmente el nombre del método o si se proporciona una firma incorrecta.²
- **@Deprecated**, para que se genere una advertencia del compilador si se utiliza este elemento.
- **@SuppressWarnings**, para desactivar las advertencias de compilador inapropiadas. Esta anotación está permitida, pero no soportada como en las versiones primeras de Java SE5 (la anotación era ignorada).

Otros cuatro tipos adicionales de anotación soportan la creación de nuevas anotaciones, aprenderemos acerca de estos tipos en este capítulo.

Cada vez que creamos clases descriptoras o interfaces que impliquen una tarea repetitiva, normalmente utilizaremos anotaciones para automatizar y simplificar el proceso. Buena parte del trabajo adicional en *Enterprise JavaBeans* (EJB), por ejemplo, se elimina mediante el uso de anotaciones en EJB3.0.

Las anotaciones permiten sustituir sistemas existentes como XDoclet, que es una herramienta independiente para *doclets* (consulte el suplemento contenido en <http://MindView.net/Books/BetterJava>) diseñado específicamente para crear *doclets*.

¹ Jeremy Meyer tuvo la gentileza de acudir a Crested Butte y pasar allí dos semanas trabajando conmigo en este capítulo. Su ayuda ha sido extremadamente valiosa.

² Esto está, sin ninguna duda, inspirado en otra característica similar disponible en C#. La característica de C# es una palabra clave y no una anotación, y está impuesta por el compilador. En otras palabras, si se sobrescribe un método en C#, es necesario utilizar la palabra clave `override`, mientras que en Java la anotación `@Override` es opcional.

con estilo de anotación. Por contraste, las anotaciones son verdaderas estructuras del lenguaje y están, por tanto, estructuradas, comprobándose sus tipos en tiempo de compilación. Al mantener toda la información en el propio código fuente y no en comentarios, el código resulta más limpio y fácil de mantener. Utilizando y ampliando la API y las herramientas de anotaciones, o empleando bibliotecas externas de manipulación del código intermedio como veremos en este capítulo, podemos realizar potentes tareas de inspección y manipulación del código fuente y del código intermedio.

Sintaxis básica

En el ejemplo siguiente, el método `testExecute()` está anotado con `@Test`. Esto no hace nada por sí mismo pero el compilador comprobará que existe una definición de la anotación `@Test` en la ruta de construcción del programa. Como veremos posteriormente en el capítulo, podemos crear una herramienta que ejecute este método por nosotros a través del mecanismo de reflexión.

```
//: annotations/Testable.java
package annotations;
import net.mindview.junit.*;

public class Testable {
    public void execute() {
        System.out.println("Executing..");
    }
    @Test void testExecute() { execute(); }
} //:-
```

Los métodos anotados no difieren de los demás métodos. La anotación `@Test` de este ejemplo puede utilizarse en combinación con cualquiera de los modificadores como `public`, `static` o `void`. Sintácticamente, las anotaciones se utilizan de forma similar a los modificadores.

Definición de anotaciones

He aquí la definición de la anotación anterior. Podemos ver que las definiciones de anotaciones se parecen bastante a las definiciones de interfaz. De hecho, se compilan en archivos de clase, al igual que cualquier otra interfaz Java:

```
//: net/mindview/junit/Test.java
// El marcador @Test.
package net.mindview.junit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} //:-
```

Aparte del símbolo `@`, la definición de `@Test` se parece bastante a la de una interfaz vacía. La definición de una anotación también requiere las *meta-anotaciones* `@Target` y `@Retention`. `@Target` define dónde se puede aplicar esta anotación (un método o un campo) `@Retention` define si las anotaciones estarán disponibles en el código fuente (**SOURCE**), en los archivos de clase (**CLASS**), o en tiempo de ejecución (**RUNTIME**).

Las anotaciones contendrán usualmente *elementos* para especificar valores para las anotaciones. Un programa o una herramienta pueden utilizar estos parámetros para procesar las anotaciones. Los elementos se asemejan a los métodos de una interfaz, excepto porque no se pueden declarar valores predeterminados.

Una anotación sin ningún elemento, como la anotación `@Test` anterior, se denomina *anotación marcadora*.

He aquí una anotación simple que controla los casos de uso en un proyecto. Los programadores anotan cada método o conjunto de métodos que satisfacen los requisitos de un caso de uso concreto. El jefe de proyecto puede hacerse una idea del progreso del proyecto contando los casos de uso implementados y los desarrolladores encargados de mantener el proyecto pueden encontrar fácilmente los casos de uso si necesitan actualizar o depurar las reglas de negocio utilizadas en el sistema.

```
//: annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description";
} //:-
```

Observe que **id** y **description** se asemejan a declaraciones de métodos. Puesto que el tipo de **id** es comprobado por el compilador, se trata de una forma fiable de enlazar una base de datos de control con el documento de casos de uso y el código fuente. El elemento **description** tiene un valor predeterminado que es seleccionado por el procesador de anotaciones si no se especifica ningún valor en el momento de anotar un método.

He aquí una clase con tres métodos anotados como casos de uso del programa:

```
//: annotations>PasswordUtils.java
import java.util.*;

public class PasswordUtils {
    @UseCase(id = 47, description =
    "Passwords must contain at least one numeric")
    public boolean validatePassword(String password) {
        return (password.matches("\\w*\\d\\w*"));
    }
    @UseCase(id = 48)
    public String encryptPassword(String password) {
        return new StringBuilder(password).reverse().toString();
    }
    @UseCase(id = 49, description =
    "New passwords can't equal previously used ones")
    public boolean checkFor newPassword(
        List<String> prevPasswords, String password) {
        return !prevPasswords.contains(password);
    }
} //:-
```

Los valores de los elementos de anotación se expresan como pares nombre-valor encerrados entre paréntesis después de la declaración **@UseCase**. A la anotación para **encryptPassword()** no se le pasa un valor en el ejemplo para el elemento **description**, por lo que cuando se procese la clase con un procesador de anotaciones aparecerá el valor predeterminado definido en **@interface UseCase**.

Podemos imaginarnos fácilmente cómo podría emplearse un sistema como éste para “esbozar” un programa y luego rellenar la funcionalidad a medida que vamos completando el diseño.

Meta-anotaciones

Actualmente sólo hay tres anotaciones estándar (descritas anteriormente) y cuatro meta-anotaciones definidas en el lenguaje Java. Las meta-anotaciones se utilizan para anotar anotaciones:

| | |
|----------------|---|
| @Target | Dónde puede aplicarse esta anotación. Los posibles argumentos de ElementType son: CONSTRUCTOR : declaración de constructor FIELD : declaración de campo (incluye constantes enum) LOCAL_VARIABLE : declaración de variable local METHOD : declaración de método PACKAGE : declaración de paquete PARAMETER : declaración de parámetro TYPE : clase, interfaz (incluyendo tipo de anotación), o declaración enum |
|----------------|---|

| | |
|--------------------|---|
| @Retention | Durante cuánto tiempo se mantiene la información de anotación. Los posibles argumentos de RetentionPolicy son: SOURCE : el compilador descarta las anotaciones. CLASS : las anotaciones están disponibles en los archivos de clases introducidos por el compilador, pero pueden ser descartados por la máquina virtual. RUNTIME : las anotaciones son retenidas por la máquina virtual en tiempo de ejecución, por lo que se pueden leer mediante el mecanismo de reflexión. |
| @Documented | Incluye esta anotación en Javadoc. |
| @Inherited | Permite a las subclases heredar anotaciones padre. |

La mayor parte del tiempo lo que haremos es definir nuestras propias anotaciones y escribir nuestros procesadores para tratar con ellas.

Escritura de procesadores de anotaciones

Si una herramienta para leerlas, las anotaciones son apenas más útiles que los comentarios. Una parte importante del proceso de uso de las anotaciones consiste en crear y utilizar *procesadores de anotaciones*. Java SE5 proporciona una serie de extensiones a la API de reflexión que nos ayudan a crear estas herramientas. También proporciona una herramienta externa denominada **apt** para ayudarnos a analizar el código fuente Java que incluya anotaciones.

He aquí un procesador de anotaciones muy simple que lee la clase anotada **PasswordUtils** y emplea el mecanismo de reflexión para buscar marcadores **@UseCase**. Dada una lista de valores **id**, enumera los casos de uso y localiza aquellos que faltan, informando de esa ausencia:

```
//: annotations/UseCaseTracker.java
import java.lang.reflect.*;
import java.util.*;

public class UseCaseTracker {
    public static void
    trackUseCases(List<Integer> useCases, Class<?> cl) {
        for(Method m : cl.getDeclaredMethods()) {
            UseCase uc = m.getAnnotation(UseCase.class);
            if(uc != null) {
                System.out.println("Found Use Case:" + uc.id() +
                    " " + uc.description());
                useCases.remove(new Integer(uc.id()));
            }
        }
        for(int i : useCases) {
            System.out.println("Warning: Missing use case-" + i);
        }
    }
    public static void main(String[] args) {
        List<Integer> useCases = new ArrayList<Integer>();
        Collections.addAll(useCases, 47, 48, 49, 50);
        trackUseCases(useCases, PasswordUtils.class);
    }
} /* Output:
Found Use Case:47 Passwords must contain at least one numeric
Found Use Case:48 no description
Found Use Case:49 New passwords can't equal previously used ones
Warning: Missing use case-50
*///:-
```

Este ejemplo utiliza tanto el método de reflexión `getDeclaredMethods()` como el método `getAnnotation()`, que proviene de la interfaz `AnnotatedElement` (clases como `Class`, `Method` y `Field` implementan esta interfaz). Este método devuelve el objeto anotación del tipo especificado, que en este caso es “`UseCase`”. Si no hay anotaciones de ese tipo concreto en el método anotado, se devuelve un valor `null`. Los valores de los elementos se extraen invocando `id()` y `description()`. Recuerde que no hemos especificado ninguna descripción en la anotación para el método `encryptPassword()`, por lo que el procesador anterior localiza el valor predeterminado “`no description`” al invocar el método `description()` para esa anotación concreta.

Elementos de anotación

El marcador `@UseCase` definido en `UseCase.java` contiene el elemento `id` de tipo `int` y el elemento `description` de tipo `String`. He aquí una lista de los tipos permitidos para los elementos de anotación:

- Todas las primitivas (`int`, `float`, `boolean` etc.)
- `String`
- `Class`
- `enum`
- `Annotation`
- Matrices de cualquiera de los tipos anteriores.

El compilador generará un error si se intenta emplear cualquier otro tipo. Observe que no está permitido utilizar ninguna de las clases envolvente de los tipos primitivos, pero gracias a la característica de conversión automática, esto no es una verdadera limitación. También podemos tener elementos que sean ellos mismos anotaciones. Como veremos un poco más adelante, las anotaciones anidadas pueden resultar muy útiles.

Restricciones de valor predeterminado

El compilador es bastante quisquilloso acerca de los valores predeterminados de los elementos. Ningún elemento puede tener un valor no especificado. Esto quiere decir que los elementos deben tener valores predeterminados o valores proporcionados por la clase que utilice la anotación.

Existe otra restricción y es que ninguno de los elementos de tipo no primitivo pueden tener `null` como valor, ni a la hora de declararlos en el código fuente ni cuando se los defina como valor predeterminado dentro de la interfaz de anotación. Esto hace que resulte difícil escribir un procesador que actúe de manera distinta dependiendo de la presencia o ausencia de un elemento, porque todos los elementos están presentes en la práctica en todas las declaraciones de anotaciones. Podemos obviar este problema tratando de comprobar si existen valores específicos, como por ejemplo cadenas de caracteres vacías o valores negativos:

```
//: annotations/SimulatingNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
    public int id() default -1;
    public String description() default "";
} //:-
```

Esta estructura sintáctica es bastante típica en las definiciones de anotaciones.

Generación de archivos externos

Las anotaciones son especialmente útiles a la hora de trabajar con sistemas que requieran algún tipo de información adicional como acompañamiento del código fuente. Tecnologías como Enterprise JavaBeans (en las versiones anteriores a EJB3) requieren numerosas interfaces y descriptores de implantación que forman una especie de “plantilla” de código, definida de

la misma forma para cada componente bean. Los servicios web, las bibliotecas de marcadores personalizados y las herramientas de mapeo objeto/relacional como Toplink y Hibernate a menudo requieren descriptores XML que son externos al código. Después de definir una clase Java, el programador debe llevar a cabo la tediosa tarea de volver a especificar informaciones tales como el nombre, el paquete, etc., es decir, informaciones que ya existen en la clase original. Cada vez que utilizamos un archivo descriptor externo, terminamos con dos fuentes de información separadas de una clase, lo que normalmente conduce a que aparezcan problemas de sincronización del código. Esto requiere también que los programadores que trabajen en el proyecto sepan cómo editar el descriptor además de cómo escribir programas Java.

Suponga que queremos proporcionar una funcionalidad básica de mapeo objeto/relacional para automatizar la creación de una tabla de base de datos con el fin de almacenar un componente JavaBean. Podríamos utilizar un archivo descriptor XML para especificar el nombre de la clase, cada de sus miembros y la información acerca de su mapeo sobre la base de datos. Sin embargo, utilizando anotaciones, podemos mantener toda la información en el archivo fuente del componente JavaBean. Para hacer esto, necesitamos anotaciones para definir el nombre de la tabla de base de datos asociada con el componente bean, sus columnas y los tipos SQL que hay que hacer corresponder con las propiedades del componente bean.

He aquí una anotación para un componente bean que le dice al procesador de anotaciones que tiene que crear una tabla de base de datos:

```
//: annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Sólo se aplica a clases
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
    public String name() default "";
} ///:-
```

Cada tipo de elemento **ElementType** que especifiquemos en la anotación **@Target** es una restricción que le dice al compilador que nuestra anotación sólo se puede aplicar a ese tipo concreto. Pódemos especificar un sólo valor de la enumeración **enum ElementType**, o bien podemos especificar una lista formada por cualquier combinación de valores separados por comas. Si queremos aplicar la anotación a cualquier tipo de elemento **ElementType**, podemos omitir la anotación **@Target**, aunque esta solución es bastante poco común.

Observe que **@DBTable** tiene un elemento **name()**, de modo que la anotación pueda suministrar un nombre para la tabla de la base de datos que el procesador tiene que crear.

He aquí las anotaciones para los campos del componente JavaBean:

```
//: annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
    boolean primaryKey() default false;
    boolean allowNull() default true;
    boolean unique() default false;
} ///:-

//: annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
    int value() default 0;
    String name() default "";
    Constraints constraints() default @Constraints;
```

```
} //:-

//: annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
    String name() default "";
    Constraints constraints() default @Constraints;
} //:-
```

La anotación **@Constraints** permite al procesador extraer los metadatos relativos a la tabla de la base de datos. Esto representa un pequeño subconjunto de las restricciones que generalmente ofrecen las bases de datos, pero nos permite hacernos una idea general. Los elementos **primaryKey()**, **allowNull()** y **unique()** tienen asignados valores predeterminados adecuados, de modo que en la mayoría de los casos un usuario de la anotación no tendrá que escribir demasiado texto.

Las otras dos anotaciones **@interface** definen tipos SQL. De nuevo, para que este sistema sea más útil, necesitamos definir una anotación para cada tipo SQL adicional. En nuestro ejemplo, dos tipos serán suficientes.

Cada uno de estos tipos tiene un elemento **name()** y un elemento **constraints()**. Este último hace uso de la característica de anotaciones anidadas, para incluir la información acerca de las restricciones de base de datos aplicable al tipo de columna. Observe que el valor predeterminado para el elemento **constraints()** es **@Constraints**. Puesto que no hay valores de elementos especificados entre paréntesis después de este tipo de anotación, el valor predeterminado de **constraints()** es en la práctica una anotación **@Constraints** con su propio conjunto de valores predeterminado. Para definir una anotación **@Constraints** anidada donde la característica de unicidad esté definida como **true** de manera predeterminada, podemos definir su elemento de la forma siguiente:

```
//: annotations/database/Uniqueness.java
// Ejemplo de anotaciones anidadas
package annotations.database;

public @interface Uniqueness {
    Constraints constraints()
        default @Constraints(unique=true);
} //:-
```

He aquí un componente bean simple que utiliza estas anotaciones:

```
//: annotations/database/Member.java
package annotations.database;

@DBTable(name = "MEMBER")
public class Member {
    @SQLString(30) String firstName;
    @SQLString(50) String lastName;
    @SQLInteger Integer age;
    @SQLString(value = 30,
    constraints = @Constraints(primaryKey = true))
    String handle;
    static int memberCount;
    public String getHandle() { return handle; }
    public String getFirstName() { return firstName; }
    public String getLastname() { return lastName; }
    public String toString() { return handle; }
    public Integer getAge() { return age; }
} //:-
```

A la anotación de clase **@DBTable** se le da el valor “MEMBER”, que se utilizará como nombre de tabla. Las propiedades de bean, **firstName** y **lastName**, están ambas anotadas con **@SQLString** y sus valores de elementos son 30 y 50, respecti-

vamente. Estas anotaciones son interesantes por dos razones: en primer lugar, utilizan el valor predeterminado en la anotación `@Constraints` anidada, y en segundo lugar emplean una característica especial de abreviatura. Si definimos un elemento de una anotación con el nombre `value`, entonces no será necesario utilizar la sintaxis basada en parejas de nombre-valor siempre y cuando sea el único tipo de elemento especificado; podemos limitarnos a especificar el valor entre paréntesis. Esto puede aplicarse a cualquiera de los tipos de elementos legales. Por supuesto, con esto estamos obligados a llamar a nuestro elemento “`value`”, pero en el caso anterior, nos permite emplear una especificación de anotación semánticamente significativa y muy fácil de leer:

```
@SQLString(30)
```

El procesador utiliza este valor para establecer el tamaño de la columna SQL que cree.

Aunque la sintaxis relativa a los valores predeterminados es bastante limpia, puede volverse muy rápidamente muy compleja. Observe la anotación correspondiente al campo `handle`. Tiene una anotación `@SQLString`, pero también necesita ser una clave principal de la base de datos, por lo que es necesario activar el tipo de elemento en `primaryKey` en la anotación `@Constraint` anidada. Aquí es donde el ejemplo comienza a ser confuso. Ahora estamos forzados a utilizar la forma, bastante larga, basada en una pareja de nombre-valor para esta anotación anidada, volviendo a especificar el nombre de elemento y el nombre de la interfaz `@interface`. Pero, como el elemento de nombre especial `value` ya no es el único valor de elemento que se está especificando, no podemos emplear la forma abreviada. Como puede ver, el resultado no es precisamente elegante.

Soluciones alternativas

Existen otras formas de crear anotaciones para llevar a cabo esta tarea. Podríamos, por ejemplo, tener una única clase de anotación denominada `@TableColumn` con un elemento `enum` que definiera valores como `STRING`, `INTEGER`, `FLOAT`, etc. Esto elimina la necesidad de definir una anotación `@interface` para cada tipo SQL, pero hace que sea imposible cualificar los tipos con elementos adicionales como `size` (tamaño) o `precision` (precisión), lo cual resulta probablemente más útil.

También podríamos utilizar un elemento de tipo `String` para describir el tipo SQL correcto, como por ejemplo, “`VARCHAR(30)`” o “`INTEGER`”. Esto nos permite cualificar los tipos, pero nos fuerza a fijar en el código la correspondencia entre el tipo Java y el tipo SQL, lo cual no es una buena práctica de diseño. No conviene tener que recompilar las clases si cambiamos las bases de datos, sería más elegante limitarnos a decirle a nuestro procesador de anotaciones que estamos usando una “versión” diferente de SQL, y dejar que el procesador lo tenga en cuenta a la hora de procesar las anotaciones.

Una tercera solución factible consiste en utilizar conjuntamente dos tipos de anotación: `@Constraints` y el tipo SQL relevante (por ejemplo, `@SQLInteger`), para anotar el campo deseado. Esto resulta ligeramente complicado, pero el compilador nos permite utilizar tantas anotaciones diferentes como queramos sobre un mismo objetivo de anotación. Observe que, al utilizar múltiples anotaciones, no podemos emplear la misma anotación dos veces.

Las anotaciones no soportan la herencia

No podemos utilizar la palabra clave `extends` con `@interfaces`. Es una pena, porque una solución elegante sería definir una anotación `@TableColumn`, como hemos sugerido anteriormente, con una anotación anidada de tipo `@SQLType`. De esa forma, podríamos heredar todos nuestros tipos SQL, como `@SQLInteger` y `@SQLString` de `@SQLType`. Esto reduciría la cantidad de texto que hay que escribir y haría más elegante la sintaxis. No parece que exista ninguna intención de que las anotaciones soporten el mecanismo de herencia en versiones futuras del lenguaje, por lo que los ejemplos anteriores parecen ser lo máximo que podemos hacer teniendo en cuenta las circunstancias actuales.

Implementación del procesador

He aquí un ejemplo de procesador de anotaciones que lee un archivo de clases, localiza sus anotaciones de base de datos y genera el comando SQL para construir la base de datos:

```
//: annotations/database/TableCreator.java
// Procesador de anotaciones basado en el mecanismo de reflexión.
// {Args: annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
```

```

import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
    public static void main(String[] args) throws Exception {
        if(args.length < 1) {
            System.out.println("arguments: annotated classes");
            System.exit(0);
        }
        for(String className : args) {
            Class<?> cl = Class.forName(className);
            DBTable dbTable = cl.getAnnotation(DBTable.class);
            if(dbTable == null) {
                System.out.println(
                    "No DBTable annotations in class " + className);
                continue;
            }
            String tableName = dbTable.name();
            // Si el nombre está vacío, utilizar el nombre de la clase:
            if(tableName.length() < 1)
                tableName = cl.getName().toUpperCase();
            List<String> columnDefs = new ArrayList<String>();
            for(Field field : cl.getDeclaredFields()) {
                String columnName = null;
                Annotation[] anns = field.getDeclaredAnnotations();
                if(anns.length < 1)
                    continue; // No es una columna de la tabla de base de datos
                if(anns[0] instanceof SQLInteger) {
                    SQLInteger sInt = (SQLInteger) anns[0];
                    // Utilizar el nombre de campo si no se especifica un nombre.
                    if(sInt.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
                        columnName = sInt.name();
                    columnDefs.add(columnName + " INT" +
                        getConstraints(sInt.constraints()));
                }
                if(anns[0] instanceof SQLString) {
                    SQLString sString = (SQLString) anns[0];
                    // Utilizar el nombre de campo si no se especifica un nombre.
                    if(sString.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
                        columnName = sString.name();
                    columnDefs.add(columnName + " VARCHAR(" +
                        sString.value() + ")" +
                        getConstraints(sString.constraints()));
                }
            }
            StringBuilder createCommand = new StringBuilder(
                "CREATE TABLE " + tableName + "(");
            for(String columnDef : columnDefs)
                createCommand.append("\n      " + columnDef + ",");
            // Eliminar coma final
            String tableCreate = createCommand.substring(
                0, createCommand.length() - 1) + ");";
            System.out.println("Table Creation SQL for " +
                className + " is :\n" + tableCreate);
        }
    }
}

```

```

private static String getConstraints(Constraints con) {
    String constraints = "";
    if(!con.allowNull())
        constraints += " NOT NULL";
    if(con.primaryKey())
        constraints += " PRIMARY KEY";
    if(con.unique())
        constraints += " UNIQUE";
    return constraints;
}
/* Output:
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30));
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
Table Creation SQL for annotations.database.Member is :
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    HANDLE VARCHAR(30) PRIMARY KEY);
*//*:-

```

El método `main()` recorre sucesivamente cada uno de los nombres de clase en la línea de comandos. Cada clase se carga utilizando `forName()` y se la comprueba para ver si incluye la anotación `@DBTable` utilizando `getAnnotation(DBTable.class)`. Si se incluye esa anotación, se localiza el nombre de la tabla y se almacena, entonces se cargan y se comprueban todos los campos de la clase con `getDeclaredAnnotations()`. Este método devuelve una matriz con todas las anotaciones definidas para un método concreto. Se utiliza el operador `instanceof` para determinar que estas anotaciones son de tipo `@SQLInteger` y `@SQLString`, y en cada caso se crea entonces el fragmento de cadena de caracteres relevante, con el nombre de la columna de la tabla. Observe que, como no existe posibilidad de herencia en las interfaces de anotación, la utilización de `getDeclaredAnnotations()` es la única forma con la que podemos aproximarnos al comportamiento polimórfico.

La anotación `@Constraint` anidada se pasa al método `getConstraints()`, que construye un objeto de tipo `String` que contiene las restricciones SQL.

Merece la pena mencionar que la técnica mostrada anteriormente es una forma un tanto ingenua de definir un mapeo objeto/relacional. Disponer de una anotación de tipo `@DBTable`, que toma el nombre de la tabla como parámetro, nos obliga a recompilar el código Java cada vez que queramos cambiar el nombre de la tabla, lo que puede resultar no muy conveniente. Hay disponibles muchos sistemas para mapear objetos sobre bases de datos relacionales, y cada vez un número mayor de ellos está haciendo uso de las anotaciones.

Ejercicio 1: (2) Implemente más tipos SQL en el ejemplo de la base de datos.

Proyecto:³ Modifique el ejemplo de la base de datos para que se conecte con una base de datos real e interactúe con ella utilizando JDBC.

Proyecto: Modifique el ejemplo de la base de datos para que cree archivos compatibles con XML en lugar de escribir código SQL.

³ Los proyectos son sugerencias que pueden utilizarse, por ejemplo, como proyectos de fin de curso. Las soluciones a los proyectos no se incluyen en la Guía de soluciones.

Utilización de apt para procesar anotaciones

La herramienta de procesamiento de anotaciones **apt** es la primera versión de Sun de este tipo de herramienta. Puesto que se trata de una versión relativamente joven, la herramienta sigue siendo un poco primitiva, pero dispone de una serie de características que pueden facilitarnos la tarea.

Como **javac**, **apt** está diseñada para ejecutarse con archivos fuente Java en lugar de con clases compiladas. De manera pre-determinada, **apt** compila los archivos fuente una vez que ha terminado de procesarlos. Esto es útil si estamos creando automáticamente nuevos archivos fuente como parte del proceso de construcción de la aplicación. De hecho, **apt** comprueba si existen anotaciones en los archivos fuente recién creados y los compila, todo ello en una pasada.

Cuando el procesador de anotaciones crea un nuevo archivo fuente, dicho archivo es comprobado a su vez en busca de anotaciones, en lo que constituye una nueva *ronda* (como se denomina en la documentación) de procesamiento. La herramienta continuará efectuando ronda tras ronda de procesamiento hasta que no se cree ningún archivo fuente. Entonces, compilará todos los archivos fuente existentes.

Cada anotación que escribamos necesitará su propio procesador, pero la herramienta **apt** permite agrupar fácilmente varios procesadores de anotación. La herramienta nos permite especificar múltiples clases que haya que procesar, lo cual resulta más fácil que tener que iterar manualmente a través de una serie de clases **File**. También podemos agregar lo que se denominan procesos escucha para recibir una notificación cuando se complete una ronda de procesamiento de anotaciones.

En el momento de escribir estas líneas, **apt** no está disponible como tarea Ant (consulte el suplemento en <http://MindView.net/Books/BetterJava>), pero mientras tanto se puede, obviamente, ejecutar la herramienta como tarea externa desde Ant. Para compilar los procesadores de anotaciones descritos en esta sección, es necesario tener **tools.jar** en la ruta de clases; esta biblioteca también contiene las interfaces **com.sun.mirror.***.

apt funciona utilizando una factoría de procesadores de anotaciones (**AnnotationProcessorFactory**) para crear el tipo apropiado de procesador para cada anotación que encuentre. Cuando se ejecuta **apt**, hay que especificar una clase factoría o una ruta de clases en la que la herramienta pueda localizar las factorías que necesite. Si no hacemos esto, **apt** se embarcará en un arcano proceso de *descubrimiento*, cuyos detalles pueden encontrarse en la sección *Developing an Annotation Processor* de la documentación de Sun.

Cuando creamos un procesador de anotaciones para utilizarlo con **apt**, no podemos emplear los mecanismos de reflexión de Java porque estamos trabajando con código fuente, no con clases compiladas.⁴ La API **mirror**⁵ resuelve este problema permitiéndonos visualizar los métodos, campos y tipos en el código fuente no compilado.

He aquí una anotación que puede utilizarse para extraer los métodos públicos de una clase y convertirlos en una interfaz:

```
//: annotations/ExtractInterface.java
// Procesamiento de anotaciones basado en APT.
package annotations;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
    public String value();
} //:-
```

El valor de **RetentionPolicy** es **SOURCE** porque no tiene ningún sentido mantener esta anotación en el archivo de clase después de haber extraído de ésta la interfaz. La siguiente clase proporciona un método público, que podría formar parte de una interfaz:

```
//: annotations/Multiplier.java
// Procesamiento de anotaciones basado en APT.
package annotations;
```

⁴ Sin embargo, utilizando la opción no estándar **-XclassesAsDecls**, se puede trabajar con anotaciones que estén contenidas en clases compiladas.

⁵ La palabra **mirror** significa espejo, así que se trata de un juego de palabras de los diseñadores Java para hacer referencia en realidad al mecanismo de reflexión.

```

@ExtractInterface("IMultiplier")
public class Multiplier {
    public int multiply(int x, int y) {
        int total = 0;
        for(int i = 0; i < x; i++)
            total = add(total, y);
        return total;
    }
    private int add(int x, int y) { return x + y; }
    public static void main(String[] args) {
        Multiplier m = new Multiplier();
        System.out.println("11*16 = " + m.multiply(11, 16));
    }
} /* Output:
11*16 = 176
*/

```

La clase **Multiplier** (que sólo funciona con enteros positivos) tiene un método **multiply()** que invoca numerosas veces el método privado **add()** para llevar a cabo la multiplicación. El método **add()** no es público, así que no forma parte de la interfaz. A la anotación se le asigna el valor de **IMultiplier**, que es el nombre de la interfaz que hay que crear.

Ahora necesitamos un procesador para realizar la extracción:

```

//: annotations/InterfaceExtractorProcessor.java
// Procesamiento de anotaciones basado en APT.
// {Exec: apt -factory
// annotations.InterfaceExtractorProcessorFactory
// Multiplier.java -s ../annotations}
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
import java.util.*;

public class InterfaceExtractorProcessor
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private ArrayList<MethodDeclaration> interfaceMethods =
        new ArrayList<MethodDeclaration>();
    public InterfaceExtractorProcessor(
        AnnotationProcessorEnvironment env) { this.env = env; }
    public void process() {
        for(TypeDeclaration typeDecl :
            env.getSpecifiedTypeDeclarations()) {
            ExtractInterface annot =
                typeDecl.getAnnotation(ExtractInterface.class);
            if(annot == null)
                break;
            for(MethodDeclaration m : typeDecl.getMethods())
                if(m.getModifiers().contains(Modifier.PUBLIC) &&
                   !(m.getModifiers().contains(Modifier.STATIC)))
                    interfaceMethods.add(m);
            if(interfaceMethods.size() > 0) {
                try {
                    PrintWriter writer =
                        env.getFiler().createSourceFile(annot.value());
                    writer.println("package " +
                        typeDecl.getPackage().getQualifiedName() + ";");
                    writer.println("public interface " +
                        annot.value() + " {");
                    for(MethodDeclaration m : interfaceMethods) {

```

El lugar donde se realiza todo el trabajo es en el método `process()`. La clase `MethodDeclaration` y su método `getModifiers()` se usan para identificar los métodos públicos (pero ignorando los estáticos) de la clase que se esté procesando. Si se encuentra algún método público, se almacena en un contenedor `ArrayList` y se emplea para crear los métodos de una nueva definición de interfaz en un archivo.java.

Observe que al constructor se le pasa un objeto `AnnotationProcessorEnvironment`. Podemos consultar este objeto para determinar todos los tipos (definiciones de clase) que la herramienta `apt` está procesando, y podemos usarlo para obtener un objeto `Messager` y un objeto `Filer`. El objeto `Messager` nos permite emitir mensajes dirigidos al usuario, por ejemplo cualquier error que pueda haberse producido en el procesamiento, junto con el lugar del código fuente donde se haya producido. El objeto `Filer` es un tipo objeto `PrintWriter` a través del cual se crean nuevos archivos. La principal razón de emplear un objeto `Filer`, en lugar de un objeto `PrintWriter` simple es que permite a `apt` llevar la cuenta de los nuevos archivos que creamos, para así poder comprobar si contienen anotaciones y compilarlas, en caso necesario.

También podemos ver que el método `createSourceFile()` abre un flujo de salida ordinario con el nombre correcto para nuestra interfaz o clase Java. No existe ningún tipo de soporte para la creación de estructuras sintácticas del lenguaje Java, así que hay que generar el código fuente Java utilizando los métodos `print()` y `println()`, un tanto primitivos. Esto quiere decir que tenemos que asegurarnos de que los corchetes estén bien emparejados y de que el código sea sintácticamente correcto.

La herramienta **apt** invoca al método **process()**, porque la herramienta necesita una factoría para proporcionar el procesador adecuado:

```
//: annotations/InterfaceExtractorProcessorFactory.java
// Procesamiento de anotaciones basado en APT.
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class InterfaceExtractorProcessorFactory
    implements AnnotationProcessorFactory {
    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {
        return new InterfaceExtractorProcessor(env);
    }
    public Collection<String> supportedAnnotationTypes() {
        return
            Collections.singleton("annotations.ExtractInterface");
    }
}
```

```

    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
} //:-/

```

Sólo hay tres métodos en la interfaz `AnnotationProcessorFactory`. Como puede ver, el que proporciona el procesador es `getProcessorFor()`, que toma un conjunto Set de declaraciones de tipo (las clases Java para las que se está ejecutando la herramienta `apt`), y el objeto `AnnotationProcessorEnvironment`, que ya hemos visto cómo se pasaba al procesador. Los otros dos métodos, `supportedAnnotationTypes()` y `supportedOptions()`, sirven para poder comprobar que disponemos de procesadores para todas las anotaciones encontradas por `apt` y que soportamos todas las anotaciones especificadas en la línea de comandos. El método `getProcessorFor()` es particularmente importante, porque si no devolvemos el nombre de clase completo de nuestro tipo de anotación dentro de la colección `String`, `apt` emitirá una advertencia informando de que no existe el procesador correspondiente y terminará su ejecución sin hacer nada.

El procesador y la factoría se encuentran en el paquete `annotations`, así que, para la estructura de directorios anterior, la línea de comandos está incrustada en el marcador de comentarios ‘`Exec`’ al principio de `InterfaceExtractorProcessor.java`. Esto le dice a `apt` que tiene que utilizar la clase factoría definida anteriormente y procesar el archivo `Multiplier.java`. La opción `-s` especifica que los nuevos archivos deben crearse en el directorio `annotations`. El archivo `IMultiplier.java` generado, como podemos adivinar examinando las instrucciones `println()` en el procesador anterior, tiene el aspecto siguiente:

```

package annotations;
public interface IMultiplier {
    public int multiply (int x, int y);
}

```

Este archivo también será compilado por `apt`, así que verá que el archivo `IMultiplier.class` aparece en el mismo directorio.

Ejercicio 2: (3) Añada al extractor de interfaces el soporte para la operación de división.

Utilización del patrón de diseño *Visitante* con `apt`

El procesamiento de anotaciones puede resultar bastante complejo. El ejemplo anterior es un procesador de anotaciones relativamente simple que sólo interpreta una anotación, a pesar de lo cual requiere de una cierta complejidad para poder llevar a cabo su tarea. Para evitar que la complejidad crezca desmesuradamente cuando tengamos más anotaciones y más procesadores, la API `mirror` proporciona clases para dar soporte al patrón de diseño *Visitante*. Este patrón de diseño es uno de los patrones clásicos del libro *Design Patterns* de Gamma *et al.*, y también puede encontrar una explicación más detallada en *Thinking in Patterns*.

Un *Visitante* recorre una estructura de datos o colección de objetos, realizando una operación con cada uno. La estructura de datos no necesita estar ordenada, y la operación que se realice con cada objeto será específica del tipo de éste. Esto hace que se desacoplen las operaciones con respecto a los propios objetos, lo que quiere decir que podemos añadir nuevas operaciones sin necesidad de añadir métodos a las definiciones de clase.

Esto hace que este patrón de diseño sea muy útil para el procesamiento de anotaciones, porque una clase Java puede considerarse como una colección de objetos `TypeDeclaration`, `FieldDeclaration`, `MethodDeclaration`, etc. Cuando utilizamos la herramienta `apt` con el patrón *Visitante*, proporcionamos una clase `Visitor` que tiene un método para gestionar cada tipo de declaración que visitemos. De este modo, podemos implementar el comportamiento apropiado para las anotaciones asociadas con métodos, clases, campos, etc.

He aquí de nuevo el generador de tablas SQL, pero esta vez con una factoría y un procesador que hace uso del patrón de diseño *Visitante*:

```

//: annotations/database/TableCreationProcessorFactory.java
// El ejemplo de la base de datos usando el patrón de diseño Visitante.
// (Exec: apt -factory
// annotations.database.TableCreationProcessorFactory
// database/Member.java -s database)
package annotations.database;

```

```

import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.util.*;
import java.util.*;
import static com.sun.mirror.util.DeclarationVisitors.*;

public class TableCreationProcessorFactory
    implements AnnotationProcessorFactory {
    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {
        return new TableCreationProcessor(env);
    }
    public Collection<String> supportedAnnotationTypes() {
        return Arrays.asList(
            "annotations.database.DBTable",
            "annotations.database.Constraints",
            "annotations.database.SQLString",
            "annotations.database.SQLInteger");
    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
    private static class TableCreationProcessor
        implements AnnotationProcessor {
        private final AnnotationProcessorEnvironment env;
        private String sql = "";
        public TableCreationProcessor(
            AnnotationProcessorEnvironment env) {
            this.env = env;
        }
        public void process() {
            for(TypeDeclaration typeDecl :
                env.getSpecifiedTypeDeclarations()) {
                typeDecl.accept(getDeclarationScanner(
                    new TableCreationVisitor(), NO_OP));
                sql = sql.substring(0, sql.length() - 1) + ");";
                System.out.println("creation SQL is :\n" + sql);
                sql = "";
            }
        }
        private class TableCreationVisitor
            extends SimpleDeclarationVisitor {
            public void visitClassDeclaration(
                ClassDeclaration d) {
                DBTable dbTable = d.getAnnotation(DBTable.class);
                if(dbTable != null) {
                    sql += "CREATE TABLE ";
                    sql += (dbTable.name().length() < 1)
                        ? d.getSimpleName().toUpperCase()
                        : dbTable.name();
                    sql += " (";
                }
            }
            public void visitFieldDeclaration(
                FieldDeclaration d) {
                String columnName = "";
                if(d.getAnnotation(SQLInteger.class) != null) {

```

La salida es idéntica a la del ejemplo DBTable anterior.

El procesador y el visitante son clases internas en este ejemplo. Observe que el método **process()** sólo añade la clase visitante e inicializa la cadena SOL.

Los dos parámetros de `getDeclarationScanner()` son visitantes; el primero se utiliza antes de visitar cada declaración y el segundo después. Este procesador sólo necesita el visitante previo visita, así que se proporciona `NO_OP` como segundo parámetro. Éste es un campo de tipo estático de la interfaz `DeclarationVisitor`, que indica un objeto `DeclarationVisitor` que no lleva a cabo ninguna tarea.

TableCreationVisitor amplía **SimpleDeclarationVisitor**, sustituyendo los dos métodos **visitClassDeclaration()** y **visitFieldDeclaration()**. **SimpleDeclarationVisitor** es un adaptador que implementa todos los métodos de la interfaz **DeclarationVisitor**, por lo que podemos concentrarnos en aquellos que necesitemos. En **visitClassDeclaration()**, se comprueba el objeto **ClassDeclaration** en busca de la anotación **DBTable**, y en caso de encontrarla, se inicializa la primera parte del objeto **String** de creación de la cadena SQL. En **visitFieldDeclaration()**, se consulta la declaración de campo para ver las anotaciones existentes y la información se extrae de forma bastante similar a como se hacia en el ejemplo original que hemos presentado anteriormente en el capítulo.

Podría parecer que esta forma de hacer las cosas resulta más complicada, pero permite obtener una solución más escalable. Si la complejidad del procesador de anotaciones se incrementa, escribir nuestro propio procesador autónomo, como en el ejemplo anterior, podría llegar pronto a resultar bastante complicado.

Ejercicio 3: (2) Añada a `TableCreationProcessorFactory.java` soporte para más tipos SQL.

Pruebas unitarias basadas en anotaciones

Las *pruebas unitarias* son la práctica de crear una o más pruebas para cada método de una clase, con el fin de comprobar de forma metódica las distintas partes de una clase y verificar que su comportamiento es correcto. La herramientas más popular de pruebas unitarias en Java se denomina *JUnit*; en el momento de escribir estas líneas, JUnit estaba a punto de ser actualizada a su versión 4, para poder incorporar anotaciones⁶. Uno de los principales problemas de las versiones de JUnit que no incorporaba el soporte de anotaciones es la cantidad de "ceremonia" necesaria para preparar y ejecutar pruebas JUnit. Esta complejidad se redujo a lo largo del tiempo, pero las anotaciones permitirán simplificar todavía más el proceso de pruebas.

Con las versiones de JUnit que no disponían de soporte de anotaciones, era necesario crear una clase separada para definir las pruebas unitarias. Con las anotaciones, podemos incluir las pruebas unitarias dentro de la clase que hay probar, reduciendo así al mínimo el tiempo y la complejidad de las pruebas unitarias. Este técnica tiene la ventaja adicional de permitir comprobar tanto los métodos privados como los públicos.

Puesto que este marco de trabajo para pruebas que utilizamos como ejemplo está basado en anotaciones, lo denominaremos **@Unit**. La forma más básica de pruebas, que es la que utilizaremos la mayor parte del tiempo, sólo necesita la anotación **@Test** para indicar lo que hay que probar. Una opción es que los métodos de prueba no tomen ningún argumento y devuelvan un valor **boolean** para indicar el éxito o el fallo de la prueba. Podemos utilizar cualquier nombre que queramos para los métodos de prueba. Asimismo, los métodos de prueba **@Unit** pueden tener cualquier tipo de acceso que deseemos, incluyendo **private**.

Para utilizar **@Unit**, todo lo que hace falta es importar **net.mindview.atunit**,⁷ marcar los métodos y campos apropiados con marcadores de prueba **@Unit** (los cuales veremos en los siguientes ejemplos) y hacer que el sistema de construcción ejecute **@Unit** con la clase resultante. He aquí un ejemplo simple:

```
//: annotations/AtUnitExample1.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample1 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @Test boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean m2() { return methodTwo() == 2; }
    @Test private boolean m3() { return true; }
    // Muestra la salida en caso de fallo:
    @Test boolean failureTest() { return false; }
    @Test boolean anotherDisappointment() { return false; }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample1");
    }
} /* Output:
annotations.AtUnitExample1
    . methodOneTest

```

⁶ Originalmente, pensé en diseñar una "versión avanzada de JUnit" basada en el diseño mostrado aquí. Sin embargo, parece que JUnit 4 también incluye muchas de las ideas que aquí se presentan, así que me resulta más sencillo utilizar la herramienta disponible.

⁷ Esta biblioteca es parte del código del libro, disponible en www.MindView.net.

```

. m2 This is methodTwo
. m3
. failureTest (failed)
. anotherDisappointment (failed)
(5 tests)

>>> 2 FAILURES <<<
annotations.AtUnitExample1: failureTest
annotations.AtUnitExample1: anotherDisappointment
*///:-
```

Las clases que hay que probar con **@Unit** deben encontrarse en paquetes.

La anotación **@Test** que va antes de los métodos **methodOne()**, **m2()**, **m3()**, **failureTest()** y **anotherDisappointment()** le dice a **@Unit** que ejecute estos métodos como pruebas unitarias. También garantiza que esos métodos no tomen ningún argumento y devuelvan un valor **boolean** o **void**. Nuestra única responsabilidad a la hora de escribir la prueba unitaria consiste en determinar si la prueba tiene éxito o falla, y devuelve **true** o **false**, respectivamente (para los métodos que devuelvan un valor **boolean**).

Si está familiarizado con JUnit, también se habrá fijado en que **@Unit** proporciona una salida más informativa: puede verse la prueba que se está ejecutando actualmente, lo que hace que la salida de dicha prueba sea más útil, y al final nos dice las clases y pruebas que han producido errores.

No estamos obligados a incluir los métodos de prueba dentro de nuestras clases, si esa solución no nos sirve. La forma más fácil de crear pruebas no embebidas es mediante el mecanismo de prueba:

```

//: annotations/AtUnitExternalTest.java
// Creación de pruebas no embebidas.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExternalTest extends AtUnitExample1 {
    @Test boolean _methodOne() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() { return methodTwo() == 2; }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExternalTest");
    }
} /* Output:
annotations.AtUnitExternalTest
. _methodOne
. _methodTwo This is methodTwo
OK (2 tests)
*///:-
```

Este ejemplo también ilustra el valor de los esquemas de denominación flexibles (a diferencia del requisito de JUnit que exige que todas nuestras pruebas comiencen por la palabra “**test**”). Aquí, los métodos **@Test** que están probando directamente otro método reciben el nombre de dicho método, pero comenzando por un guion bajo (no estoy sugiriendo que este estilo resulte ideal, sino simplemente mostrando una posibilidad).

También podemos utilizar el mecanismo de composición para crear pruebas no embebidas:

```

//: annotations/AtUnitComposition.java
// Creación de pruebas no embebidas.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;
```

```

public class AtUnitComposition {
    AtUnitExample1 testObject = new AtUnitExample1();
    @Test boolean _methodOne() {
        return
            testObject.methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() {
        return testObject.methodTwo() == 2;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitComposition");
    }
} /* Output:
annotations.AtUnitComposition
  - _methodOne
  - _methodTwo This is methodTwo

OK (2 tests)
*///:-
```

Para cada prueba se crea un nuevo miembro **testObject**, ya que se crea para cada prueba un objeto **AtUnitComposition**.

No existen métodos especiales “assert” como en JUnit, pero la segunda forma del método **@Test** permite devolver **void** (o **boolean**, si seguimos queriendo devolver **true** o **false** en este caso). Para comprobar que la prueba ha tenido éxito, podemos utilizar instrucciones **assert** de Java. Las aserciones de Java normalmente tienen que ser habilitadas con el indicador **-ea** en la línea de comandos **java**, pero **@Unit** se encarga automáticamente de habilitarlas. Para indicar el fallo, podemos incluso emplear una excepción. Uno de los objetivos de diseño de **@Unit** es imponer la menor cantidad posible de sintaxis adicional, y las excepciones e instrucciones **assert** de Java son lo único necesario para informar acerca de los errores. Una aserción fallida o una excepción generada dentro del método de prueba se tratan como una prueba fallida, pero **@Unit** no se detiene en este caso, sino que continúa hasta haber ejecutado todas las pruebas.

```

//: annotations/AtUnitExample2.java
// Se pueden utilizar aserciones y excepciones en las pruebas.
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample2 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @Test void assertExample() {
        assert methodOne().equals("This is methodOne");
    }
    @Test void assertFailureExample() {
        assert 1 == 2: "What a surprise!";
    }
    @Test void exceptionExample() throws IOException {
        new FileInputStream("nofile.txt"); // Genera excepción
    }
    @Test boolean assertAndReturn() {
        // Aserción con mensaje:
        assert methodTwo() == 2: "methodTwo must equal 2";
        return methodOne().equals("This is methodOne");
    }
}
```

```

    }
    public static void main(String[] args) throws Exception {
        OSEExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample2");
    }
} /* Output:
annotations.AtUnitExample2
. assertExample
. assertFailureExample java.lang.AssertionError: What a surprise!
(failed)
. exceptionExample java.io.FileNotFoundException: myfile.txt (The system cannot find
the file specified)
(failed)
. assertAndReturn This is methodTwo

(4 tests)

>>> 2 FAILURES <<<
annotations.AtUnitExample2: assertFailureExample
annotations.AtUnitExample2: exceptionExample
*///:-
```

He aquí un ejemplo utilizando pruebas no embebidas con aserciones, en el que se realizan algunas pruebas simples de `java.util.HashSet`:

```

//: annotations/HashSetTest.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
    HashSet<String> testObject = new HashSet<String>();
    @Test void initialization() {
        assert testObject.isEmpty();
    }
    @Test void _contains() {
        testObject.add("one");
        assert testObject.contains("one");
    }
    @Test void _remove() {
        testObject.add("one");
        testObject.remove("one");
        assert testObject.isEmpty();
    }
    public static void main(String[] args) throws Exception {
        OSEExecute.command(
            "java net.mindview.atunit.AtUnit HashSetTest");
    }
} /* Output:
annotations.HashSetTest
. initialization
. _remove
. _contains
OK (3 tests)
*///:-
```

La solución basada en la herencia parece más simple, en ausencia de otras restricciones.

Ejercicio 4: (3) Verifique que se crea un nuevo objeto `testObject` antes de cada prueba.

Ejercicio 5: (1) Modifique el ejemplo anterior para utilizar la solución basada en la herencia.

Ejercicio 6: (1) Pruebe `LinkedList` utilizando la técnica mostrada en `HashSetTest.java`.

Ejercicio 7: (1) Modifique el ejercicio anterior para utilizar la solución basada en la herencia.

Para cada prueba unitaria, `@Unit` crea un objeto de la clase que se está probando utilizando un constructor predeterminado. Se invoca la prueba para dicho objeto, y a continuación se descarta el objeto para evitar que se deslicen efectos secundarios en otras pruebas unitarias. Esta solución utiliza el constructor predeterminado para crear los objetos. Si no disponemos de un constructor predeterminado o necesitamos un mecanismo más sofisticado de construcción de los objetos, hay que crear un método estático para generar el objeto y asociar la anotación `@TestObjectCreate`, como en el ejemplo siguiente:

```
//: annotations/AtUnitExample3.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample3 {
    private int n;
    public AtUnitExample3(int n) { this.n = n; }
    public int getN() { return n; }
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @TestObjectCreate static AtUnitExample3 create() {
        return new AtUnitExample3(47);
    }
    @Test boolean initialization() { return n == 47; }
    @Test boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean m2() { return methodTwo() == 2; }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample3");
    }
} /* Output:
annotations.AtUnitExample3
: initialization
: methodOneTest
: m2 This is methodTwo
OK (3 tests)
*///:-
```

El método `@TestObjectCreate` debe ser estático y debe devolver un objeto del tipo que estemos probando; el programa `@Unit` se encargará de comprobar que esto es así.

En ocasiones, necesitamos campos adicionales para realizar las pruebas unitarias. Podemos utilizar la anotación `@TestProperty` para marcar aquellos campos que sólo se utilicen en las pruebas unitarias (con el fin de poderlos eliminar antes de entregar el producto al cliente). He aquí un ejemplo que lee valores de un objeto `String` que se descompone utilizando el método `String.split()`. Esta entrada se emplea para generar objetos de prueba:

```
//: annotations/AtUnitExample4.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
```

```

import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitExample4 {
    static String theory = "All brontosauruses " +
        "are thin at one end, much MUCH thicker in the " +
        "middle, and then thin again at the far end.";
    private String word;
    private Random rand = new Random(); // Semilla basada en tiempo
    public AtUnitExample4(String word) { this.word = word; }
    public String getWord() { return word; }
    public String scrambleWord() {
        List<Character> chars = new ArrayList<Character>();
        for(Character c : word.toCharArray())
            chars.add(c);
        Collections.shuffle(chars, rand);
        StringBuilder result = new StringBuilder();
        for(char ch : chars)
            result.append(ch);
        return result.toString();
    }
    @TestProperty static List<String> input =
        Arrays.asList(theory.split(" "));
    @TestProperty
        static Iterator<String> words = input.iterator();
    @TestObjectCreate static AtUnitExample4 create() {
        if(words.hasNext())
            return new AtUnitExample4(words.next());
        else
            return null;
    }
    @Test boolean words() {
        print("'" + getWord() + "'");
        return getWord().equals("are");
    }
    @Test boolean scramble1() {
        // Cambiar a una semilla específica para obtener resultados verificables:
        rand = new Random(47);
        print("'" + getWord() + "'");
        String scrambled = scrambleWord();
        print(scrambled);
        return scrambled.equals("lAl");
    }
    @Test boolean scramble2() {
        rand = new Random(74);
        print("'" + getWord() + "'");
        String scrambled = scrambleWord();
        print(scrambled);
        return scrambled.equals("tsaeborornussu");
    }
    public static void main(String[] args) throws Exception {
        System.out.println("starting");
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample4");
    }
} /* Output:
starting
annotations.AtUnitExample4
    . scramble1 'All'
lAl

```

```

. scramble2 'brontosauruses'
tsaeborornussu.

. words 'are'

OK (3 tests)
*///:-
```

También podemos usar `@TestProperty` para marcar métodos que pueden ser utilizados durante las pruebas, pero que no sean pruebas en sí mismos.

Observe que este programa depende del orden de ejecución de las pruebas, lo cual no es, en general, una buena práctica.

Si nuestro proceso de creación de objetos de pruebas realiza una inicialización que requiera una posterior limpieza, podemos añadir opcionalmente un método estático `@TestObjectCleanup` para realizar la limpieza cuando hayamos terminado de usar el objeto de prueba. En este ejemplo, `@TestObjectCreate` abre un archivo para crear cada objeto de prueba, así que es necesario cerrar el archivo antes de descartar el objeto de prueba:

```

//: annotations/AtUnitExample5.java
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample5 {
    private String text;
    public AtUnitExample5(String text) { this.text = text; }
    public String toString() { return text; }
    @TestProperty static PrintWriter output;
    @TestProperty static int counter;
    @TestObjectCreate static AtUnitExample5 create() {
        String id = Integer.toString(counter++);
        try {
            output = new PrintWriter("Test" + id + ".txt");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return new AtUnitExample5(id);
    }
    @TestObjectCleanup static void
    cleanup(AtUnitExample5 tobj) {
        System.out.println("Running cleanup");
        output.close();
    }
    @Test boolean test1() {
        output.print("test1");
        return true;
    }
    @Test boolean test2() {
        output.print("test2");
        return true;
    }
    @Test boolean test3() {
        output.print("test3");
        return true;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
            "java net.mindview.atunit.AtUnit AtUnitExample5");
    }
} /* Output:
```

```

annotations.AtUnitExample5
    . test1
Running cleanup
    . test2
Running cleanup
    . test3
Running cleanup
OK (3 tests)
*///:-
```

Podemos ver, analizando la salida, que después de cada prueba se ejecuta automáticamente el método de limpieza.

Utilización de @Unit con genéricos

Los genéricos plantean un problema especial, porque no resulta posible "probar genéricamente". Debemos efectuar las pruebas para un parámetro de tipo específico o un conjunto de parámetros de tipo. La solución es simple: heredar una clase de prueba a partir de una versión especificada de la clase genérica.

He aquí una implementación simple de una pila:

```

//: annotations/StackL.java
// Un pila construida sobre un contenedor linkedList.
package annotations;
import java.util.*;

public class StackL<T> {
    private LinkedList<T> list = new LinkedList<T>();
    public void push(T v) { list.addFirst(v); }
    public T top() { return list.getFirst(); }
    public T pop() { return list.removeFirst(); }
} //:-
```

Para probar una versión `String`, hereda una clase de prueba de `StackL<String>`:

```

//: annotations/StackLStringTest.java
// Aplicación de @Unit a genéricos.
package annotations;
import net.mindview.junit.*;
import net.mindview.util.*;

public class StackLStringTest extends StackL<String> {
    @Test void _push() {
        push("one");
        assert top().equals("one");
        push("two");
        assert top().equals("two");
    }
    @Test void _pop() {
        push("one");
        push("two");
        assert pop().equals("two");
        assert pop().equals("one");
    }
    @Test void _top() {
        push("A");
        push("B");
        assert top().equals("B");
        assert top().equals("B");
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(
```

```

    "java net.mindview.atunit.AtUnit StackLStringTest");
}

/* Output:
annotations.StackLStringTest
  .push
  .pop
  .top
OK (3 tests)
*///:-
```

La única desventaja potencial de la herencia es que perdemos la capacidad de acceder a los métodos privados en la clase que se está probando. Si esto constituye un problema, podemos definir el método en cuestión como **protected**, o añadir un método no privado **@TestProperty** que invoque al método privado (el método **@TestProperty** será luego eliminado del código de producción por la herramienta **AtUnitRemover** que se muestra más adelante en el capítulo).

Ejercicio 8: (2) Cree una clase con un método privado y añada un método **@TestProperty** no privado como se ha descrito anteriormente. Invoque este método en su código de pruebas.

Ejercicio 9: (2) Escriba pruebas **@Unit** básicas para **HashMap**.

Ejercicio 10: (2) Seleccione un ejemplo de algún otro lugar del libro y añada pruebas **@Unit**.

No hace falta ningún “agrupamiento”

Una de las grandes ventajas de **@Unit** sobre JUnit es que no hacen falta “agrupamientos”. En JUnit, necesitamos poder decir de alguna forma a la herramienta de pruebas unitarias qué es lo que necesitamos probar, y esto requiere la introducción de “agrupamientos” de pruebas, para que JUnit pueda encontrarlos y ejecutar las pruebas.

@Unit simplemente busca archivos de clase que contengan las anotaciones apropiadas, y ejecuta a continuación los métodos **@Test**. Uno de los principales objetivos que me planteé al desarrollar el sistema de pruebas **@Unit** es que fuera enormemente transparente, para que los desarrolladores pudieran comenzar a utilizarlo simplemente añadiendo métodos **@Test**, sin ningún otro código especial y sin ningún conocimiento adicional como los requeridos por JUnit y muchos otros sistemas de pruebas unitarias. Ya es suficientemente difícil escribir pruebas sin añadir nuevos errores, como para también perder el tiempo con complicaciones innecesarias, así que **@Unit** trataba de hacer que la tarea de definir las tareas unitarias sea trivial. De esta forma, resulta más probable que el diseñador se anime a escribir esas pruebas.

Implementación de **@Unit**

En primer lugar, necesitamos definir todos los tipos de anotación. Se trata de marcadores simples que no tienen ningún campo. El marcador **@Test** ya se ha definido al principio del capítulo y aquí están el resto de las anotaciones:

```

//: net/mindview/atunit/TestObjectCreate.java
// El marcador @Unit @TestObjectCreate.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~

//: net/mindview/atunit/TestObjectCleanup.java
// El marcador @Unit @TestObjectCleanup.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~

//: net/mindview/atunit/TestProperty.java
```

```
// El marcador @Unit @TestProperty.
package net.mindview.atunit;
import java.lang.annotation.*;

// Se pueden marcar como propiedades tanto los campos como los métodos:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} //:-~
```

Todas las pruebas tienen un tipo retención igual a **RUNTIME**, porque el sistema **@Unit** debe descubrir las pruebas en el código compilado.

Para implementar el sistema que ejecuta las pruebas, utilizamos el mecanismo de reflexión para extraer las anotaciones. El programa utiliza esta información para decidir cómo construir los objetos de prueba y para ejecutar las pruebas sobre ellos. Gracias a las anotaciones el sistema es sorprendentemente pequeño y sencillo:

```
//: net/mindview/atunit/AtUnit.java
// Un sistema de pruebas unitarias basadas en anotaciones.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnit implements ProcessFiles.Strategy {
    static Class<?> testClass;
    static List<String> failedTests= new ArrayList<String>();
    static long testsRun = 0;
    static long failures = 0;
    public static void main(String[] args) throws Exception {
        ClassLoader.getSystemClassLoader()
            .setDefaultAssertionStatus(true); // Habilitar aserciones
        new ProcessFiles(new AtUnit(), "class").start(args);
        if(failures == 0)
            print("OK (" + testsRun + " tests)");
        else {
            print("(" + testsRun + " tests)");
            print("\n>> " + failures + " FAILURE" +
                (failures > 1 ? "S" : "") + " <<<");
            for(String failed : failedTests)
                print(" " + failed);
        }
    }
    public void process(File cFile) {
        try {
            String cName = ClassNameFinder.thisClass(
                BinaryFile.read(cFile));
            if(!cName.contains("."))
                return; // Ignorar clases no empaquetadas
            testClass = Class.forName(cName);
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        TestMethods testMethods = new TestMethods();
        Method creator = null;
        Method cleanup = null;
        for(Method m : testClass.getDeclaredMethods()) {
            testMethods.addIfTestMethod(m);
            if(creator == null)
```

```

        creator = checkForCreatorMethod(m);
        if(cleanup == null)
            cleanup = checkForCleanupMethod(m);
    }
    if(testMethods.size() > 0) {
        if(creator == null)
            try {
                if(!Modifier.isPublic(testClass
                    .getDeclaredConstructor().getModifiers()))
                    print("Error: " + testClass +
                        " default constructor must be public");
                System.exit(1);
            }
        } catch(NoSuchMethodException e) {
            // Constructor determinado sintetizado; OK
        }
        print(testClass.getName());
    }
    for(Method m : testMethods) {
        printnb(" " + m.getName() + " ");
        try {
            Object testObject = createTestObject(creator);
            boolean success = false;
            try {
                if(m.getReturnType().equals(boolean.class))
                    success = (Boolean)m.invoke(testObject);
                else {
                    m.invoke(testObject);
                    success = true; // Si no falla ninguna aserción
                }
            } catch(InvocationTargetException e) {
                // La excepción en si está dentro de e:
                print(e.getCause());
            }
            print(success ? "" : "(failed)");
            testsRun++;
            if(!success) {
                failures++;
                failedTests.add(testClass.getName() +
                    ": " + m.getName());
            }
            if(cleanup != null)
                cleanup.invoke(testObject, testObject);
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

static class TestMethods extends ArrayList<Method> {
    void addIfTestMethod(Method m) {
        if(m.getAnnotation(Test.class) == null)
            return;
        if(!(m.getReturnType().equals(boolean.class) ||
            m.getReturnType().equals(void.class)))
            throw new RuntimeException("@Test method" +
                " must return boolean or void");
        m.setAccessible(true); // En caso de que sea privado, etc.
        add(m);
    }
}

```

```

private static Method checkForCreatorMethod(Method m) {
    if(m.getAnnotation(TestObjectCreate.class) == null)
        return null;
    if(!m.getReturnType().equals(testClass))
        throw new RuntimeException("@TestObjectCreate " +
            "must return instance of Class to be tested");
    if((m.getModifiers() &
        java.lang.reflect.Modifier STATIC) < 1)
        throw new RuntimeException("@TestObjectCreate " +
            "must be static.");
    m.setAccessible(true);
    return m;
}

private static Method checkForCleanupMethod(Method m) {
    if(m.getAnnotation(TestObjectCleanup.class) == null)
        return null;
    if(!m.getReturnType().equals(void.class))
        throw new RuntimeException("@TestObjectCleanup " +
            "must return void");
    if((m.getModifiers() &
        java.lang.reflect.Modifier STATIC) < 1)
        throw new RuntimeException("@TestObjectCleanup " +
            "must be static.");
    if(m.getParameterTypes().length == 0 ||
       m.getParameterTypes()[0] != testClass)
        throw new RuntimeException("@TestObjectCleanup " +
            "must take an argument of the tested type.");
    m.setAccessible(true);
    return m;
}

private static Object createTestObject(Method creator) {
    if(creator != null) {
        try {
            return creator.invoke(testClass);
        } catch(Exception e) {
            throw new RuntimeException("Couldn't run " +
                "@TestObject (creator) method.");
        }
    } else { // Utilizar el constructor predeterminado:
        try {
            return testClass.newInstance();
        } catch(Exception e) {
            throw new RuntimeException("Couldn't create a " +
                "test object. Try using a @TestObject method.");
        }
    }
}
}

```

`AtUnit.java` utiliza la herramienta `ProcessFiles` de `net.mindview.util`. La clase `AtUnit` implementa `ProcessFiles.Strategy`, que contiene el método `process()`. De esta forma, se puede pasar una instancia de `AtUnit` al constructor `ProcessFiles`. El segundo argumento del constructor le dice a `ProcessFiles` que busque todos los archivos que tengan la extensión "class".

Si no proporcionamos un argumento de línea de comandos, el programa recorrerá el árbol de directorios actual. También podemos proporcionar múltiples argumentos que pueden ser archivos de clase (con o sin la extensión `.class`) o directorios. Puesto que `@Unit` encuentra automáticamente las clases y métodos que son susceptibles de prueba, no hace falta ningún mecanismo de “agrupamiento”¹⁸.

⁸ No está claro por qué el constructor predeterminado de la clase que estemos probando debe ser público, pero si no lo es, la llamada a `newInstance()` se cuelga (sin generar una excepción).

Uno de los problemas que `AtUnit.java` debe resolver cuando descubre archivos de clase es que el nombre de clase real cualificado (incluyendo el paquete) no resulta evidente a partir del nombre de archivo de clase. Para descubrir esta información, debe analizarse el archivo de clase, lo cual no es trivial, aunque tampoco imposible.⁹ Por tanto, lo primero que sucede cuando se localiza un archivo `.class` es que se abre y sus datos binarios son leídos y entregados a `ClassNameFinder.thisClass()`. Aquí, nos estamos introduciendo en el campo de la “ingeniería de código intermedio”, porque lo que estamos haciendo es analizar el contenido de un archivo de clase:

```
//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
    public static String thisClass(byte[] classBytes) {
        Map<Integer, Integer> offsetTable =
            new HashMap<Integer, Integer>();
        Map<Integer, String> classNameTable =
            new HashMap<Integer, String>();
        try {
            DataInputStream data = new DataInputStream(
                new ByteArrayInputStream(classBytes));
            int magic = data.readInt(); // 0xcafebabe
            int minorVersion = data.readShort();
            int majorVersion = data.readShort();
            int constant_pool_count = data.readShort();
            int[] constant_pool = new int[constant_pool_count];
            for(int i = 1; i < constant_pool_count; i++) {
                int tag = data.read();
                int tableSize;
                switch(tag) {
                    case 1: // UTF
                        int length = data.readShort();
                        char[] bytes = new char[length];
                        for(int k = 0; k < bytes.length; k++)
                            bytes[k] = (char) data.read();
                        String className = new String(bytes);
                        classNameTable.put(i, className);
                        break;
                    case 5: // LONG
                    case 6: // DOUBLE
                        data.readLong(); // descartar 8 bytes
                        i++; // Salto especial necesario
                        break;
                    case 7: // CLASS
                        int offset = data.readShort();
                        offsetTable.put(i, offset);
                        break;
                    case 8: // STRING
                        data.readShort(); // descartar 2 bytes
                        break;
                    case 3: // INTEGER
                    case 4: // FLOAT
                    case 9: // FIELD_REF
                    case 10: // METHOD_REF
                    case 11: // INTERFACE_METHOD_REF
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

⁹ Jeremy Meyer y yo nos pasamos la mayor parte de una jornada tratando de descubrir la solución.

```

        case 12: // NAME_AND_TYPE
            data.readInt(); // descartar 4 bytes;
            break;
        default:
            throw new RuntimeException("Bad tag " + tag);
    }
}
short access_flags = data.readShort();
int this_class = data.readShort();
int super_class = data.readShort();
return classNameTable.get(
    offsetTable.get(this_class)).replace('/', '.');
} catch(Exception e) {
    throw new RuntimeException(e);
}
}
// Ilustración:
public static void main(String[] args) throws Exception {
    if(args.length > 0) {
        for(String arg : args)
            print(thisClass(BinaryFile.read(new File(arg))));
    } else
        // Recorrer todo el árbol:
        for(File klass : Directory.walk(".", ".*\\".class"))
            print(thisClass(BinaryFile.read(klass)));
}
} //:~
```

Aunque no podemos analizar aquí todos los detalles, cada archivo de clase se ajusta a un formato concreto y hemos tratado en el ejemplo de utilizar nombres de campo significativos para los fragmentos de datos extraídos del flujo de datos `ByteArrayInputStream`; también podemos ver el tamaño de cada fragmento examinando la longitud de la lectura realizada en el flujo de entrada. Por ejemplo, los primeros 32 bits de cualquier archivo de clase son siempre el “número mágico” `0xCAFEBABE`,¹⁰ y los dos siguientes valores `short` son la información de versión. La sección de constantes contiene las constantes del programa y es, por tanto, de tamaño variable; el siguiente valor `short` nos dice cuál es el tamaño para poder asignar una matriz del tamaño apropiado. Cada entrada de la sección de constantes puede ser un valor de tamaño fijo o variable, así que tenemos que examinar el marcador con el que comienza cada uno para ver qué hay que hacer con él, ésa es la razón de la instrucción `switch`. Aquí, no estamos tratando de analizar con precisión todos los datos del archivo de clase, sino simplemente recorrer ésta y extraer los fragmentos de interés, así que, como puede ver en el ejemplo, se descarta una gran cantidad de datos. La información acerca de las clases está almacenada en las tablas `classNameTable` y `offsetTable`. Después de leer la sección de constantes, podemos encontrar la información `this_class` que es un índice para la tabla `offsetTable`, que produce un índice para la tabla `classNameTable`, en la que podemos leer el nombre de la clase.

Volviendo a `AtUnit.java`, el método `process()` ahora dispone del nombre de la clase y puede tratar de determinar si contiene ‘.’, lo que quiere decir que está dentro de un paquete. Las clases no incluidas en un paquete se ignoran. Si una clase se encuentra en un paquete se utiliza el cargador de clases estándar para cargar la clase con `Class.forName()`. Ahora podemos analizar la clase en busca de anotaciones `@Unit`.

Sólo necesitamos buscar tres cosas: métodos `@Test`, que están almacenados en una lista `TestMethods`, y si existen métodos `@TestObjectCreate` y `@TestObjectCleanup`. Estos métodos se descubren mediante las llamadas a método asociadas que se pueden ver en el código, que buscan las correspondientes anotaciones.

Si se encuentra algún método `@Test`, se imprime el nombre de la clase para que podamos ver lo que está sucediendo, a continuación de lo cual se ejecuta cada prueba. Esto implica imprimir el nombre de un método, luego invocar `createTestObject()`, el cual utilizará el método `@TestObjectCreate` si existe o utilizará el constructor predeterminado si no existe. Una vez creado el objeto de prueba, se invoca el método de prueba para dicho objeto. Si la prueba devuelve un

¹⁰ Hay varias leyendas relativas al significado de este número mágico, pero como Java fue creado por auténticos frikies, podemos suponer, razonablemente, que tiene algo que ver con fantasías adolescentes acerca de una mujer en una cafetería.

valor **boolean**, se captura el resultado. Si no, presuponemos que la prueba ha tenido éxito a menos que se genere una excepción (que es lo que sucedería en caso de que se produzca una aserción fallida o cualquier otro tipo de excepción). Si se genera una excepción, se imprime la información de excepción para mostrar la causa. Si se produce cualquier fallo, se incrementa el contador de fallos y se añade el nombre de la clase y el método a **failedTests** para poder incluirlos en el informe que se genera al final de la ejecución.

Ejercicio 11: (5) Añada una anotación **@TestNote** a **@Unit**, para que la nota asociada se visualice simplemente durante las pruebas.

Eliminación del código de prueba

Aunque en muchos proyectos no pasa nada si dejamos el código de prueba en el código final (especialmente si definimos todos los métodos de prueba como **private**, cosa que siempre podemos hacer), en algunos casos conviene eliminar el código de prueba, para que el tamaño del producto sea menor o para que ese código no esté al alcance del cliente.

Esto requiere prácticas de ingeniería de código intermedio demasiado sofisticadas como para realizarlas manualmente. Sin embargo, la biblioteca de código abierto Javassist¹¹ hace posible la ingeniería de código intermedio. El siguiente programa admite un indicador **-r** opcional como primer argumento; si incluimos el indicador, eliminará las anotaciones **@Test**, mientras que si no lo incluimos se limitará a mostrar esas anotaciones. También se emplea aquí **ProcessFiles** para recorrer los archivos y directorios que hayamos elegido:

```
//: net/mindview/atunit/AtUnitRemover.java
// Visualiza las anotaciones @Unit existentes en los archivos de
// clase compilados. Si el primer argumento es "-r", se eliminan
// las anotaciones @Unit.
// (Args: ...)
// (Requires: javassist.bytecode.ClassFile;
// Debe instalar la biblioteca Javassist disponible en
// http://sourceforge.net/projects/jboss/ )
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitRemover
implements ProcessFiles.Strategy {
    private static boolean remove = false;
    public static void main(String[] args) throws Exception {
        if(args.length > 0 && args[0].equals("-r")) {
            remove = true;
            String[] nargs = new String[args.length - 1];
            System.arraycopy(args, 1, nargs, 0, nargs.length);
            args = nargs;
        }
        new ProcessFiles(
            new AtUnitRemover(), "class").start(args);
    }
    public void process(File cFile) {
        boolean modified = false;
        try {
            String cName = ClassNameFinder.thisClass(

```

¹¹ Gracias al Dr. Shigeru Chiba por crear esta biblioteca, y por toda la ayuda que me prestó a la hora de desarrollar **AtUnitRemover.java**.

ClassPool es una especie de resumen de todas las clases del sistema que estemos modificando. Garantiza la coherencia de todas las clases modificadas. Podemos extraer cada clase **CtClass** de **ClassPool**, de forma similar a como el cargador de clases y **Class.forName()** cargan las clases en la máquina JVM.

CtClass contiene el código intermedio de un objeto de clase y nos permite generar información acerca de la clase y manipular el código de la misma. Aquí, invocamos `getDeclaredMethods()` (al igual que el mecanismo de reflexión de Java) y obtenemos un objeto **MethodInfo** a partir de cada método **CtMethod**. Con esto, podemos examinar las anotaciones. Si algún método tiene una anotación en el paquete `net.mindview.atunit`, se elimina dicho método.

Si la clase ha sido modificada, se sobreescribe el archivo de clase original con la nueva clase.

En el momento de escribir estas líneas, se acababa de añadir la funcionalidad de “eliminación” de Javassist¹², y descubrimos que eliminar los campos `@TestProperty` resulta más complejo que eliminar los métodos. Dado que pueden existir operaciones de inicialización estática que hagan referencia a esos campos, no podemos limitarnos a borrarlos. Por tanto, la versión anterior del código sólo elimina los métodos `@Unit`. Sin embargo, consulte el sitio web de Javassist para ver si existen actualizaciones; es posible que en el futuro se añada la funcionalidad de eliminación de campos. Mientras tanto, observe que el método de prueba externo mostrado en `AtUnitExternalTest.java` permite eliminar todas las pruebas simplemente borrando todos los archivos de clase creados por el código de prueba.

Resumen

Resulta muy de agradecer que se hayan añadido las anotaciones a Java. Constituyen una forma estructurada (y con comprobación de tipos) de añadir metadatos al código sin hacer que éste se complique innecesariamente y resulte ilegible. Las

¹² El Dr. Shigeru Chiba fue tan amable de añadir el método CtClass.removeMethod() a solicitud mías.

anotaciones pueden ayudarnos a eliminar la tediosa tarea de escribir descriptores de implantación y otros archivos generados. El hecho de que el marcador Javadoc `@deprecated` haya sido sustituido por la anotación `@Deprecated` es simplemente una indicación de hasta qué punto las anotaciones son mucho más convenientes que los comentarios para describir la información de las clases.

Java SE5 sólo incluye un pequeño número de anotaciones. Esto quiere decir que, si no utiliza una biblioteca de otro fabricante, necesitará crear sus propias anotaciones, junto con la lógica asociada. Con la herramienta `apt`, podemos compilar los archivos recién generados en un único paso, facilitándose así el proceso de construcción de aplicaciones, pero actualmente la API `mirror` tan sólo incluye la funcionalidad básica para ayudarnos a identificar los elementos de las definiciones de clases Java. Como hemos visto, podemos utilizar Javassist para las tareas de ingeniería de código intermedio, o bien podemos escribir a mano nuestras propias herramientas de manipulación de código intermedio.

La situación mejorará sin ninguna duda en el futuro, y los fabricantes de interfaces API y sistemas comenzarán a proporcionar anotaciones como parte de sus herramientas. Como puede imaginarse al analizar el sistema `@Unit`, resulta bastante previsible que las anotaciones provoquen cambios significativos en la forma de programar en Java.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Concurrencia

21

Hasta este momento, hemos estado hablando de *programación secuencial*. Todo lo que sucede en un programa sucede paso a paso.

Una gran cantidad de problemas de programación pueden resolverse utilizando programación secuencial. Sin embargo, para algunos problemas, resulta conveniente e incluso esencial ejecutar varias partes del programa en paralelo, de modo que dichas partes parezcan estarse ejecutando concurrentemente o, si hay disponibles varios procesadores, se ejecuten realmente de manera simultánea.

La programación paralela puede mejorar enormemente la velocidad de ejecución de los programas, proporcionar un modelo más sencillo para el diseño de ciertos tipos de programas, o ambas cosas a la vez. Sin embargo, llegar a familiarizarse con la teoría y las técnicas de la programación concurrente es algo situado a un nivel superior que las técnicas de programación que hemos aprendido hasta ahora en el libro, y representa un tema de nivel intermedio o avanzado. Este capítulo tan sólo puede proporcionar una introducción al tema, y después de estudiarlo será mucho el camino que le quede para llegar a ser un buen programador concurrente.

Como veremos, el problema real de la concurrencia es el que se presenta cuando una serie de tareas que se están ejecutando en paralelo comienzan a interferir entre sí. Esto puede suceder de una manera tan sutil y ocasional que probablemente resulte bastante apropiado decir que la concurrencia es “teóricamente determinista pero prácticamente no determinista”. En otras palabras, podemos demostrar que resulta posible escribir programas concurrentes que, con el suficiente cuidado y con las necesarias inspecciones de código, funcionen correctamente. Sin embargo, en la práctica, resulta mucho más fácil escribir programas concurrentes que únicamente “parezcan” funcionar correctamente pero que, dadas las condiciones adecuadas, fallarán. Es posible que estas condiciones nunca lleguen a darse o que se den de una manera tan infrecuente que jamás aparezcan los fallos durante las pruebas. De hecho, puede que no seamos capaces de escribir código de pruebas que permita generar las condiciones de fallo de nuestros programas concurrentes. Los fallos resultantes sólo ocurrirán ocasionalmente, y como resultado aparecerán en forma de quejas de los clientes. Éste es uno de los argumentos principales de estudiar el tema de la concurrencia: si lo ignoramos, lo más probable es que los problemas terminen por asaltarnos.

La concurrencia parece estar, por tanto, rodeada de peligros, y si eso le hace sentirse un tanto atemorizado, mejor que mejor. Aunque Java SE5 ha realizado mejoras significativas en lo que respecta a la concurrencia, siguen sin existir sistemas de protección como la verificación en tiempo de compilación o las excepciones comprobadas, para informarnos de cuándo hemos cometido un error. Con la concurrencia, toda la responsabilidad recae sobre nosotros, y sólo si somos suspicaces y agresivos podremos escribir código multihebra en Java que sea lo suficientemente fiable.

Hay algunas personas que sugieren que la concurrencia es un tema demasiado avanzado como para incluirlo en un libro de introducción al lenguaje. Su argumento es que la concurrencia es un tema autónomo que puede tratarse independientemente y que los pocos casos en los que la concurrencia aparece durante las tareas cotidianas de programación (como por ejemplo, con las interfaces gráficas de usuario) pueden tratarse sin necesidad de recurrir a estructuras especiales del lenguaje. ¿Por qué introducir un tema tan complejo si podemos evitarlo?

¡Ojalá fuera así! Lamentablemente, la decisión de si nuestros programas Java utilizarán hebras no está en nuestras manos. El sólo hecho de que nosotros no creemos ninguna hebra no quiere decir que vayamos a ser capaces de evitar escribir código basado en hebras. Por ejemplo, los sistemas web constituyen una de las aplicaciones Java más comunes y la clase básica de la biblioteca web, *servlet*, es inherentemente multihebra; esto resulta esencial porque los servidores web contienen a menudo múltiples procesadores y la concurrencia es una forma ideal de emplear esos procesadores. Aunque un *servlet* puede

parecer muy simple, es necesario que entendamos los problemas de la concurrencia con el fin de utilizar los *servlets* apropiadamente. Lo mismo podríamos decir de la programación de las interfaces gráficas de usuario, como veremos en el Capítulo 22, *Interfaces gráficas de usuario*. Aunque las bibliotecas Swing y SWT disponen de mecanismos para la seguridad de las hebras resulta difícil utilizar dichos mecanismos adecuadamente sin entender el tema de la concurrencia.

Java es un lenguaje multihebra y los problemas de concurrencia están presentes, con independencia de que seamos conscientes de su existencia. Como resultado, existen muchos programas Java que funcionan simplemente por accidente o que funcionan la mayor parte de las veces y que fallan misteriosamente de vez en cuando debido a problemas de concurrencia no localizados. En ocasiones, estos fallos son benignos, pero otras veces pueden representar la pérdida de datos de gran valor, y si no somos al menos conscientes de los problemas concurrencia, podemos terminar asumiendo que el problema se encuentra en algún otro lugar en vez de en nuestro software. Este tipo de problemas también pueden verse expuestos o amplificados si se transfiere un programa a un sistema multiprocesador. Básicamente, conocer el tema de la concurrencia nos permite ser conscientes de que existe una posibilidad de que programas aparentemente correctos puedan exhibir un comportamiento incorrecto.

La programación concurrente es como desembarcar en un nuevo mundo y aprender un nuevo lenguaje, o al menos un nuevo conjunto de conceptos del lenguaje. Comprender la programación concurrente tiene el mismo nivel de dificultad que comprender la programación orientada a objetos. Si nos aplicamos, podemos llegar a entender el mecanismo básico, pero generalmente es necesario un estudio profundo y un cierto nivel de práctica para llegar a dominar realmente la materia. El objetivo de este capítulo es proporcionar una panorámica de los fundamentos básicos de la concurrencia, para que se puedan entender los conceptos y se puedan escribir programas multihebra de una complejidad razonable, pero tenga en cuenta que resulta fácil confiarse demasiado. En cuanto comience a desarrollar soluciones de una cierta complejidad, necesitará estudiar libros específicamente dedicados a esta materia.

Las múltiples caras de la concurrencia

Una de las razones principales por las que la programación concurrente puede resultar confusa es que hay más de un problema que resolver utilizando la concurrencia y más de una técnica para implementar la concurrencia, y no existe una clara correspondencia entre estos dos aspectos (e incluso, a menudo, las líneas de separación son completamente difusas). Como resultado, estamos obligados a tratar de entender todos los problemas y los casos especiales para poder emplear la concurrencia de manera efectiva.

Los problemas que se resuelven mediante la concurrencia pueden clasificarse, de manera un tanto burda, en dos categorías: "velocidad" y "gestionabilidad del diseño".

Ejecución más rápida

El tema de la velocidad parece simple a primera vista: si queremos que un programa se ejecute más rápidamente, lo descomponemos en fragmentos y ejecutamos cada uno de estos fragmentos en un procesador distinto. La concurrencia es una herramienta fundamental para la programación multiprocesador. Hoy día, como se está agotando la Ley de Moore (al menos para los chips convencionales), las mejoras de velocidad se producen en la forma de procesadores multinúcleo en lugar de mediante chips más rápidos. Para hacer que los programas se ejecuten más rápidamente es necesario aprender a aprovechar dichos procesadores adicionales, y ésa es una de las cosas que la concurrencia hace posible.

Si disponemos de una máquina multiprocesador se pueden distribuir múltiples tareas entre los distintos procesadores, lo que permite incrementar enormemente el rendimiento. Esto es lo que suele suceder con los potentes servidores web multiprocesador, que pueden distribuir un gran número de solicitudes de usuario entre las distintas CPU, dentro de un programa que asigne una hebra a cada solicitud.

Sin embargo, la concurrencia puede también, a menudo, mejorar el rendimiento de programas que se estén ejecutando en un único procesador.

Esto puede parecer poco intuitivo. Si pensamos en ello, un programa concurrente que se esté ejecutando en un único procesador debería tener un gasto de procesamiento administrativo *mayor* que si todas las partes del programa se ejecutaran secuencialmente, debido al coste añadido del *cambio de contexto* (cambio de una tarea a otra). A primera vista, parece que debería ser más rápido ejecutar todas las partes del programa como una única tarea, ahorrándose el coste asociado al cambio de contexto.

Lo que hace que la concurrencia pueda mejorar el rendimiento en estos casos es el *bloqueo*. Si una tarea del programa no puede continuar con su procesamiento debido a alguna condición que no está bajo control del programa (normalmente operaciones de E/S), decimos que la tarea o la hebra se *bloquea*. Sin la concurrencia, todo el programa tendrá que detenerse ante esa condición externa; sin embargo, si se ha escrito el programa utilizando concurrencia, las otras tareas del programa pueden continuar ejecutándose cuando una tarea se bloquee, con lo que el programa continuará avanzado. Desde el punto de vista del rendimiento, no tiene sentido utilizar la concurrencia en una máquina con un único procesador, a menos que alguna de las tareas pueda llegar a bloquearse.

Un ejemplo muy común de mejora de rendimiento en los sistemas monoprocesador es la *programación dirigida por sucesos*. De hecho, una de las razones más imperiosas para utilizar la concurrencia es la de construir una interfaz de usuario con una buena capacidad de respuesta. Pensemos en un programa que tenga que realizar algún tipo de operación de larga duración y que termine, por tanto, ignorando la entrada del usuario, sin dar a éste ninguna respuesta. Si disponemos de un botón para salir del programa, no queremos tener que consultar si ese botón se ha apretado en cada fragmento de código que escribamos. Si lo hicieramos, el código tendría un aspecto terrible, y además no existiría ninguna garantía de que un programador no se olvidara de realizar esa comprobación. Sin la concurrencia, la única forma de tener una interfaz gráfica de usuario con una buena respuesta es que todas las tareas comprueben periódicamente la entrada de usuario. Sin embargo, al crear una hebra de ejecución separada para responder a las entradas de usuario, incluso aunque esta hebra estará bloqueada la mayor parte del tiempo, el programa permitirá garantizar un cierto nivel de respuesta.

El programa necesita continuar realizando sus operaciones, y al mismo tiempo necesita también devolver el control a la interfaz de usuario para poder responder a éste. Pero un método convencional no puede continuar llevando a cabo sus operaciones y al mismo tiempo devolver el control al resto del programa. De hecho, parece que esto fuera imposible, como si estuviéramos exigiendo a la CPU que estuviera en dos sitios a la vez; pero, ésta es, la ilusión que la concurrencia permite (en el caso de los sistemas multiprocesador se trata de algo más que una ilusión).

Una forma muy sencilla de implementar la concurrencia es en el nivel del sistema operativo, utilizando *procesos*. Un proceso es un programa auto-contenido que se ejecuta en su propio espacio de direcciones. Un sistema operativo *multitarea* puede ejecutar más de un proceso (programa) simultáneamente, conmutando periódicamente la CPU de un proceso a otro, al mismo tiempo que parece que cada proceso estuviera ejecutándose por separado. Los procesos resultan muy atractivos, porque el sistema operativo se encarga normalmente de aislar un proceso de otro de modo que no puedan interferir entre sí, lo que hace que la programación basada en procesos sea relativamente sencilla. Por contraste, los sistemas concurrentes, como el que se utiliza en Java, comparten recursos tales como la memoria y la E/S, por lo que la dificultad fundamental a la hora de escribir programas multihebra es la de coordinar el uso de estos recursos entre distintas tareas dirigidas por hebras, de modo que no haya más de una tarea en cada momento que pueda acceder a un determinado recurso.

He aquí un ejemplo simple donde se utilizan procesos del sistema operativo: mientras yo escribía este libro, solía hacer múltiples copias de seguridad redundantes del estado actual del libro. Hacía una copia en un directorio local, otra en un dispositivo USB, otra en un disco Zip y otra en un sitio FTP remoto. Para automatizar este proceso, escribí un pequeño programa, (en Python, pero los conceptos son los mismos) que comprime el libro en un archivo, incluyendo un número de versión en el nombre y luego realizaba las copias. Inicialmente, realizaba todas las copias secuencialmente, esperando a que cada una se completara antes de dar comienzo a la siguiente. Pero entonces me di cuenta de que cada operación de copia requería una cantidad de tiempo distinta, dependiendo de la velocidad de E/S del medio. Puesto que estaba utilizando un sistema operativo multitarea, podía iniciar cada operación de copia como un proceso separado y dejarlas ejecutarse en paralelo, lo que aceleraba la ejecución completa del programa. Mientras que uno de los procesos estaba bloqueado otro podía continuar con su tarea.

Éste es un ejemplo ideal de concurrencia. Cada tarea se ejecuta como un proceso en su propio espacio de direcciones, así que no existe la posibilidad de interferencias entre las distintas tareas. Lo más importante es que no hay *ninguna necesidad* de que las tareas se comuniquen entre sí, porque todas ellas son completamente independientes. El sistema operativo se encarga de todos los detalles necesarios para garantizar que todos los archivos se copien apropiadamente. Como resultado, no existe ningún riesgo y lo que obtenemos es un programa más rápido sin ningún coste adicional.

Algunos autores llevan incluso a defender que los procesos son la única solución razonable para la concurrencia,¹ pero lamentablemente existen, por regla general, limitaciones relativas al número de procesos y al gasto administrativo adicional asociado con cada uno que impiden que esa solución basada en procesos pueda aplicarse a todo el conjunto de problemas de concurrencia.

¹ Eric Raymond, por ejemplo, hace un encendida defensa de esta idea en *The Art of UNIX Programming* (Addison-Wesley, 2004).

Algunos lenguajes de programación están diseñados para aislar las tareas concurrentes unas de otras. Estos programas se denominan, generalmente, *lenguajes funcionales*, y en ellos cada llamada a función no produce ningún efecto secundario (no pudiendo así interferir con otras funciones) y se la pueda ejecutar como una tarea independiente. Erlang es uno de dichos lenguajes e incluye mecanismos seguros para que una tarea se comunique con otra. Si nos encontramos con que una parte de nuestro programa tiene que hacer un uso intensivo de la concurrencia y tropezamos con demasiados problemas a la hora de desarrollar esa parte, podemos considerar como posible solución el escribir esa parte del programa en un lenguaje concurrente dedicado, como Erlang.

Java adoptó la solución más tradicional que consiste en añadir el soporte para hebras por encima de un lenguaje secuencial.² En lugar de iniciar procesos externos en un sistema operativo multitarea, el mecanismo de hebras crea las distintas tareas *dentro* de un único proceso, representado por el programa que se está ejecutando. Una de las ventajas que esta solución proporciona es la transparencia con respecto al sistema operativo, que era uno de los principales objetivos de diseño en Java. Por ejemplo, las versiones pre-OSX del sistema operativo Macintosh (que era objetivo relativamente importante para las primeras versiones de Java) no soportaba la multitarea. Si no se hubiera añadido el mecanismo multihebra a Java, los programas Java concurrentes no habrían podido portarse a Macintosh ni a otras plataformas similares, incurriendo así en el requisito de que los programas deberían “escribirse una vez y ejecutarse en todas partes”.³

Mejora del diseño del código

Un programa que use múltiples tareas en una máquina de un único procesador seguirá haciendo una única cosa cada vez, por lo que debería ser teóricamente posible escribir el mismo programa sin utilizar tareas. Sin embargo, la concurrencia proporciona un beneficio organizativo de gran importancia: el diseño del programa puede simplificarse enormemente. Algunos tipos de problemas, como la simulación, son difíciles de resolver si no se incorpora el soporte para la concurrencia.

La mayoría de las personas han tenido la oportunidad de ver algún tipo de simulación u otro, bien en forma de juego informático o bien como animaciones generadas por computadora en alguna película. Generalmente, las simulaciones involucran muchos elementos que interactúan entre sí, cada uno de los cuales tiene “su propio cerebro”. Aunque es cierto que, en una máquina de un solo procesador, cada elemento de simulación está siendo controlado por ese único procesador, desde el punto de vista de la programación resulta mucho más fácil actuar como si cada elemento de simulación tuviera su propio procesador y fuera una tarea independiente.

Una simulación de gran envergadura puede incluir un gran número de tareas, lo que se corresponde con el hecho de que cada elemento de una simulación puede actuar de manera independiente; esto incluye no sólo los elfos y los brujos sino también las puertas o las piedras. Los sistemas multihebra tienen a menudo un límite relativamente pequeño en cuanto al número de hebras disponibles, estando dicho límite, en ocasiones, en el borde de las decenas o los centenares. Este número puede variar fuera del control del programa: puede depender de la plataforma, o en el caso de Java, de la versión de la máquina JVM. En Java, podemos asumir, por regla general, que no existirán suficientes hebras disponibles como para asignar una a cada elemento de una simulación de gran envergadura.

Una técnica típica para resolver este problema consiste en utilizar lo que se denomina multihebra *cooperativa*. El mecanismo de hebras de Java es *con desalojo*, lo que significa que hay un mecanismo de planificación que proporciona franjas temporales para cada hebra, interrumpiendo periódicamente a una hebra y efectuando un cambio de contexto a otra hebra, de modo que a cada una se le asigne una cantidad de tiempo razonable como para poder realizar la tarea que tenga asignada. En un sistema cooperativo, cada tarea cede el control voluntariamente, lo que requiere que el programador inserte a propósito algún tipo de instrucción de cesión de control dentro de cada tarea. La ventaja de un sistema cooperativo es doble: el cambio de contexto es mucho menos costoso que, normalmente, en un sistema con desalojo, y además no existe ningún límite teórico al número de tareas independientes que pueden ejecutarse simultáneamente. Cuando estamos tratando con un gran número de elementos de simulación, ésta puede ser la solución ideal. Observe, sin embargo, que algunos sistemas cooperativos no están diseñados para distribuir las tareas entre los distintos procesadores, lo que puede resultar muy restrictivo.

En el otro extremo, la concurrencia representa un modelo muy útil (porque refleja muy bien lo que sucede) cuando estamos trabajando con los modernos sistemas de *mensajería*, que involucran a muchas computadoras independientes distribuidas a

² Se podría argumentar que tratar de agregar la concurrencia a un lenguaje secuencial es un enfoque condenado al fracaso, pero que cada cual saque sus propias conclusiones.

³ Este requisito nunca ha llegado a satisfacerse del todo y Sun ya no pone tanto énfasis en él. Irónicamente, una de las razones de que este requisito no llegar a satisfacerse puede ser, precisamente, los problemas relativos al sistema de hebras, y puede que se hayan solventado en Java SE5.

lo largo de una red. En este caso, todos los procesos se ejecutan de forma completamente independiente unos de otros y no existe ni siquiera la posibilidad de compartir recursos. Sin embargo, seguimos teniendo que sincronizar la transparencia de información entre los distintos procesos, de modo que el sistema de mensajería, entendido como un todo, no pierda información ni introduzca información en los instantes incorrectos. Incluso si no pretende utilizar la concurrencia a menudo en el futuro inmediato, resulta muy útil entender los conceptos implicados para poder comprender las arquitecturas de mensajería, que se están convirtiendo en la forma predominante de crear sistemas distribuidos.

La concurrencia tiene sus costes asociados, incluyendo la complejidad inherente a este tipo soluciones, pero estos costes son más que compensados por las mejoras en el diseño del programa, por el equilibrado de recursos y por la mayor comodidad de los usuarios. En general, las hebras nos permiten crear un diseño con un acoplamiento más débil; si no fuera por ellas, determinadas partes de nuestro código se verían obligadas a prestar una atención explícita a determinadas actividades de cuya gestión se encargan normalmente las hebras.

Conceptos básicos sobre hebras

La programación concurrente permite particionar un programa en una serie de tareas separadas y que se ejecutan de forma independiente. Usando un mecanismo multihebra, cada una de estas tareas independientes (también denominadas subtareas) se asigna a una *hebra de ejecución*. Una *hebra* es un único flujo de control secuencial dentro de un proceso. Un único proceso puede, por tanto, tener múltiples tareas que se ejecuten concurrentemente, pero a la hora de programar actuamos como si cada tarea dispusiera del procesador para ella sola. Un mecanismo subyacente se encarga de dividir el tiempo de procesador de manera transparente, sin que en general tengamos que prestar atención a este mecanismo.

El modelo de hebras es una utilidad de programación que simplifica la tarea de realizar varias operaciones al mismo tiempo dentro de un mismo programa: el procesador irá saltando de una tarea a otra, asignando a cada una parte de su tiempo.⁴ Cada tarea piensa que tiene asignado el procesador de manera continua, pero lo cierto es que el tiempo del procesador se distribuye entre todas las tareas (excepto cuando el programa esté de hecho ejecutándose sobre múltiples procesadores). Una de las mayores ventajas del mecanismo de hebras es que el programador puede abstraerse completamente de este nivel, de modo que el código no necesita saber si está ejecutándose sobre un único procesador o sobre varios. De esta manera, la utilización de hebras constituye una forma de crear programas que sean transparentemente escalables: si un programa se está ejecutando de forma demasiado lenta, podemos acelerarlo fácilmente añadiendo más procesadores a la computadora. Los mecanismos multitarea y multihebra tienden a ser las formas más razonables de utilizar los sistemas multiprocesador.

Definición de las tareas

Una hebra se encarga de dirigir una cierta tarea, por lo que necesitamos una forma de describir dicha tarea. Para ello, se emplea la interfaz **Runnable**. Para definir una tarea, basta con implementar **Runnable** y escribir un método **run()** para hacer que la tarea realice su correspondiente trabajo.

Por ejemplo, la siguiente tarea **LiftOff** se encarga de mostrar una cuenta atrás antes de un lanzamiento:

```
//: concurrency/LiftOff.java
// Ilustración de la interfaz Runnable.

public class LiftOff implements Runnable {
    protected int countDown = 10; // Predeterminado
    private static int taskCount = 0;
    private final int id = taskCount++;
    public LiftOff() {}
    public LiftOff(int countDown) {
        this.countDown = countDown;
    }
    public String status() {
        return "#" + id + "(" +
            countDown + ")";
    }
}
```

⁴ Esto es cierto cuando el sistema utiliza un mecanismo de franjas temporales (por ejemplo, Windows). Solaris utiliza un modelo de concurrencia basado en una cola FIFO: a menos que se despierte una hebra de mayor prioridad, la hebra actual continuará ejecutándose hasta que se bloquee o termine. Eso significa que otras hebras con la misma prioridad no podrán ejecutarse hasta que la hebra actual ceda el control de procesador.

```

        (countDown > 0 ? countDown : "Liftoff!") + ", ";
    }
    public void run() {
        while(countDown-- > 0) {
            System.out.print(status());
            Thread.yield();
        }
    }
} //:-

```

El identificador **id** distingue entre múltiples instancias de la tarea. Es de tipo **final** porque no se espera que cambie una vez que ha sido inicializado.

El método **run()** de una tarea suele tener algún tipo de bucle que continúa ejecutándose hasta que la tarea deja de ser necesaria, por lo que es preciso establecer la condición de salida de este bucle (una opción consiste simplemente en ejecutar una instrucción **return** desde **run()**). A menudo, **run()** se implementa en la forma de un bucle infinito, lo que quiere decir que si no aparece un factor que haga que **run()** termine, este método continuará ejecutándose para siempre (posteriormente en el capítulo veremos cómo terminar las tareas de manera segura).

La llamada al método estático **Thread.yield()** dentro de **run()** es una sugerencia para el *planificador de hebras* (la parte del mecanismo de hebras de Java que conmuta el procesador de una hebra a la siguiente). Dicha sugerencia dice: “Acabo de finalizar las partes importantes de mi ciclo y este sería un buen momento para conmutar a otra tarea durante un rato”. Es completamente opcional, pero utilizamos dicho método aquí porque tiende a producir una salida más interesante en estos tipos de ejemplo: tenemos más probabilidades de ver cómo se cambia de unas tareas a otras.

En el siguiente ejemplo, el método **run()** de la tarea no está dirigido por una hebra separada, sino que simplemente se le invoca desde **main()** (en realidad, *sí* que estamos usando una hebra: la que siempre se asigna a **main()**):

```

//: concurrency/MainThread.java

public class MainThread {
    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run();
    }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
*//:-

```

Cuando derivamos una clase de **Runnable**, debe tener un método **run()**, pero esto no tiene nada de especial: no produce ninguna capacidad innata de gestión de hebras. Para conseguir disponer del mecanismo de hebras tenemos que asociar explícitamente una tarea a una hebra.

La clase Thread

La forma tradicional de transformar un objeto **Runnable** en una tarea funcional consiste en entregárselo a un constructor **Thread** (hebra). Este ejemplo muestra cómo asignar una hebra a un objeto **LiftOff** utilizando un objeto **Thread**:

```

//: concurrency/BasicThreads.java
// El uso más básico de la clase Thread.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for Liftoff");
    }
} /* Output: (90% match)
Waiting for Liftoff
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
*//:-

```

Un constructor **Thread** sólo necesita un objeto **Runnable**. Al invocar el método **start()** de un objeto **Thread** se realizará la inicialización necesaria para la hebra y a continuación se invocará el método **run()** del objeto **Runnable** para iniciar la tarea dentro de la nueva hebra.

Aún cuando **start()** parezca estar haciendo una llamada a un método de larga duración, podemos ver a la salida (el mensaje “Waiting for LiftOff” aparece antes de completarse la cuenta atrás) que **start()** vuelve rápidamente. En la práctica, hemos hecho una llamada al método **LiftOff.run()**, y dicho método no ha finalizado todavía, pero como **LiftOff.run()** está siendo ejecutado por una hebra distinta, podemos continuar realizando otras operaciones en la hebra **main()** (esta capacidad no está restringida a la hebra **main()**: cualquier hebra puede iniciar otra hebra). Así, el programa está ejecutando dos métodos a la vez: **main()** y **LiftOff.run()**. El método **run()** es el código que se ejecuta “simultáneamente” con las otras hebras del programa.

Podemos añadir fácilmente más hebras para controlar más tareas. A continuación podemos ver cómo todas las tareas se ejecutan de manera concertada:⁵

```
//: concurrency/MoreBasicThreads.java
// Adición de más hebras.

public class MoreBasicThreads {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Thread(new LiftOff()).start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (Sample)
Waiting for LiftOff
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:-
```

La salida muestra que la ejecución de las diferentes tareas está entremezclada a medida que se comuta de una tarea a otra. El planificador de hebras controla esta commutación de forma automática. Si tenemos múltiples procesadores en la máquina, el procesador de hebras distribuirá de manera transparente las hebras entre los distintos procesadores.⁶

La salida de este programa será diferente en cada ejecución, porque el mecanismo de planificación de hebras no es determinístico. De hecho, podemos ver enormes diferencias en la salida de este programa en una versión del JDK y en la siguiente. Por ejemplo, una versión anterior del JDK no efectuaba demasiado a menudo la commutación de hebras, por lo que la hebra 1 podía recorrer todas las pasadas del bucle hasta terminar, luego la hebra 2 completaría todas las pasadas de su bucle, etc. En la práctica, esto equivalía a llamar a una rutina que realizaría todos los bucles de manera consecutiva, salvo porque el iniciar todas esas hebras resulta bastante más costoso. Las versiones anteriores del JDK parecen exhibir un mejor comportamiento de asignación de franjas temporales, con lo que cada hebra parece recibir un servicio más regular. Generalmente, estos tipos de cambios de comportamiento en el JDK no son mencionados por Sun, así que no podemos basar nuestros planes en ninguna previsión coherente relativa al comportamiento del mecanismo de hebras. La mejor solución consiste en ser lo más conservador posible a la hora de escribir código basado en hebras.

Cuando **main()** crea los objetos **Thread**, no está capturando las referencias de ninguno de ellos. Con un objeto normal, esto haría que el objeto fuera candidato para la depuración de memoria, pero eso no sucede así con los objetos **Thread**. Cada objeto **Thread** “se registra” a sí mismo, por lo que existe de hecho una referencia a ese objeto en algún lugar, y el depurador de memoria no puede borrar el objeto hasta que la tarea salga de su método **run()** y termine. Podemos ver, analizando

⁵ En este caso, una única hebra (**main()**) está creando todas las hebras **LiftOff**. Sin embargo, si tenemos múltiples hebras creando hebras **LiftOff** es posible que más de una hebra **LiftOff** tenga el mismo valor **id**. Posteriormente en el capítulo veremos cuál es la razón.

⁶ Esto no era así en algunas de las versiones anteriores de Java.

la salida, que todas las tareas se ejecutan efectivamente hasta su conclusión, por lo que una hebra crea una hebra de ejecución separada que persiste después de que se complete la llamada a `start()`.

Ejercicio 1: (2) Implemente una clase **Runnable**. Dentro de `run()`, imprima un mensaje y luego invoque `yield()`. Repita este proceso tres veces, y luego vuelva desde `run()`. Ponga un mensaje de inicio en el constructor y un mensaje de terminación cuando la tarea termine. Cree varias de estas tareas y contórelas utilizando hebras.

Ejercicio 2: (2) Siguiendo la forma de `generics/Fibonacci.java`, cree una tarea que genere una secuencia de **n** números de Fibonacci, donde **n** sea un parámetro proporcionado al constructor de la tarea. Cree varias de estas tareas y contórelas mediante hebras.

Utilización de Executor

Los *Ejecutores* `java.util.concurrent` de Java SE5 simplifican la programación concurrente, encargándose de gestionar los objeto **Thread** por nosotros. Los ejecutores proporcionan un nivel de indirección entre un cliente y la ejecución de una tarea, en lugar de ejecutar una tarea directamente, hay un objeto intermedio que se encarga de ejecutar la tarea. Los ejecutores permiten gestionar la ejecución de tareas asíncronas sin tener que gestionar de manera explícita el ciclo de vida de las hebras. Los ejecutores son el método preferido de inicio de tareas en Java SE5/6.

Podemos utilizar un ejecutor (**Executor**) en lugar de crear explícitamente objetos **Thread** en `MoreBasicThreads.java`. Un objeto **LiftOff** sabe cómo ejecutar una tarea específica; al igual que el patrón de diseño *Comando*, expone un único método para ser ejecutado. Un objeto **ExecutorService** (un objeto **Executor** con un ciclo de vida de servicio, por ejemplo, apagar) sabe cómo construir el contexto apropiado para ejecutar objetos **Runnable**. En el siguiente ejemplo, el objeto **CachedThreadPool** crea una hebra por cada tarea. Observe que se crea un objeto **ExecutorService** utilizando un método **Executors** estático que determina el tipo de objeto **Executor** que tiene que ser:

```
//: concurrency/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:-
```

A menudo, puede utilizarse un único **Executor** para crear y gestionar todas las tareas del sistema.

La llamada a `shutdown()` impide que se envíen nuevas tareas a ese objeto **Executor**. La hebra actual (en este caso, la que controla `main()`) continuará ejecutando todas las tareas enviadas antes de `shutdown()`. El programa finalizará en cuanto finalice todas las tareas del objeto **Executor**.

Podemos sustituir fácilmente el objeto **CachedThreadPool** del ejemplo anterior por un tipo diferente de **Executor**. Un objeto **FixedThreadPool** utiliza un conjunto limitado de hebras para ejecutar las tareas indicadas:

```
//: concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
```

```

public static void main(String[] args) {
    // El argumento del constructor es el número de tareas:
    ExecutorService exec = Executors.newFixedThreadPool(5);
    for(int i = 0; i < 5; i++)
        exec.execute(new LiftOff());
    exec.shutdown();
}
} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:-
```

Con **FixedThreadPool**, realizamos la costosa asignación de hebras una única vez, por adelantado, con lo que limitamos el número de hebras. Esto permite ahorrar tiempo, porque no tenemos que pagar constantemente el coste asociado a la creación de una hebra para cada tarea individual. Asimismo, en un sistema dirigido por sucesos, las rutinas de tratamiento de sucesos que requieren hebras pueden ser servidas con la rapidez que queramos, extrayendo simplemente hebras de ese conjunto preasignado. Con esta solución, no podemos agotar los recursos disponibles porque el objeto **FixedThreadPool** utiliza un número limitado de objetos **Thread**.

Observe que en cualquiera de los dos tipos de conjuntos de hebras que hemos examinado, las hebras existentes se reutilizan de manera automática siempre que sea posible.

Aunque en este libro utilizaremos conjuntos de hebras de tipo **CachedThreadPool**, también puede considerar utilizar **FixedThreadPool** en el código de producción. **CachedThreadPool** creará generalmente tantas hebras como necesite durante la ejecución de un programa y luego dejará de crear nuevas hebras a medida que vaya pudiendo reciclar las antiguas, por lo que resulta razonable elegir en primer lugar este tipo de objeto **Executor**. Sólo si esta técnica causa problemas necesitaremos cambiar a un conjunto de hebras de tipo **FixedThreadPool**.

Un ejecutor **SingleThreadExecutor** es como **FixedThreadPool** pero con un tamaño de una única hebra.⁷ Esto resulta útil para cualquier cosa que queramos ejecutar de manera continua en otra hebra (una tarea de larga duración), como por ejemplo una tarea que se dedique a escuchar a las conexiones *socket* entrantes. También es útil para tareas de corta duración que queramos ejecutar dentro de una hebra, por ejemplo, un registro de sucesos local o remoto, o también para una hebra que se emplee para despachar sucesos.

Si se envía más de una tarea a un ejecutor **SingleThreadExecutor**, las tareas se pondrán en cola y cada una de ellas se ejecutará hasta completarse antes de que se inicie la siguiente tarea, utilizando todas ellas la misma hebra. En el siguiente ejemplo, podemos ver cómo cada tarea se completa en el orden en que fue enviada antes de que dé comienzo la siguiente. Así, un ejecutor **SingleThreadExecutor** serializa las tareas que se le envían y mantiene su propia cola (oculta) de tareas pendientes.

```

//: concurrency/SingleThreadExecutor.java
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output:
```

⁷ También ofrece una importante garantía de concurrencia que los otros tipos de ejecutores no ofrecen: no se pueden invocar concurrentemente dos tareas. Esto hace que cambien los requisitos de bloqueo de las tareas (hablaremos sobre el bloqueo más adelante en el capítulo).

```

#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(Liftoff!), #1(9), #1(8), #1(7), #1(6), #1(5),
#1(4), #1(3), #1(2), #1(1), #1(Liftoff!), #2(9), #2(8),
#2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1),
#2(Liftoff!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4),
#3(3), #3(2), #3(1), #3(Liftoff!), #4(9), #4(8), #4(7),
#4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Liftoff!),
*///:-

```

Veamos otro ejemplo. Suponga que tenemos una serie de hebras que están controlando tareas que utilizan el sistema de archivos. Podemos ejecutar estas tareas con **SingleThreadExecutor** para garantizar que en cada momento sólo haya una tarea ejecutándose en cualquier hebra. De esta forma, no tenemos que preocuparnos de la sincronización en lo que respecta al recurso compartido (y además no colapsaremos el sistema de archivos). En ocasiones, una mejor solución consiste en sincronizarse con el recurso (de lo que hablaremos más adelante en el capítulo). Pero **SingleThreadExecutor** nos permite obviar los problemas de coordinación a la hora, por ejemplo, de construir el prototipo de un sistema. Serializando las tareas, podemos eliminar la necesidad de serializar los objetos.

Ejercicio 3: (1) Repita el Ejercicio 1 utilizando los diferentes tipos de ejecutores mostrados en esta sección.

Ejercicio 4: (1) Repita el Ejercicio 2 utilizando los diferentes tipos de ejecutores mostrados en esta sección.

Producción de valores de retorno de las tareas

Un objeto **Runnable** es una tarea independiente que realiza un cierto trabajo, pero que no devuelve un valor. Si queremos que la tarea devuelva un valor cuando finalice, podemos implementar la interfaz **Callable** en lugar de la interfaz **Runnable**. **Callable**, introducida en Java SE5, es un genérico con un parámetro de tipo que representa el valor de retorno del método **call()** (en lugar de **run()**), y debe invocarse utilizando un método **submit()** de **ExecutorService**. He aquí un ejemplo simple:

```

//: concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }
    public String call() {
        return "result of TaskWithResult " + id;
    }
}

public class CallableDemo {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results =
            new ArrayList<Future<String>>();
        for(int i = 0; i < 10; i++)
            results.add(exec.submit(new TaskWithResult(i)));
        for(Future<String> fs : results)
            try {
                // get() se bloquea hasta completarse:
                System.out.println(fs.get());
            } catch(InterruptedIOException e) {
                System.out.println(e);
                return;
            } catch(ExecutionException e) {
                System.out.println(e);
            } finally {

```

```

        exec.shutdown();
    }
}
} /* Output:
result of TaskWithResult 0
result of TaskWithResult 1
result of TaskWithResult 2
result of TaskWithResult 3
result of TaskWithResult 4
result of TaskWithResult 5
result of TaskWithResult 6
result of TaskWithResult 7
result of TaskWithResult 8
result of TaskWithResult 9
*///:~

```

El método **submit()** produce un objeto **Future**, parametrizado para el tipo particular de resultado devuelto por el objeto **Callable**. Podemos consultar el objeto **Future** con **isDone()** para ver si se ha completado. Cuando la tarea se ha completado y dispone de un resultado, podemos invocar **get()** para extraer éste. También podemos simplemente invocar **get()** sin comprobar **isDone()**, en cuyo caso **get()** se bloqueará hasta que el resultado esté listo. Podemos asimismo invocar **get()** con una temporización, o invocar **isDone()** para ver si la tarea se ha completado, antes de tratar de llamar a **get()** para extraer el resultado.

El método sobrecargado **Executors.callable()** toma un objeto **Runnable** y produce un objeto **Callable**. **ExecutorService** tiene algunos métodos de “invocación” que ejecutan colecciones de objetos **Callable**.

Ejercicio 5: (2) Modifique el Ejercicio 2 de modo que la tarea sea un objeto **Callable** que sume los valores de todos los número de Fibonacci. Cree varias tareas y muestre los resultados.

Cómo dormir una tarea

Una forma simple de modificar el comportamiento de las tareas consiste en invocar **sleep()** para detener (bloquear) la ejecución de dicha tarea durante un cierto tiempo. En la clase **LiftOff**, si sustituimos la llamada a **yield()** por una llamada a **sleep()**, obtenemos lo siguiente:

```

//: concurrency/SleepingTask.java
// Llamada a sleep() para detenerse durante un tiempo.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
    public void run() {
        try {
            while(countDown-- > 0) {
                System.out.print(status());
                // A la antigua usanza:
                // Thread.sleep(100);
                // Al estilo Java SE5/6:
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch(InterruptedException e) {
            System.err.println("Interrupted");
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new SleepingTask());
        exec.shutdown();
    }
} /* Output:

```

```

#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:-
```

La llamada a `sleep()` puede generar una excepción `InterruptedException`, y como podemos ver, esta excepción se atrapa en `run()`. Puesto que las excepciones no se propagan entre unas hebras y otras para volver a `main()`, es necesario gestionar de manera local dentro de cada tarea todas las excepciones que puedan generarse.

Java SE5 ha introducido la versión más explícita de `sleep()` como parte de la clase `TimeUnit`, como se muestra en el ejemplo anterior. Esto proporciona una mayor legibilidad, al permitirnos especificar las unidades del retardo asociado con `sleep()`. `TimeUnit` también puede usarse para realizar conversiones, como veremos más adelante en el capítulo.

Dependiendo de la plataforma, podríamos observar que las tareas se ejecutan en orden “perfectamente distribuido”: cero a cuatro y luego vuelta de nuevo a cero. Esto tiene bastante sentido, porque después de cada instrucción de impresión cada tarea pasa a dormir (se bloquea), lo que permite al planificador de hebras conmutar a otra hebra distinta, que se encarga de dirigir otra tarea. Sin embargo, el comportamiento secuencial descansa sobre el mecanismo subyacente de hebras, que es diferente entre un sistema y otro, así que no podemos confiar en que las cosas sean siempre así. Si necesitamos controlar el orden de ejecución de las tareas, lo mejor que podemos hacer es emplear controles de sincronización (descritos más adelante) o, en algunos casos, no utilizar hebras en absoluto, sino en su lugar escribir nuestras propias rutinas cooperativas que se entreguen unas a otras el control, en un orden especificado.

Ejercicio 6: (2) Cree una tarea que duerma durante una cantidad aleatoria de tiempo comprendida entre 1 y 10 segundos, y que luego muestre el tiempo durante el que ha estado dormida y salga. Cree y ejecute un cierto número (indicado en la línea de comandos) de estas tareas.

Prioridad

La *prioridad* de una hebra le indica al planificador la importancia de esa hebra. Aunque el orden en que el procesador ejecuta una serie de hebras es indeterminado, el planificador tenderá a ejecutar primero la hebra en espera que tenga la mayor prioridad. Sin embargo, esto no significa que las hebras con una menor prioridad no se ejecuten (así que es imposible que se produzca un interbloqueo debido a las prioridades). Simplemente, las hebras de menor prioridad tienden a ejecutarse menos a menudo.

La inmensa mayoría de las veces, todas las hebras deberían ejecutarse con la prioridad predeterminada. Tratar de manipular las prioridades de las hebras suele ser un error.

He aquí un ejemplo que ilustra los niveles de prioridad. Podemos leer la prioridad de una hebra existente con `getPriority()` y cambiarla en cualquier momento con `setPriority()`.

```

//: concurrency/SimplePriorities.java
// Muestra el uso de las prioridades de las hebras.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
    private int countDown = 5;
    private volatile double d; // Sin optimización
    private int priority;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
    public String toString() {
        return Thread.currentThread() + ":" + countDown;
    }
    public void run() {
        Thread.currentThread().setPriority(priority);
    }
}
```

```

        while(true) {
            // Una operación costosa, interrumpible:
            for(int i = 1; i < 100000; i++) {
                d += (Math.PI + Math.E) / (double)i;
                if(i % 1000 == 0)
                    Thread.yield();
            }
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }

public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
        exec.execute(
            new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(
        new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown();
}

} /* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:-

```

toString() se sustituye para utilizar **Thread.toString()**, que imprime el nombre de la hebra, el nivel de prioridad y el “grupo de hebras” al que la hebra pertenece. Podemos establecer el nombre de las hebras nosotros mismos a través del constructor; aquí se generan automáticamente como **pool-1-thread-1**, **pool-1-thread-2**, etc. El método **toString()** sustituido también muestra el valor de cuenta atrás de la tarea. Observe que podemos obtener, dentro de una tarea, una referencia al objeto **Thread** que está dirigiendo la tarea, invocando **Thread.currentThread()**.

Podemos ver que el nivel de prioridad de la última hebra es el más alto, y que el resto de las hebras tienen el nivel más bajo. Observe que la prioridad se fija al comienzo de **run()**; fijar la prioridad en el constructor no sería adecuado, ya que el objeto **Executor** no ha comenzado la tarea en dicho instante.

Dentro de **run()**, se realizan 100.000 repeticiones de un cálculo en coma flotante bastante costoso, que implica la suma y división de valores **double**. La variable **d** es de tipo **volatile** para tratar de garantizar que no se realicen optimizaciones del compilador. Sin este cálculo, no podríamos ver el efecto de establecer los niveles de prioridad (inténtelo: desactive mediante un comentario el bucle **for** que contiene los cálculos de tipo **double**). Con el cálculo, podemos ver que la hebra con **MAX_PRIORITY** recibe una preferencia mayor por parte del planificador de hebras (al menos, éste era el comportamiento en la máquina Windows XP). Aún cuando imprimir en la consola también representa un comportamiento relativamente costoso, no es posible percibir los niveles de prioridad de esa forma, porque la impresión en la consola no puede verse interrumpida (en caso contrario, la visualización en la consola mostraría mensajes entremezclados al emplear hebras), mientras que los cálculos matemáticos sí que se pueden interrumpir. Los cálculos duran lo suficiente como para que el mecanismo de planificación intervenga, commute dos tareas y preste atención a las prioridades, de modo que las hebras de alta prioridad obtienen preferencia. Sin embargo, para garantizar que se produzca un cambio de contexto, se invocan regularmente instrucciones **yield()**.

Aunque el JDK tiene 10 niveles de prioridad, este número no se corresponde excesivamente bien con los mecanismos utilizados por muchos sistemas operativos. Por ejemplo, Windows tiene 7 niveles de prioridad que no son fijos, por lo que esa

correspondencia es indeterminada en el caso de este sistema operativo. El sistema Solaris de Sun tiene 2^{31} niveles. El único enfoque portable consiste en limitarse a utilizar **MAX_PRIORITY**, **NORM_PRIORITY** y **MIN_PRIORITY** a la hora de ajustar los niveles de prioridad.

Cesión del control

Si sabemos que ya hemos llevado a cabo lo que queríamos hacer durante la pasada de un bucle en nuestro método **run()**, podemos proporcionar una indicación al mecanismo de planificación de hebras, en el sentido de que ya hemos realizado una tarea suficiente y que puede permitirse a otra tarea disponer del procesador. Esta indicación (*y es una indicación: no hay ninguna garantía de que una implementación concreta respete esa indicación*) toma la forma de una llamada al método **yield()**. Cuando invocamos **yield()**, estamos sugiriendo que se pueden ejecutar otras hebras *de la misma prioridad*.

LiftOff.java utiliza **yield()** para distribuir de manera adecuada el procesamiento entre las diversas tareas **LiftOff**. Pruebe a desactivar mediante comentarios la llamada a **Thread.yield()** en **LiftOff.run()** para ver la diferencia. Sin embargo, en general, no podemos confiar en **yield()** para ninguna tarea seria de control o de optimización de la aplicación. De hecho, **yield()** se emplea muy a menudo de manera incorrecta.

Hebras demonio

Una hebra “demonio” pretende proporcionar un servicio general en segundo plano mientras el programa se ejecuta, pero sin formar parte de la esencia del programa. Por tanto, cuando todas las hebras no demonio se completan, el programa se termina, terminando también durante el proceso todas las hebras demonio. A la inversa, si existe alguna hebra no demonio que todavía se esté ejecutando, el programa no puede terminar. Existe, por ejemplo, una hebra no demonio que ejecuta el método **main()**.

```
//: concurrency/SimpleDaemons.java
// Las hebras demonio no impiden que el programa termine.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < 10; i++) {
            Thread daemon = new Thread(new SimpleDaemons());
            daemon.setDaemon(true); // Hay que invocarla antes de start()
            daemon.start();
        }
        print("All daemons started");
        TimeUnit.MILLISECONDS.sleep(175);
    }
} /* Output: (Sample)
All daemons started
Thread[Thread-0,5,main] SimpleDaemons@530daa
Thread[Thread-1,5,main] SimpleDaemons@a62fc3
Thread[Thread-2,5,main] SimpleDaemons@89ae9e
Thread[Thread-3,5,main] SimpleDaemons@1270b73
Thread[Thread-4,5,main] SimpleDaemons@60aeb0
```

```

Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
*///:~

```

Debemos definir la hebra como demonio invocando **setDaemon()** antes de iniciarla.

No hay nada que impida al programa terminar una vez que **main()** finaliza su trabajo, ya que lo único que queda ejecutándose son hebras demonio. Para poder ver el resultado de iniciar todas las hebras demonio, la hebra **main()** se pone brevemente a dormir. Sin esto, sólo veríamos parte de los resultados de la creación de las hebras demonio (pruebe a realizar llamadas a **sleep()** de diversas duraciones para ver este comportamiento).

SimpleDaemons.java crea objetos **Thread** explícitos para poder activar el indicador que los define como hebras demonio. Se pueden personalizar los atributos (demonio, prioridad, nombre) de las hebras creadas por objetos **Executor** escribiendo una factoría **ThreadFactory** personalizada:

```

//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setDaemon(true);
        return t;
    }
} //://:~

```

La única diferencia con respecto a un objeto **ThreadFactory** normal es que éste asigna el valor **true** al indicador que identifica las hebras demonio. Ahora podemos pasar un nuevo objeto **DaemonThreadFactory** como argumento a **Executors.newCachedThreadPool()**:

```

//: concurrency/DaemonFromFactory.java
// Utilización de una factoría de hebras para crear demonios.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch(InterruptedException e) {
            print("Interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool(
            new DaemonThreadFactory());
        for(int i = 0; i < 10; i++)
            exec.execute(new DaemonFromFactory());
        print("All daemons started");
        TimeUnit.MILLISECONDS.sleep(500); // Ejecutar durante un tiempo
    }
} /* (Ejecutar para ver la salida) *///:~

```

Cada uno de los métodos de creación estáticos `ExecutorService` se sobrecarga para tomar un objeto `ThreadFactory` que se utilizará para crear nuevas hebras.

Podemos llevar este enfoque un paso más allá y crear una utilidad `DaemonThreadPoolExecutor`:

```
//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor
extends ThreadPoolExecutor {
    public DaemonThreadPoolExecutor() {
        super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
              new SynchronousQueue<Runnable>(),
              new DaemonThreadFactory());
    }
} //:~
```

Para obtener los valores para llamada al constructor de la clase base, simplemente hemos echado un vistazo al código fuente `Executors.java`.

Podemos averiguar si una hebra es de tipo demonio invocando `isDaemon()`. Si una hebra es un demonio, entonces todas las hebras que cree serán también demonios automáticamente, como demuestra el siguiente ejemplo:

```
//: concurrency/Daemons.java
// Las hebras demonio crean otras hebras demonio.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Daemon implements Runnable {
    private Thread[] t = new Thread[10];
    public void run() {
        for(int i = 0; i < t.length; i++) {
            t[i] = new Thread(new DaemonSpawn());
            t[i].start();
            println("DaemonSpawn " + i + " started, ");
        }
        for(int i = 0; i < t.length; i++)
            println("t[" + i + "].isDaemon() = " +
                   t[i].isDaemon() + ", ");
        while(true)
            Thread.yield();
    }
}

class DaemonSpawn implements Runnable {
    public void run() {
        while(true)
            Thread.yield();
    }
}

public class Daemons {
    public static void main(String[] args) throws Exception {
        Thread d = new Thread(new Daemon());
        d.setDaemon(true);
        d.start();
        println("d.isDaemon() = " + d.isDaemon() + ", ");
        // Permitir que las hebras demonio finalicen
        // sus procesos de arranque:
        TimeUnit.SECONDS.sleep(1);
    }
}
```

```

        }
} /* Output: (Sample)
d.isDaemon() = true, DaemonSpawn 0 started, DaemonSpawn 1
started, DaemonSpawn 2 started, DaemonSpawn 3 started,
DaemonSpawn 4 started, DaemonSpawn 5 started, DaemonSpawn 6
started, DaemonSpawn 7 started, DaemonSpawn 8 started,
DaemonSpawn 9 started, t[0].isDaemon() = true,
t[1].isDaemon() = true, t[2].isDaemon() = true,
t[3].isDaemon() = true, t[4].isDaemon() = true,
t[5].isDaemon() = true, t[6].isDaemon() = true,
t[7].isDaemon() = true, t[8].isDaemon() = true,
t[9].isDaemon() = true,
*///:-
```

La hebra **Daemon** se configura en modo demonio. A continuación, esa hebra inicia una serie de otras hebras (que *no se definen explícitamente* como demonios), para demostrar de todos modos que esas hebras serán de tipo demonio. A continuación, **Daemon** entra en un bucle infinito que llama a **yield()** para ceder el control a los otros procesos.

Es necesario tener en cuenta que las hebras demonio terminarán sus métodos **run()** sin ejecutar cláusulas **finally**:

```

//: concurrency/DaemonsDontRunFinally.java
// Las hebras demonio no ejecutan la cláusula finally
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADaemon implements Runnable {
    public void run() {
        try {
            print("Starting ADaemon");
            TimeUnit.SECONDS.sleep(1);
        } catch(InterruptedException e) {
            print("Exiting via InterruptedException");
        } finally {
            print("This should always run?");
        }
    }
}

public class DaemonsDontRunFinally {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new ADaemon());
        t.setDaemon(true);
        t.start();
    }
} /* Output:
Starting ADaemon
*///:-
```

Cuando ejecutamos este programa, vemos que la cláusula **finally** no se ejecuta, pero si desactivamos mediante comentarios la llamada a **setDaemon()**, la cláusula **finally** *sí* que se ejecutará.

Este comportamiento es correcto, aún cuando resulte algo inesperado, teniendo en cuenta las explicaciones dadas antes para **finally**. Los demonios se terminan “abruptamente” cuando termina la última de las hebras no demonio. Por tanto, en cuanto salimos de **main()**, la máquina JVM termina todos los demonios inmediatamente sin ninguna de las formalidades que cabría esperar. Dado que no se pueden finalizar los demonios de una manera limpia, las hebras demonio no suelen ser muy convenientes. Generalmente, resulta mejor emplear objetos **Executor** no demonio, ya que todas las tareas controladas por un objeto **Executor** pueden terminarse de una sola vez. Como veremos posteriormente en el capítulo, esa terminación tiene lugar, en este caso, de una manera ordenada.

Ejercicio 7: (2) Experimente con diferentes tiempos de dormir en **Daemons.java**, para ver lo que sucede.

Ejercicio 8: (1) Modifique **MoreBasicThreads.java** para que todas las hebras sean de tipo demonio y el programa termine en cuanto **main()** sea capaz de terminar.

Ejercicio 9: (3) Modifique **SimplePriorities.java** para que una factoría personalizada **ThreadFactory** establezca las prioridades de todas las hebras.

Variaciones de código

En los ejemplos que hemos visto hasta ahora, todas las clases de tareas implementan la interfaz **Runnable**. En algunos casos muy simples, podemos utilizar la técnica alternativa de heredar directamente de **Thread**, como en este ejemplo:

```
//: concurrency/SimpleThread.java
// Herencia directa de la clase Thread.

public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        // Almacenar el nombre de la hebra:
        super(Integer.toString(++threadCount));
        start();
    }
    public String toString() {
        return "#" + getName() + "(" + countDown + ")";
    }
    public void run() {
        while(true) {
            System.out.print(this);
            if(--countDown == 0)
                return;
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new SimpleThread();
    }
} /* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3),
#2(2), #2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5),
#4(4), #4(3), #4(2), #4(1), #5(5), #5(4), #5(3), #5(2),
#5(1),
*///:-
```

Proporcionamos a los objetos **Thread** nombres específicos invocando el constructor **Thread** apropiado. Este nombre se extrae en **toString()** utilizando **getName()**.

Otra estructura de código que puede que se encuentre alguna vez es la del objeto **Runnable** auto-gestionado:

```
//: concurrency/SelfManaged.java
// Un objeto Runnable que contiene su propia hebra directora.

public class SelfManaged implements Runnable {
    private int countDown = 5;
    private Thread t = new Thread(this);
    public SelfManaged() { t.start(); }
    public String toString() {
        return Thread.currentThread().getName() +
               "(" + countDown + ")";
    }
    public void run() {
        while(true) {
```

```

        System.out.print(this);
        if(--countDown == 0)
            return;
    }
}
public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
        new SelfManaged();
}
/* Output:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-0(1),
Thread-1(5), Thread-1(4), Thread-1(3), Thread-1(2), Thread-1(1),
Thread-2(5), Thread-2(4), Thread-2(3), Thread-2(2), Thread-2(1),
Thread-3(5), Thread-3(4), Thread-3(3), Thread-3(2), Thread-3(1),
Thread-4(5), Thread-4(4), Thread-4(3), Thread-4(2), Thread-4(1),
*//*:-

```

Esta técnica no difiere especialmente de la de heredar de **Thread**, salvo porque la sintaxis es ligeramente más abstrusa. Sin embargo, implementar una interfaz nos permite heredar de una clase distinta, cosa que no se puede hacer si heredamos desde **Thread**.

Observe que **start()** se invoca dentro del constructor. Este ejemplo es bastante simple y resulta, por tanto, probablemente seguro, pero hay que tener en cuenta que iniciar hebras dentro de un constructor puede resultar bastante problemático, porque otra tarea podría empezar a ejecutarse antes de que el constructor se haya completado, lo que quiere decir que la tarea pudiera ser capaz de acceder al objeto en un estado inestable. Ésta es otra razón adicional de preferir objetos **Executor** a la creación explícita de objetos **Thread**.

En ocasiones, resulta conveniente ocultar el código de gestión de hebras dentro de la clase utilizando una clase interna, como se muestra a continuación:

```

//: concurrency/ThreadVariations.java
// Creación de hebras con clases internas.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Utilización de una clase interna nominada:
class InnerThread1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner extends Thread {
        Inner(String name) {
            super(name);
            start();
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    sleep(10);
                }
            } catch(InterruptedException e) {
                print("interrupted");
            }
        }
        public String toString() {
            return getName() + ": " + countDown;
        }
    }
    public InnerThread1(String name) {
        inner = new Inner(name);
    }
}

```

```

        }
    }

    // Utilización de una clase interna anónima:
    class InnerThread2 {
        private int countDown = 5;
        private Thread t;
        public InnerThread2(String name) {
            t = new Thread(name);
            public void run() {
                try {
                    while(true) {
                        print(this);
                        if(--countDown == 0) return;
                        sleep(10);
                    }
                } catch(InterruptedException e) {
                    print("sleep() interrupted");
                }
            }
            public String toString() {
                return getName() + ":" + countDown;
            }
        };
        t.start();
    }
}

// Utilización de una implementación Runnable nominada:
class InnerRunnable1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner implements Runnable {
        Thread t;
        Inner(String name) {
            t = new Thread(this, name);
            t.start();
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch(InterruptedException e) {
                print("sleep() interrupted");
            }
        }
        public String toString() {
            return t.getName() + ":" + countDown;
        }
    }
    public InnerRunnable1(String name) {
        inner = new Inner(name);
    }
}

// Utilización de una implementación Runnable anónima:
class InnerRunnable2 {
    private int countDown = 5;

```

```

private Thread t;
public InnerRunnable2(String name) {
    t = new Thread(new Runnable() {
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch(InterruptedException e) {
                print("sleep() interrupted");
            }
        }
        public String toString() {
            return Thread.currentThread().getName() +
                ":" + countDown;
        }
    }, name);
    t.start();
}
}

// Un método separado para ejecutar un cierto código como una tarea:
class ThreadMethod {
    private int countDown = 5;
    private Thread t;
    private String name;
    public ThreadMethod(String name) { this.name = name; }
    public void runTask() {
        if(t == null) {
            t = new Thread(name) {
                public void run() {
                    try {
                        while(true) {
                            print(this);
                            if(--countDown == 0) return;
                            sleep(10);
                        }
                    } catch(InterruptedException e) {
                        print("sleep() interrupted");
                    }
                }
                public String toString() {
                    return getName() + ":" + countDown;
                }
            };
            t.start();
        }
    }
}

public class ThreadVariations {
    public static void main(String[] args) {
        new InnerThread1("InnerThread1");
        new InnerThread2("InnerThread2");
        new InnerRunnable1("InnerRunnable1");
        new InnerRunnable2("InnerRunnable2");
        new ThreadMethod("ThreadMethod").runTask();
    }
} /* (Ejecutar para ver la salida) */:-
```

InnerThread1 crea una clase interna nominada que amplía **Thread** y crea una instancia de esta clase interna dentro del constructor. Esto tiene sentido si la clase interna dispone de capacidades especiales (nuevos métodos) a los que necesitamos acceder desde otros métodos. Sin embargo, la mayor parte de las veces la razón para crear una hebra es únicamente utilizar las capacidades de la clase **Thread**, por lo que no es necesario crear una clase interna nominada. **InnerThread2** muestra cuál es la alternativa: dentro del constructor se crea una subclase interna anónima de **Thread** y se la generaliza a una referencia **t** a **Thread**. Si otros métodos de la clase necesitan acceder a **t**, pueden hacerlo a través de la interfaz **Thread**, y no necesitan conocer el tipo exacto del objeto.

La tercera y cuarta clases del ejemplo repiten las dos primeras clases, pero en lugar de utilizar la interfaz **Runnable** emplean la clase **Thread**.

La clase **ThreadMethod** muestra la creación de una hebra dentro de un método. Si invocamos el método una vez que estamos listos para ejecutar la hebra, el método termina antes de que la hebra dé comienzo. Si la hebra sólo está realizando una operación auxiliar en lugar de alguna otra cosa más fundamental para la clase, probablemente este enfoque resulte más útil y apropiado que iniciar una hebra dentro del constructor de la clase.

Ejercicio 10: (4) Modifique el Ejercicio 5 de acuerdo con el ejemplo de la clase **ThreadMethod**, de modo que **runTask()** tome un argumento que especifique la cantidad de números de Fibonacci que hay que sumar, y que cada vez que invoquemos **runTask()** devuelva el objeto **Future** producido por la llamada a **submit()**.

Terminología

Como muestra la sección anterior, existen diversas alternativas a la hora de implementar programas concurrentes en Java, y dichas alternativas pueden resultar confusas. A menudo, el problema procede de la terminología empleada a la hora de describir la tecnología de programas concurrentes, especialmente en aquellos casos que hay implicadas hebras.

A estas alturas, debería ya entender que existe una distinción entre la tarea que se está ejecutando y la hebra que la dirige; esta distinción resulta especialmente clara en las bibliotecas Java, porque realmente no tenemos ningún control sobre la clase **Thread** (y esta separación es todavía más clara con los ejecutores, que se encargan de crear y gestionar las hebras por nosotros). Lo que hacemos es crear una tarea y asociar una hebra con cada tarea, de modo que la hebra se encargue de dirigirla.

En Java, la clase **Thread** no hace nada por sí misma. Se limita a dirigir la tarea que se le indique. A pesar de ello, sobre los mecanismos de gestión de hebras, suele utilizar expresiones como “la hebra realiza esta acción o esta otra”. La impresión que se obtiene al leer esto es que la hebra *es* la tarea, y cuando yo tropecé con las hebras de Java, esta impresión era tan fuerte que para mí existía una relación de tipo “es-un” muy clara, y de lo cual yo deducía que era necesario heredar una tarea de un objeto **Thread**. Añadimos a esto la inadecuada elección del nombre de la interfaz **Runnable** (ejecutable), que debería haberse denominado, mucho más apropiadamente “**Task**” (tarea).

El problema es que los niveles de abstracción están mezclados. Conceptualmente, queremos crear una tarea que se ejecute independientemente de otras tareas, por lo que deberíamos poder definir una tarea y decir “ejecutar” sin preocuparnos de los detalles. Pero físicamente las hebras pueden resultar muy costosas de crear, por lo que es necesario conservarlas y gestionarlas adecuadamente. Por tanto, tiene sentido, *desde el punto de vista de la implementación*, separar las tareas de las hebras. Además, el mecanismo de hebras en Java está basado en la solución de *pthreads* de bajo nivel proveniente de C, que es una solución en la que es necesario sumergirse y en la que es preciso entender todos los detalles de lo que está ocurriendo. Parte de esta naturaleza de bajo nivel ha terminado deslizándose en la implementación Java, por lo que para permanecer en un nivel mayor de abstracción, es necesario ser disciplinado a la hora de escribir el código (trataremos de mostrar esa disciplina a lo largo del capítulo).

Para clarificar estas explicaciones, trataré de utilizar el término “tarea” cuando me refiera al trabajo que hay que realizar y “hebra” únicamente a la hora de referirme al mecanismo específico que se ocupa de dirigir la tarea. Por tanto, si estamos analizando un sistema en un nivel conceptual podemos limitarnos a emplear el término “tarea” sin necesidad de mencionar en absoluto el mecanismo encargado de dirigir esa tarea.

Absorción de una hebra

Una hebra puede invocar **join()** sobre otra hebra para esperar que esa segunda hebra se complete antes de que la primera continúe con su trabajo. Si una hebra invoca **t.join()** sobre otra hebra **t**, entonces la hebra invocante se suspende hasta que la hebra objetivo **t** finalice (cuando **t.isAlive()** es **false**).

También podemos invocar **join()** con un argumento de fin de temporización (en milisegundos o en milisegundos y nanosegundos), de modo que si la hebra objetivo no finaliza en dicho período de tiempo, la llamada a **join()** vuelve de todos modos.

La llamada a **join()** puede abortarse invocando **interrupt()** sobre la hebra invocante, para lo que hace falta una cláusula **try-catch**.

Todas estas operaciones se ilustran en el siguiente ejemplo:

```
//: concurrency/Joining.java
// Ejemplo de join().
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
    private int duration;
    public Sleeper(String name, int sleepTime) {
        super(name);
        duration = sleepTime;
        start();
    }
    public void run() {
        try {
            sleep(duration);
        } catch(InterruptedException e) {
            print(getName() + " was interrupted. " +
                  "isInterrupted(): " + isInterrupted());
            return;
        }
        print(getName() + " has awakened");
    }
}

class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();
    }
    public void run() {
        try {
            sleeper.join();
        } catch(InterruptedException e) {
            print("Interrupted");
        }
        print(getName() + " join completed");
    }
}

public class Joining {
    public static void main(String[] args) {
        Sleeper
            sleepy = new Sleeper("Sleepy", 1500),
            grumpy = new Sleeper("Grumpy", 1500);
        Joiner
            dopey = new Joiner("Dopey", sleepy),
            doc = new Joiner("Doc", grumpy);
        grumpy.interrupt();
    }
} /* Output:
```

```

Grumpy was interrupted. isInterrupted(): false
Doc join completed
Sleepy has awakened
Dopey join completed
*///:~

```

Un objeto **Sleeper** es una hebra que pasa a dormir durante un tiempo especificado en su constructor. En **run()**, la llamada a **sleep()** puede terminar cuando finaliza el tiempo especificado, pero también puede ser interrumpida. Dentro de la cláusula **catch**, se informa de la interrupción, junto con el valor de **isInterrupted()**. Cuando otra hebra invoca **interrupt()** sobre esta hebra, se activa un indicador para mostrar que la hebra ha sido interrumpida. Sin embargo, este indicador se borra en el momento de tratar la excepción, por lo que el resultado será siempre **false** dentro de la cláusula **catch**. El indicador se utiliza para otras situaciones en las que una hebra puede examinar su estado de interrupción, de forma independiente de la excepción.

Un objeto **Joiner** es una tarea para que un objeto **Sleeper** se despierte invocando **join()** sobre ese objeto **Sleeper**. En **main()**, cada objeto **Sleeper** tiene un objeto **Joiner** y podemos ver a la salida que si el objeto **Sleeper** es interrumpido o finaliza normalmente, el objeto **Joiner** completa su tarea en conjunción con el objeto **Sleeper**.

Observe que las bibliotecas **java.util.concurrent** de Java SE5 contienen herramientas tales como **CyclicBarrier** (que se ilustra más adelante en este capítulo) que pueden ser más apropiadas que **join()**, que formaba parte de la biblioteca de hebras original.

Creación de interfaces de usuario de respuesta rápida

Como hemos indicado anteriormente, uno de los motivos para la utilización de hebras consiste en crear una interfaz de usuario de rápida respuesta. Aunque no vamos a sumergirnos en las interfaces gráficas hasta el Capítulo 22, *Interfaces gráficas de usuario*, el siguiente ejemplo es un simple prototipo de una interfaz de usuario basada en consola. El ejemplo tiene dos versiones: una que se queda bloqueada en un cálculo y nunca puede leer la entrada de la consola y una segunda que inserta el cálculo dentro de una tarea y puede, por tanto, estar realizando a la vez el cálculo y escuchando la entrada de la consola.

```

//: concurrency/ResponsiveUI.java
// Capacidad de respuesta de la interfaz de usuario
// {RunByHand}

class UnresponsiveUI {
    private volatile double d = 1;
    public UnresponsiveUI() throws Exception {
        while(d > 0)
            d = d + (Math.PI + Math.E) / d;
        System.in.read(); // Nunca llega aquí
    }
}

public class ResponsiveUI extends Thread {
    private static volatile double d = 1;
    public ResponsiveUI() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
            d = d + (Math.PI + Math.E) / d;
        }
    }
    public static void main(String[] args) throws Exception {
        //! new UnresponsiveUI(); // Hay que matar este proceso
        new ResponsiveUI();
        System.in.read();
    }
}

```

```

        System.out.println(d); // Mostrar el progreso
    }
} //:~

```

UnresponsiveUI realiza un cálculo dentro de un bucle **while** infinito, por lo que nunca, obviamente, alcanza la línea de entrada de datos de la consola (hemos engañado al compilador para que piense que la línea de entrada es alcanzable utilizando el **while** condicional). Si desactivamos mediante un comentario la línea que crea una interfaz **UnresponsiveUI**, tendremos que matar el proceso para poder salir.

Para hacer que el programa tenga una adecuada capacidad de respuesta, hay que colocar el cálculo dentro de un método **run()** para permitir que se lo pueda desalojar del procesador, y cuando pulsemos la tecla Intro, veremos que el cálculo ha estado ejecutándose en segundo plano mientras se esperaba a que se produjera la entrada del usuario.

Grupos de hebras

Un *grupo de hebras* mantiene una colección de hebras. La ventaja de los grupos de hebras puede resumirse citando las palabras de Joshua Bloch,⁸ el arquitecto de software que, mientras estaba en Sun, corrigió y mejoró enormemente la biblioteca de colecciones de Java en el JDK 1.2 (entre otras contribuciones):

"Los grupos de hebras podrían definirse como un experimento que no tuvo éxito, así que se puede simplemente ignorar su existencia".

Si el lector ha invertido tiempo y esfuerzo tratando de comprender el valor de los grupos de hebras (como es mi caso), podría preguntarse por qué no se ha producido ningún anuncio más oficial de Sun acerca de este tema: la misma cuestión podría plantearse acerca de varios otros cambios que Java ha sufrido a lo largo de los años. El economista Joseph Stiglitz laureado con el Premio Nobel tiene una filosofía de la vida que podría aplicarse aquí.⁹ Se denomina *La teoría del compromiso delegado*:

"El coste de continuar con los errores es soportado por otros, mientras que el coste de admitirlos es soportado por nosotros."

Captura de excepciones

Debido a la naturaleza de las hebras no podemos capturar una excepción que haya escapado de una hebra. Una vez que una excepción sale del método **run()** de una tarea, no se propagará hacia la consola a menos que adoptemos medidas especiales para capturar esas excepciones "vagabundas". Antes de Java SE5, se utilizaban los grupos de hebras para capturar estas excepciones, pero con Java SE5 podemos resolver el problema mediante objetos **Executor**, por lo que deja de ser necesario saber *nada* acerca de los grupos de hebras (salvo para entender el código heredado, véase *Thinking in Java, 2ª Edición*, descargable de www.MindView.net, para conocer más detalles sobre los grupos de hebras).

He aquí una tarea que siempre genera una excepción que se propaga fuera de su método **run()** y un método **main()** que muestra lo que sucede al ejecutarlo:

```

//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} //:~

```

⁸ *Effective JavaTM Programming Language Guide*, de Joshua Bloch (Addison-Wesley, 2001), p. 211.

⁹ Y en varias otras ocasiones relacionadas con la utilización de Java. Bueno, en realidad, ¿por qué detener esto? En el pasado he prestado labores de consultoría en bastantes proyectos en los que esta filosofía era aplicable.

La salida es (después de quitar algunos cualificadores para que quepa en la página):

```
java.lang.RuntimeException
    at ExceptionThread.run(ExceptionThread.java:7)
    at ThreadPoolExecutor$Worker.runTask(Unknown Source)
    at ThreadPoolExecutor$Worker.run(Unknown Source)
    at java.lang.Thread.run(Unknown Source)
```

No se pierde nada al encerrar el cuerpo del método principal dentro un bloque **try-catch**:

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;

public class NaiveExceptionHandling {
    public static void main(String[] args) {
        try {
            ExecutorService exec =
                Executors.newCachedThreadPool();
            exec.execute(new ExceptionThread());
        } catch(RuntimeException ue) {
            // !Esta instrucción NO se ejecutará!
            System.out.println("Exception has been handled!");
        }
    }
} //:-
```

Esto produce el mismo resultado que el ejemplo anterior: una excepción no capturada.

Para resolver el problema, cambiemos la forma en que el objeto **Executor** produce las hebras. **Thread.UncaughtExceptionHandler** es una nueva interfaz en Java SE5; que nos permite asociar una rutina de tratamiento de excepciones a cada objeto **Thread**. **Thread.UncaughtExceptionHandler.uncaughtException()** se invoca automáticamente cuando esa hebra está a punto de morir debido a una excepción no capturada. Para usarla, creamos un nuevo tipo de factoría **ThreadFactory** que asocia un nuevo objeto **Thread.UncaughtExceptionHandler** a cada nuevo objeto **Thread** que crea. Pasamos dicha factoría al método **Executors** que crea un nuevo objeto **ExecutorService**:

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("run() by " + t);
        System.out.println(
            "eh = " + t.getUncaughtExceptionHandler());
        throw new RuntimeException();
    }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("caught " + e);
    }
}

class HandlerThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        System.out.println(this + " creating new Thread");
        Thread t = new Thread(r);
        System.out.println("created " + t);
        return t;
    }
}
```

```

        t.setUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        System.out.println(
            "eh = " + t.getUncaughtExceptionHandler());
        return t;
    }
}

public class CaptureUncaughtException {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool(
            new HandlerThreadFactory());
        exec.execute(new ExceptionThread2());
    }
} /* Output: (90% match)
HandlerThreadFactory@de6ced creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
run() by Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
caught java.lang.RuntimeException
*///:-

```

Hemos añadido facilidades de traza adicionales para verificar que las hebras creadas por la factoría reciben el nuevo objeto **UncaughtExceptionHandler**. Podemos ver que las excepciones no capturadas están siendo ahora capturadas **uncaughtException**.

El ejemplo anterior nos permite configurar la rutina de tratamiento caso por caso. Si sabemos que vamos a utilizar la misma rutina de tratamiento de excepciones en todas partes, una técnica todavía más simple consiste en definir la rutina *predeterminada* de tratamiento de excepciones no capturadas, que configura un campo estático dentro de la clase **Thread**:

```

//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} /* Output:
caught java.lang.RuntimeException
*///:-

```

Esta rutina de tratamiento sólo se invoca si no existe una rutina de tratamiento de excepciones no capturadas para la hebra concreta. El sistema comprueba si la hebra dispone de una rutina y, en caso contrario, mira a ver si el grupo de hebras especializa su método **uncaughtException()**; en caso contrario, invoca la rutina predeterminada **defaultUncaughtExceptionHandler**.

Compartición de recursos

Podemos pensar en un programa que tenga una sola hebra como si fuera una entidad solitaria que se mueve en nuestro espacio de problema y que hace una sola cosa cada vez. Puesto que sólo hay una entidad, no tenemos que pensar en el problema de que dos entidades intenten utilizar el mismo recurso al mismo tiempo: problemas que son similares al caso de dos personas que intentaran aparcar en el mismo sitio, que estuvieran tratando de pasar por una misma puerta al tiempo o que estuvieran incluso intentando hablar simultáneamente.

Con la concurrencia, no tienen por qué existir entidades solitarias, sino que tenemos la posibilidad de que haya dos o más tareas interfiriendo entre sí. Si no evitamos las colisiones, podemos encontrarnos con que las dos tareas traten de acceder a la misma cuenta corriente al mismo tiempo, imprimir en la misma impresora, ajustar la misma válvula, etc.

Acceso inapropiado a los recursos

Consideremos el siguiente ejemplo en el que una tarea genera números pares y otras tareas consumen dichos números. Aquí, el único trabajo de las tareas consumidoras consiste en comprobar la validez de los números pares.

En primer lugar, definimos **EvenChecker**, la tarea consumidora, ya que la vamos a reutilizar en todos los ejemplos subsiguientes. Para desacoplar **EvenChecker** de los varios tipos de generadores con los que vamos a estar experimentando, creamos una clase abstracta denominada **IntGenerator**, que contiene el mínimo número de métodos necesarios que **EvenChecker** debe utilizar: dispone de un método **next()** y el generador puede ser cancelado. Esta clase no implementa la interfaz **Generator**, porque debe generar un valor **int**, y los genéricos no soportan parámetros primitivos.

```
//: concurrency/IntGenerator.java
public abstract class IntGenerator {
    private volatile boolean canceled = false;
    public abstract int next();
    // Permitir que cancelarlo:
    public void cancel() { canceled = true; }
    public boolean isCanceled() { return canceled; }
} //:-~
```

IntGenerator tiene un método **cancel()** para cambiar el estado de un indicador booleano **canceled**, así como un método **isCanceled()** para ver si el objeto ha sido cancelado. Puesto que el indicador **canceled** es de tipo **boolean**, es *atómico*, lo que significa que las operaciones simples como la asignación y la devolución de un valor, tienen lugar sin que se puedan producir interrupciones, así que es posible ver ese campo en un estado intermedio durante la realización de esas operaciones simples. El indicador **canceled** también es de tipo **volatile**, con el fin de asegurar la *visibilidad*. Hablaremos más en detalle de la atomicidad y la visibilidad posteriormente en el capítulo.

Cualquier objeto **IntGenerator** puede ser probado con la siguiente clase **EvenChecker**:

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
    private IntGenerator generator;
    private final int id;
    public EvenChecker(IntGenerator g, int ident) {
        generator = g;
        id = ident;
    }
    public void run() {
        while(!generator.isCanceled()) {
            int val = generator.next();
            if(val % 2 != 0) {
                System.out.println(val + " not even!");
                generator.cancel(); // Cancela todos los objetos EvenChecker
            }
        }
    }
    // Probar cualquier tipo de IntGenerator:
    public static void test(IntGenerator gp, int count) {
        System.out.println("Press Control-C to exit");
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < count; i++)
            exec.execute(new EvenChecker(gp, i));
        exec.shutdown();
    }
    // Valor predeterminado de recuento:
    public static void test(IntGenerator gp) {
        test(gp, 10);
    }
} //:-~
```

Observe que en este ejemplo la clase que puede cancelarse no es de tipo **Runnable**. En su lugar, todas las tareas **EvenChecker** que dependen del objeto **IntGenerator** lo comprueban para ver si ha sido cancelado, como podemos ver en **run()**. De esta forma, las tareas que comparten un recurso común (el objeto **IntGenerator**) observan dicho recurso para ver la señal de terminación. Esto elimina las denominadas condiciones de carrera, en las que dos o más tareas compiten para responder a una condición y, por tanto, colisionan o producen de alguna otra manera resultados incoherentes. Es necesario pensar con cuidado todas las posibles formas en que un sistema concurrente puede fallar y protegerse frente a ellas. Por ejemplo, una tarea no puede depender de otra tarea porque el orden de terminación de las tareas no está garantizado. Aquí, haciendo que las tareas dependan de un objeto que no es una tarea, eliminamos esa potencial condición de carrera.

El método **test()** prepara y realiza una prueba de cualquier tipo de **IntGenerator**, iniciando una serie de objetos **EvenChecker** que usan el mismo objeto **IntGenerator**. Si **IntGenerator** provoca un fallo, **test()** informará de él y volverá. En caso contrario, es necesario pulsar Control-C para terminar el método.

Las tareas **EvenChecker** están constantemente leyendo y probando los valores de su objeto **IntGenerator** asociado. Observe que si **generator.isCanceled()** es **true**, **run()** vuelve, lo que dice al objeto **Executor** en **EvenChecker.test()** que la tarea está completa. Cualquier tarea **EvenChecker** puede invocar **cancel()** sobre su objeto **IntGenerator** asociado, lo que hará que todas las tareas **EvenChecker** que estén usando **IntGenerator** terminen de manera grácil. En secciones posteriores, veremos que Java ofrece mecanismos más generales que estos para la terminación de las hebras.

El primer objeto **IntGenerator** que vamos a examinar dispone de un método **next()** que produce una serie de valores pares:

```
//: concurrency/EvenGenerator.java
// Cuando las hebras colisionan.

public class EvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public int next() {
        ++currentEvenValue; // ¡Punto de peligro!
        ++currentEvenValue;
        return currentEvenValue;
    }

    public static void main(String[] args) {
        EvenChecker.test(new EvenGenerator());
    }
} /* Output: (Sample)
Press Control-C to exit
89476993 not even!
89476993 not even!
*///:~
```

Resulta posible que una tarea invoque **next()** después de que otra tarea haya realizado el primer incremento de **currentEvenValue** pero no el segundo (en el lugar del código que tiene el comentario “¡Punto de peligro!”). Esto hace que el valor quede en un estado “incorrecto”. Para probar que esto puede suceder, **EvenChecker.test()** crea un grupo de objetos **EvenChecker** para leer continuamente la salida de un objeto **EvenGenerator** y ver que cada valor es par. Si no lo es, se informa del error y el programa termina.

Este programa terminará por fallar, porque las tareas **EvenChecker** tienen que acceder a la información de **EvenGenerator** mientras que esa información está en un estado “incorrecto”. Sin embargo, puede que el problema no se detecte hasta que el objeto **EvenGenerator** haya completado muchos ciclos, dependiendo de las particularidades de nuestro sistema operativo y de otros detalles de implementación. Si queremos ver mucho más rápido cómo falla el programa, podemos incluir una llamada a **yield()** entre el primer y el segundo incremento. Esto es parte del problema con los programas multihebra: pueden parecer correctos a pesar de que existe un error, siempre y cuando la probabilidad de fallo sea baja.

Es importante comprobar que la propia operación de incremento requiere múltiples pasos y que la tarea puede ser suspendida por el mecanismo de gestión de hebras en mitad de una operación de incremento; en otras palabras, el incremento no es una operación átomica en Java. Así que, incluso no sería seguro realizar una operación de incremento sin proteger la correspondiente tarea.

Resolución de la contienda por los recursos compartidos

El ejemplo anterior muestra un problema fundamental a la hora de emplear hebras: nunca sabemos cuándo va a ejecutarse una hebra. Imagine que nos encontramos ante una mesa, sentados con un tenedor y a punto de tomar el último bocado de un plato, y que al acercar nuestro tenedor a ese bocado éste se desvanece súbitamente, porque nuestra hebra fue suspendida y otro comensal se adelantó y se comió ese bocado. Ése es el problema con el que estamos tratando cuando escribimos programas concurrentes. Para que la concurrencia funcione, necesitamos alguna forma de impedir que dos tareas accedan al mismo recurso simultáneamente, al menos durante los períodos críticos.

Evitar este tipo de colisión es cuestión, simplemente, de bloquear un recurso mientras que una tarea lo esté utilizando. La primera tarea que acceda a un recurso debe bloquearlo y entonces las otras tareas no podrán acceder a él hasta que se desbloquee, en cuyo momento otra tarea lo bloqueará y lo usará, y así sucesivamente.

Para resolver el problema de la colisión de hebras, casi todos los esquemas de concurrencia *serializan el acceso a los recursos compartidos*. Esto quiere decir que sólo se permite que una sola tarea acceda al recurso compartido en cada momento. Esto se suele conseguir, normalmente, colocando una cláusula alrededor de un fragmento de código, de forma que sólo se permita que una única tarea pase en cada momento a través de ese código. Puesto que esta cláusula produce una *exclusión mutua*, un nombre común que se emplea para este tipo de mecanismo es el de *mutex*.

Considere el cuarto de aseo de nuestra casa; varias personas (tareas dirigidas por hebras) podrían querer el uso exclusivo del cuarto de baño (el recurso compartido). Para acceder al cuarto de baño, una persona llama a la puerta para ver si está ocupado. Si no lo está, entra y bloquea la puerta. Cualquier otra tarea que quiera utilizar el cuarto de baño estará “bloqueada” y no podrá utilizarlo, así que esas tareas esperarán en la puerta hasta que el cuarto de baño quede disponible.

La analogía resulta algo menos apropiada en lo que respecta al instante en el que el cuarto de aseo queda libre y llega el momento de proporcionar acceso a otra tarea. En realidad, no existe una cola de personas y no estamos seguros de quién ocupará a continuación el cuarto de baño, porque el planificador de hebras no es determinista en ese sentido. En lugar de ello, es como si hubiera un grupo de tareas bloqueadas dando vueltas en las cercanías del cuarto de baño y cuando la tarea que ha ocupado el cuarto de baño lo desbloquea y sale de él, la tarea que esté más cerca de la puerta aprovechará para introducirse. Como hemos indicado anteriormente, podemos enviar sugerencias al planificador de hebras mediante `yield()` y `setPriority()`, pero puede que esas sugerencias no tengan demasiado éxito dependiendo de la plataforma y de la implementación de la máquina JVM.

Para impedir las colisiones relativas a los recursos, Java tiene un soporte integrado, basado en el uso de la palabra clave **synchronized**. Cuando una tarea quiere ejecutar un fragmento de código protegido por la palabra clave **synchronized**, comprueba si el bloqueo está disponible, lo adquiere, ejecuta el código y lo libera.

El recurso compartido es, normalmente, simplemente un cierto espacio de memoria, en forma de un objeto, pero también podría ser un archivo, un puerto de E/S o algo más complejo como una impresora. Para controlar el acceso a un recurso compartido, primero ponemos ese recurso dentro de un objeto. Entonces cualquier método que emplee el recurso puede sincronizarse con **synchronized**. Si una tarea está enmarcada en una llamada a los métodos **synchronized**, todas las demás tareas se verán impedidas de entrar en *ninguno* de los métodos **synchronized** hasta que la primera tarea vuelva de su llamada.

En el código de producción, ya hemos visto que conviene hacer que los elementos de datos de una clase sean privados y, de manera que se acceda a esa memoria únicamente a través de métodos. Podemos impedir las colisiones declarando dichos métodos como de tipo **synchronized**, de la forma siguiente:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Todos los objetos contienen, automáticamente, un único bloqueo (también denominado *monitor*). Cuando invocamos cualquier método **synchronized**, dicho objeto se bloquea y no podrá llamarse a ningún otro método **synchronized** de ese objeto hasta que el primero termine y libere el bloqueo. Para los métodos del ejemplo anterior, si una tarea invoca `f()` para un objeto, ninguna otra tarea podrá invocar a `f()` o `g()` para el mismo objeto hasta que `f()` se haya completado y libere el bloqueo. Por tanto, existe un único bloqueo que es compartido por todos los métodos **synchronized** de un objeto concreto, y este bloqueo puede emplearse para evitar que haya más de una tarea escribiendo en la memoria del objeto simultáneamente.

Observe que resulta especialmente importante que los campos sean de tipo **private** a la hora de trabajar con concurrencia; en caso contrario, la palabra clave **synchronized** no podrá evitar que otra tarea acceda a un campo directamente, y por tanto se produzcan colisiones.

Una tarea puede adquirir el bloqueo de un objeto múltiples veces. Esto sucede si un método invoca un segundo método sobre el mismo objeto, que a su vez invoque otro método sobre el mismo objeto, etc. La máquina JVM lleva la cuenta del número de veces que el objeto ha sido bloqueado. Si el objeto está desbloqueado, ese valor de recuento será igual a cero. Cuando una tarea adquiere un bloqueo por primera vez, el valor de recuento pasa a ser uno. Cada vez que la misma tarea adquiere otro bloqueo sobre el mismo objeto, se incrementa el valor de recuento. Naturalmente, esa adquisición múltiple de bloqueos sólo está permitida para la tarea que haya adquirido el bloqueo en primer lugar. Cada vez que la tarea abandona un método **synchronized**, el valor de recuento se decrementa, hasta que llegue a valer cero, lo que hace que se libere el bloqueo completamente y que pueda ser usado por otras tareas.

También existe un único bloqueo *por clase* (como parte del objeto **Class** de dicha clase), de modo que los métodos **synchronized** estáticos pueden impedir que otros métodos similares accedan simultáneamente a los datos estáticos de la clase.

¿Cuándo debemos efectuar una sincronización? Le recomendamos que aplique la *Regla de Brian de la sincronización*:¹⁰

Si estamos escribiendo una variable que pueda ser leída a continuación por otra hebra, o leyendo una variable que pueda haber sido escrita en último lugar por otra hebra, es necesario utilizar la sincronización; además, tanto el lector como el escritor deben sincronizarse usando el mismo bloqueo monitor.

Si tenemos más de un método en nuestra clase que trate con los datos críticos, será necesario sincronizar todos los métodos relevantes. Si sólo sincronizamos uno de los métodos, entonces los otros podrán ignorar el bloqueo del objeto y podrán ser invocados con impunidad. Éste es un punto muy importante: todo método que acceda a un recurso compartido crítico deberá estar sincronizado, porque sino el programa no funcionará.

Sincronización de EvenGenerator

Añadiendo **synchronized** a **EvenGenerator.java**, podemos impedir los accesos no deseados de las hebras:

```
//: concurrency/SynchronizedEvenGenerator.java
// Simplificación de los mutex con la palabra clave synchronized.
// {RunByHand}

public class SynchronizedEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public synchronized int next() {
        ++currentEvenValue;
        Thread.yield(); // Provoca el fallo más rápidamente
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new SynchronizedEvenGenerator());
    }
} ///:~
```

Hemos insertado a **Thread.yield()** entre dos incrementos, para generar la probabilidad de que se produzca un cambio de contexto mientras que **currentEvenValue** se encuentra en un estado incorrecto. Puesto que el mutex evita que haya más de una tarea en la sección crítica simultáneamente, esto no produce un fallo, pero invocar **yield()** es una forma muy útil de aumentar la probabilidad de fallo en caso de que éste vaya a suceder.

La primera tarea que entra en **next()** adquiere el bloqueo, y todas las tareas sucesivas que intenten adquirir el bloqueo no podrán hacerlo hasta que la primera tarea lo libere. En dicho punto, el mecanismo de planificación seleccionará otra tarea que esté esperando a adquirir el bloqueo. De esta forma, sólo puede haber una tarea en cada momento pasando por el código protegido por el mutex.

¹⁰ En honor de Brian Goetz, autor de *Java Concurrency in Practice*, por Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes y Doug Lea (Addison-Wesley, 2006).

Ejercicio 11: (3) Cree una clase que contenga dos campos de datos y un método que manipule dichos campos en un proceso multipaso, y que durante la ejecución de dicho método, dichos campos pasen por “estados incorrectos” (de acuerdo con una cierta definición que usted mismo puede establecer). Añada métodos para leer los campos y cree múltiples hebras para invocar los distintos métodos y mostrar que los datos están visibles en su “estado incorrecto”. Corrija el problema empleando la palabra clave **synchronized**.

Uso de objetos Lock explícitos

La biblioteca `java.util.concurrent` de Java SE5 también contiene un mecanismo explícito de mutex definido en `java.util.concurrent.locks`. Para el objeto **Lock** es necesario crearlo, bloquearlo y desbloquearlo explícitamente; por tanto, produce un código menos elegante que el mecanismo integrado. Sin embargo, es más flexible para resolver ciertos tipos de problemas. He aquí `SynchronizedEvenGenerator.java` reescrito para utilizar bloqueos **Lock** explícitos:

```
//: concurrency/MutexEvenGenerator.java
// Prevención de las colisiones de hebras mediante mutex.
// {RunByHand}
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    private Lock lock = new ReentrantLock();
    public int next() {
        lock.lock();
        try {
            ++currentEvenValue;
            Thread.yield(); // Provoca un fallo más rápidamente
            ++currentEvenValue;
            return currentEvenValue;
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        EvenChecker.test(new MutexEvenGenerator());
    }
} ///:-
```

`MutexEvenGenerator` añade un mutex denominado **lock** que utiliza los métodos **lock()** y **unlock()** para crear una sección crítica dentro de **next()**. Cuando utilizamos objetos **Lock**, es importante internalizar la estructura sintáctica que aquí se muestra: justo después de la llamada a **lock()**, hay que insertar una instrucción **try-finally** con **unlock()** en la cláusula **finally**, ésta es la única forma de garantizar que el bloqueo se libere siempre. Observe que la instrucción **return** debe estar dentro de la cláusula **try** para garantizar que el método **unlock()** no se ejecute demasiado pronto, exponiendo los datos a la posible manipulación por parte de otra tarea.

Aunque la cláusula **try-finally** requiere más código que utilizar la palabra clave **synchronized**, también representa una de las ventajas de los objetos **Lock** explícitos. Si algo falla empleando la palabra clave **synchronized**, se genera una excepción, pero no tenemos la posibilidad de realizar ninguna tarea de limpieza para poder mantener el sistema en un estado correcto. Con los objetos **Lock** explícitos, podemos mantener el estado correcto del sistema empleando la cláusula **finally**.

En general, cuando utilizamos **synchronized**, necesitaremos escribir menos código y la oportunidad de que se produzcan errores se reduce enormemente, por lo que sólo utilizaremos normalmente los objetos **Lock** explícitos cuando estemos resolviendo problemas especiales. Por ejemplo, con la palabra clave **synchronized**, no podemos realizar intentos fallidos de adquirir un bloqueo, ni tampoco tratar de adquirir un bloqueo durante un cierto espacio de tiempo y luego liberarlo; para hacer estas cosas, es necesario emplear la biblioteca **concurrent**:

```
//: concurrency/AttemptLocking.java
// Los bloqueos de la biblioteca concurrent nos permiten
// desistir al intentar adquirir un bloqueo.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

```

public class AttemptLocking {
    private ReentrantLock lock = new ReentrantLock();
    public void untimed() {
        boolean captured = lock.tryLock();
        try {
            System.out.println("tryLock(): " + captured);
        } finally {
            if(captured)
                lock.unlock();
        }
    }
    public void timed() {
        boolean captured = false;
        try {
            captured = lock.tryLock(2, TimeUnit.SECONDS);
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
        try {
            System.out.println("tryLock(2, TimeUnit.SECONDS): " +
                captured);
        } finally {
            if(captured)
                lock.unlock();
        }
    }
    public static void main(String[] args) {
        final AttemptLocking al = new AttemptLocking();
        al.untimed(); // True -- el bloqueo está disponible
        al.timed(); // True -- el bloqueo está disponible
        // Ahora crear una tarea separada para establecer el bloqueo:
        new Thread() {
            { setDaemon(true); }
            public void run() {
                al.lock.lock();
                System.out.println("acquired");
            }
        }.start();
        Thread.yield(); // Dar una oportunidad a la segunda tarea
        al.untimed(); // False -- bloqueo adquirido por la tarea
        al.timed(); // False -- bloqueo adquirido por la tarea
    }
} /* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*///:~

```

Un bloqueo **ReentrantLock** nos permite intentar adquirir el bloqueo sin éxito por lo que si alguien más ha adquirido el bloqueo, podemos decidir renunciar a adquirirlo por el momento y hacer alguna otra cosa mientras en lugar de esperar a que se libere, como podemos ver en el método **untimed()**. En **timed()**, se realiza un intento de adquirir el bloqueo que puede fallar después de 2 segundos (observe el uso de la clase **TimeUnit** de Java SE5 para especificar las unidades). En **main()**, se crea un objeto **Thread** separado como una clase anónima y éste adquiere el bloqueo para que los métodos **untimed()** y **timed()** tengan algo con lo que trabajar.

El objeto **Lock** explícito también nos proporciona un control de granularidad masiva sobre el bloqueo y el desbloqueo de lo que permite el bloqueo **synchronized** integrado. Esto resulta útil para implementar estructuras de sincronización especializadas, tales como por ejemplo la de *bloqueo mano a mano* (también conocida como *acoplamiento de bloqueos*), utilizada

para recorrer los nodos de una lista enlazada, el código que recorre la lista debe capturar el bloqueo del nodo siguiente antes de liberar el bloqueo del nodo actual.

Atomicidad y volatilidad

Una idea incorrecta pero que se repite a menudo en las explicaciones sobre el mecanismo de hebras de Java es “las operaciones atómicas no necesitan sincronizarse”. Una *operación atómica* es aquella que no puede ser interrumpida por el planificador de hebras: si la operación se inicia, entonces continuará ejecutándose hasta completarse, sin que pueda producirse un cambio de contexto durante el proceso. Confiar en la atomicidad resulta peligroso: sólo debemos tratar de emplear la atomicidad en lugar de la sincronización si somos auténticos expertos en concurrencia o si disponemos de ayuda de uno de tales expertos. Si considera que es lo suficientemente hábil como para jugar con fuego, haga esta prueba:

*La Prueba de Goetz*¹¹: si eres capaz de escribir una máquina JVM de altas prestaciones para un único procesador moderno, entonces podrás *comenzar a pensar* si puedes ahorrarte la sincronización.¹²

Resulta útil *saber* acerca de la atomicidad, y saber también que ésta se utilizó, junto con otras técnicas avanzadas, para implementar algunos de los componentes más inteligentes de la biblioteca `java.util.concurrent`. Pero no caiga en la tentación de depender usted mismo de la atomicidad; tenga siempre presente la Regla de Brian de la sincronización, presentada anteriormente.

La atomicidad se aplica a las “operaciones simples” sobre los tipos primitivos, excepto para valores **long** y **double**. Leer y escribir variables primitivas de **long** y **double** es, de manera garantizada, una operación de acceso hacia y desde la memoria absolutamente indivisible (atómica). Sin embargo, la máquina JVM está autorizada a realizar lecturas y escrituras de valores de 64 bits (variables **long** y **double**) como dos operaciones de 32 bits separadas, haciendo surgir la posibilidad de que se produzca un cambio de contexto en mitad de una lectura o escritura, con lo que diferentes tareas podrían ver resultados incorrectos (esto se denomina en ocasiones *desgajamiento de palabra*, porque existe la posibilidad de ver el valor después de que sólo se haya cambiado una parte del mismo). Sin embargo, si que tenemos atomicidad (para asignaciones y devoluciones simples) si utilizamos la palabra clave **volatile** al definir una variable **long** o **double** (observe que **volatile** no funcionaba adecuadamente antes de Java SE5). Las diferentes máquinas JVM son libres de proporcionar garantías más fuertes, pero tenga en cuenta que no se debe confiar en las características que sean específicas de determinadas plataformas.

Por tanto, las operaciones atómicas no son interrumpibles por el mecanismo de gestión de hebras. Los programadores expertos pueden aprovechar esto para escribir *código libre de bloqueos*, que no necesita sincronizarse. Pero incluso esto es una simplificación excesiva. En ocasiones, aún cuando parezca que una operación atómica debería ser segura, puede que no lo sea. Los lectores de este libro no podrán, normalmente, pasar la Prueba de Goetz antes mencionada, por lo que no deberían pensar en sustituir la sincronización por operaciones atómicas. Tratar de eliminar la sincronización es realmente un signo de optimización prematura, que generará una gran cantidad de problemas, probablemente, sin ganar nada a cambio, o muy poco.

En los sistemas multiprocesador (que ahora están apareciendo en la forma de procesadores *multinúcleo*: múltiples procesadores en un mismo chip), la *visibilidad* más que la atomicidad suele ser mucho más importante que en los sistemas de un solo procesador. Los cambios realizados por una tarea, incluso si son atómicos en el sentido de que no son interrumpibles, puede que no sean visibles para otras tareas (los cambios deben estar almacenados temporalmente en una caché del procesador local, por ejemplo), de modo que diferentes tareas tendrán una visión distinta del estado de la aplicación. El mecanismo de sincronización, por el contrario, fuerza a que los cambios realizados por una tarea en un sistema multiprocesador sean visibles para toda la aplicación. Sin la sincronización no resulta posible determinar cuándo serán visibles los cambios.

La palabra clave **volatile** también asegura la visibilidad para toda la aplicación. Si declaramos que un campo es de tipo **volatile**, esto quiere decir que, tan pronto como se realice una escritura en dicho campo, todas las lecturas podrán ver el cambio. Esto es cierto incluso si se utilizan cachés locales: los campos volátiles se escriben inmediatamente en la memoria principal y las lecturas se realizan también en la memoria principal.

¹¹ Llamada así por el antes mencionado Brian Goetz, un experto en concurrencia que me ha ayudado a elaborar este capítulo, que está parcialmente basado en algunos comentarios que me hizo.

¹² Un corolario de esta prueba es: “Si alguien sugiere que la gestión de hebras es sencilla, asegúrate de que esa persona no tenga bajo sus responsabilidades el tomar decisiones importantes acerca del proyecto. Si esa persona está en disposición de tomar esas decisiones, tienes un problema”.

Es importante entender que la atomicidad y la volatilidad son conceptos distintos. Una operación atómica en un campo no volátil no será necesariamente volcada en la memoria principal, por lo que otra tarea que lea dicho campo no tendría por qué, necesariamente, ver el nuevo valor. Si hay múltiples tareas accediendo a un campo, dicho campo debe ser de tipo volátil; en caso contrario, sólo debería accederse al campo utilizando la sincronización. La sincronización también provoca el volcado en memoria principal, por lo que si un campo está completamente protegido por bloques o métodos sincronizados, no es necesario definirlo como de tipo volátil.

Cualquier escritura que una tarea realice será visible para dicha tarea, por lo que no es necesario un campo volátil si ese campo sólo es consultado dentro de una tarea.

La palabra **volatile** no funciona cuando el valor de un campo depende de su valor anterior (por ejemplo, el incrementar un contador), ni tampoco funciona con aquellos campos cuyos valores están restringidos por los valores de otros campos, como por ejemplo, los límites inferior (**lower**) y superior (**upper**) de una clase **Range** (rango) que deba obedecer la restricción **lower <= upper**.

Normalmente, sólo resulta seguro **volatile** en lugar de **synchronized** si la clase sólo tiene un campo modificable. De nuevo, nuestra primera opción debería ser emplear la palabra clave **synchronized**; éste es el enfoque más seguro e intentar hacer cualquier otra cosa resulta arriesgado.

¿Qué cosas pueden llegar a ser operaciones atómicas? La asignación y la devolución del valor de un campo serán normalmente atómicas. Sin embargo, en C++ incluso las siguientes instrucciones *podrían* ser atómicas:

```
i++; // Podría ser atómica en C++
i += 2; // Podría ser atómica en C++
```

Pero en C++, esto depende del compilador y del procesador. No podemos escribir código inter-plataforma en C++ que dependa de la atomicidad, porque C++ no tiene un *modelo de memoria* coherente, a diferencia de Java (en Java SE5).¹³

En Java, las operaciones anteriores son, sin ninguna duda, *no atómicas*, como se puede ver analizando las instrucciones JVM generadas por los siguientes métodos:

```
//: concurrency/Atomicity.java
// {Exec: javap -c Atomicity}

public class Atomicity {
    int i;
    void f1() { i++; }
    void f2() { i += 3; }
} /* Output: (Sample)
...
void f1();
Code:
  0:   aload_0
  1:   dup
  2:   getfield      #2; //Campo i:I
  5:   iconst_1
  6:   iadd
  7:   putfield      #2; //Campo i:I
 10:  return
void f2();
Code:
  0:   aload_0
  1:   dup
  2:   getfield      #2; //Campo i:I
  5:   iconst_3
  6:   iadd
  7:   putfield      #2; //Campo i:I
 10:  return
*////:-
```

¹³ Esto se está tratando de remediar en el siguiente estándar C++ que se va a publicar.

Cada instrucción produce una operación “get” y una operación “put”, con instrucciones entremedias. Por tanto, entre el instante de obtener el valor y el instante de almacenar el valor corregido, otra tarea podría modificar el campo, así que las operaciones no son atómicas.

Si aplicamos ciegamente la idea de la atomicidad, podemos ver que `getValue()` en el siguiente programa se ajusta a la descripción:

```
//: concurrency/AtomicityTest.java
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
    private int i = 0;
    public int getValue() { return i; }
    private synchronized void evenIncrement() { i++; i++; }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicityTest at = new AtomicityTest();
        exec.execute(at);
        while(true) {
            int val = at.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} /* Output: (Sample)
191583767
*///:-
```

Sin embargo, el programa encontrará valores no pares y terminará. Aunque `return i` es ciertamente una operación atómica, la falta de sincronización permite que el valor se lea mientras que el objeto se encuentra en un estado intermedio inestable. Además, puesto que `i` también es de tipo volátil, habrá problemas de visibilidad. Tanto `getValue()` como `evenIncrement()` deben ser sincronizados. Sólo los expertos en concurrencia deberían intentar realizar optimizaciones en situaciones como ésta. De nuevo, recuerde siempre la Regla de Brian de la sincronización.

Como segundo ejemplo vamos a considerar algo todavía más simple: una clase que produce números de serie.¹⁴ Cada vez que se llama a `nextSerialNumber()`, debe devolver un valor único al llamante:

```
//: concurrency/SerialNumberGenerator.java
public class SerialNumberGenerator {
    private static volatile int serialNumber = 0;
    public static int nextSerialNumber() {
        return serialNumber++; // No es seguro con las hebras
    }
} /*:-
```

`SerialNumberGenerator` es una clase de lo más simple que podríamos imaginar, y para cualquiera que proceda de C++ o que tenga alguna otra experiencia previa similar de bajo nivel, cabría esperar que la de incremento fuera una operación atómica, porque un incremento en C++ a menudo puede implementarse como una instrucción de microprocesador (aunque no en una forma inter-plataforma fiable). Sin embargo, como hemos indicado antes, una operación incremento en Java *no* es atómica e implica tanto una lectura como una escritura, por lo que existen problemas con las hebras incluso en operaciones tan simples como ésta. Como veremos, el problema aquí no es la volatilidad, el problema real es que `nextSerialNumber()` accede a un valor compartido y mutable sin sincronización.

¹⁴ Ejemplo inspirado en el libro *Effective Java™ Programming Language Guide* de Joshua Bloch (Addison-Wesley, 2001), p. 190.

El campo **serialNumber** es volátil porque es posible que cada hebra tenga una pila local y mantenga en ella copias de algunas variables. Si definimos una variable como volátil, esto le dice al compilador que no realice ninguna optimización que pudiera eliminar las lecturas y escrituras que mantienen al campo en sincronización exacta con los datos locales almacenados en las hebras. En la práctica, las lecturas y escrituras se realizan directamente en memoria y no se almacenan valores en caché. La palabra clave **volatile** también restringe la posibilidad de que el compilador realice reordenaciones de los accesos durante la optimización. Sin embargo, **volatile** no afecta al hecho de que la de incremento no es una operación atómica.

Básicamente, debemos definir un campo como **volatile** si hay varias tareas que pueden acceder a la vez a dicho campo y al menos uno de esos accesos es una escritura. Por ejemplo, un campo que se utilice como indicador para detener una tarea debe ser declarado como de tipo **volatile**; en caso contrario, dicho indicador podría estar almacenado en un registro de caché, y cuando se hicieran cambios en el indicador desde fuera de la tarea, el valor de la caché no sería modificado y la tarea nunca sabría que debe detenerse.

Para probar **SerialNumberGenerator**, necesitamos un conjunto que no se quede sin memoria, por si acaso se tarda un largo tiempo en detectar el problema. El conjunto **CircularSet** mostrado aquí reutiliza la memoria usada para almacenar valores enteros, en la suposición de que para cuando volvamos al principio del conjunto, la posibilidad de una colisión con los valores sobreescritos será mínima. Los métodos **add()** y **contains()** están sincronizados para impedir las colisiones entre hebras:

```
//: concurrency/SerialNumberChecker.java
// Determinadas operaciones que pueden parecer seguras no lo son
// cuando hay hebras presentes.
// {Args: 4}
import java.util.concurrent.*;

// Reutiliza el almacenamiento para no agotar la memoria:
class CircularSet {
    private int[] array;
    private int len;
    private int index = 0;
    public CircularSet(int size) {
        array = new int[size];
        len = size;
        // Inicializar con un valor no producido por
        // el generador SerialNumberGenerator:
        for(int i = 0; i < size; i++)
            array[i] = -1;
    }
    public synchronized void add(int i) {
        array[index] = i;
        // Implementar circularmente el index y reescribir los elementos
        // antiguos:
        index = ++index % len;
    }
    public synchronized boolean contains(int val) {
        for(int i = 0; i < len; i++)
            if(array[i] == val) return true;
        return false;
    }
}

public class SerialNumberChecker {
    private static final int SIZE = 10;
    private static CircularSet serials =
        new CircularSet(1000);
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static class SerialChecker implements Runnable {
        public void run() {
            while(true) {
```

```

        int serial =
            SerialNumberGenerator.nextSerialNumber();
        if(serials.contains(serial)) {
            System.out.println("Duplicate: " + serial);
            System.exit(0);
        }
        serials.add(serial);
    }
}

public static void main(String[] args) throws Exception {
    for(int i = 0; i < SIZE; i++)
        exec.execute(new SerialChecker());
    // Detenerse después de n segundos si hay un argumento:
    if(args.length > 0) {
        TimeUnit.SECONDS.sleep(new Integer(args[0]));
        System.out.println("No duplicates detected");
        System.exit(0);
    }
}
} /* Output: (Sample)
Duplicate: 8468656
*///:-
```

SerialNumberChecker contiene un conjunto estático **CircularSet** que almacena todos los números de serie que se han producido y una clase **SerialChecker** anidada que garantiza que los números de serie sean únicos. Creando múltiples tareas para contender con el acceso a los números de serie, descubriremos que las tareas llegan a obtener un número de serie duplicado, si dejamos que el programa se ejecute el tiempo suficiente. Para resolver el problema, añada la palabra clave **synchronized** a **nextSerialNumber()**.

Las operaciones atómicas que se supone que son seguras son las de lectura y asignación de primitivas. Sin embargo, como hemos visto en **AtomicityTest.java**, sigue siendo posible utilizar una operación atómica que acceda a nuestro objeto, mientras que éste se encuentre en un estado intermedio inestable. Realizar suposiciones acerca de este tema resulta muy peligroso. Lo mejor que puede hacerse es aplicar la Regla de Brian de la sincronización.

Ejercicio 12: (3) Corrija **AtomicityTest.java** utilizando la palabra clave **synchronized**. ¿Puede demostrar que ahora es correcto?

Ejercicio 13: (1) Corrija **SerialNumberChecker.java** utilizando la palabra clave **synchronized**. ¿Puede demostrar que ahora es correcto?

Clases atómicas

Java SE5 introduce clases de variables atómicas especiales como **AtomicInteger**, **AtomicLong**, **AtomicReference**, etc., que proporcionan una operación condicional de actualización atómica de la forma:

```
boolean compareAndSet(expectedValue, updateValue);
```

Este ejemplo de método compararía y actualizaría en una operación atómica una determinada variable, proporcionándose como argumentos el valor esperado y el valor de actualización. Este tipo de método se utiliza para realizar una optimización avanzada, aprovechando la atomicidad del nivel de la máquina disponible en algunos procesadores modernos, por lo que generalmente no tenemos por qué preocuparnos de utilizarlos. En ocasiones, pueden resultar útiles para los programas normales, pero, de nuevo, sólo cuando se esté intentando realizar una optimización. Por ejemplo, podemos escribir de nuevo **AtomicityTest.java** para utilizar **AtomicInteger**:

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
```

```

private AtomicInteger i = new AtomicInteger(0);
public int getValue() { return i.get(); }
private void evenIncrement() { i.addAndGet(2); }
public void run() {
    while(true)
        evenIncrement();
}
public static void main(String[] args) {
    new Timer().schedule(new TimerTask() {
        public void run() {
            System.err.println("Aborting");
            System.exit(0);
        }
    }, 5000); // Terminar después de 5 segundos
ExecutorService exec = Executors.newCachedThreadPool();
AtomicIntegerTest ait = new AtomicIntegerTest();
exec.execute(ait);
while(true) {
    int val = ait.getValue();
    if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
    }
}
}
} //:-

```

Aquí hemos eliminado la palabra clave **synchronized** utilizando en su lugar **AtomicInteger**. Puesto que el programa no falla, hemos añadido un objeto **Timer** para abortar automáticamente la ejecución después de 5 segundos.

He aquí **MutexEvenGenerator.java** reescrito para utilizar **AtomicInteger**:

```

//: concurrency/AtomicEvenGenerator.java
// Las clases atómicas son útiles en ocasiones en los programas normales.
// {RunByHand}
import java.util.concurrent.atomic.*;
public class AtomicEvenGenerator extends IntGenerator {
    private AtomicInteger currentEvenValue =
        new AtomicInteger(0);
    public int next() {
        return currentEvenValue.addAndGet(2);
    }
    public static void main(String[] args) {
        EvenChecker.test(new AtomicEvenGenerator());
    }
}
} //:-

```

De nuevo, todas las demás formas de sincronización se han eliminado empleando **AtomicInteger**.

Es necesario recalcar que las clases **Atomic** fueron diseñadas para construir las clases de **java.util.concurrent**, y que sólo debemos utilizarlas en nuestro propio código en circunstancias especiales, e incluso en ese caso únicamente cuando podemos garantizar que no van a surgir otros posibles problemas. Generalmente, es más seguro utilizar los bloqueos (bien la palabra clave **synchronized** o bien objetos **Lock** explícitos).

Ejercicio 14: (4) Demuestre que **java.util.Timer** se puede escalar para utilizar números de gran magnitud creando un programa que genere muchos objetos **Timer** que realicen alguna tarea simple cuando se produzca el fin de temporización.

Secciones críticas

En ocasiones, lo único que queremos evitar es que múltiples hebras accedan a parte del código dentro de un método, en lugar de al método completo. La sección de código que queramos aislar de esta manera se denomina *sección crítica* y se crea uti-

lizando la palabra clave **synchronized**. Aquí, **synchronized** se utiliza para especificar el objeto cuyo bloqueo se está empleando para sincronizar el código incluido en la sincronización:

```
synchronized(syncObject) {
    // En cada momento, sólo una tarea puede
    // acceder a este código
}
```

Esto se denomina también *bloqueo sincronizado*; lo que quiere decir que antes de poder entrar en esa sección de código, hay que adquirir el bloqueo para **syncObject**. Si alguna otra tarea ha adquirido ya este bloqueo, entonces no podrá entrarse en la sección crítica hasta que dicho bloqueo se libere.

El siguiente ejemplo compara ambas técnicas de sincronización, demostrando que el tiempo disponible para que otras tareas accedan a un objeto se incrementa significativamente empleando un bloqueo **synchronized** en lugar de sincronizar el método completo. Además, muestra cómo se puede utilizar una clase no protegida en entornos multihebra si se la controla y protege mediante otra clase:

```
//: concurrency/CriticalSection.java
// Sincronización de bloqueo en lugar de métodos completos. También
// ilustra la protección de una clase no protegida mediante
// otra clase protegida.
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // No protegida frente a hebras
    private int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Pair() { this(0, 0); }
    public int getX() { return x; }
    public int getY() { return y; }
    public void incrementX() { x++; }
    public void incrementY() { y++; }
    public String toString() {
        return "x: " + x + ", y: " + y;
    }
    public class PairValuesNotEqualException
        extends RuntimeException {
        public PairValuesNotEqualException() {
            super("Pair values not equal: " + Pair.this);
        }
    }
    // Invariante arbitrario -- ambas variables deben ser iguales:
    public void checkState() {
        if(x != y)
            throw new PairValuesNotEqualException();
    }
}

// Proteger un objeto Pair dentro de una clase protegida frente a hebras:
abstract class PairManager {
    AtomicInteger checkCounter = new AtomicInteger(0);
    protected Pair p = new Pair();
    private List<Pair> storage =
        Collections.synchronizedList(new ArrayList<Pair>());
    public synchronized Pair getPair() {
```

```

// Hacer una copia para que el original esté seguro:
return new Pair(p.getX(), p.getY());
}
// Asumimos que ésta es una operación de larga duración
protected void store(Pair p) {
    storage.add(p);
    try {
        TimeUnit.MILLISECONDS.sleep(50);
    } catch(InterruptedException ignore) {}
}
public abstract void increment();
}

// Sincronizar el método completo:
class PairManager1 extends PairManager {
    public synchronized void increment() {
        p.incrementX();
        p.incrementY();
        store(getPair());
    }
}

// Utilizar una sección crítica:
class PairManager2 extends PairManager {
    public void increment() {
        Pair temp;
        synchronized(this) {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        }
        store(temp);
    }
}

class PairManipulator implements Runnable {
    private PairManager pm;
    public PairManipulator(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true)
            pm.increment();
    }
    public String toString() {
        return "Pair: " + pm.getPair() +
            " checkCounter = " + pm.checkCounter.get();
    }
}

class PairChecker implements Runnable {
    private PairManager pm;
    public PairChecker(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true) {
            pm.checkCounter.incrementAndGet();
            pm.getPair().checkState();
        }
    }
}

```

```

        }
    }

public class CriticalSection {
    // Probar las dos técnicas:
    static void
    testApproaches(PairManager pman1, PairManager pman2) {
        ExecutorService exec = Executors.newCachedThreadPool();
        PairManipulator
            pm1 = new PairManipulator(pman1),
            pm2 = new PairManipulator(pman2);
        PairChecker
            pcheck1 = new PairChecker(pm1),
            pcheck2 = new PairChecker(pm2);
        exec.execute(pm1);
        exec.execute(pm2);
        exec.execute(pcheck1);
        exec.execute(pcheck2);
        try {
            TimeUnit.MILLISECONDS.sleep(500);
        } catch(InterruptedException e) {
            System.out.println("Sleep interrupted");
        }
        System.out.println("pm1: " + pm1 + "\npm2: " + pm2);
        System.exit(0);
    }
    public static void main(String[] args) {
        PairManager
            pman1 = new PairManager1(),
            pman2 = new PairManager2();
        testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
pm1: Pair: x: 15, y: 15 checkCounter = 272565
pm2: Pair: x: 16, y: 16 checkCounter = 3956974
*///:-

```

Como se indica, **Pair** no es seguro de cara a las hebras, porque su invariante (o porque su invariante es arbitrario) requiere que ambas variables mantengan los mismos valores. Además, como hemos visto anteriormente en el capítulo, las operaciones de incremento no son seguras respecto a las hebras, y como ninguno de los métodos está sincronizado, no podemos confiar en que un objeto **Pair** permanezca sin corromperse dentro de un programa con hebras.

Imagine que alguien nos entrega la clase **Pair** no protegida frente a hebras, y que necesitamos utilizarla en un entorno de hebras. Para hacer esto, creamos la clase **PairManager**, que almacena un objeto **Pair** y controla todo el acceso al mismo. Observe que los únicos métodos públicos son **getPair()**, que está sincronizado y el método abstracto **increment()**. La sincronización de **increment()** se gestionará cuando se lo implemente.

La estructura de **PairManager**, en la que la funcionalidad implementada en la clase base utiliza uno o más métodos abstractos definidos en las clases derivadas se denomina *Método de plantillas* en la jerga de los *Patrones de diseño*.¹⁵ Los patrones de diseño nos permiten encapsular los cambios que nuestro código pueda sufrir; aquí, la parte que cambia es el método **increment()**. En **PairManager1** todo el método **increment()** está sincronizado, mientras que en **PairManager2** sólo se sincroniza parte del método **increment()** utilizando un bloque **synchronized**. Observe que la palabra clave **synchronized** no forma parte de la firma del método y que, por tanto, puede ser añadida al sustituirlo en una clase derivada.

El método **store()** añade un objeto **Pair** a un contenedor **ArrayList** sincronizado, por lo que esta operación es segura con respecto a las hebras. Por tanto, no es necesario protegerla, y se la coloca fuera del bloque **synchronized** en **PairManager2**.

¹⁵ Véase *Design Patterns*, por Gamma et al. (Addison-Wesley, 1995).

PairManipulator se crea para probar los dos diferentes tipos de objetos **PairManager**, invocando **increment()** en una tarea mientras se ejecuta la comprobación **PairChecker** desde otra tarea. Para ver con qué frecuencia es capaz de ejecutar la prueba, **PairChecker** incrementa **checkCounter** cada vez que tiene éxito. En **main()**, se crean dos objetos **PairManipulator** y se les permite ejecutarse durante un tiempo, después de lo cual se muestran los resultados de cada objeto **PairManipulator**.

Aunque probablemente pueda percibir una gran variación a la salida entre una ejecución del programa y la siguiente, en general verá que **PairManager1.increment()** no permite a **PairChecker** unos accesos tan frecuentes como a **PairManager2.increment()**, que tiene el bloque **synchronized** y goza, por tanto, de un mayor tiempo con los bloqueos liberados. Ésta es, normalmente, la razón para utilizar un bloque **synchronized** en lugar de la sincronización del método completo: para permitir que las otras tareas puedan acceder más frecuentemente (siempre y cuando sea seguro hacerlo).

También se pueden usar objetos **Lock** explícitos para crear secciones críticas:

```
//: concurrency/ExplicitCriticalSection.java
// Uso de objetos Lock explícitos para crear secciones críticas.
package concurrency;
import java.util.concurrent.locks.*;

// Sincronizar el método completo:
class ExplicitPairManager1 extends PairManager {
    private Lock lock = new ReentrantLock();
    public synchronized void increment() {
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            store(getPair());
        } finally {
            lock.unlock();
        }
    }
}

// Usar una sección crítica:
class ExplicitPairManager2 extends PairManager {
    private Lock lock = new ReentrantLock();
    public void increment() {
        Pair temp;
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        } finally {
            lock.unlock();
        }
        store(temp);
    }
}

public class ExplicitCriticalSection {
    public static void main(String[] args) throws Exception {
        PairManager
            pman1 = new ExplicitPairManager1(),
            pman2 = new ExplicitPairManager2();
        CriticalSection.testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
pm1: Pair: x: 15, y: 15 checkCounter = 174035
pm2: Pair: x: 16, y: 16 checkCounter = 2608588
*///:-
```

Este programa reutiliza la mayor parte de **CriticalSection.java** y crea nuevos tipos de **PairManager** que utilizan objetos **Lock** explícitos. **ExplicitPairManager2** muestra la creación de una sección crítica mediante un objeto **Lock**; la llamada a **store()** se encuentra fuera de la sección crítica.

Sincronización sobre otros objetos

A un bloque **synchronized** hay que proporcionarle un objeto con el que sincronizarse, y normalmente el objeto más apropiado para este propósito es el objeto actual para el que esté siendo invocado el método: **synchronized(this)**, que es la técnica usada en **PairManager2**. De esta forma, cuando se adquiere el bloqueo para el bloque **synchronized**, no se pueden invocar otros métodos **synchronized** y secciones críticas del objeto. Por tanto, el efecto de la sección crítica, al sincronizarse sobre **this**, consiste simplemente en reducir el ámbito de sincronización.

En ocasiones, es necesario sincronizarse con otro objeto, pero si hacemos esto debemos garantizar que todas las tareas relevantes se sincronicen con el mismo objeto. El siguiente ejemplo demuestra que dos tareas pueden entrar en un objeto si los métodos de dicho objeto se sincronizan con bloques diferentes:

DualSync.f() se sincroniza con **this** (sincronizando el método completo), y **g()** tiene un bloque **synchronized** que se sincroniza con **syncObject**. Así, las dos sincronizaciones son independientes. Esto se ilustra en **main()** creando un objeto **Thread** que invoca **f()**. La hebra en **main()** se utiliza para llamar a **g()**. Podemos ver, analizando la salida, que ambos métodos se ejecutan al mismo tiempo, así que ninguno de ellos está bloqueado por la sincronización del otro.

Ejercicio 15: (1) Cree una clase con tres métodos que contengan secciones críticas, y que todas se sincronicen con el mismo objeto. Cree múltiples tareas para demostrar que sólo uno de estos métodos puede ejecutarse cada vez. Ahora, modifique los métodos de modo que cada uno de ellos se sincronice con un objeto distinto y muestre que los tres métodos pueden ejecutarse simultáneamente.

Ejercicio 16: (1) Modifique el Ejercicio 15 para usar objetos **Lock** explícitos.

Almacenamiento local de las hebras

Una segunda forma de evitar que las tareas colisionen o accedan a recursos compartidos consiste en eliminar la compartición de variables. El *almacenamiento local de las hebras* es un mecanismo que crea automáticamente un espacio de almacenamiento distinto para la misma variable, para cada hebra diferente que esté utilizando un objeto. Así, si tenemos cinco hebras utilizando un objeto con una variable **x**, el almacenamiento local de las hebras genera cinco espacios diferentes de almacenamiento para **x**. Básicamente, este mecanismo nos permite asociar un cierto estado con cada hebra.

La creación y gestión del almacenamiento local de la hebra son realizados por la clase **java.lang.ThreadLocal**, como puede verse aquí:

```
//: concurrency/ThreadLocalVariableHolder.java
// Asignación automática de su propio espacio
// de almacenamiento a cada hebra.
import java.util.concurrent.*;
import java.util.*;

class Accessor implements Runnable {
    private final int id;
    public Accessor(int idn) { id = idn; }
    public void run() {
        while(!Thread.currentThread().isInterrupted()) {
            ThreadLocalVariableHolder.increment();
            System.out.println(this);
            Thread.yield();
        }
    }
    public String toString() {
        return "#" + id + ":" + ThreadLocalVariableHolder.get();
    }
}

public class ThreadLocalVariableHolder {
    private static ThreadLocal<Integer> value =
        new ThreadLocal<Integer>() {
        private Random rand = new Random(47);
        protected synchronized Integer initialValue() {
            return rand.nextInt(10000);
        }
    };
    public static void increment() {
        value.set(value.get() + 1);
    }
    public static int get() { return value.get(); }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
```

```

        for(int i = 0; i < 5; i++)
            exec.execute(new Accessor(i));
        TimeUnit.SECONDS.sleep(3); // Ejecutar durante un tiempo
        exec.shutdownNow(); // Todos los objetos Accessors terminarán
    }
} /* Output: (Sample)
#0: 9259
#1: 556
#2: 6694
#3: 1862
#4: 962
#0: 9260
#1: 557
#2: 6695
#3: 1863
#4: 963
...
*///:~

```

Los objetos **ThreadLocal** normalmente se almacenan como campos estáticos. Cuando creamos el objeto **ThreadLocal**, sólo podemos acceder al contenido del objeto con los métodos **get()** y **set()**. El método **get()** devuelve una copia del objeto asociado con dicha hebra, mientras que **set()** inserta su argumento en el objeto almacenado para dicha hebra, devolviendo el objeto anterior que estuviera almacenado. Los métodos **increment()** y **get()** ilustran este mecanismo en **ThreadLocalVariableHolder**. Observe que **increment()** y **get()** no están sincronizados, porque **ThreadLocal** garantiza que no se produzca ninguna condición de carrera.

Cuando ejecute este programa, podrá ver que a cada hebra individual se le asigna su propio espacio de almacenamiento, ya que cada una de ellas mantiene su propio valor de recuento, aún cuando sólo existe un objeto **ThreadLocalVariableHolder**.

Terminación de tareas

En algunos de los ejemplos anteriores, los métodos **cancel()** e **isCancelled()** se colocan en una clase que es visible para todas las tareas. Las tareas comprueban **isCancelled()** para determinar cuándo deben finalizar. Esta solución resulta bastante razonable. Sin embargo, en algunas situaciones, la tarea debe terminarse de manera más abrupta. En esta sección, veremos qué problemas existen con respecto a dicha finalización.

En primer lugar, veamos un ejemplo que no sólo ilustra el problema de la terminación, sino que también es un ejemplo adicional de la compartición de recursos.

El jardín ornamental

En esta simulación, el comité director del jardín quiere saber cuántas personas entran el jardín cada día a través de las distintas puertas. Cada puerta tiene un torno o algún otro tipo de contador, y después de incrementar el contador del torno, se incrementa una cuenta compartida que representa el número total de personas que hay en el jardín.

```

//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
    private int count = 0;
    private Random rand = new Random(47);
    // Eliminar la palabra clave synchronized para ver cómo falla el recuento:
    public synchronized int increment() {
        int temp = count;
        if(rand.nextBoolean()) // Seguir el control la mitad del tiempo
            Thread.yield();

```

```

        return (count = ++temp);
    }
    public synchronized int value() { return count; }
}

class Entrance implements Runnable {
    private static Count count = new Count();
    private static List<Entrance> entrances =
        new ArrayList<Entrance>();
    private int number = 0;
    // No necesita sincronización para leer:
    private final int id;
    private static volatile boolean canceled = false;
    // Operación atómica sobre un campo volátil:
    public static void cancel() { canceled = true; }
    public Entrance(int id) {
        this.id = id;
        // Mantener esta tarea en una lista. También impide
        // que se aplique la depuración de memoria a las tareas muertas:
        entrances.add(this);
    }
    public void run() {
        while(!canceled) {
            synchronized(this) {
                ++number;
            }
            print(this + " Total: " + count.increment());
            try {
                TimeUnit.MILLISECONDS.sleep(100);
            } catch(InterruptedException e) {
                print("sleep interrupted");
            }
        }
        print("Stopping " + this);
    }
    public synchronized int getValue() { return number; }
    public String toString() {
        return "Entrance " + id + ": " + getValue();
    }
    public static int getTotalCount() {
        return count.value();
    }
    public static int sumEntrances() {
        int sum = 0;
        for(Entrance entrance : entrances)
            sum += entrance.getValue();
        return sum;
    }
}

public class OrnamentalGarden {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Entrance(i));
        // Ejecutar durante un tiempo, luego detenerse y recopilar los datos:
        TimeUnit.SECONDS.sleep(3);
        Entrance.cancel();
        exec.shutdown();
    }
}

```

```

if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
    print("Some tasks were not terminated!");
print("Total: " + Entrance.getTotalCount());
print("Sum of Entrances: " + Entrance.sumEntrances());
}
} /* Output: (Sample)
Entrance 0: 1 Total: 1
Entrance 2: 1 Total: 3
Entrance 1: 1 Total: 2
Entrance 4: 1 Total: 5
Entrance 3: 1 Total: 4
Entrance 2: 2 Total: 6
Entrance 4: 2 Total: 7
Entrance 0: 2 Total: 8
...
Entrance 3: 29 Total: 143
Entrance 0: 29 Total: 144
Entrance 4: 29 Total: 145
Entrance 2: 30 Total: 147
Entrance 1: 30 Total: 146
Entrance 0: 30 Total: 149
Entrance 3: 30 Total: 148
Entrance 4: 30 Total: 150
Stopping Entrance 2: 30
Stopping Entrance 1: 30
Stopping Entrance 0: 30
Stopping Entrance 3: 30
Stopping Entrance 4: 30
Total: 150
Sum of Entrances: 150
*///:~

```

Un único objeto **Count** mantiene la cuenta maestra de los visitantes del jardín y se almacena como campo estático en la clase **Entrance**. **Count.increment()** y **Count.value()** están sincronizados para controlar el acceso al campo **count**. El método **increment()** utiliza un objeto **Random** para ejecutar el método **yield()** aproximadamente la mitad de las veces, entre la operación de extraer **count** para almacenarlo en **temp** y la de incrementar y almacenar de nuevo **temp** en **count**. Si desactivamos mediante un comentario la palabra clave **synchronized** en **increment()**, el programa deja de funcionar porque habrá múltiples tareas accediendo a **count** modificándolo simultáneamente (el método **yield()** hace que el problema se produzca más rápidamente).

Cada tarea **Entrance** (que representa una de las entradas del jardín) mantiene un valor local **number** que contiene el número de visitantes que han pasado a través de esa entrada concreta. Esto proporciona una forma de verificar el objeto **count**, para comprobar que está registrando el número correcto de visitantes. **Entrance.run()** simplemente incrementa **number** y el objeto **count** y pasa a dormir durante 100 milisegundos.

Puesto que **Entrance.canceled** es un indicador booleano volátil, que sólo se lee y se configura (y que nunca es leído en combinación con otros campos), sería posible no sincronizar el acceso al mismo. Pero, siempre que tenga alguna duda acerca de si sincronizar algo o no, lo mejor es utilizar **synchronized**.

Este programa hace un esfuerzo especial para finalizar todas las cosas de una manera estable. En parte, la razón de hacer esto es ilustrar lo cuidadoso que hay que ser a la hora de terminar un programa multihebra; por otro lado, pretendemos con ello ilustrar el valor de **interrupt()**, del que hablaremos en breve.

Después de 3 segundos, **main()** envía el mensaje estático **cancel()** a **Entrance**, de modo que llama a **shutdown()** para el objeto **exec**, y luego invoca a **awaitTermination()** sobre **exec**. **ExecutorService.awaitTermination()** espera a que cada tarea se complete y si todo se completa antes de que finalice la temporización devuelve **true**, en caso contrario, devuelve **false** para indicar que no se han completado todas las tareas. Aunque esto hace que cada tarea salga de su método **run()** y terminen por tanto como tarea, los objetos **Entrance** seguirán siendo válidos, porque en el constructor cada objeto **Entrance** está almacenado en un contenedor estático **List<Entrance>** denominado **entrances**. Por tanto, **sumEntrances()** seguirá funcionando con objetos **Entrance** válidos.

A medida que se ejecuta este programa, podemos ver cómo se muestran el recuento total y el recuento correspondiente a cada entrada a medida que las personas pasan por los tornos. Si eliminamos la declaración `synchronized` de `Count.increment()`, observaremos que el número total de personas no es el esperado. El número de personas contabilizadas por cada torno será distinto del valor almacenado en `count`. Mientras utilicemos el mutex para sincronizar el acceso a `Count`, las cosas sucederán correctamente. Recuerde que `Count.increment()` exagera la probabilidad de fallos, utilizando `temp` y `yield()`. En los problemas reales de los programas multihebra, la posibilidad de fallo puede ser estadísticamente pequeña, así que podemos caer fácilmente en la trampa de pensar que las cosas están funcionando correctamente. Al igual que en el ejemplo anterior, resulta posible que existan problemas ocultos que no se nos hayan ocurrido, por lo que debe tratar de ser lo más diligente posible a la hora de revisar el código concurrente.

Ejercicio 17: (2) Cree un contador de radiaciones que pueda tener cualquier número de sensores remotos.

Terminación durante el bloqueo

El método `Entrance.run()` del ejemplo anterior incluye una llamada a `sleep()` dentro de su bucle. Sabemos que `sleep()` termina por despertarse y que la tarea alcanzará la parte superior del bucle donde tendrá la oportunidad de salir del mismo, comprobando el indicador `cancelled`. Sin embargo, `sleep()` es simplemente una de las situaciones en las que la ejecución de una tarea puede quedar bloqueada, y existen ocasiones en las que es necesario terminar una tarea bloqueada.

Estados de las hebras

Una hebra puede estar en uno de cuatro estados:

1. *Nueva*: una hebra permanece en este estado sólo momentáneamente, mientras se la está creando, se asignan los recursos del sistema necesarios y se realiza la inicialización. Después de esto, la hebra pasa a ser elegible para recibir tiempo de procesador. El planificador hará entonces que la hebra efectúe una transición a los estados ejecutable o bloqueado.
2. *Ejecutable*: significa que una hebra puede ejecutarse cuando el mecanismo de gestión de franjas temporales tenga ciclos de procesador para esa hebra. Así, la hebra puede o no estarse ejecutando en cualquier momento dado, pero no hay nada que la impida ejecutarse si el planificador así lo decide. Por tanto, la hebra no está ni muerta ni bloqueada.
3. *Bloqueada*: la hebra puede ejecutarse pero hay algo que lo impide. Mientras que una hebra se encuentra en el estado bloqueado, el planificador simplemente la ignorará y no la asignará tiempo de procesador. Hasta que una hebra no vuelve a entrar en el estado ejecutable, no realizará ninguna operación.
4. *Muerta*: una hebra en el estado muerto o *terminado* ya no es tenida en cuenta por el planificador y no puede recibir tiempo de procesador. Su tarea se ha completado y la hebra ya no es ejecutable. Una forma de que una tarea muera es volviendo de su método `run()`, pero la hebra de una tarea también puede interrumpirse como veremos en breve.

Formas de bloquearse

Una tarea puede llegar a estar bloqueada por las siguientes razones:

- Hemos mandado la tarea a dormir invocando `sleep(milliseconds)`, en cuyo caso no podrá ejecutarse durante el tiempo especificado.
- Hemos suspendido la ejecución de la hebra con `wait()`. No volverá a ser ejecutable de nuevo hasta que la hebra reciba el mensaje `notify()` o `notifyAll()` (o los mensajes equivalentes `signal()` o `signalAll()` para las herramientas de la biblioteca de Java SE5 `java.util.concurrent`). Examinaremos este caso en una sección posterior.
- La tarea está esperando a que se complete una operación de E/S.
- La tarea está tratando de invocar un método `synchronized` sobre otro objeto y el bloqueo no está disponible porque ya ha sido adquirido por otra tarea.

En los programas antiguos, puede que también vea que se usan los métodos `suspend()` y `resume()` para bloquear y desbloquear las hebras, pero estos métodos son desaconsejados en los programas Java modernos (porque son proclives a los

interbloqueos), así que no los examinaremos en el libro. También se desaconseja el método **stop()**, porque no libera los bloqueos que la hebra haya adquirido, y si los objetos están en un estado incoherente (“dañados”), otras tareas podrían consultarlos y modificarlos en dicho estado. Los problemas resultantes pueden ser muy sutiles y difíciles de detectar.

El problema que ahora debemos examinar es el siguiente: en ocasiones, queremos terminar una tarea que se encuentra en el estado bloqueado. Si no podemos esperar a que la hebra alcance un punto en el código en el que ella misma pueda comprobar un valor de estado y decidir terminar por cuenta propia, tenemos que forzar a que la tarea salga de su estado bloqueado.

Interrupción

Como puede imaginarse, resulta mucho más complicado salir en mitad de un método **Runnable.run()** que esperar a que ese método alcance un punto donde se compruebe un indicador de “cancelación”, o algún otro lugar en el que el programador esté listo para abandonar el método. Cuando salimos abruptamente de una tarea bloqueada, puede que tengamos que efectuar tareas de limpieza de los recursos. Debido a esto, salir en mitad del método **run()** de una tarea se parece más a generar una excepción que a ninguna otra cosa, por lo que en las hebras Java se utilizan las excepciones para este tipo de interrupción¹⁶ (esto significa que estamos caminando sobre el filo que separa el uso adecuado e inadecuado de las excepciones, porque quiere decir que a menudo se las emplea para control del flujo del ejecución). Para volver a un estado aceptable conocido cuando se termina una tarea de esta forma, es necesario analizar con cuidado los caminos de ejecución del código y escribir la cláusula **catch** para limpiar adecuadamente las cosas.

Para poder terminar una tarea bloqueada, la clase **Thread** contiene el método **interrupt()**. Este método activa el indicador de estado interrumpido para la hebra. Una hebra que tenga activado el indicador de estado interrumpido generará una interrupción **InterruptedException** si ya está bloqueada o si se trata de realizar una operación de bloqueo. El indicador de estado interrumpido será reinicializado cuando se genere la excepción o si la tarea llama a **Thread.interrupted()**. Como puede ver, **Thread.interrupted()** proporciona una segunda forma de abandonar el bucle **run()**, sin generar una excepción.

Para invocar **interrupt()**, es necesario disponer de un objeto **Thread**. Puede que el lector haya observado que la nueva biblioteca **concurrent** parece evitar la manipulación directa de objetos **Thread** y trata en su lugar de realizar todas las tareas a través de objetos **Executors**. Si llamamos a **shutdownNow()** sobre un objeto **Executor**, se generará una llamada **interrupt()** dirigida a cada una de las hebras que el ejecutor haya iniciado. Esto tiene bastante sentido, porque normalmente querremos terminar de una sola vez todas las tareas para un ejecutor concreto, cuando hayamos finalizado parte de un programa o el programa completo. Sin embargo, hay veces en las que puede que sólo queramos interrumpir una única tarea. Si estamos usando ejecutores, podemos obtener información sobre el contexto de una tarea en el momento de iniciarla llamando a **submit()** en lugar de a **execute()**. **submit()** devuelve un genérico **Future<?>**, con un parámetro no especificado (porque nunca se va a llamar **get()** para ese parámetro); el motivo de guardar este tipo de objeto **Future** es que se puede invocar **cancel()** sobre el objeto y usarlo para interrumpir una tarea completa. Si pasamos el valor **true** a **cancel()**, esta hebra tendrá permiso para invocar **interrupt()** sobre dicha hebra, con el fin de detenerla; por tanto **cancel()** es una forma de interrumpir hebras individuales iniciadas mediante un ejecutor.

He aquí un ejemplo que muestra los fundamentos básicos de utilización de **interrupt()** empleando ejecutores:

```
//: concurrency/Interrupting.java
// Interrupción de una hebra bloqueada.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class SleepBlocked implements Runnable {
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(100);
        } catch(InterruptedException e) {
            print("InterruptedException");
        }
    }
}
```

¹⁶ Sin embargo, las excepciones nunca se generan asincronamente. Por tanto, no hay ningún riesgo de que algo se interrumpe en mitad de la ejecución de una instrucción o de una llamada a método. Y, siempre que utilicemos la estructura **try-finally** al utilizar objetos mutex (en lugar de la palabra clave **synchronized**), esos mutex serán liberados automáticamente si se genera una excepción.

```

        }
        print("Exiting SleepBlocked.run()");
    }
}

class IOBlocked implements Runnable {
    private InputStream in;
    public IOBlocked(InputStream is) { in = is; }
    public void run() {
        try {
            print("Waiting for read():");
            in.read();
        } catch(IOException e) {
            if(Thread.currentThread().isInterrupted()) {
                print("Interrupted from blocked I/O");
            } else {
                throw new RuntimeException(e);
            }
        }
        print("Exiting IOBlocked.run()");
    }
}

class SynchronizedBlocked implements Runnable {
    public synchronized void f() {
        while(true) // Nunca libera el bloqueo
            Thread.yield();
    }
    public SynchronizedBlocked() {
        new Thread() {
            public void run() {
                f(); // Bloqueo adquirido por esta hebra
            }
        }.start();
    }
    public void run() {
        print("Trying to call f()");
        f();
        print("Exiting SynchronizedBlocked.run()");
    }
}

public class Interrupting {
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static void test(Runnable r) throws InterruptedException{
        Future<?> f = exec.submit(r);
        TimeUnit.MILLISECONDS.sleep(100);
        print("Interrupting " + r.getClass().getName());
        f.cancel(true); // Interrumpe si se está ejecutando
        print("Interrupt sent to " + r.getClass().getName());
    }
    public static void main(String[] args) throws Exception {
        test(new SleepBlocked());
        test(new IOBlocked(System.in));
        test(new SynchronizedBlocked());
        TimeUnit.SECONDS.sleep(3);
        print("Aborting with System.exit(0)");
        System.exit(0); // ... puesto que las 2 últimas interrupciones fallaron
    }
}

```

```

} /* Output: (95% match)
Interrupting SleepBlocked
InterruptedException
Exiting SleepBlocked.run()
Interrupt sent to SleepBlocked
Waiting for read():
Interrupting IOBlocked
Interrupt sent to IOBlocked
Trying to call f()
Interrupting SynchronizedBlocked
Interrupt sent to SynchronizedBlocked
Aborting with System.exit(0)
*///:~

```

Cada tarea representa un tipo diferente de bloqueo. **SleepBlock** es un ejemplo de bloqueo interrumpible, mientras que **IOBlocked** y **SynchronizedBlocked** son bloqueos no interrumpibles.¹⁷ El programa demuestra que las operaciones de E/S y de espera sobre un bloqueo **synchronized** no sean interrumpibles, cosa que también puede deducirse examinando el código: no se requiere ninguna rutina de tratamiento de **InterruptedException** ni para la E/S ni para los intentos de invocar un método sincronizado.

Las dos primeras clases son bastante simples: el método **run()** llama a **sleep()** en la primera clase y a **read()** en la segunda. Sin embargo, para ilustrar **SynchronizedBlocked**, debemos primero adquirir el bloqueo. Esto se lleva a cabo en el constructor creando una instancia de una clase **Thread** anónima que adquiere el bloqueo del objeto invocando **f()** (la hebra debe ser diferente de aquella que está dirigiendo **run()** para **SynchronizedBlock**, porque una misma hebra sí que puede adquirir múltiples veces un bloqueo sobre un objeto). Dado que **f()** nunca vuelve, ese bloqueo nunca es liberado. **SynchronizedBlock.run()** trata de invocar **f()** y se queda bloqueado esperando a que el bloqueo se libere.

Podemos ver, analizando la salida, que se puede interrumpir una llamada a **sleep()** (o cualquier llamada que requiera que capturemos **InterruptedException**). Sin embargo, no se puede interrumpir una tarea que esté tratando de adquirir un bloqueo sincronizado o que esté tratando de efectuar una operación de E/S. Esto es un poco desconcertante, especialmente si estamos creando una tarea que realice operaciones de E/S, porque significa que la E/S tiene el potencial de bloquear el programa multihilo. Obviamente esto constituiría un auténtico problema, especialmente para programas basados en la Web.

Una solución un tanto drástica pero en ocasiones bastante efectiva para este problema, consiste en cerrar el recurso subyacente que hace que la tarea esté bloqueada:

```

//: concurrency/CloseResource.java
// Interrupción de una tarea bloqueada
// cerrando el recurso subyacente.
// {RunByHand}
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class CloseResource {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InputStream socketInput =
            new Socket("localhost", 8080).getInputStream();
        exec.execute(new IOBlocked(socketInput));
        exec.execute(new IOBlocked(System.in));
        TimeUnit.MILLISECONDS.sleep(100);
        print("Shutting down all threads");
        exec.shutdownNow();
    }
}

```

¹⁷ Algunas versiones del JDK también proporcionaban soporte para la instrucción **InterruptedException**. Sin embargo, este soporte sólo estaba parcialmente implementado, y únicamente en algunas plataformas. Si se genera esta excepción hace que los objetos de E/S sean inutilizables. Resulta poco probable que las futuras versiones sigan proporcionando soporte para esta excepción.

```

TimeUnit.SECONDS.sleep(1);
print("Closing " + socketInput.getClass().getName());
socketInput.close(); // Libera la hebra bloqueada
TimeUnit.SECONDS.sleep(1);
print("Closing " + System.in.getClass().getName());
System.in.close(); // Libera la hebra bloqueada
}
} /* Output: (85% match)
Waiting for read():
Waiting for read():
Shutting down all threads
Closing java.net.SocketInputStream
Interrupted from blocked I/O
Exiting IOBlocked.run()
Closing java.io.BufferedInputStream
Exiting IOBlocked.run()
*///:-
```

Después de invocar `shutdownNow()`, los retardos utilizados antes de llamar a `close()` para los dos flujos de datos de entrada permiten resaltar que las tareas se desbloquean una vez que el recurso subyacente se ha cerrado. Resulta interesante observar que la interrupción aparece cuando estamos cerrando un objeto `Socket` pero no al cerrar `System.in`.

Afortunadamente, las clases `nio` presentadas en el Capítulo 18, *Entrada/salida*, permiten una interrupción más civilizada de las interrupciones de E/S. Los canales `nio` bloqueados responden automáticamente a las interrupciones:

```

//: concurrency/NIOInterruptedException.java
// Interrupción de un canal NIO bloqueado.
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
    private final SocketChannel sc;
    public NIOBlocked(SocketChannel sc) { this.sc = sc; }
    public void run() {
        try {
            print("Waiting for read() in " + this);
            sc.read(ByteBuffer.allocate(1));
        } catch(ClosedByInterruptException e) {
            print("ClosedByInterruptException");
        } catch(AsynchronousCloseException e) {
            print("AsynchronousCloseException");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        print("Exiting NIOBlocked.run() " + this);
    }
}

public class NIOInterruptedException {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InetSocketAddress isa =
            new InetSocketAddress("localhost", 8080);
        SocketChannel sc1 = SocketChannel.open(isa);
        SocketChannel sc2 = SocketChannel.open(isa);
```

```

Future<?> f = exec.submit(new NIOBlocked(sc1));
exec.execute(new NIOBlocked(sc2));
exec.shutdown();
TimeUnit.SECONDS.sleep(1);
// Producir una interrupción mediante cancel:
f.cancel(true);
TimeUnit.SECONDS.sleep(1);
// Liberar el bloqueo cerrando un canal:
sc2.close();
}
} /* Output: (Sample)
Waiting for read() in NIOBlocked@7a84e4
Waiting for read() in NIOBlocked@15c7850
ClosedByInterruptException
Exiting NIOBlocked.run() NIOBlocked@15c7850
AsynchronousCloseException
Exiting NIOBlocked.run() NIOBlocked@7a84e4
*///:-
```

Como se muestra, también podemos cerrar el canal subyacente para liberar el bloqueo, aunque esto sólo debería ser necesario en raras ocasiones. Observe que utilizando `execute()` para iniciar ambas tareas e invocar `e.shutdownNow()` permite terminar fácilmente cualquier cosa. La captura del objeto `Future` en el ejemplo anterior sólo era necesaria para enviar la interrupción a una hebra y no a la otra.¹⁸

Ejercicio 18: (2) Cree una clase que no sea de tipo tarea con un método que llame a `sleep()` durante un intervalo de larga duración. Cree una tarea que llame al método contenido en la clase que haya definido. En `main()`, inicie la tarea y luego llame a `interrupt()` para terminarla. Asegúrese que la tarea termina de manera segura.

Ejercicio 19: (4) Modifique `OrnamentalGarden.java` para que utilice `interrupt()`.

Ejercicio 20: (1) Modifique `CachedThreadPool.java` para que todas las tareas reciban una interrupción (con `interrupt()`) antes de completarse.

Tareas bloqueadas por un mutex

Como vimos en `Interrupting.java`, si tratamos de llamar a un método sincronizado sobre un objeto cuyo bloqueo ya haya sido adquirido, la tarea llamante será suspendida (bloqueada) hasta que el objeto esté disponible. El siguiente ejemplo muestra cómo puede una misma tarea adquirir múltiples veces el mismo mutex:

```

//: concurrency/MultiLock.java
// Una hebra puede volver a adquirir el mismo bloqueo.
import static net.mindview.util.Print.*;
public class MultiLock {
    public synchronized void f1(int count) {
        if(count-- > 0) {
            print("f1() calling f2() with count " + count);
            f2(count);
        }
    }
    public synchronized void f2(int count) {
        if(count-- > 0) {
            print("f2() calling f1() with count " + count);
            f1(count);
        }
    }
    public static void main(String[] args) throws Exception {
        final MultiLock multiLock = new MultiLock();
        new Thread() {
```

¹⁸ Ervin Varga me ayudó en la investigaciones relacionadas con esta sección.

```

        public void run() {
            multiLock.f1(10);
        }
    }.start();
}
} /* Output:
f1() calling f2() with count 9
f2() calling f1() with count 8
f1() calling f2() with count 7
f2() calling f1() with count 6
f1() calling f2() with count 5
f2() calling f1() with count 4
f1() calling f2() with count 3
f2() calling f1() with count 2
f1() calling f2() with count 1
f2() calling f1() with count 0
*///:-
```

En **main()**, se crea un objeto **Thread** para invocar **f1()**; luego **f1()** y **f2()** se llaman el uno al otro hasta que el valor de **count** pasa a ser cero. Puesto que la tarea ya ha adquirido el bloqueo del objeto **multiLock** dentro de la primera llamada a **f1()**, esa misma tarea lo estará volviendo a adquirir en la llamada a **f2()**, y así sucesivamente. Esto tiene sentido, porque una tarea debe ser capaz de invocar otros métodos sincronizados contenidos dentro del mismo objeto, ya que dicha tarea ya posee el bloqueo.

Como hemos observado anteriormente al hablar de la E/S ininterrumpible, cada vez que una tarea pueda bloquearse de tal forma que no pueda ser interrumpida, existirá la posibilidad de que el programa se quede bloqueado. Una de las características añadidas a las bibliotecas de concurrencia de Java SE5 es la posibilidad de que las tareas bloqueadas en bloqueos de tipo **ReentrantLock** sean interrumpidas, a diferencia de las tareas bloqueadas en métodos sincronizados o secciones críticas:

```

//: concurrency/Interrupting2.java
// Interrupción de una tarea bloqueada con un bloqueo ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
    private Lock lock = new ReentrantLock();
    public BlockedMutex() {
        // Adquirirlo directamente, para demostrar la interrupción
        // de la tarea bloqueada en un bloqueo ReentrantLock:
        lock.lock();
    }
    public void f() {
        try {
            // Esto no estará nunca disponible para una segunda tarea
            lock.lockInterruptibly(); // Llamada especial
            print("lock acquired in f()");
        } catch(InterruptedException e) {
            print("Interrupted from lock acquisition in f()");
        }
    }
}

class Blocked2 implements Runnable {
    BlockedMutex blocked = new BlockedMutex();
    public void run() {
        print("Waiting for f() in BlockedMutex");
        blocked.f();
        print("Broken out of blocked call");
    }
}
```

```

public class Interrupting2 {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Blocked2());
        t.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Issuing t.interrupt()");
        t.interrupt();
    }
} /* Output:
Waiting for f() in BlockedMutex
Issuing t.interrupt()
Interrupted from lock acquisition in f()
Broken out of blocked call
*///:-

```

La clase **BlockedMutex** tiene un constructor que adquiere el propio bloqueo del objeto y nunca lo libera. Por esa razón, si tratamos de invocar **f()** desde una segunda tarea (diferente de la que haya creado el objeto **BlockedMutex**), siempre nos quedaremos bloqueados, porque el objeto **Mutex** no puede ser adquirido. En **Blocked2**, el método **run()** se detendrá en la llamada a **blocked.f()**. Cuando ejecutamos el programa, vemos que, a diferencia de una llamada de E/S, **interrupt()** permite salir de una llamada que esté bloqueada por un mutex.¹⁹

Comprobación de la existencia de una interrupción

Observe que cuando invocamos **interrupt()** para una hebra, el único instante en que se produce la interrupción es cuando la tarea entra, o se encuentra ya dentro, de una operación bloqueante (excepto, como hemos visto, en el caso de los métodos sincronizados bloqueados o la E/S ininterrumpibles, en cuyo caso no hay nada que podamos hacer). Pero ¿qué sucede si hemos escrito código que pueda o no hacer esa llamada bloqueante dependiendo de las condiciones en las que se la ejecute? Si sólo podemos salir generando una excepción en una llamada bloqueante, no siempre seremos capaces de abandonar el bucle **run()**. Por tanto, si invocamos **interrupt()** para detener una tarea, la tarea necesita una segunda forma de salir en caso de que el bucle **run()** no esté realizando ninguna llamada bloqueante.

Esta oportunidad se presenta gracias al *estado interrumpido*, que es fijado por la llamada a **interrupt()**. Comprobamos si la tarea está en estado interrumpido llamando a **interrupted()**. Esto no sólo nos dice si se ha llamado a **interrupt()**, sino que también borra el estado interrumpido. Borrar el estado interrumpido garantiza que el sistema no nos notifique dos veces que se ha interrumpido una tarea. La notificación la recibiremos a través de una única excepción **InterruptedException** o una única comprobación con éxito del método **Thread.interrupted()**. Si queremos comprobar de nuevo si hemos sido interrumpidos, podemos almacenar el resultado al invocar **Thread.interrupted()**.

El siguiente ejemplo muestra la sintaxis típica que se usaría en el método **run()** para gestionar ambas posibilidades (bloqueada y no bloqueada) cuando está activado el estado interrumpido:

```

//: concurrency/InterruptingIdiom.java
// Sintaxis general para interrumpir una tarea.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class NeedsCleanup {
    private final int id;
    public NeedsCleanup(int ident) {
        id = ident;
        print("NeedsCleanup " + id);
    }
    public void cleanup() {
        print("Cleaning up " + id);
    }
}

```

¹⁹ Observe que, aunque resulta poco probable, la llamada a **t.interrupt()** podría llegar a suceder antes que la llamada a **blocked.f()**.

```

class Blocked3 implements Runnable {
    private volatile double d = 0.0;
    public void run() {
        try {
            try {
                while(!Thread.interrupted()) {
                    // punto1
                    NeedsCleanup n1 = new NeedsCleanup(1);
                    // Iniciar try-finally inmediatamente después de la definición
                    // de n1, para garantizar una limpieza apropiada de n1:
                    try {
                        print("Sleeping");
                        TimeUnit.SECONDS.sleep(1);
                        // Punto2
                        NeedsCleanup n2 = new NeedsCleanup(2);
                        // Garantizar una limpieza apropiada de n2:
                        try {
                            print("Calculating");
                            // Una operación no bloqueante de larga duración:
                            for(int i = 1; i < 2500000; i++)
                                d = d + (Math.PI + Math.E) / d;
                            print("Finished time-consuming operation");
                        } finally {
                            n2.cleanup();
                        }
                    } finally {
                        n1.cleanup();
                    }
                }
                print("Exiting via while() test");
            } catch(InterruptedException e) {
                print("Exiting via InterruptedException");
            }
        }
    }
}

public class InterruptingIdiom {
    public static void main(String[] args) throws Exception {
        if(args.length != 1) {
            print("usage: java InterruptingIdiom delay-in-mS");
            System.exit(1);
        }
        Thread t = new Thread(new Blocked3());
        t.start();
        TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
        t.interrupt();
    }
} /* Output: (Sample)
NeedsCleanup 1
Sleeping
NeedsCleanup 2
Calculating
Finished time-consuming operation
Cleaning up 2
Cleaning up 1
NeedsCleanup 1
Sleeping
Cleaning up 1
Exiting via InterruptedException
*///:-

```

La clase **NeedsCleanup** enfatiza la necesidad de efectuar una limpieza apropiada de los recursos si se abandona el bucle mediante una excepción. Observe que todos los recursos de **NeedsCleanup** creados en **Blocked3.run()** deben ir seguidos inmediatamente de cláusulas **try-finally** para garantizar que siempre se invoque el método **cleanup()**.

Debe proporcionar al programa un argumento de linea de comandos que es el retardo en milisegundos antes de que se invoque **interrupt()**. Utilizando diferentes retardos, podemos salir de **Blocked3.run()** en diferentes puntos del bucle: en la llamada **sleep()** bloqueante y en el cálculo matemático no bloqueante. Como vemos, si se invoca **interrupt()** después del comentario “punto2” (durante la operación no bloqueante), se completa primero el bucle, después se destruyen todos los objetos locales y finalmente se sale del bucle por la parte superior gracias a la instrucción **while**. Sin embargo, si se invoca **interrupt()** entre “punto1” y “punto2” (después de la instrucción **while** pero antes o durante la operación bloqueante **sleep()**), la tarea sale a través de la excepción **InterruptedException**, la primera vez que se intente una operación bloqueante. En ese caso, sólo se limpian los objetos **NeedsCleanup** que hayan sido creados hasta el punto en que se genera la excepción, y tenemos la oportunidad de crear cualquier otra tarea de limpieza dentro de la cláusula **catch**.

Una clase diseñada para responder a una interrupción deberá establecer una política para garantizar que permanezca en un estado coherente. Esto significa, generalmente, que la creación de todos los objetos que requieran limpieza deberá ir seguida por cláusulas **try-finally** de modo que esa limpieza tenga lugar independientemente de cómo se salga del bucle **run()**. El código de este tipo puede funcionar bien, aunque hay que señalar que, debido a la falta de llamadas automáticas a destructores en Java, depende de que el programador de clientes describa las cláusulas **try-finally** apropiadas.

Cooperación entre tareas

Como hemos visto, cuando se utilizan hebras para ejecutar más de una tarea simultáneamente, podemos evitar que unas tareas interfieran con los recursos de otras utilizando un bloqueo (mutex) para sincronizar el comportamiento de las dos tareas. En otras palabras, si dos tareas están interfiriendo en lo que respecta a un recurso compartido (normalmente la memoria), utilizamos un mutex para permitir que sólo una tarea acceda en cada momento a dicho recurso.

Con ese problema resuelto, el siguiente paso consiste en aprender lo que hay que hacer para que las tareas puedan cooperar entre sí, de modo que múltiples tareas puedan trabajar juntas para resolver un cierto problema. Ahora, la cuestión no es qué interferencias se producen entre unas tareas y otras, sino cómo trabajar al unísono, ya que partes de un cierto problema deberán ser resueltas antes de que se puedan resolver otras partes. Esto se parece bastante a la planificación de proyectos: primero hay que hacer los cimientos de la casa, pero mientras, se pueden ir haciendo los perfiles de aluminio o fabricando los ladrillos, y estas dos tareas tienen que estar finalizadas antes de que el resto de la casa pueda completarse. Asimismo, la fontanería deberá estar terminada antes de hacer las paredes, las paredes deberán haber sido acabadas antes de poder finalizar los interiores, etc. Algunas de estas tareas pueden hacerse en paralelo, pero ciertos pasos requieren que determinadas tareas previas se completen antes de poder continuar.

La cuestión clave cuando hay una serie de tareas cooperando es la negociación que se produce entre dichas tareas. Para llevar a cabo esa negociación, utilizamos la misma base: el mutex, que en este caso garantiza que sólo haya una tarea que pueda responder a una señal. Esto elimina cualquier posible condición de carrera. Además del mutex, tenemos que añadir una forma de que una tarea suspenda su ejecución hasta que un cierto estado externo cambie (por ejemplo, “la fontanería ha sido acabada”), lo que indicará que será el momento de que dicha tarea continúe. En esta sección, vamos a examinar el tema de la negociación entre tareas, que se puede implementar utilizando los métodos **wait()** y **notifyAll()** de **Object**. La biblioteca de concurrencia de Java SE5 también proporciona objetos **Condition** con métodos **await()** y **signal()**. Veremos los problemas que pueden surgir, junto con sus correspondientes soluciones.

wait() y notifyAll()

wait() nos permite esperar a que se produzca un cambio en cierta condición que está fuera del control del método actual. A menudo, esta condición será modificada por otra tarea. Lo que no queremos es permanecer inactivos dentro de un bucle mientras comprobamos la condición de la tarea; este tipo de espera se denomina *espera activa*, y representa usualmente un mal uso de los ciclos de procesador. Por ello, el método **wait()** suspende la tarea mientras espera a que el mundo exterior cambie, y sólo cuando tiene lugar una llamada a **notify()** o **notifyAll()** (que sugieren que puede haber ocurrido un cierto suceso de interés) se despertará la tarea y comprobará si se han producido cambios. De este modo, **wait()** proporciona una forma de sincronizar las actividades entre las tareas.

Es importante comprender que **sleep()** no libera el bloqueo del objeto cuando se lo invoca, pero tampoco lo hace **yield()**. Por otro lado, cuando una tarea entra en una llamada a **wait()** dentro de un método, se suspende la ejecución de esa hebra y se libera el bloqueo sobre ese objeto. Puesto que **wait()** libera el bloqueo, quiere decir que ese bloqueo podrá ser adquirido por otra tarea, por lo que durante una espera con **wait()** podrán invocarse otros métodos sincronizados en el (ahora desbloqueado) objeto. Esto resulta esencial, porque esos otros métodos son normalmente los que provocan el cambio que hacen que sea interesante que se vuelva a despertar la tarea suspendida. Así, cuando llamamos a **wait()**, estamos diciendo: "he hecho todo lo que puedo por ahora, así que voy a esperar aquí, pero quiero permitir que otras operaciones sincronizadas puedan tener lugar, si es que pueden".

Existen dos formas de **wait()**. Una versión toma un argumento en milisegundos que tiene el mismo significado que **sleep()**: "efectúa una pausa durante este período de tiempo". Pero, a diferencia de **sleep()**, con **wait(pausa)**:

1. El bloqueo del objeto se libera durante la ejecución de **wait()**.
2. También se puede salir de la llamada a **wait()** debido a la recepción de **notify()** o **notifyAll()**, además de permitir que la temporización finalice.

La segunda forma de **wait()** más común no toma ningún argumento. Este tipo de **wait()** continuará indefinidamente hasta que la hebra reciba un mensaje **notify()** o **notifyAll()**.

Una aspecto bastante distintivo de **wait()**, **notify()** y **notifyAll()** es que estos métodos forman parte de la clase base **Object** y no de **Thread**. Aunque esto parece algo extraño a primera vista (tener algo exclusivo del mecanismo de hebras como parte de la clase base universal), resulta esencial porque estos métodos manipulan el bloqueo que también forma parte de todos los objetos. Como resultado, podemos incluir una llamada a **wait()** dentro de cualquier método sincronizado, independientemente de si dicha clase amplía a **Thread** o implementa **Runnable**. De hecho, el único lugar en que se puede llamar a **wait()**, **notify()** o **notifyAll()** es dentro de un método o bloque **synchronized**, (**sleep()** puede invocarse dentro de métodos no sincronizados ya que no manipula el bloqueo). Si invocamos cualquiera de estos métodos dentro de un método que no sea de tipo **synchronized**, el programa se podrá compilar, pero al ejecutarlo se obtendrá una excepción **IllegalMonitorStateException** con el poco intuitivo mensaje de "current thread not owner" (la hebra actual no es la propietaria). Este mensaje quiere decir que la tarea que está invocando **wait()**, **notify()** o **notifyAll()** debe "poseer" (adquirir) el bloqueo del objeto antes de poder invocar ninguno de sus métodos.

Podemos pedir a otro objeto que realice una operación que manipula su propio bloqueo. Para hacer esto, tenemos primero que capturar el bloqueo de ese objeto. Por ejemplo, si queremos enviar **notifyAll()** a un objeto **x**, deberemos hacerlo dentro de un bloque sincronizado que adquiere el bloqueo para **x**:

```
synchronized(x) {
    x.notifyAll();
}
```

Examinemos un ejemplo simple. **WaxOMatic.java** tiene dos procesos: uno para aplicar cera a un objeto coche (representado por un objeto **Car**) y otro para pulirlo. La tarea de pulido no puede llevar a cabo su trabajo hasta que haya finalizado la tarea de aplicación de la cera y la tarea de aplicación de cera hasta que la tarea de pulido haya finalizado, antes de poner otra capa de cera. Tanto **WaxOn** como **WaxOff** utilizan el objeto **Car**, que emplea **wait()** y **notifyAll()** para suspender y reiniciar las tareas mientras éstas están esperando a que una condición cambie:

```
//: concurrency/waxomatic/WaxOMatic.java
// Cooperación básica entre tareas.
package concurrency.waxomatic;
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Car {
    private boolean waxOn = false;
    public synchronized void waxed() {
        waxOn = true; // Listo para pulido
        notifyAll();
    }
    public synchronized void buffed() {
        waxOn = false; // Listo para otra capa de cera
    }
}
```

```

        notifyAll();
    }
    public synchronized void waitForWaxing()
    throws InterruptedException {
        while(waxOn == false)
            wait();
    }
    public synchronized void waitForBuffing()
    throws InterruptedException {
        while(waxOn == true)
            wait();
    }
}

class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax On task");
    }
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5); // Ejecutar durante cierto tiempo...
        exec.shutdownNow(); // Interrumpir todas las tareas
    }
}

```

```

} /* Output: (95% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax On task
Exiting via interrupt
Ending Wax Off task
*///:-
```

Aquí, **Car** tiene un único campo booleano **waxOn**, que indica el estado del proceso de aplicación-pulido.

En **waitForWaxing()**, se comprueba el indicador **waxOn** y, si es **false**, se suspende la tarea llamante invocando **wait()**. Es importante que esto tenga lugar dentro un método sincronizado, en el que la tarea haya adquirido el bloqueo. Cuando invocamos **wait()**, la hebra se suspende y el bloqueo se libera. Resulta esencial que se libere el bloqueo, porque para cambiar con seguridad el estado del objeto (por ejemplo, para cambiar **waxOn** a **true**, que es algo que tiene que ocurrir para que la tarea suspendida pueda continuar), dicho bloqueo debe estar disponible para que lo adquiera alguna otra tarea. En este ejemplo, cuando otra tarea invoca **waxed()** para indicar que es el momento de hacer algo, hay que adquirir el bloqueo para poder cambiar **waxOn** a **true**. Después, **waxed()** invoca a **notifyAll()**, que despierta a la tarea que había sido suspendida en la llamada a **wait()**. Para que la tarea pueda despertarse de una llamada a **wait()**, deberá primero readquirir el bloqueo que liberó en el momento de entrara en **wait()**. La tarea no se despertará hasta que dicho bloqueo esté disponible.²⁰

WaxOn.run() representa el primer paso dentro de un proceso de aplicación de la cera al coche, así que lleva a cabo su operación: una llamada a **sleep()** para simular el tiempo necesario para la aplicación de la cera. A continuación, le dice al coche que la aplicación de la cera se ha completado y llama a **waitForBuffing()**, que suspende esta tarea con una llamada a **wait()** hasta que la tarea **WaxOff** llama a **buffed()** para el coche, cambiando el estado y llamando a **notifyAll()**. **WaxOff.run()**, por otro lado, entra inmediatamente en **waitForWaxing()** y se suspende, por tanto, hasta que la cera haya sido aplicada por **WaxOn** y se invoque a **waxed()**. Cuando se ejecuta este programa, podemos ver cómo este proceso en dos pasos se repite continuamente a medida que las dos tareas se ceden la una a la otra el control. Después de cinco segundos, **interrupt()** detiene ambas hebras; cuando se invoca **shutdownNow()** para un objeto **ExecutorService**, éste invoca a **interrupt()** para todas las tareas que esté controlando.

El ejemplo anterior resalta el hecho de que hay que rodear una llamada a **wait()** con un bucle **while** que compruebe la condición o condiciones de interés. Esto es importante porque:

- Puede que tengamos múltiples tareas que estén esperando a un determinado bloqueo por la misma razón, y la primera tarea que se despierte puede cambiar la situación (incluso si no hacemos esto, alguien podría heredar de nuestra clase y hacerlo). En este caso, dicha tarea debería ser suspendida de nuevo hasta que su condición de interés cambiara.
- En el momento en que esta tarea se despierte de su llamada a **wait()**, es posible que alguna otra tarea haya cambiado las cosas de modo que esta tarea sea incapaz de realizar su operación en este momento, o no le interesa realizarla. De nuevo, debería volver a ser suspendida invocando de nuevo a **wait()**.
- También es posible que las tareas estuvieran esperando el bloqueo del objeto *por razones distintas* (en cuyo caso, es necesario utilizar **notifyAll()**). En este caso, necesitamos comprobar si se nos ha despertado por la razón correcta y, en caso contrario, volver a invocar **wait()**.

Por tanto, resulta esencial que comprobemos nuestra condición de interés concreta y que volvamos a **wait()** si dicha condición no se cumple. La estructura sintáctica para hacer esto es un bucle **while**.

Ejercicio 21: (2) Cree dos clases **Runnable**, una con un método **run()** que se inicie e invoque **wait()**. La segunda clase debe capturar las referencias del objeto **Runnable**. Su método **run()** debería invocar **notifyAll()** para la

²⁰ En algunas plataformas, existe una tercera forma de salir de una llamada a **wait()**: el denominado *despertar espúreo*. Un despertar espúreo significa, esencialmente, que una hebra puede abandonar el bloqueo prematuramente (mientras está esperando de acuerdo con un semáforo o con una variable de condición) sin que ello venga desencadenado por un mensaje a **notify()** o **notifyAll()** (o sus equivalentes para los nuevos objetos **Condition**). La hebra simplemente se despierta aparentemente por sí misma. Estos despertares espúreos existen porque la implementación de hebras POSIX, o sus equivalentes, no es siempre tan sencilla como debería ser en algunas plataformas. Permitir estos despertares espúreos hace que la tarea de construir una biblioteca como *pthreads* sea más fácil en esas plataformas.

primera tarea después de que haya pasado un cierto número de segundos, de modo que la primera tarea pueda mostrar un mensaje. Pruebe las dos clases utilizando un objeto **Executor**.

Ejercicio 22: (4) Cree un ejemplo de *espera activa*. Una tarea debe dormir durante un cierto tiempo y luego asignar el valor **true** a un indicador. La segunda tarea deberá comprobar dicho indicador dentro de un bucle **while** (ésta es la espera activa) y cuando el indicador sea **true**, deberá asignarle de nuevo el valor **false** e informar del cambio a través de la consola. Observe cuánto tiempo desperdicia el programa dentro de la espera activa, y cree una segunda versión del programa que emplee **wait()** en lugar de esa espera activa.

Señales perdidas

Cuando se coordinan dos hebras utilizando **notify()**/**wait()** o **notifyAll()**/**wait()**, resulta posible perder una señal. Suponga que **T1** es una hebra que notifica a **T2**, y que las dos hebras se implementan utilizando el siguiente enfoque (incorrecto):

```
T1:
synchronized(sharedMonitor) {
    <condición de configuración para T2>
    sharedMonitor.notify();
}

T2:
while(someCondition) {
    // Punto 1
    synchronized(sharedMonitor) {
        sharedMonitor.wait();
    }
}
```

La *<condición de configuración para T2>* es una acción destinada a impedir que **T2** invoque **wait()**, si es que no lo ha hecho ya.

Suponga que **T2** evalúa **someCondition** y encuentra que esa condición es verdadera. En **Punto 1**, el planificador de hebras podría commutar a **T1**. **T1** ejecutaría su configuración, y entonces invocaría **notify()**. Cuando **T2** continúe ejecutándose, es demasiado tarde para que **T2** se dé cuenta de que la condición se ha modificado mientras tanto, por lo que entrará ciegamente en **wait()**. El mensaje **notify()** se perderá y **T2** esperará indefinidamente a recibir una señal que ya había sido enviada, lo que producirá un interbloqueo.

La solución consiste en impedir la condición de carrera que afecta a la variable **someCondition**. He aquí la técnica correcta para **T2**:

```
synchronized(sharedMonitor) {
    while(someCondition)
        sharedMonitor.wait();
}
```

Ahora, si se ejecuta primero **T1**, cuando el control vuelve a **T2** éste podrá ver que la condición se ha modificado, y *no* entrará en **wait()**. A la inversa, si se ejecuta primero **T2**, entrará en **wait()** y será posteriormente despertado por **T1**. De este modo, no puede perderse ninguna señal.

notify() y notifyAll()

Puesto que técnicamente podría darse el caso de que hubiera más de una tarea esperando con **wait()** con un único objeto **Car**, resulta más seguro invocar **notifyAll()** que simplemente **notify()**. Sin embargo, la estructura del programa anterior es tal que sólo habrá una tarea esperando con **wait()**, por lo que podemos perfectamente usar **notify()** en lugar de **notifyAll()**.

Utilizar **notify()** en lugar de **notifyAll()** es una optimización. Sólo se despertará con **notify()** a una de las tareas de las muchas posibles que estén esperando con bloqueo, así que si tratamos de utilizar **notify()** debemos estar seguros de que se despertará la tarea correcta. Además, todas las tareas deberán estar esperando por la misma condición si queremos utilizar **notify()**, porque si hubiera tareas que estuvieran esperando por condiciones diferentes, no sabríamos si se despertará la tarea correcta. Si empleamos **notify()**, sólo una tarea deberá aprovecharse cuando se modifique la condición. Finalmente, estas

restricciones deben cumplirse para todas las subclases posibles. Si no se puede cumplir alguna de estas reglas, es necesario emplear **notifyAll()** en lugar de **notify()**.

Una de las afirmaciones confusas que a menudo se hacen a la hora de explicar el mecanismo de herramientas de Java es que **notifyAll()** desperta "a todas las tareas en espera". ¿Quiere esto decir que cualquier tarea que se encuentre esperando con **wait()** en cualquier lugar del programa, será despertada por cualquier llamada a **notifyAll()**? En el siguiente ejemplo, el código asociado con **Task2** demuestra que esto no es así; de hecho, cuando se invoque **notifyAll()** para un cierto bloqueo sólo se despertarán las tareas que estén esperando por ese bloqueo concreto:

```
//: concurrency/NotifyVsNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blocker {
    synchronized void waitingCall() {
        try {
            while(!Thread.interrupted()) {
                wait();
                System.out.print(Thread.currentThread() + " ");
            }
        } catch(InterruptedException e) {
            // OK salir de esta forma
        }
    }
    synchronized void prod() { notify(); }
    synchronized void prodAll() { notifyAll(); }
}

class Task implements Runnable {
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
    // Un objeto Blocker separado:
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

public class NotifyVsNotifyAll {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Task());
        exec.execute(new Task2());
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            boolean prod = true;
            public void run() {
                if(prod)
                    System.out.print("\nnotify() ");
                Task.blocker.prod();
                prod = false;
            } else {
                System.out.print("\nnotifyAll() ");
                Task.blocker.prodAll();
                prod = true;
            }
        }, 400, 400); // Ejecutar cada 4 segundos
    }
}
```

```

        TimeUnit.SECONDS.sleep(5); // Ejecutar durante un tiempo...
        timer.cancel();
        System.out.println("\nTimer canceled");
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.print("Task2.blocker.prodAll() ");
        Task2.blocker.prodAll();
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.println("\nShutting down");
        exec.shutdownNow(); // Interrumpir todas las tareas
    }
} /* Output: (Sample)
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-5,5,main] Thread[pool-1-thread-4,5,main]
Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-
thread-2,5,main] Thread[pool-1-thread-3,5,main]
Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
Timer canceled
Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
Shutting down
*///:-
```

Task y **Task2** tienen, cada uno de ellos, su propio objeto **Blocker**, de modo que cada objeto **Task** se bloquea sobre **Task.blocker**, y cada objeto **Task2** se bloquea sobre **Task2.blocker**. En **main()**, se configura un objeto **java.util.Timer** para ejecutar su método **run()** cada 4/10 segundos y ese método **run()** llama alternativamente a los métodos **notify()** y **notifyAll()** sobre **Task.blocker**, a través de los métodos “prod”.

Analizando la salida, podemos ver que, aunque existe un objeto **Task2** que no está bloqueado sobre **Task2.blocker**, ninguna de las llamadas **notify()** o **notifyAll()** sobre **Task.blocker** hace que el objeto **Task2** se despierte. De forma similar, al final de **main()**, se invoca **cancel()** para **timer** y, aun cuando se cancela el temporizador, las primeras cinco tareas siguen ejecutándose y siguen bloqueadas en sus llamadas a **Task.blocker.waitingCall()**. La salida correspondiente a la llamada **Task2.blocker.prodAll()** no incluye ninguna de las tareas que está esperando por el bloqueo de **Task.blocker**.

Esto también tiene sentido si examinamos **prod()** y **prodAll()** en **Blocker**. Estos métodos son sincronizados, lo que quiere decir que tienen su propio bloqueo, de manera que cuando invocan **notify()** o **notifyAll()**, resulta lógico que sólo estén invocando dichos métodos para ese bloqueo, y que sólo despierten a las tareas que estén esperando por ese bloqueo concreto.

Blocker.waitingCall() es lo suficientemente simple como para que podamos escribir **for(;;)** en lugar de **while(!Thread.interrupted())**, y conseguir el mismo efecto en este caso, porque en este ejemplo no hay diferencia entre abandonar el bucle con una excepción y abandonarlo comprobando el indicador **interrupted()**; en ambos casos se ejecuta el mismo código. Sin embargo, por cuestión de estilo, este ejemplo comprueba **interrupted()**, porque existen dos formas

distintas de salir del bucle. Si decidimos posteriormente añadir más código al bucle, correríamos el riesgo de introducir un error si no se cubren ambas formas de salir del bucle.

Ejercicio 23: (7) Demuestre que **WaxOMatic.java** funciona adecuadamente cuando se emplea **notify()** en lugar de **notifyAll()**.

Productores y consumidores

Considere un restaurante que tiene un cocinero y un camarero. El camarero tiene que esperar a que el cocinero prepare un plato. Cuando el cocinero tiene un plato preparado, se lo notifica al camarero, que toma el plato y lo entrega al cliente y vuelve a quedar en espera. Éste es un ejemplo de cooperación entre tareas: el cocinero representa al *productor* y el camarero representa al *consumidor*. Ambas tareas deben negociar a medida que se producen y consumen los platos y el sistema tiene que ser capaz de terminar de una manera ordenada. He aquí este ejemplo modelado en código Java:

```
//: concurrency/Restaurant.java
// La técnica productor-consumidor para cooperación entre tareas.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
    private final int orderNum;
    public Meal(int orderNum) { this.orderNum = orderNum; }
    public String toString() { return "Meal " + orderNum; }
}

class WaitPerson implements Runnable {
    private Restaurant restaurant;
    public WaitPerson(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal == null)
                        wait(); // ... para que el cocinero准备 un plato.
                }
                print("Waitperson got " + restaurant.meal);
                synchronized(restaurant.chef) {
                    restaurant.meal = null;
                    restaurant.chef.notifyAll(); // Listo para otro
                }
            }
        } catch(InterruptedException e) {
            print("WaitPerson interrupted");
        }
    }
}

class Chef implements Runnable {
    private Restaurant restaurant;
    private int count = 0;
    public Chef(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal != null)
                        wait(); // ... para que se lleven el plato
                }
            }
        } catch(InterruptedException e) {
            print("Chef interrupted");
        }
    }
}
```

```

        if(++count == 10) {
            print("Out of food, closing");
            restaurant.exec.shutdownNow();
        }
        printnb("Order up! ");
        synchronized(restaurant.waitPerson) {
            restaurant.meal = new Meal(count);
            restaurant.waitPerson.notifyAll();
        }
        TimeUnit.MILLISECONDS.sleep(100);
    }
} catch(InterruptedException e) {
    print("Chef interrupted");
}
}

public class Restaurant {
    Meal meal;
    ExecutorService exec = Executors.newCachedThreadPool();
    WaitPerson waitPerson = new WaitPerson(this);
    Chef chef = new Chef(this);
    public Restaurant() {
        exec.execute(chef);
        exec.execute(waitPerson);
    }
    public static void main(String[] args) {
        new Restaurant();
    }
} /* Output:
Order up! Waitperson got Meal 1
Order up! Waitperson got Meal 2
Order up! Waitperson got Meal 3
Order up! Waitperson got Meal 4
Order up! Waitperson got Meal 5
Order up! Waitperson got Meal 6
Order up! Waitperson got Meal 7
Order up! Waitperson got Meal 8
Order up! Waitperson got Meal 9
Out of food, closing
WaitPerson interrupted
Order up! Chef interrupted
*///:~

```

El objeto **Restaurant** es el punto focal tanto para el camarero (**WaitPerson**) como para el cocinero (**Chef**). Ambos deben saber para qué objeto **Restaurant** están trabajando, porque ambos deben colocar o tomar los platos en el “mostrador” del mismo restaurante, **restaurant.meal**. En **run()**, el objeto **WaitPerson** entra en modo **wait()**, deteniéndose esta tarea hasta que ésta es despertada mediante un mensaje **notifyAll()** procedente del objeto **Chef**. Puesto que esto es un programa muy simple, sabemos que sólo habrá una tarea esperando por el bloqueo correspondiente a **WaitPerson**: la propia tarea **WaitPerson**. Por esta razón, sería teóricamente posible invocar **notify()** en lugar de **notifyAll()**. Sin embargo, en situaciones más complejas, puede que haya múltiples tareas esperando por el bloqueo concreto de un objeto, así que no sabremos qué tarea debe ser despertada. Por tanto, resulta más seguro invocar **notifyAll()**, que despierta a todas las tareas que estén esperando por ese bloqueo. Cada tarea deberá entonces decidir si la notificación es relevante.

Una vez que el objeto **Chef** entrega un plato (un objeto **Meal**) y notifica al objeto **WaitPerson**, el **Chef** espera hasta que **WaitPerson** toma el plato y lo notifica al **Chef**, que entonces puede producir el siguiente objeto **Meal**.

Observe que la llamada a **wait()** está encerrada dentro de una instrucción **while()** que comprueba esa misma condición por la que se está esperando. Esto parece un poco extraño a primera vista: si estamos esperando un pedido, una vez que despertamos, ese pedido tendrá que estar disponible, ¿verdad? Como hemos indicado anteriormente, el problema es que en una

aplicación concurrente, alguna otra tarea podría interferir y hacer el pedido mientras que el objeto **WaitPerson** se está despertando. El único enfoque seguro consiste en utilizar *siempre* la siguiente sintaxis para una llamada a **wait()** (empleando, por supuesto, la adecuada sincronización y preparando el programa para que no exista la posibilidad de que se pierdan señales):

```
while (noSeCumpleCondición)
    wait();
```

Esto garantiza que la condición se habrá cumplido antes de salir del bucle de espera y que si se nos ha notificado algo que no afecta a la condición (como puede ocurrir con **notifyAll()**), o la condición cambia antes de que salgamos del todo del bucle de espera, estará garantizado que volveremos a entrar en espera.

Observe que la llamada a **notifyAll()** tiene que capturar primero el bloqueo sobre **waitPerson**. La llamada **wait()** en **WaitPerson.run()** libera automáticamente el bloqueo, así que esto resulta posible. Dado que el bloqueo deberá haber sido adquirido para poder invocar **notifyAll()**, estará garantizado que no puedan interferir dos tareas que estén tratando de invocar **notifyAll()** sobre un mismo objeto.

Ambos métodos **run()** están diseñados para poder efectuar una terminación ordenada encerrando el método **run()** completo dentro del bloque **try**. La cláusula **catch** se cierra justo antes de la llave de cierre del método **run()**, por lo que si la tarea recibe una excepción **InterruptedException**, terminará inmediatamente después de capturar la excepción.

En **Chef**, observe que después de invocar **shutdownNow()**, *podríamos* simplemente volver (con **return**) de **run()**, y eso es lo que haremos normalmente. Sin embargo, resulta un poco más interesante hacerlo de la forma en que se lleva a cabo en el ejemplo. Recuerde que **shutdownNow()** envía una notificación **interrupt()** a todas las tareas que hayan iniciado el objeto **ExecutorService**. Pero en el caso de **Chef**, la tarea no se termina inmediatamente después de recibir la notificación **interrupt()**, porque la interrupción sólo genera **InterruptedException** cuando la tarea intenta iniciar una operación bloqueante (interrumpible). Por tanto, primero veremos que se muestra el mensaje “Order up!” y luego se genera **InterruptedException** cuando el objeto **Chef** trata de invocar **sleep()**. Si eliminamos la llamada a **sleep()**, la tarea alcanzará la parte superior del bucle **run()** y saldrá de la comprobación **Thread.interrupted()**, sin generar una excepción.

El ejemplo anterior sólo tiene un lugar cuando una tarea pueda almacenar un objeto de modo que otra tarea pueda utilizar ese objeto posteriormente. Sin embargo, en una implementación típica productor-consumidor, se utilizaría una cola de tipo FIFO para almacenar los objetos que estén siendo producidos y consumidos. Aprenderemos más acerca de dichas colas posteriormente en el capítulo.

Ejercicio 24: (1) Resuelva un problema de un único productor y un único consumidor utilizando **wait()** y **notifyAll()**. El productor no debe desbordar el *buffer* del receptor, lo que podría ocurrir si el productor fuera más rápido que el consumidor. Si el consumidor es más rápido que el productor, entonces no deberá leer los mismos datos más de una vez. No realice ninguna suposición acerca de las velocidades relativas del productor y el consumidor.

Ejercicio 25: (1) En la clase **Chef** de **Restaurant.java**, vuelva del método **run()** después de invocar **shutdownNow()** y observe la diferencia de comportamiento.

Ejercicio 26: (8) Añada una clase **BusBoy** (ayudante) a **Restaurant.java**. Después de entregar un plato, **WaitPerson** debe notificar a **BusBoy** que tiene que efectuar la limpieza.

Utilización de objetos Lock y Condition explícitos

Existen herramientas adicionales explícitas dentro de la biblioteca **java.util.concurrent** de Java SE5 que puede utilizarse para reescribir **WaxOMatic.java**. La clase básica que utiliza un mutex y permite la suspensión de tareas es **Condition**, y puede suspender una tarea llamando a **await()** sobre un objeto **Condition**. Cuando tiene lugar un cambio de estado externo que pudiera implicar que una tarea puede continuar con su procesamiento, enviamos una notificación a la tarea invocando el método **signal()**, para despertar a una sola tarea, o **signalAll()**, para despertar a todas las tareas que se hayan suspendido a sí mismas para esperar por ese objeto **Condition** (al igual que sucede con **notifyAll()**, **signalAll()** es la técnica más segura).

He aquí un programa **WaxOMatic.java** reescrito para incluir un objeto **Condition** que se utiliza para suspender una tarea dentro de **waitForWaxing()** o **waitForBuffing()**:

```

//: concurrency/waxomatic2/WaxOMatic2.java
// Utilización de objetos Lock y Condition.
package concurrency.waxomatic2;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean waxOn = false;
    public void waxed() {
        lock.lock();
        try {
            waxOn = true; // Listo para pulir
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void buffed() {
        lock.lock();
        try {
            waxOn = false; // Listo para otra capa de cera
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void waitForWaxing() throws InterruptedException {
        lock.lock();
        try {
            while(waxOn == false)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
    public void waitForBuffing() throws InterruptedException{
        lock.lock();
        try {
            while(waxOn == true)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
}
class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
            }
        }
    }
}

```

```

        car.waitForBuffing();
    }
} catch(InterruptedException e) {
    print("Exiting via interrupt");
}
print("Ending Wax On task");
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic2 {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* Output: (90% match)
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
Ending Wax Off task
Exiting via interrupt
Ending Wax On task
*///:-
```

En el constructor de **Car**, un único objeto **Lock** produce un objeto **Condition** que se utiliza para gestionar la comunicación inter-tareas. Sin embargo, el objeto **Condition** no contiene ninguna información acerca del estado del proceso, así que es necesario gestionar información adicional que indique este estado; esa información es el campo **waxOn** de tipo **boolean**.

Cada llamada a **lock()** debe ir seguida inmediatamente de una cláusula **try-finally** para garantizar que el desbloqueo se produzca en todos los casos. Al igual que sucede con las versiones integradas, una tarea debe poseer el bloqueo antes de poder invocar a **await()**, **signal()** o **signalAll()**.

Observe que esta solución es más compleja que la anterior y que esa complejidad no nos proporciona ninguna ventaja adicional en este caso. Los objetos **Lock** y **Condition** sólo son necesarios para otros problemas de gestión de hebras más complicados.

Ejercicio 27: (2) Modifique **Restaurant.java** para utilizar objetos **Lock** y **Condition** explícitos.

Productores-consumidores y colas

Los métodos `wait()` y `notifyAll()` resuelven el problema de la cooperación entre tareas a bastante bajo nivel, efectuando una negociación para cada iteración. En muchos casos, podemos movernos un nivel de abstracción hacia arriba y resolver los problemas de la cooperación entre tareas utilizando una *cola sincronizada*, que sólo permite que una tarea inserte o elimine un elemento cada vez. Este tipo de funcionalidad la proporciona la interfaz `java.util.concurrent.BlockingQueue`, que tiene varias implementaciones estándar. Normalmente utilizaremos `LinkedBlockingQueue`, que es una cola no limitada; la cola `ArrayBlockingQueue` tiene un tamaño fijo, por lo que sólo se puede introducir en ella un cierto número de elementos antes de que se bloquee.

Estas colas también suspenden una tarea consumidora si dicha tarea trata de extraer un objeto de la cola estando ésta vacía; la tarea se reanudará cuando haya más elementos disponibles. Las colas bloqueantes como éstas pueden resolver un gran número de problemas de una forma mucho más simple y más fiable que `wait()` y `notifyAll()`.

He aquí una prueba simple que serializa la ejecución de objetos `LiftOff`. El consumidor es `LiftOffRunner`, que extrae cada objeto `LiftOff` de la cola bloqueante `BlockingQueue` y lo ejecuta directamente (es decir, utiliza su propia hebra invocando explícitamente `run()` en lugar de iniciar una nueva hebra para cada tarea).

```
//: concurrency/TestBlockingQueues.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class LiftOffRunner implements Runnable {
    private BlockingQueue<LiftOff> rockets;
    public LiftOffRunner(BlockingQueue<LiftOff> queue) {
        rockets = queue;
    }
    public void add(LiftOff lo) {
        try {
            rockets.put(lo);
        } catch(InterruptedException e) {
            print("Interrupted during put()");
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                LiftOff rocket = rockets.take();
                rocket.run(); // Utilizar esta hebra
            }
        } catch(InterruptedException e) {
            print("Waking from take()");
        }
        print("Exiting LiftOffRunner");
    }
}

public class TestBlockingQueues {
    static void getKey() {
        try {
            // Compensar las diferencias entre Windows/Linux en cuanto
            // a la longitud del resultado producido por la tecla Intro:
            new BufferedReader(
                new InputStreamReader(System.in)).readLine();
        } catch(java.io.IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

}

static void getkey(String message) {
    print(message);
    getKey();
}

static void
test(String msg, BlockingQueue<LiftOff> queue) {
    print(msg);
    LiftOffRunner runner = new LiftOffRunner(queue);
    Thread t = new Thread(runner);
    t.start();
    for(int i = 0; i < 5; i++)
        runner.add(new LiftOff(5));
    getKey("Press 'Enter' (" + msg + ")");
    t.interrupt();
    print("Finished " + msg + " test");
}

public static void main(String[] args) {
    test("LinkedBlockingQueue", // Tamaño ilimitado
         new LinkedBlockingQueue<LiftOff>());
    test("ArrayBlockingQueue", // Tamaño fijo
         new ArrayBlockingQueue<LiftOff>(3));
    test("SynchronousQueue", // Tamaño igual a 1
         new SynchronousQueue<LiftOff>());
}
} //:-

```

Las tareas son introducidas en la cola **BlockingQueue** por **main()** y son extraídas de **BlockingQueue** por el objeto **LiftOffRunner**. Observe que **LiftOffRunner** puede ignorar los problemas de sincronización porque éstos son resueltos por la cola **BlockingQueue**.

Ejercicio 28: (3) Modifique **TestBlockingQueues.java** añadiendo una nueva tarea que introduzca objetos **LiftOff** en la cola **BlockingQueue**, en lugar de hacerlo en **main()**.

Ejemplo de uso de **BlockingQueue**

Como ejemplo de utilización de las colas de tipo **BlockingQueue**, considere una máquina que tiene tres tareas: una para hacer una tostada, otra para untarla de mantequilla y otra para poner mermelada sobre la tostada ya untada con mantequilla. Podemos ir haciendo pasar la tostada por las distintas colas **BlockingQueue** que sirven de comunicación entre los procesos:

```

//: concurrency/ToastOMatic.java
// Una tostadora basada en colas.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Toast {
    public enum Status { DRY, BUTTERED, JAMMED }
    private Status status = Status.DRY;
    private final int id;
    public Toast(int idn) { id = idn; }
    public void butter() { status = Status.BUTTERED; }
    public void jam() { status = Status.JAMMED; }
    public Status getStatus() { return status; }
    public int getId() { return id; }
    public String toString() {
        return "Toast " + id + ": " + status;
    }
}

```

```

class ToastQueue extends LinkedBlockingQueue<Toast> {}

class Toaster implements Runnable {
    private ToastQueue toastQueue;
    private int count = 0;
    private Random rand = new Random(47);
    public Toaster(ToastQueue tq) { toastQueue = tq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(
                    100 + rand.nextInt(500));
                // Hacer tostada
                Toast t = new Toast(count++);
                print(t);
                // Insertar en la cola
                toastQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Toaster interrupted");
        }
        print("Toaster off");
    }
}

// Untar de mantequilla la tostada:
class Butterer implements Runnable {
    private ToastQueue dryQueue, butteredQueue;
    public Butterer(ToastQueue dry, ToastQueue buttered) {
        dryQueue = dry;
        butteredQueue = buttered;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Se bloquea hasta que haya otra tostada disponible:
                Toast t = dryQueue.take();
                t.butter();
                print(t);
                butteredQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Butterer interrupted");
        }
        print("Butterer off");
    }
}

// Poner mermelada sobre la tostada untada de mantequilla:
class Jammer implements Runnable {
    private ToastQueue butteredQueue, finishedQueue;
    public Jammer(ToastQueue buttered, ToastQueue finished) {
        butteredQueue = buttered;
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Se bloquea hasta que haya otra tostada disponible:

```

```

        Toast t = butteredQueue.take();
        t.jam();
        print(t);
        finishedQueue.put(t);
    }
} catch(InterruptedException e) {
    print("Jammer interrupted");
}
print("Jammer off");
}

// Consumir la tostada:
class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finished) {
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Se bloquea hasta que haya otra tostada disponible:
                Toast t = finishedQueue.take();
                // Verificar que la tostada se ha preparado correctamente
                // y que todas las tostadas llevan mermelada:
                if(t.getId() != counter++) ||
                    t.getStatus() != Toast.Status.JAMMED) {
                    print(">>> Error: " + t);
                    System.exit(1);
                } else
                    print("Chomp! " + t);
            }
        } catch(InterruptedException e) {
            print("Eater interrupted");
        }
        print("Eater off");
    }
}

public class ToastOMATIC {
    public static void main(String[] args) throws Exception {
        ToastQueue dryQueue = new ToastQueue(),
                    butteredQueue = new ToastQueue(),
                    finishedQueue = new ToastQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Toaster(dryQueue));
        exec.execute(new Butterer(dryQueue, butteredQueue));
        exec.execute(new Jammer(butteredQueue, finishedQueue));
        exec.execute(new Eater(finishedQueue));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* (Ejecutar para ver la salida) */:-

```

Toast es un excelente ejemplo de la ventaja de emplear valores **enum**. Observe que no hay ninguna sincronización explícita (utilizando objetos **Lock** o la palabra clave **synchronized**), porque la sincronización es gestionada de manera implícita por las colas (que se sincronizan internamente) y por el diseño del sistema: cada objeto **Toast** sólo es manipulado por una única tarea cada vez. Puesto que las colas son bloqueantes, los procesos se suspenden y se reanudan automáticamente.

Podemos ver que **BlockingQueue** puede simplificar el problema enormemente. El acoplamiento entre las clases que existiría con instrucciones `wait()` y `notifyAll()` explícitas se elimina, porque cada clase se comunica sólo con sus colas **BlockingQueue**.

Ejercicio 29: (8) Modifique **ToastOMatic.java** para fabricar bocadillos de mantequilla de cacahuete y mermelada, utilizando dos líneas de fabricación separadas (una para el pan con mantequilla, otra para el pan con mermelada y luego mezclando las dos líneas).

Utilización de canalizaciones para la E/S entre tareas

A menudo, resulta útil que las tareas se comuniquen entre sí utilizando mecanismos de E/S. Las bibliotecas de gestión de hebras pueden proporcionar soporte para la E/S inter-tareas en la forma de canalizaciones (*pipes*). Estas canalizaciones existen en la biblioteca E/S de Java en forma de las clases **PipedWriter** (que permite a una tarea escribir en una canalización) y **PipedReader** (que permite a otra tarea distinta leer de la misma canalización). Podríamos considerar esto como una variante del problema productor-consumidor, siendo la canalización una solución prediseñada. La canalización es básicamente una cola bloqueante, y existía ya como solución en las versiones de Java anteriores a la introducción de **BlockingQueue**.

He aquí un ejemplo simple en el que dos tareas utilizan una canalización para comunicarse:

```
//: concurrency/PipedIO.java
// Utilización de canalizaciones para la E/S inter-tareas
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
    private Random rand = new Random(47);
    private PipedWriter out = new PipedWriter();
    public PipedWriter getPipedWriter() { return out; }
    public void run() {
        try {
            while(true)
                for(char c = 'A'; c <= 'z'; c++)
                    out.write(c);
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
        } catch(IOException e) {
            print(e + " Sender write exception");
        } catch(InterruptedException e) {
            print(e + " Sender sleep interrupted");
        }
    }
}

class Receiver implements Runnable {
    private PipedReader in;
    public Receiver(Sender sender) throws IOException {
        in = new PipedReader(sender.getPipedWriter());
    }
    public void run() {
        try {
            while(true) {
                // Se bloquea hasta que haya caracteres:
                printnb("Read: " + (char)in.read() + ", ");
            }
        } catch(IOException e) {
            print(e + " Receiver read exception");
        }
    }
}
```

```

    }
}

public class PipedIO {
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(sender);
        exec.execute(receiver);
        TimeUnit.SECONDS.sleep(4);
        exec.shutdownNow();
    }
} /* Output: (65% match)
Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read:
G, Read: H, Read: I, Read: J, Read: K, Read: L, Read: M,
java.lang.InterruptedException: sleep interrupted Sender
sleep interrupted
java.io.InterruptedIOException Receiver read exception
*///:-

```

Sender y **Receiver** representan tareas que necesitan comunicarse entre sí. **Sender** crea un objeto escritor **PipedWriter**, que es un objeto autónomo, pero dentro de **Receiver** la creación del objeto lector **PipedReader** debe asociarse con un objeto **PipedWriter** dentro del constructor. **Sender** pone datos sobre el objeto **Writer** y duerme una cantidad aleatoria de tiempo. Sin embargo, **Receiver** no tiene ningún método **sleep()** o **wait()**. Sin embargo, cuando invoca el método de lectura **read()**, la canalización se bloquea automáticamente si no existen más datos.

Observe que los objetos **sender** y **receiver** se inician en **main()**, después de que los objetos hayan sido completamente construidos. Si no comenzamos con objetos completamente construidos, la canalización puede presentar un comportamiento incoherente en las distintas plataformas (observe que las colas de tipo **BlockingQueue** son más robustas y fáciles de usar).

Una diferencia importante entre un objeto **PipedReader** y la E/S normal es la que podemos ver cuando se invoca **shutdownNow()**: el lector **PipedReader** es interrumpible, mientras que si cambiáramos, por ejemplo, la llamada **in.read()** por **System.in.read()**, la interrupción **interrupt()** no conseguiría salir de la llamada a **read()**.

Ejercicio 30: (1) Modifique **PipedIO.java** para utilizar una cola **BlockingQueue** en lugar de una canalización.

Interbloqueo

A estas alturas ya sabemos que un objeto puede tener métodos sincronizados u otras formas de bloqueo que impidan a las tareas acceder a dicho objeto hasta que se libere el mutex. También hemos visto que las tareas pueden bloquearse. Por tanto, es posible que una tarea se quede esperando por otra tarea, que a su vez espere por otra tarea, etc., hasta que la cadena se cierre de nuevo con una tarea que esté esperando por la primera. En esta situación, tendríamos un ciclo continuo de tareas esperando unas por otras y ninguna de ellas podría continuar con su procesamiento. Esta situación se denomina *interbloqueo (deadlock)*.²¹

Si tratamos de ejecutar un programa y se produce directamente un interbloqueo, podemos intentar localizar inmediatamente el error. El problema real se produce cuando nuestro programa parece estar funcionamiento correctamente pero tiene la posibilidad oculta de interbloquearse. En este caso, puede que no tengamos ninguna indicación de que ese interbloqueo es posible, así que el error estará latente en el programa hasta que se presente de manera inesperada cuando un cliente lo ejecute (en una forma que casi siempre será muy difícil de reproducir). Por tanto, la prevención del interbloqueo mediante un diseño cuidadoso del programa es una parte crítica del desarrollo de sistemas concurrentes.

El problema de *la cena de los filósofos*, inventado por Edsger Dijkstra, es una ilustración clásica del interbloqueo. La descripción básica especifica cinco filósofos (aunque el ejemplo mostrado aquí permitiría cualquier número de ellos). Estos

²¹ También podemos tener lo que se denomina *bloqueo activo (livelock)* cuando hay dos tareas que son capaces de cambiar su estado (no están bloqueadas), pero nunca consiguen realizar ningún progreso útil.

filósofos pasan parte de su tiempo pensando y parte de su tiempo comiendo. Mientras están pensando, no necesitan ningún recurso compartido, pero todos ellos comen usando un número limitado de utensilios. En la descripción original del problema, los utensilios eran tenedores, y se requieren dos tenedores para tomar espagueti de una fuente situada en el centro de la mesa, aunque parece que tiene más sentido decir que esos utensilios sean palillos. Claramente, cada filósofo necesitará dos palillos para poder comer.

Introducimos una dificultad en el problema: como filósofos, tienen muy poco dinero, así que sólo pueden permitirse comprar cinco palillos (o más generalmente, el mismo número de palillos que de filósofos). Esos palillos están espaciados alrededor de la mesa entre los filósofos. Cuando un filósofo quiere comer, debe tomar el palillo situado a su izquierda y el situado a su derecha. Si alguno de los filósofos sentado a su lado está usando uno de los palillos que necesita, nuestro filósofo deberá esperar hasta que los palillos necesarios estén disponibles.

```
//: concurrency/Chopstick.java
// Palillos para la cena de los filósofos.

public class Chopstick {
    private boolean taken = false;
    public synchronized
    void take() throws InterruptedException {
        while(taken)
            wait();
        taken = true;
    }
    public synchronized void drop() {
        taken = false;
        notifyAll();
    }
} //:-
```

No puede haber dos filósofos (objeto **Philosopher**) que tomen (con el método **take()**) el mismo palillo (objeto **Chopstick**) al mismo tiempo. Además, si el objeto **Chopstick** ya ha sido tomado por un objeto **Philosopher**, otro filósofo podrá esperar (**wait()**) hasta que el objeto **Chopstick** quede disponible cuando su propietario actual invoque **drop()** (soltar palillo).

Cuando una tarea **Philosopher** invoca **take()**, esa tarea espera que el indicador **taken** (ocupado) valga **false** (es decir, hasta que el objeto **Philosopher** que actualmente posee el objeto **Chopstick** lo libere). Entonces, la tarea asigna al indicador **taken** el valor **true** para indicar que el nuevo objeto **Philosopher** posee ahora el objeto **Chopstick**. Cuando este objeto **Philosopher** haya terminado de usar el objeto **Chopstick**, invocará **drop()** para cambiar el indicador y llamará a **notifyAll()** para notificar a todos los demás objetos **Philosopher** que puedan estar esperando a utilizar el objeto **Chopstick**.

```
//: concurrency/Philosopher.java
// Un filósofo comensal
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
    private Chopstick left;
    private Chopstick right;
    private final int id;
    private final int ponderFactor;
    private Random rand = new Random(47);
    private void pause() throws InterruptedException {
        if(ponderFactor == 0) return;
        TimeUnit.MILLISECONDS.sleep(
            rand.nextInt(ponderFactor * 250));
    }
    public Philosopher(Chopstick left, Chopstick right,
        int ident, int ponder) {
        this.left = left;
        this.right = right;
```

```

        id = ident;
        ponderFactor = ponder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(this + " " + "thinking");
                pause();
                // El filósofo tiene hambre
                print(this + " " + "grabbing right");
                right.take();
                print(this + " " + "grabbing left");
                left.take();
                print(this + " " + "eating");
                pause();
                right.drop();
                left.drop();
            }
        } catch(InterruptedException e) {
            print(this + " " + "exiting via interrupt");
        }
    }
    public String toString() { return "Philosopher " + id; }
} /**:~
```

En **Philosopher.run()**, cada objeto **Philosopher** simplemente piensa y come de manera continua. El método **pause()** efectúa una llamada a **sleeps()** durante un período aleatorio si el factor **ponderFactor** es distinto de cero. Utilizando esto, vemos que el filósofo está pensando durante un intervalo de tiempo aleatorio y que luego intenta tomar los palillos izquierdo y derecho, comiendo durante un intervalo de tiempo aleatorio, y luego volviendo a repetir el ciclo.

Ahora podemos escribir una versión del programa, versión que estará sometida al problema del interbloqueo:

```

//: concurrency/DeadlockingDiningPhilosophers.java
// Ilustra cómo puede haber interbloqueos ocultos en un programa.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Chopstick[] sticks = new Chopstick[size];
        for(int i = 0; i < size; i++)
            sticks[i] = new Chopstick();
        for(int i = 0; i < size; i++)
            exec.execute(new Philosopher(
                sticks[i], sticks[(i+1) % size], i, ponder));
        if(args.length == 3 && args[2].equals("timeout"))
            TimeUnit.SECONDS.sleep(5);
        else {
            System.out.println("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* (Ejecutar para ver la salida) */://:~
```

Podrá observar que si los objetos **Philosopher** pasan demasiado poco tiempo pensando, todos ellos competirán por los objetos **Chopstick** cuando traten de comer, de modo que el interbloqueo se producirá mucho más rápidamente.

El primer argumento de la línea de comandos ajusta el factor **ponder**, que afecta al intervalo de tiempo que cada objeto **Philosopher** dedica a pensar. Si tenemos muchos objetos **Philosopher** o éstos pasan una gran cantidad de tiempo pensando, puede que nunca lleguemos a ver un problema de interbloqueo, aunque éste seguirá siendo posible. Un argumento de la línea de comandos igual a cero tiende a hacer que el programa se interbloquee muy rápidamente.

Observe que los objetos **Chopstick** no necesitan identificadores internos; se los identifica por su posición dentro de la matriz **sticks**. A cada constructor **Philosopher** se le proporciona una referencia a sendos objetos **Chopstick** izquierdo y derecho. Cada objeto **Philosopher** excepto el último se inicializa situando dicho objeto **Philosopher** entre el siguiente par de objetos **Chopstick**. Al último objeto **Philosopher** se le asigna el objeto **Chopstick** situado en la posición cero como palillo derecho, con lo que se completa la mesa redonda. Esto se debe a que el último filósofo está situado justo al lado del primero y ambos comparten ese palillo número cero. Ahora, será posible que todos los objetos **Philosopher** traten de comer, quedando todos ellos a la espera de que el objeto **Philosopher** situado a continuación de ellos libere su objeto **Chopstick**. Esto hará que el programa se interbloquee.

Si nuestros filósofos invierten más tiempo pensando que comiendo, tendrán una posibilidad mucho menor de requerir los recursos compartidos (los palillos), con lo que podríamos quedarnos convencidos de que el programa no sufre interbloqueos (utilizando un valor de **ponder** distinto de cero o un gran número de objetos **Philosopher**), aunque en realidad no es así. Este ejemplo es interesante precisamente porque ilustra que un programa puede parecer estar ejecutándose correctamente y sin embargo ser capaz de interbloquearse.

Para corregir el problema, es necesario entender que el interbloqueo puede producirse si se cumplen simultáneamente cuatro condiciones:

1. Exclusión mutua. Al menos uno de los recursos utilizados por las tareas no debe ser compatible. En este caso, un objeto **Chopstick** sólo puede ser utilizado por un objeto **Philosopher** cada vez.
2. Al menos una tarea debe poseer un recurso y estar esperando a adquirir otro recurso que actualmente es propiedad de otra tarea. Es decir, para que el interbloqueo se produzca, un objeto **Philosopher** deberá poseer un objeto **Chopstick** y estar esperando a adquirir otro.
3. Los recursos no pueden ser desalojados de una tarea. La liberación de recursos por parte de tareas sólo puede producirse como un suceso normal. En otras palabras, nuestros filósofos son muy educados y no arrebatan los palillos a los otros filósofos.
4. Puede producirse una espera circular, según la cual una tarea estará esperando por un recurso que posee otra tarea, que a su vez estará esperando por un recurso que posee otra tarea, y así sucesivamente, hasta que una de las tareas esté esperando por un recurso poseído por la primera tarea, lo que hace que se produzca una cadena circular de bloqueo. En **DeadlockingDiningPhilosophers.java**, esta espera circular se produce porque cada filósofo trata de obtener primero el palillo derecho y luego el izquierdo.

Como *todas* estas condiciones deben producirse para provocar un interbloqueo, basta con que impidamos que una de ellas se produzca para conseguir que no haya interbloqueos. En este programa, la forma más fácil de impedir el interbloqueo consiste en impedir que se dé la cuarta condición. Esta condición sucede porque cada filósofo trata de tomar los correspondientes palillos en un orden concreto, primero el derecho y luego el izquierdo. Debido a ello, resulta posible encontrarse en una situación en la que cada uno de los filósofos haya tomado su palillo derecho y esté esperando a poder conseguir el izquierdo, provocando la aparición de la condición de espera circular. Sin embargo, si inicializamos el último filósofo para que trate de obtener primero el palillo izquierdo y luego el derecho, ese filósofo nunca impedirá que el filósofo situado inmediatamente a su derecha tome los dos palillos que necesita. En este caso, se impide la espera circular. Ésta sólo es una de las posibles soluciones del problema; también podríamos evitar el problema impidiendo que se cumpla alguna de las otras condiciones (consulte algún otro libro para conocer más detalles sobre la gestión avanzada de hebras):

```
//: concurrency/FixedDiningPhilosophers.java
// La cena de los filósofos sin interbloqueo.
// {Args: 5 5 timeout}
import java.util.concurrent.*;

public class FixedDiningPhilosophers {
```

```

public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
        ponder = Integer.parseInt(args[0]);
    int size = 5;
    if(args.length > 1)
        size = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for(int i = 0; i < size; i++)
        sticks[i] = new Chopstick();
    for(int i = 0; i < size; i++)
        if(i < (size-1))
            exec.execute(new Philosopher(
                sticks[i], sticks[i+1], i, ponder));
        else
            exec.execute(new Philosopher(
                sticks[0], sticks[i], i, ponder));
    if(args.length == 3 && args[2].equals("timeout"))
        TimeUnit.SECONDS.sleep(5);
    else {
        System.out.println("Press 'Enter' to quit");
        System.in.read();
    }
    exec.shutdownNow();
}
} /* (Ejecutar para ver la salida) *///:-
```

Garantizando que el último filósofo tome y deje el palillo izquierdo antes que el derecho, eliminamos el interbloqueo y el programa funcionará sin problemas.

No hay ningún soporte del lenguaje para ayudarnos a prevenir el interbloqueo; es un problema que deberemos resolver nosotros mismos realizando un diseño cuidadoso. Ya sé que estas palabras no sirven de mucho consuelo a aquellas personas que estén tratando de depurar un programa sometido al interbloqueo.

Ejercicio 31: (8) Cambie **DeadlockingDiningPhilosophers.java** de modo que, cuando un filósofo haya terminado de emplear sus palillos, los deje en una bandeja. Cuando un filósofo quiera comer, tomará los dos palillos siguientes que estén disponibles en la bandeja. ¿Elimina esto la posibilidad del interbloqueo? ¿Podemos reintroducir el interbloqueo simplemente reduciendo el número de palillos disponibles?

Nuevos componentes de biblioteca

La biblioteca **java.util.concurrent** de Java SE5 introduce un número significativo de nuevas clases diseñadas para resolver los problemas de concurrencia. Aprender a emplear estas clases puede ayudarnos a escribir programas concurrentes más simples y robustos.

Esta sección incluye un conjunto representativo de ejemplos de diversos componentes, aunque algunos de los componentes no los analizaremos aquí (aquellos que es menos probable que el lector se encuentre a la hora de analizar programas o utilice a la hora de escribirlos).

Puesto que estos componentes permiten resolver varios problemas, no existe una forma clara de organizarlos, así que trataré de comenzar con ejemplos sencillos y continuar con una serie de ejemplos de creciente complejidad.

CountDownLatch

Esta clase se usa para sincronizar una o más tareas forzándolas a esperar a que se complete un conjunto de operaciones que estén siendo realizadas por otras tareas.

Al objeto **CountDownLatch** le proporcionamos un valor de recuento inicial y cualquier tarea que invoque **await()** sobre dicho objeto se bloqueará hasta que el recuento alcance el valor cero. Otras tareas pueden invocar **countDown()** sobre el

objeto para reducir el valor de recuento, presumiblemente cuando esas tareas finalicen su trabajo. **CountDownLatch** está diseñado para utilizarlo una sola vez, el valor de recuento no puede reinicializarse. Si necesitamos una versión donde pueda reinicializar el valor de recuento, podemos emplear en su lugar **CyclicBarrier**.

Las tareas que invocan **countDown()** no se bloquean cuando hacen esa llamada. Sólo la llamada a **await()** se bloquea hasta que el recuento alcanza el valor cero.

Un uso normal consiste en dividir un problema en n tareas independientemente resolubles y crear un objeto **CountDownLatch** con un valor n . Cuando cada una de las tareas finaliza invoca **countDown()** para el objeto encargado del recuento. Las tareas que están esperando a que el problema se resuelva pueden invocar a **await()** sobre el objeto encargado del recuento para esperar a que el problema esté completamente resuelto. He aquí un esqueleto de ejemplo que ilustra esta técnica:

```
//: concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Realiza cierta parte de una tarea:
class TaskPortion implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private static Random rand = new Random(47);
    private final CountDownLatch latch;
    TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch(InterruptedException ex) {
            // Forma aceptable de salir
        }
    }
    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
        print(this + " completed");
    }
    public String toString() {
        return String.format("%1$-3d", id);
    }
}

// Espera sobre CountDownLatch:
class WaitingTask implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final CountDownLatch latch;
    WaitingTask(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            print("Latch barrier passed for " + this);
        } catch(InterruptedException ex) {
            print(this + " interrupted");
        }
    }
    public String toString() {
```

```

        return String.format("WaitingTask %1$-3d ", id);
    }

public class CountDownLatchDemo {
    static final int SIZE = 100;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // Todos deben compartir un único objeto CountDownLatch:
        CountDownLatch latch = new CountDownLatch(SIZE);
        for(int i = 0; i < 10; i++)
            exec.execute(new WaitingTask(latch));
        for(int i = 0; i < SIZE; i++)
            exec.execute(new TaskPortion(latch));
        print("Launched all tasks");
        exec.shutdown(); // Salir cuando todas las tareas se hayan completado
    }
} /* (Ejecutar para ver la salida) *///:~

```

TaskPortion duerme durante un periodo aleatorio para simular la terminación de parte del proceso, y **WaitingTask** indica que una parte del sistema tiene que esperar hasta que el problema inicial se haya completado. Todas las tareas funcionan con el mismo contador **CountDownLatch**, que se define en **main()**.

Ejercicio 32: (7) Utilice un objeto **CountDownLatch** para resolver el problema de correlación de los resultados de las distintas entradas de **OrnamentalGarden.java**. Elimine el código innecesario en la nueva versión del ejemplo.

Seguridad de las hebras en la biblioteca

Observe que **TaskPortion** contiene un objeto estático **Random**, lo que significa que puede haber múltiples tareas invocando **Random.nextInt()** al mismo tiempo. ¿Es esto seguro?

Si existe un problema, puede resolverse en este caso asignando dos **TaskPortion** en su propio objeto **Random**, es decir, eliminando el especificado **static**. Pero esa misma cuestión seguirá siendo pertinente, con carácter general, para todos los métodos de la biblioteca estándar de Java: ¿cuáles de esos métodos son seguros de cara a las hebras y cuáles no lo son?

Lamentablemente, la documentación del JDK no es muy explicativa en este aspecto. Resulta que **Random.nextInt()** sí que es seguro respecto a las hebras, pero es necesario descubrir en cada caso si un cierto método lo es, efectuando una búsqueda en la Web o inspeccionando el código de la biblioteca Java. Evidentemente, esta situación no resulta particularmente atractiva para un lenguaje de programación que está diseñado, al menos en teoría, para soportar la concurrencia.

CyclicBarrier

La clase **CyclicBarrier** (barrera circular) se utiliza en aquellas situaciones en las que queremos crear un grupo de tareas para realizar un cierto trabajo en paralelo, y luego esperar a que todas hayan finalizado, antes de continuar con el siguiente paso (algo parecido a **join()**, podríamos decir). Esta clase alinea todas las tareas paralelas en la barrera circular, de modo que podamos continuar hacia adelante al unísono. Se trata de una solución muy similar a **CountDownLatch**, excepto porque **CountDownLatch** representa un suceso que sólo se produce una vez, mientras que **CyclicBarrier** puede reutilizarse muchas veces.

Las simulaciones me han fascinado desde que empecé a utilizar computadoras y la concurrencia es un factor clave a la hora de realizar simulaciones. El primer programa que recuerdo haber escrito²² era una simulación: un juego de carreras de caballos escrito en BASIC y denominado (debido a las limitaciones de los nombres de archivos) HOSRAC.BAS. He aquí la versión orientada a objetos y con hebras de dicho programa, utilizando una clase **CyclicBarrier**:

```

//: concurrency/HorseRace.java
// Utilización de CyclicBarrier.

```

²² Cuando estaba en la universidad; en la laboratorio disponíamos de un teletipo SR-33 con un nodo de acoplamiento de acústico de 110 baudios mediante el que se accedía a una computadora HP-1000.

```

import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random rand = new Random(47);
    private static CyclicBarrier barrier;
    public Horse(CyclicBarrier b) { barrier = b; }
    public synchronized int getStrides() { return strides; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    strides += rand.nextInt(3); // Produce 0, 1 o 2
                }
                barrier.await();
            }
        } catch(InterruptedException e) {
            // Una forma legítima de salir
        } catch(BrokenBarrierException e) {
            // Ésta queremos probarla
            throw new RuntimeException(e);
        }
    }
    public String toString() { return "Horse " + id + " "; }
    public String tracks() {
        StringBuilder s = new StringBuilder();
        for(int i = 0; i < getStrides(); i++)
            s.append("*");
        s.append(id);
        return s.toString();
    }
}
public class HorseRace {
    static final int FINISH_LINE = 75;
    private List<Horse> horses = new ArrayList<Horse>();
    private ExecutorService exec =
        Executors.newCachedThreadPool();
    private CyclicBarrier barrier;
    public HorseRace(int nHorses, final int pause) {
        barrier = new CyclicBarrier(nHorses, new Runnable() {
            public void run() {
                StringBuilder s = new StringBuilder();
                for(int i = 0; i < FINISH_LINE; i++)
                    s.append("="); // La valla en el hipódromo
                print(s);
                for(Horse horse : horses)
                    print(horse.tracks());
                for(Horse horse : horses)
                    if(horse.getStrides() >= FINISH_LINE) {
                        print(horse + "won!");
                        exec.shutdownNow();
                        return;
                    }
            }
        });
        try {
            TimeUnit.MILLISECONDS.sleep(pause);
        }
    }
}

```

```

        } catch(InterruptedException e) {
            print("barrier-action sleep interrupted");
        }
    });
for(int i = 0; i < nHorses; i++) {
    Horse horse = new Horse(barrier);
    horses.add(horse);
    exec.execute(horse);
}
}

public static void main(String[] args) {
    int nHorses = 7;
    int pause = 200;
    if(args.length > 0) { // Argumento opcional
        int n = new Integer(args[0]);
        nHorses = n > 0 ? n : nHorses;
    }
    if(args.length > 1) { // Argumento opcional
        int p = new Integer(args[1]);
        pause = p > -1 ? p : pause;
    }
    new HorseRace(nHorses, pause);
}
} /* (Ejecutar para ver la salida) */:-
```

Un objeto **CyclicBarrier** puede proporcionar una “acción de barrera”, que es un objeto **Runnable** que se ejecuta automáticamente cuando el contador alcanza el valor cero; ésta es otra distinción entre **CyclicBarrier** y **CountDownLatch**. Aquí, la acción de barrera se define mediante una clase anónima que se entrega al constructor de **CyclicBarrier**.

Intenté que cada caballo pudiera imprimirse a sí mismo, pero el orden de visualización dependía del gestor de tareas. **CyclicBarrier** permite que cada caballo haga lo que necesita hacer para poder continuar adelante, y luego tiene que esperar en la barrera hasta que todos los demás caballos hayan avanzado. Cuando todos los caballos se han desplazado, **CyclicBarrier** llama automáticamente a su objeto **Runnable** que define la acción de barrera para mostrar los caballos por orden junto con la valla.

Una vez que todas las tareas han pasado la barrera, ésta estará automáticamente lista para la siguiente ronda.

Para obtener el efecto de una animación muy simple, reduzca el tamaño de la consola para que sólo se muestren los caballos.

DelayQueue

Se trata de una cola **BlockingQueue** de objetos sin limitación de tamaño que implementa la interfaz **Delayed**. Un objeto sólo puede ser extraído de la cola una vez que su retardo haya transcurrido. La cola está ordenada, de modo que el objeto situado al principio es el que tiene el retardo que ha transcurrido hace más tiempo. Si no ha transcurrido ninguno de los retardos, no habrá ningún elemento de cabecera y el método **poll()** devolverá **null** (debido a esto, no pueden insertarse elementos **null** en la cola).

He aquí un ejemplo donde los objetos **Delayed** son tareas y el consumidor **DelayedTaskConsumer** extrae la tarea más “urgente” (aquella cuyo retardo haya caducado hace más tiempo) de la cola y la ejecuta. Observe que **DelayQueue** es, por tanto, una variante de una cola con prioridad.

```

//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
    private static int counter = 0;
    private final int id = counter++;
    ...
}
```

```

private final int delta;
private final long trigger;
protected static List<DelayedTask> sequence =
    new ArrayList<DelayedTask>();
public DelayedTask(int delayInMilliseconds) {
    delta = delayInMilliseconds;
    trigger = System.nanoTime() +
        NANOSECONDS.convert(delta, MILLISECONDS);
    sequence.add(this);
}
public long getDelay(TimeUnit unit) {
    return unit.convert(
        trigger - System.nanoTime(), NANOSECONDS);
}
public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
}
public void run() { println(this + " "); }
public String toString() {
    return String.format("[%1$-4d]", delta) +
        " Task " + id;
}
public String summary() {
    return "(" + id + ":" + delta + ")";
}
public static class EndSentinel extends DelayedTask {
    private ExecutorService exec;
    public EndSentinel(int delay, ExecutorService e) {
        super(delay);
        exec = e;
    }
    public void run() {
        for(DelayedTask pt : sequence) {
            println(pt.summary() + " ");
        }
        print();
        print(this + " Calling shutdownNow()");
        exec.shutdownNow();
    }
}
}
class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> q;
    public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
        this.q = q;
    }
    public void run() {
        try {
            while(!Thread.interrupted())
                q.take().run(); // Ejecutar tarea con la hebra actual
        } catch(InterruptedException e) {
            // Forma aceptable de salir
        }
        print("Finished DelayedTaskConsumer");
    }
}

```

```

public class DelayQueueDemo {
    public static void main(String[] args) {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        DelayQueue<DelayedTask> queue =
            new DelayQueue<DelayedTask>();
        // Rellenar con tareas que tengan retardos aleatorios:
        for(int i = 0; i < 20; i++)
            queue.put(new DelayedTask(rand.nextInt(5000)));
        // Establecer el punto de detención
        queue.add(new DelayedTask.EndSentinel(5000, exec));
        exec.execute(new DelayedTaskConsumer(queue));
    }
} /* Output:
[128] Task 11 [200] Task 7 [429] Task 5 [520] Task 18
[555] Task 1 [961] Task 4 [998] Task 16 [1207] Task 9
[1693] Task 2 [1809] Task 14 [1861] Task 3 [2278] Task 15
[3288] Task 10 [3551] Task 12 [4258] Task 0 [4258] Task 19
[4522] Task 8 [4589] Task 13 [4861] Task 17 [4868] Task 6
(0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868)
(7:200) (8:4522) (9:1207) (10:3288) (11:128) (12:3551)
(13:4589) (14:1809) (15:2278) (16:998) (17:4861) (18:520)
(19:4258) (20:5000)
[5000] Task 20 Calling shutdownNow()
Finished DelayedTaskConsumer
*///:-
```

DelayedTask contiene una lista **List<DelayedTask>** denominada **sequence** que preserva el orden en que fueron creadas las tareas, para poder comprobar que efectivamente se está realizando una ordenación.

La interfaz **Delayed** tiene un método, **getDelay()**, que dice cuánto queda para que caduque el retardo o cuánto tiempo hace que ha caducado. Este método nos fuerza a utilizar la clase **TimeUnit** porque es el argumento de tipo. Esta clase resulta ser bastante útil, porque podemos convertir fácilmente las unidades sin realizar ningún cálculo. Por ejemplo, el valor de **delta** se almacena en milisegundos, pero el método **System.nanoTime()** de Java SE5 devuelve el tiempo en nanosegundos. Podemos convertir el valor de **delta** indicando en qué unidades está y en qué unidades lo queremos como en el ejemplo siguiente:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

En **getDelay()**, pasamos las unidades deseadas como argumento **unit**, y las usamos para convertir el intervalo de tiempo transcurrido desde el instante de disparo a las unidades solicitadas por el llamante, sin siquiera tener por qué conocer qué unidades son esas (éste es un ejemplo simple del patrón de diseño *Estrategia*, según el cual parte del algoritmo se pasa como argumento).

Para la ordenación, la interfaz **Delayed** también hereda la interfaz **Comparable**, así que habrá que implementar **compareTo()** para que realice una comparación razonable. **toString()** y **summary()** se encargan del formato de la salida y la clase anidada **EndSentinel** proporciona una forma de terminar todo, colocándola como último elemento de la cola.

Observe que, como **DelayedTaskConsumer** es ella misma una tarea, dispone de su propio objeto **Thread** que puede usar para ejecutar cada tarea que se extraiga de la cola. Puesto que las tareas se están ejecutando según el orden de prioridad de la cola, no hay necesidad en este ejemplo de iniciar hebras separadas para ejecutar las tareas **DelayedTask**.

Podemos ver, analizando la salida, que el orden en que se crean las tareas no tiene ningún efecto sobre el orden de ejecución, en lugar de ello, las tareas se ejecutan según el orden de sus retardos, como cabría esperar.

PriorityBlockingQueue

Se trata, básicamente, de una cola con prioridad que tiene operaciones de extracción bloqueantes. He aquí un ejemplo en el que los objetos de la cola con prioridad son tareas que salen de la cola según el orden de prioridad. A cada tarea con prioridad **PrioritizedTask** se le proporciona un número de prioridad para fijar el orden:

```

//: concurrency/PriorityBlockingQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class PrioritizedTask implements
Runnable, Comparable<PrioritizedTask> {
    private Random rand = new Random(47);
    private static int counter = 0;
    private final int id = counter++;
    private final int priority;
    protected static List<PrioritizedTask> sequence =
        new ArrayList<PrioritizedTask>();
    public PrioritizedTask(int priority) {
        this.priority = priority;
        sequence.add(this);
    }
    public int compareTo(PrioritizedTask arg) {
        return priority < arg.priority ? 1 :
            (priority > arg.priority ? -1 : 0);
    }
    public void run() {
        try {
            TimeUnit.MILLISECONDS.sleep(rand.nextInt(250));
        } catch(InterruptedException e) {
            // Forma aceptable de salir
        }
        print(this);
    }
    public String toString() {
        return String.format("[%1$-3d]", priority) +
            " Task " + id;
    }
    public String summary() {
        return "(" + id + ":" + priority + ")";
    }
    public static class EndSentinel extends PrioritizedTask {
        private ExecutorService exec;
        public EndSentinel(ExecutorService e) {
            super(-1); // La menor prioridad en este programa
            exec = e;
        }
        public void run() {
            int count = 0;
            for(PrioritizedTask pt : sequence) {
                printnb(pt.summary());
                if(++count % 5 == 0)
                    print();
            }
            print();
            print(this + " Calling shutdownNow()");
            exec.shutdownNow();
        }
    }
}

class PrioritizedTaskProducer implements Runnable {
    private Random rand = new Random(47);
    private Queue<Runnable> queue;

```

```

private ExecutorService exec;
public PrioritizedTaskProducer(
    Queue<Runnable> q, ExecutorService e) {
    queue = q;
    exec = e; // Utilizado para EndSentinel
}
public void run() {
    // Cola no limitada; nunca se bloquea.
    // Rellenarla rápidamente con prioridades aleatorias:
    for(int i = 0; i < 20; i++) {
        queue.add(new PrioritizedTask(rand.nextInt(10)));
        Thread.yield();
    }
    // Introducir tareas de prioridad máxima:
    try {
        for(int i = 0; i < 10; i++) {
            TimeUnit.MILLISECONDS.sleep(250);
            queue.add(new PrioritizedTask(10));
        }
        // Añadir tareas, primero las de menor prioridad:
        for(int i = 0; i < 10; i++)
            queue.add(new PrioritizedTask(i));
        // Un continela para detener todas las tareas:
        queue.add(new PrioritizedTask.EndSentinel(exec));
    } catch(InterruptedException e) {
        // Forma aceptable de salir
    }
    print("Finished PrioritizedTaskProducer");
}
}

class PrioritizedTaskConsumer implements Runnable {
    private PriorityBlockingQueue<Runnable> q;
    public PrioritizedTaskConsumer(
        PriorityBlockingQueue<Runnable> q) {
        this.q = q;
    }
    public void run() {
        try {
            while(!Thread.interrupted())
                // Utilizar la hebra actual para ejecutar la tarea:
                q.take().run();
        } catch(InterruptedException e) {
            // Forma aceptable de salir
        }
        print("Finished PrioritizedTaskConsumer");
    }
}

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws Exception {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        PriorityBlockingQueue<Runnable> queue =
            new PriorityBlockingQueue<Runnable>();
        exec.execute(new PrioritizedTaskProducer(queue, exec));
        exec.execute(new PrioritizedTaskConsumer(queue));
    }
} /* (Ejecutar para ver la salida) *///:~

```

Al igual que en el ejemplo anterior, la secuencia de creación de los objetos **PrioritizedTask** se almacena en la lista **sequence**, para compararla con el orden real de ejecución. El método **run()** durante un corto período de tiempo aleatorio imprime la información del objeto, mientras que **EndSentinel** proporciona la misma funcionalidad que antes al garantizar que es el último objeto de la cola.

Los objetos **PrioritizedTaskProducer** y **PrioritizedTaskConsumer** se interconectan a través de una cola **PriorityBlockingQueue**. Puesto que la naturaleza bloqueante de la cola proporciona todos los mecanismos necesarios de sincronización, observe que no hace falta ninguna sincronización explícita: no tenemos que preocuparnos por si la cola tiene algún elemento en el momento de leer de ella, porque ésta simplemente bloqueará al lector cuando no tenga ningún elemento.

El controlador de invernadero implementado con ScheduledExecutor

En el Capítulo 10, *Clases internas*, introdujimos el ejemplo de un sistema de control aplicado a un invernadero hipotético, donde se encendían y apagaban diversos elementos o se los ajustaba de alguna manera. Éste puede verse como un tipo de problema de concurrencia, siendo cada suceso deseado en el invernadero una tarea que se ejecuta en un instante predefinido. La clase **ScheduledThreadPoolExecutor** proporciona el servicio necesario para resolver el problema. Utilizando **schedule()** (para ejecutar una tarea una vez) o **scheduleAtFixedRate()** (para repetir una tarea a intervalos regulares), configuraremos objetos **Runnable** que haya que ejecutar en un cierto instante futuro. Compare el código siguiente con la técnica utilizada en el Capítulo 10, *Clases internas*, para observar cómo se simplifican las cosas cuando podemos emplear una herramienta predefinida como **ScheduledThreadPoolExecutor**:

```
//: concurrency/GreenhouseScheduler.java
// Reescritura de innerclasses/GreenhouseController.java
// para usar la clase ScheduledThreadPoolExecutor.
// {Args: 5000}
import java.util.concurrent.*;
import java.util.*;

public class GreenhouseScheduler {
    private volatile boolean light = false;
    private volatile boolean water = false;
    private String thermostat = "Day";
    public synchronized String getThermostat() {
        return thermostat;
    }
    public synchronized void setThermostat(String value) {
        thermostat = value;
    }
    ScheduledThreadPoolExecutor scheduler =
        new ScheduledThreadPoolExecutor(10);
    public void schedule(Runnable event, long delay) {
        scheduler.schedule(event,delay,TimeUnit.MILLISECONDS);
    }
    public void
    repeat(Runnable event, long initialDelay, long period) {
        scheduler.scheduleAtFixedRate(
            event, initialDelay, period, TimeUnit.MILLISECONDS);
    }
    class LightOn implements Runnable {
        public void run() {
            // Incluir aquí el código de control del hardware
            // para encender físicamente la iluminación.
            System.out.println("Turning on lights");
            light = true;
        }
    }
    class LightOff implements Runnable {
        public void run() {
```

```

    // Incluir aquí el código de control del hardware
    // para apagar físicamente la iluminación.
    System.out.println("Turning off lights");
    light = false;
}
}
class WaterOn implements Runnable {
    public void run() {
        // Incluir aquí el código de control del hardware.
        System.out.println("Turning greenhouse water on");
        water = true;
    }
}
class WaterOff implements Runnable {
    public void run() {
        // Incluir aquí el código de control del hardware.
        System.out.println("Turning greenhouse water off");
        water = false;
    }
}
class ThermostatNight implements Runnable {
    public void run() {
        // Incluir aquí el código de control del hardware.
        System.out.println("Thermostat to night setting");
        setThermostat("Night");
    }
}
class ThermostatDay implements Runnable {
    public void run() {
        // Incluir aquí el código de control del hardware.
        System.out.println("Thermostat to day setting");
        setThermostat("Day");
    }
}
class Bell implements Runnable {
    public void run() { System.out.println("Bing!"); }
}
class Terminate implements Runnable {
    public void run() {
        System.out.println("Terminating");
        scheduler.shutdownNow();
        // Hay que iniciar una tarea separada para hacer este trabajo,
        // ya que el planificador ha sido terminado:
        new Thread() {
            public void run() {
                for(DataPoint d : data)
                    System.out.println(d);
            }
        }.start();
    }
}
// Nueva característica: recopilación de datos
static class DataPoint {
    final Calendar time;
    final float temperature;
    final float humidity;
    public DataPoint(Calendar d, float temp, float hum) {
        time = d;
        temperature = temp;
    }
}

```

```

        humidity = hum;
    }
    public String toString() {
        return time.getTime() +
            String.format(
                " temperature: %1$.1f humidity: %2$.2f",
                temperature, humidity);
    }
}
private Calendar lastTime = Calendar.getInstance();
{ // Ajustar la hora con medias horas
    lastTime.set(Calendar.MINUTE, 30);
    lastTime.set(Calendar.SECOND, 00);
}
private float lastTemp = 65.0f;
private int tempDirection = +1;
private float lastHumidity = 50.0f;
private int humidityDirection = +1;
private Random rand = new Random(47);
List<DataPoint> data = Collections.synchronizedList(
    new ArrayList<DataPoint>());
class CollectData implements Runnable {
    public void run() {
        System.out.println("Collecting data");
        synchronized(GreenhouseScheduler.this) {
            // Simular que el intervalo es mayor de lo que es:
            lastTime.set(Calendar.MINUTE,
                lastTime.get(Calendar.MINUTE) + 30);
            // Una oportunidad entre 5 de invertir la dirección:
            if(rand.nextInt(5) == 4)
                tempDirection = -tempDirection;
            // Almacenar valor anterior:
            lastTemp = lastTemp +
                tempDirection * (1.0f + rand.nextFloat());
            if(rand.nextInt(5) == 4)
                humidityDirection = -humidityDirection;
            lastHumidity = lastHumidity +
                humidityDirection * rand.nextFloat();
            // Es preciso clonar Calendar, ya que en caso contrario todos
            // los puntos de datos almacenarían referencias al mismo valor
            // lastTime. Para un objeto básico como Calendar, clone() es OK.
            data.add(new DataPoint((Calendar)lastTime.clone(),
                lastTemp, lastHumidity));
        }
    }
}
public static void main(String[] args) {
    GreenhouseScheduler gh = new GreenhouseScheduler();
    gh.schedule(gh.new Terminate(), 5000);
    // La anterior clase "Restart" no es necesaria:
    gh.repeat(gh.new Bell(), 0, 1000);
    gh.repeat(gh.new ThermostatNight(), 0, 2000);
    gh.repeat(gh.new LightOn(), 0, 200);
    gh.repeat(gh.new LightOff(), 0, 400);
    gh.repeat(gh.new WaterOn(), 0, 600);
    gh.repeat(gh.new WaterOff(), 0, 800);
    gh.repeat(gh.new ThermostatDay(), 0, 1400);
    gh.repeat(gh.new CollectData(), 500, 500);
}
} /* (Ejecutar para ver la salida) */:-

```

Esta versión reorganiza el código y añade una nueva característica: recopilar las lecturas de temperatura y humedad en el invernadero. Cada objeto **DataPoint** (punto de datos) almacena y visualiza un único dato, mientras que **CollectData** es la tarea planificada que genera los elementos simulados y los añade al contenedor **List<DataPoint>** de **Greenhouse** cada vez que se la ejecuta.

Observe el uso tanto de **volatile** como de **synchronized** en los lugares apropiados, para impedir que las tareas interfieran unas con otras. Todos los métodos de la lista que almacena los objetos **DataPoint** se sincronizan empleando la utilidad **synchronizedList()** de **java.util.Collections** en el momento de crear la lista.

Ejercicio 33: (7) Modifique **GreenhouseScheduler.java** para que utilice un objeto **DelayQueue** en lugar de **ScheduledExecutor**.

Semaphore

Un bloqueo normal (de **concurrent.locks** o el bloqueo integrado **synchronized**) sólo permite a una única tarea acceder a un recurso cada vez. Un *semáforo contador* permite que n tareas accedan al recurso simultáneamente. También podemos pensar en un semáforo como algo que entrega “permisos” para usar un recurso, aunque en realidad no se utiliza ningún objeto permiso.

Como ejemplo, consideremos el concepto de *conjunto compartido de objetos*, que gestiona un número limitado de objetos permitiendo que esos objetos se extraigan del conjunto para utilizarlos y se devuelvan una vez que el usuario haya terminado con ellos. Esta funcionalidad puede encapsularse en una clase genérica:

```
//: concurrency/Pool.java
// Utilización de un semáforo dentro de conjunto compartido,
// para restringir el número de tareas que pueden usar un recurso.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();
    private volatile boolean[] checkedOut;
    private Semaphore available;
    public Pool(Class<T> classObject, int size) {
        this.size = size;
        checkedOut = new boolean[size];
        available = new Semaphore(size, true);
        // Cargar conjunto con objetos que puedan extraerse:
        for(int i = 0; i < size; ++i)
            try {
                // Presupone un constructor predeterminado:
                items.add(classObject.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
    public T checkOut() throws InterruptedException {
        available.acquire();
        return getItem();
    }
    public void checkIn(T x) {
        if(releaseItem(x))
            available.release();
    }
    private synchronized T getItem() {
        for(int i = 0; i < size; ++i)
            if(!checkedOut[i])
                checkedOut[i] = true;
```

```

        return items.get(i);
    }
    return null; // El semáforo impide llegar aquí
}
private synchronized boolean releaseItem(T item) {
    int index = items.indexOf(item);
    if(index == -1) return false; // No está en la lista
    if(checkedOut[index]) {
        checkedOut[index] = false;
        return true;
    }
    return false; // No ha sido extraído
}
} //:-

```

En esta forma simplificada, el constructor utiliza **newInstance()** para cargar de objetos el conjunto compartido. Si necesitamos un nuevo objeto, invocamos **checkOut()**, y cuando hemos finalizado con un objeto, lo devolvemos con **checkIn()**.

La matriz booleana **checkedOut** lleva la cuenta de los objetos que han sido extraídos y está gestionada por los métodos **getItem()** y **releaseItem()**. Estos, a su vez, están protegidos por el semáforo **available**, de modo que, en **checkOut()**, **available** bloquea el progreso de la llamada si no hay permisos de semáforo disponibles (lo que quiere decir que no hay más objetos en el conjunto compartido). En **checkIn()**, si el objeto que se está devolviendo es válido, se devuelve un permiso al semáforo.

Para construir un ejemplo, podemos usar **Fat**, un tipo de objeto que resulta costoso de crear, porque su constructor tarda bastante tiempo en ejecutarse:

```

//: concurrency/Fat.java
// Objetos que son costosos de crear.

public class Fat {
    private volatile double d; // Impedir optimización
    private static int counter = 0;
    private final int id = counter++;
    public Fat() {
        // Operación costosa e interrumpible:
        for(int i = 1; i < 10000; i++) {
            d += (Math.PI + Math.E) / (double)i;
        }
    }
    public void operation() { System.out.println(this); }
    public String toString() { return "Fat id: " + id; }
}
} //:-

```

Agruparemos estos objetos en un conjunto compartido para limitar el impacto de su constructor. Podemos probar la clase **Pool** creando una tarea que extraiga objetos **Fat**, los conserve durante un cierto período de tiempo y luego los devuelva:

```

//: concurrency/SemaphoreDemo.java
// Prueba de la clase Pool
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Una tarea para extraer un recurso de un conjunto compartido:
class CheckoutTask<T> implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private Pool<T> pool;
    public CheckoutTask(Pool<T> pool) {
        this.pool = pool;
    }
}

```

```

public void run() {
    try {
        T item = pool.checkOut();
        print(this + "checked out " + item);
        TimeUnit.SECONDS.sleep(1);
        print(this + "checking in " + item);
        pool.checkIn(item);
    } catch(InterruptedException e) {
        // Forma aceptable de terminar
    }
}
public String toString() {
    return "CheckoutTask " + id + " ";
}

public class SemaphoreDemo {
    final static int SIZE = 25;
    public static void main(String[] args) throws Exception {
        final Pool<Fat> pool =
            new Pool<Fat>(Fat.class, SIZE);
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < SIZE; i++)
            exec.execute(new CheckoutTask<Fat>(pool));
        print("All CheckoutTasks created");
        List<Fat> list = new ArrayList<Fat>();
        for(int i = 0; i < SIZE; i++) {
            Fat f = pool.checkOut();
            printnb(i + ": main() thread checked out ");
            f.operation();
            list.add(f);
        }
        Future<?> blocked = exec.submit(new Runnable() {
            public void run() {
                try {
                    // El semáforo impide extracciones adicionales,
                    // por lo que se bloquea la llamada:
                    pool.checkOut();
                } catch(InterruptedException e) {
                    print("checkOut() Interrupted");
                }
            }
        });
        TimeUnit.SECONDS.sleep(2);
        blocked.cancel(true); // Salir de la llamada bloqueada
        print("Checking in objects in " + list);
        for(Fat f : list)
            pool.checkIn(f);
        for(Fat f : list)
            pool.checkIn(f); // Segunda devolución ignorada
        exec.shutdown();
    }
} /* (Ejecutar para ver la salida) *///:-

```

En **main()**, se crea un objeto **Pool** para almacenar los objetos **Fat** y un conjunto de tareas **ChecoutTask** comienza a gestionar el conjunto compartido. Entonces, la hebra **main()** comienza a extraer objetos **Fat sin devolverlos**. Una vez que ha extraído todos los objetos del conjunto compartido, el semáforo no permitirá ninguna extracción adicional. El método **run()** de **blocked** se ve así bloqueado, y después de dos segundos se invoca el método **cancel()** para salir de **Future**. Observe que las extracciones redundantes son ignoradas por el objeto **Pool**.

Este ejemplo depende de que el cliente de **Pool** sea riguroso y devuelva voluntariamente los elementos, lo cual es la solución más simple, siempre que funcione. Si no podemos confiar siempre en esto, *Thinking in Patterns* (en www.MindView.net) contiene análisis adicionales de formas que pueden emplearse para gestionar los objetos extraídos de conjuntos de objetos compartidos.

Exchanger

Un intercambiador (**Exchanger**) es una barrera que intercambia objetos entre dos tareas. Cuando las tareas entran en la barrera, cada una de ellas tiene un objeto, y cuando salen tienen el objeto que anteriormente era propiedad de la otra tarea. Los intercambiadores se utilizan típicamente cuando una tarea está creando objetos que son caros de producir y otra tarea está consumiendo dichos objetos; de esta forma, pueden crearse más objetos al mismo tiempo que están siendo consumidos.

Para probar la clase **Exchanger**, vamos a crear tareas productoras y consumidoras que, mediante genéricos y objetos **Generator**, funcionarán con cualquier tipo de objeto, y luego las aplicaremos a la clase **Fat**. Los objetos **Exchanger-Producer** y **ExchangerConsumer** utilizan una lista **List<T>** como el objeto que hay que intercambiar; cada uno contiene un objeto **Exchanger** para este contenedor **List<T>**. Cuando se invoca el método **Exchanger.exchange()**, éste se bloquea hasta que la otra tarea invoca su método **exchange()**, y cuando ambos métodos **exchange()** se han completado, los contenidos **List<T>** habrán sido intercambiados:

```
//: concurrency/ExchangerDemo.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ExchangerProducer<T> implements Runnable {
    private Generator<T> generator;
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    ExchangerProducer(Exchanger<List<T>> exchg,
        Generator<T> gen, List<T> holder) {
        exchanger = exchg;
        generator = gen;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                for(int i = 0; i < ExchangerDemo.size; i++)
                    holder.add(generator.next());
                // Intercambiar el contenedor vacío por el lleno:
                holder = exchanger.exchange(holder);
            }
        } catch(InterruptedException e) {
            // OK terminar de esta forma.
        }
    }
}

class ExchangerConsumer<T> implements Runnable {
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    private volatile T value;
    ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder) {
        exchanger = ex;
        this.holder = holder;
    }
    public void run() {
        try {
```

```

        while(!Thread.interrupted()) {
            holder = exchanger.exchange(holder);
            for(T x : holder) {
                value = x; // Extraer valor
                holder.remove(x); // OK para CopyOnWriteArrayList
            }
        }
    } catch(InterruptedException e) {
        // OK terminar de esta forma.
    }
    System.out.println("Final value: " + value);
}

public class ExchangerDemo {
    static int size = 10;
    static int delay = 5; // Segundos
    public static void main(String[] args) throws Exception {
        if(args.length > 0)
            size = new Integer(args[0]);
        if(args.length > 1)
            delay = new Integer(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();
        List<Fat>
            producerList = new CopyOnWriteArrayList<Fat>(),
            consumerList = new CopyOnWriteArrayList<Fat>();
        exec.execute(new ExchangerProducer<Fat>(xc,
            BasicGenerator.create(Fat.class), producerList));
        exec.execute(
            new ExchangerConsumer<Fat>(xc, consumerList));
        TimeUnit.SECONDS.sleep(delay);
        exec.shutdownNow();
    }
} /* Output: (Sample)
Final value: Fat id: 29999
*///:~

```

En `main()`, se crea un único objeto `Exchanger` para que lo empleen ambas tareas, y dos contenedores `CopyOnWriteArrayList` para intercambiar. Esta variante concreta de `List` permite que se invoque el método `remove()` mientras que se está recorriendo la lista sin que se genere una excepción `ConcurrentModificationException`. `ExchangerProducer` rellena una lista y luego intercambia la lista llena por la vacía que `ExchangerConsumer` le entrega. Debido a la existencia del objeto `Exchanger`, el llenado de una lista y el consumo de la otra pueden tener lugar simultáneamente.

Ejercicio 34: (1) Modifique `ExchangerDemo.java` para utilizar su propia clase en lugar de `Fat`.

Simulación

Uno de los usos más interesantes y atractivos de la concurrencia es el de crear simulaciones. Utilizando la concurrencia, cada componente de una simulación puede ser su propia tarea, y esto hace que la simulación resulte mucho más fácil de programar. Muchos juegos informáticos y animaciones por computadora en las películas son simulaciones, y `HorseRace.java` y `GreenhouseScheduler.java`, mostrados anteriormente, también podrían considerarse simulaciones.

Simulación de un cajero

Esta simulación clásica puede representar cualquier situación en la que aparecen objetos aleatoriamente, y estos objetos requieren un intervalo de tiempo aleatorio para ser servido por un número limitado de servidores. Es posible construir la simulación para determinar el número ideal de servidores.

En este ejemplo, cada cliente del banco requiere una cierta cantidad de tiempo de servicio, que es el número de unidades de tiempo que el cajero debe invertir en el cliente para satisfacer sus necesidades. La cantidad de tiempo de servicio será diferente para cada cliente y se determinará aleatoriamente. Además, no sabemos cuántos clientes llegarán en cada intervalo, por lo que esto también se determinará aleatoriamente.

```

//: concurrency/BankTellerSimulation.java
// Utilización de colas y mecanismos multihebra.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Los objetos de sólo lectura no requieren sincronización
class Customer {
    private final int serviceTime;
    public Customer(int tm) { serviceTime = tm; }
    public int getServiceTime() { return serviceTime; }
    public String toString() {
        return "[" + serviceTime + "]";
    }
}

// Mostrar a la fila de clientes cómo visualizarse:
class CustomerLine extends ArrayBlockingQueue<Customer> {
    public CustomerLine(int maxLineSize) {
        super(maxLineSize);
    }
    public String toString() {
        if(this.size() == 0)
            return "[Empty]";
        StringBuilder result = new StringBuilder();
        for(Customer customer : this)
            result.append(customer);
        return result.toString();
    }
}

// Añadir clientes aleatoriamente a una cola:
class CustomerGenerator implements Runnable {
    private CustomerLine customers;
    private static Random rand = new Random(47);
    public CustomerGenerator(CustomerLine cq) {
        customers = cq;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(300));
                customers.put(new Customer(rand.nextInt(1000)));
            }
        } catch(InterruptedException e) {
            System.out.println("CustomerGenerator interrupted");
        }
        System.out.println("CustomerGenerator terminating");
    }
}

class Teller implements Runnable, Comparable<Teller> {
    private static int counter = 0;
    private final int id = counter++;

```

```

// Clientes servidos durante este intervalo:
private int customersServed = 0;
private CustomerLine customers;
private boolean servingCustomerLine = true;
public Teller(CustomerLine cq) { customers = cq; }
public void run() {
    try {
        while(!Thread.interrupted()) {
            Customer customer = customers.take();
            TimeUnit.MILLISECONDS.sleep(
                customer.getServiceTime());
            synchronized(this) {
                customersServed++;
                while(!servingCustomerLine)
                    wait();
            }
        }
    } catch(InterruptedException e) {
        System.out.println(this + "interrupted");
    }
    System.out.println(this + "terminating");
}
public synchronized void doSomethingElse() {
    customersServed = 0;
    servingCustomerLine = false;
}
public synchronized void serveCustomerLine() {
    assert !servingCustomerLine:"already serving: " + this;
    servingCustomerLine = true;
    notifyAll();
}
public String toString() { return "Teller " + id + " "; }
public String shortString() { return "T" + id; }
// Usado por la cola de prioridad:
public synchronized int compareTo(Teller other) {
    return customersServed < other.customersServed ? -1 :
        (customersServed == other.customersServed ? 0 : 1);
}

}

class TellerManager implements Runnable {
    private ExecutorService exec;
    private CustomerLine customers;
    private PriorityQueue<Teller> workingTellers =
        new PriorityQueue<Teller>();
    private Queue<Teller> tellersDoingOtherThings =
        new LinkedList<Teller>();
    private int adjustmentPeriod;
    private static Random rand = new Random(47);
    public TellerManager(ExecutorService e,
        CustomerLine customers, int adjustmentPeriod) {
        exec = e;
        this.customers = customers;
        this.adjustmentPeriod = adjustmentPeriod;
        // Comenzar con un único cajero:
        Teller teller = new Teller(customers);
        exec.execute(teller);
        workingTellers.add(teller);
    }
}

```

```

public void adjustTellerNumber() {
    // Esto es realmente un sistema de control. Ajustando
    // los números, podemos descubrir problemas de estabilidad
    // en el mecanismo de control.
    // Si la fila es demasiado larga, añadir otro cajero:
    if(customers.size() / workingTellers.size() > 2) {
        // Si los cajeros están descansando o haciendo
        // otra tarea, decir a uno que venga:
        if(tellersDoingOtherThings.size() > 0) {
            Teller teller = tellersDoingOtherThings.remove();
            teller.serveCustomerLine();
            workingTellers.offer(teller);
            return;
        }
        // en caso contrario, crear (contratar) un nuevo cajero
        Teller teller = new Teller(customers);
        exec.execute(teller);
        workingTellers.add(teller);
        return;
    }
    // Si la fila es lo suficientemente corta, eliminar un cajero:
    if(workingTellers.size() > 1 &&
       customers.size() / workingTellers.size() < 2)
        reassignOneTeller();
    // Si no hay una fila, sólo hace falta un cajero:
    if(customers.size() == 0)
        while(workingTellers.size() > 1)
            reassignOneTeller();
}
// Dar a un cajero un trabajo diferente o un descanso:
private void reassignOneTeller() {
    Teller teller = workingTellers.poll();
    teller.doSomethingElse();
    tellersDoingOtherThings.offer(teller);
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            TimeUnit.MILLISECONDS.sleep(adjustmentPeriod);
            adjustTellerNumber();
            System.out.print(customers + " { ");
            for(Teller teller : workingTellers)
                System.out.print(teller.shortString() + " ");
            System.out.println("}");
        }
    } catch(InterruptedException e) {
        System.out.println(this + "interrupted");
    }
    System.out.println(this + "terminating");
}
public String toString() { return "TellerManager"; }
}

public class BankTellerSimulation {
    static final int MAX_LINE_SIZE = 50;
    static final int ADJUSTMENT_PERIOD = 1000;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // Si la fila es muy larga, los clientes se irán:
        CustomerLine customers =

```

```

    new CustomerLine(MAX_LINE_SIZE);
exec.execute(new CustomerGenerator(customers));
// El director añadirá o quitará cajeros según sea necesario:
exec.execute(new TellerManager(
    exec, customers, ADJUSTMENT_PERIOD));
if(args.length > 0) // Optional argument
    TimeUnit.SECONDS.sleep(new Integer(args[0]));
else {
    System.out.println("Press 'Enter' to quit");
    System.in.read();
}
exec.shutdownNow();
}
} /* Output: (Sample)
[429] [200] [207] { T0 T1 }
[861] [258] [140] [322] { T0 T1 }
[575] [342] [804] [826] [896] [984] { T0 T1 T2 }
[984] [810] [141] [12] [689] [992] [976] [368] [395] [354] { T0 T1 T2 T3 }
Teller 2 interrupted
Teller 2 terminating
Teller 1 interrupted
Teller 1 terminating
TellerManager interrupted
TellerManager terminating
Teller 3 interrupted
Teller 3 terminating
Teller 0 interrupted
Teller 0 terminating
CustomerGenerator interrupted
CustomerGenerator terminating
*///:-
```

Los objetos **Customer** son muy simples, conteniendo únicamente un campo **final int**. Puesto que estos objetos nunca cambian, son objetos de *sólo lectura* y no requieren sincronización ni el uso de **volatile**. Además, cada tarea **Teller** (cajero) sólo elimina un objeto **Customer** cada vez de la cola de entrada, y trabaja sobre un objeto **Customer** hasta que se haya completado, por lo que en cualquier caso a cada objeto **Customer** sólo accederá una tarea en cada momento.

CustomerLine representa una única fila en la que los clientes esperan antes de ser servidos por un objeto **Teller**. Se trata simplemente de una cola **ArrayBlockingQueue** que tiene un método **toString()** que imprime los resultados de la forma deseada.

Con cada objeto **CustomerLine** se asocia un objeto **CustomerGenerator**, que introduce clientes en la cola a intervalos aleatorios.

Cada objeto **Teller** extrae clientes de la cola **CustomerLine** y los procesa de uno en uno, llevando la cuenta del número de clientes a los que ha servido durante ese intervalo concreto. Se le puede decir al cajero que haga alguna otra cosa con **doSomethingElse()** cuando no haya suficientes clientes o que atienda a la fila con **serveCustomerLine()** cuando haya muchos clientes. Para seleccionar el siguiente cajero que hay que poner a atender a los clientes, el método **compareTo()** examina el número de clientes servidos, de modo que una cola **PriorityQueue** puede seleccionar automáticamente al cajero que haya realizado un menor trabajo hasta el momento.

El objeto **TellerManager** es el centro de actividad. Lleva el control de todos los cajeros y de lo que está sucediendo con los clientes. Uno de los aspectos interesantes de esta simulación es que se intenta descubrir el número óptimo de cajeros para un flujo de clientes determinado. Podemos ver esto en el método **adjustTellerNumber()**, que es un sistema de control para agregar y eliminar cajeros de una manera estable. Todos los sistemas de control tienen problemas de estabilidad; si reaccionan demasiado rápido a un cambio, son inestables y si reaccionan demasiado lento, el sistema se desplaza a uno de sus extremos.

Ejercicio 35: (8) Modifique **BankTellerSimulation.java** para que represente clientes web que estén enviando solicitudes a un número fijo de servidores. El objetivo es determinar la carga que el grupo de servidores puede gestionar.

Simulación de un restaurante

Esta simulación retoma el ejemplo simple **Restaurant.java** mostrado anteriormente en el capítulo, añadiendo más componentes de simulación, como los pedidos (**Order**) y los platos (**Plate**), y reutiliza las clases **menu** del Capítulo 19, *Tipos enumerados*.

También introduce la cola **SynchronousQueue** de Java SE5, que es una cola bloqueante que no tienen capacidad interna, por lo que cada operación **put()** (inserción) debe esperar a que se produzca una operación **take()** (extracción), y viceversa. Es como si estuviéramos entregando un objeto a alguien (no hay ninguna mesa sobre el que ponerlo, por lo que sólo funciona si la otra persona está tendiendo una mano hacia nosotros y lista para recibir el objeto). En este ejemplo, **SynchronousQueue** representa el espacio situado delante del comensal, para hacer hincapié en la idea de que sólo se puede servir un plato cada vez.

El resto de las clases de la funcionalidad de este ejemplo se ajustan a la estructura de **Restaurant.java** o pretenden ser una plasmación bastante directa de las operaciones de un restaurante real:

```
//: concurrency/restaurant2/RestaurantWithQueues.java
// {Args: 5}
package concurrency.restaurant2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Se entrega al camarero, que a su vez se lo da al:
class Order { // (un objeto de transferencia de datos)
    private static int counter = 0;
    private final int id = counter++;
    private final Customer customer;
    private final WaitPerson waitPerson;
    private final Food food;
    public Order(Customer cust, WaitPerson wp, Food f) {
        customer = cust;
        waitPerson = wp;
        food = f;
    }
    public Food item() { return food; }
    public Customer getCustomer() { return customer; }
    public WaitPerson getWaitPerson() { return waitPerson; }
    public String toString() {
        return "Order: " + id + " item: " + food +
            " for: " + customer +
            " served by: " + waitPerson;
    }
}

// Esto es lo que el chef devuelve:
class Plate {
    private final Order order;
    private final Food food;
    public Plate(Order ord, Food f) {
        order = ord;
        food = f;
    }
    public Order getOrder() { return order; }
    public Food getFood() { return food; }
    public String toString() { return food.toString(); }
}

class Customer implements Runnable {
```

```

private static int counter = 0;
private final int id = counter++;
private final WaitPerson waitPerson;
// Sólo se puede recibir un plato cada vez:
private SynchronousQueue<Plate> placeSetting =
    new SynchronousQueue<Plate>();
public Customer(WaitPerson w) { waitPerson = w; }
public void
deliver(Plate p) throws InterruptedException {
    // Sólo se bloquea si el cliente está todavía
    // comiendo el plato anterior:
    placeSetting.put(p);
}
public void run() {
    for(Course course : Course.values()) {
        Food food = course.randomSelection();
        try {
            waitPerson.placeOrder(this, food);
            // Se bloquea hasta que se entregue el plato:
            print(this + "eating " + placeSetting.take());
        } catch(InterruptedException e) {
            print(this + "waiting for " +
                course + " interrupted");
            break;
        }
    }
    print(this + "finished meal, leaving");
}
public String toString() {
    return "Customer " + id + " ";
}
}

class WaitPerson implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    BlockingQueue<Plate> filledOrders =
        new LinkedBlockingQueue<Plate>();
    public WaitPerson(Restaurant rest) { restaurant = rest; }
    public void placeOrder(Customer cust, Food food) {
        try {
            // No debería bloquearse en la práctica, porque es una cola
            // LinkedBlockingQueue sin límite de tamaño:
            restaurant.orders.put(new Order(cust, this, food));
        } catch(InterruptedException e) {
            print(this + " placeOrder interrupted");
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Se bloquea hasta que está listo un plato
                Plate plate = filledOrders.take();
                print(this + "received " + plate +
                    " delivering to " +
                    plate.getOrder().getCustomer());
                plate.getOrder().getCustomer().deliver(plate);
            }
        }
    }
}

```

```

        } catch(InterruptedException e) {
            print(this + " interrupted");
        }
        print(this + " off duty");
    }
    public String toString() {
        return "WaitPerson " + id + " ";
    }
}

class Chef implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    private static Random rand = new Random(47);
    public Chef(Restaurant rest) { restaurant = rest; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Se bloquea hasta que aparece un pedido:
                Order order = restaurant.orders.take();
                Food requestedItem = order.item();
                // El tiempo para preparar el pedido:
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                Plate plate = new Plate(order, requestedItem);
                order.getWaitPerson().filledOrders.put(plate);
            }
        } catch(InterruptedException e) {
            print(this + " interrupted");
        }
        print(this + " off duty");
    }
    public String toString() { return "Chef " + id + " "; }
}

class Restaurant implements Runnable {
    private List<WaitPerson> waitPersons =
        new ArrayList<WaitPerson>();
    private List<Chef> chefs = new ArrayList<Chef>();
    private ExecutorService exec;
    private static Random rand = new Random(47);
    BlockingQueue<Order>
        orders = new LinkedBlockingQueue<Order>();
    public Restaurant(ExecutorService e, int nWaitPersons,
        int nChefs) {
        exec = e;
        for(int i = 0; i < nWaitPersons; i++) {
            WaitPerson waitPerson = new WaitPerson(this);
            waitPersons.add(waitPerson);
            exec.execute(waitPerson);
        }
        for(int i = 0; i < nChefs; i++) {
            Chef chef = new Chef(this);
            chefs.add(chef);
            exec.execute(chef);
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {

```

```

        // Llega un nuevo cliente; asignar un camarero:
        WaitPerson wp = waitPersons.get(
            rand.nextInt(waitPersons.size()));
        Customer c = new Customer(wp);
        exec.execute(c);
        TimeUnit.MILLISECONDS.sleep(100);
    }
} catch(InterruptedException e) {
    print("Restaurant interrupted");
}
print("Restaurant closing");
}

}

public class RestaurantWithQueues {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        Restaurant restaurant = new Restaurant(exec, 5, 2);
        exec.execute(restaurant);
        if(args.length > 0) // Argumento opcional
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
        else {
            print("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* Output: (Sample)
WaitPerson 0 received SPRING_ROLLS delivering to Customer 1
Customer 1 eating SPRING_ROLLS
WaitPerson 3 received SPRING_ROLLS delivering to Customer 0
Customer 0 eating SPRING_ROLLS
WaitPerson 0 received BURRITO delivering to Customer 1
Customer 1 eating BURRITO
WaitPerson 3 received SPRING_ROLLS delivering to Customer 2
Customer 2 eating SPRING_ROLLS
WaitPerson 1 received SOUP delivering to Customer 3
Customer 3 eating SOUP
WaitPerson 3 received VINDALOO delivering to Customer 0
Customer 0 eating VINDALOO
WaitPerson 0 received FRUIT delivering to Customer 1
...
*///:~

```

Un aspecto muy importante de este ejemplo es la gestión de la complejidad utilizando colas para la comunicación entre tareas. Esta técnica simplifica enormemente el proceso de la programación concurrente, al invertir el control: las tareas no interfieren directamente entre sí. En su lugar, las tareas se intercambian objetos a través de colas. La tarea receptora gestiona el objeto, tratándola como un mensaje, en lugar de recibir directamente mensajes. Si seguimos esta técnica siempre que podamos, tendremos una mayor posibilidad de construir sistemas concurrentes robustos.

Ejercicio 36: (10) Modifique **RestaurantWithQueues.java** para que haya un objeto **OrderTicket** (nota de pedido) por cada mesa. Cambie **order** por **orderTicket**, y añada la clase **Table** (mesa), con múltiples clientes (**Customer**) por mesa.

Distribución de trabajo

He aquí un ejemplo de simulación simple donde se aúnan muchos de los conceptos vistos en el capítulo. Considere una hipotética línea de montaje robotizada para automóviles. Cada objeto automóvil (**Car**) será construido en varias etapas, comenzando por la fabricación del chasis y siguiendo por el montaje del motor, de la transmisión y de las ruedas.

```

//: concurrency/CarBuilder.java
// Un ejemplo complejo de tareas que funcionan conjuntamente.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Car {
    private final int id;
    private boolean
        engine = false, driveTrain = false, wheels = false;
    public Car(int idn) { id = idn; }
    // Objeto Car vacío:
    public Car() { id = -1; }
    public synchronized int getId() { return id; }
    public synchronized void addEngine() { engine = true; }
    public synchronized void addDriveTrain() {
        driveTrain = true;
    }
    public synchronized void addWheels() { wheels = true; }
    public synchronized String toString() {
        return "Car " + id + " [" + " engine: " + engine
            + " driveTrain: " + driveTrain
            + " wheels: " + wheels + " ]";
    }
}

class CarQueue extends LinkedBlockingQueue<Car> {}

class ChassisBuilder implements Runnable {
    private CarQueue carQueue;
    private int counter = 0;
    public ChassisBuilder(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(500);
                // Hacer chasis:
                Car c = new Car(counter++);
                print("ChassisBuilder created " + c);
                // Insertar en la cola
                carQueue.put(c);
            }
        } catch(InterruptedException e) {
            print("Interrupted: ChassisBuilder");
        }
        print("ChassisBuilder off");
    }
}

class Assembler implements Runnable {
    private CarQueue chassisQueue, finishingQueue;
    private Car car;
    private CyclicBarrier barrier = new CyclicBarrier(4);
    private RobotPool robotPool;
    public Assembler(CarQueue cq, CarQueue fq, RobotPool rp){
        chassisQueue = cq;
        finishingQueue = fq;
        robotPool = rp;
    }
}

```

```

public Car car() { return car; }
public CyclicBarrier barrier() { return barrier; }
public void run() {
    try {
        while(!Thread.interrupted()) {
            // Se bloque hasta que esté disponible un chasis:
            car = chassisQueue.take();
            // Comprar robots para realizar el trabajo:
            robotPool.hire(EngineRobot.class, this);
            robotPool.hire(DriveTrainRobot.class, this);
            robotPool.hire(WheelRobot.class, this);
            barrier.await(); // Until the robots finish
            // Insertar coche en la cola de acabado (finishingQueue)
            // para trabajos adicionales
            finishingQueue.put(car);
        }
    } catch(InterruptedException e) {
        print("Exiting Assembler via interrupt");
    } catch(BrokenBarrierException e) {
        // Queremos que nos informen de esta excepción
        throw new RuntimeException(e);
    }
    print("Assembler off");
}
}

class Reporter implements Runnable {
    private CarQueue carQueue;
    public Reporter(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(carQueue.take());
            }
        } catch(InterruptedException e) {
            print("Exiting Reporter via interrupt");
        }
        print("Reporter off");
    }
}

abstract class Robot implements Runnable {
    private RobotPool pool;
    public Robot(RobotPool p) { pool = p; }
    protected Assembler assembler;
    public Robot assignAssembler(Assembler assembler) {
        this.assembler = assembler;
        return this;
    }
    private boolean engage = false;
    public synchronized void engage() {
        engage = true;
        notifyAll();
    }
    // La parte de run() que es diferente para cada robot:
    abstract protected void performService();
    public void run() {
        try {
            powerDown(); // Esperar hasta que haga falta

```

```

        while(!Thread.interrupted()) {
            performService();
            assembler.barrier().await(); // Sincronizar
            // Hemos finalizado con este trabajo...
            powerDown();
        }
    } catch(InterruptedException e) {
        print("Exiting " + this + " via interrupt");
    } catch(BrokenBarrierException e) {
        // Queremos que nos informen de esta excepción
        throw new RuntimeException(e);
    }
    print(this + " off");
}
private synchronized void
powerDown() throws InterruptedException {
    engage = false;
    assembler = null; // Desconectar del ensamblador (Assembler)
    // Volver a ponernos en la cola de disponibles:
    pool.release(this);
    while(engage == false) // Desconectar alimentación
        wait();
}
public String toString() { return getClass().getName(); }
}

class EngineRobot extends Robot {
    public EngineRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing engine");
        assembler.car().addEngine();
    }
}

class DriveTrainRobot extends Robot {
    public DriveTrainRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing DriveTrain");
        assembler.car().addDriveTrain();
    }
}

class WheelRobot extends Robot {
    public WheelRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing Wheels");
        assembler.car().addWheels();
    }
}

class RobotPool {
    // Impide sileciosamente que existan entrada idénticas:
    private Set<Robot> pool = new HashSet<Robot>();
    public synchronized void add(Robot r) {
        pool.add(r);
        notifyAll();
    }
    public synchronized void
    hire(Class<? extends Robot> robotType, Assembler d)
}

```

```

throws InterruptedException {
    for(Robot r : pool)
        if(r.getClass().equals(robotType)) {
            pool.remove(r);
            r.assignAssembler(d);
            r.engage(); // Encenderlo para realizar la tarea
            return;
        }
    wait(); // Ninguno disponible
    hire(robotType, d); // Intentar de nuevo recursivamente
}
public synchronized void release(Robot r) { add(r); }
}

public class CarBuilder {
    public static void main(String[] args) throws Exception {
        CarQueue chassisQueue = new CarQueue();
        finishingQueue = new CarQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        RobotPool robotPool = new RobotPool();
        exec.execute(new EngineRobot(robotPool));
        exec.execute(new DriveTrainRobot(robotPool));
        exec.execute(new WheelRobot(robotPool));
        exec.execute(new Assembler(
            chassisQueue, finishingQueue, robotPool));
        exec.execute(new Reporter(finishingQueue));
        // Comenzar a funcionar produciendo un chasis:
        exec.execute(new ChassisBuilder(chassisQueue));
        TimeUnit.SECONDS.sleep(7);
        exec.shutdownNow();
    }
} /* (Ejecutar para ver la salida) *///:~

```

Los objetos **Car** se transportan de un lugar a otro mediante una cola **CarQueue**, que es un tipo de **LinkedBlockingQueue**. Un objeto **ChassisBuilder** crea un chasis de coche y lo coloca en una cola **CarQueue**. El objeto **Assembler** extrae los objetos **Car** de una cola **CarQueue** y adquiere objetos **Robot** para trabajar con ellos. Una barrera **CyclicBarrier** permite que **Assembler** espere hasta que todos los objetos **Robot** hayan terminado, en cuyo momento coloca el objeto **Car** en la cola **CarQueue** de salida para transportarlo a la siguiente operación. El consumidor de la cola **CarQueue** final es un objeto **Reporter**, que simplemente imprime los datos del objeto **Car** para demostrar que las tareas se han completado apropiadamente.

Los objetos **Robot** se gestionan mediante un conjunto compartido y cuando hace falta realizar un trabajo, se extrae el objeto **Robot** apropiado de ese conjunto. Después de completar el trabajo, el objeto **Robot** se devuelve al conjunto compartido.

En **main()**, se crean todos los objetos necesarios y se inicializan las tareas, inciando **ChassisBuilder** en último lugar para comenzar con el proceso (sin embargo, debido al comportamiento de la cola **LinkedBlockingQueue**, no importaría si iniciáramos la tarea de construcción del chasis en primer lugar). Observe que este programa se ajusta a todas las directrices relativas al tiempo de vida de los objetos presentadas en este capítulo, de manera que el proceso de terminación resulta seguro.

Observará que **Car** tiene definidos todos sus métodos como **synchronized**. En realidad, *en este ejemplo*, esto es redundante, porque dentro de la fábrica, los objetos **Car** se desplazan a través de colas y sólo una tarea puede estar trabajando en un cierto coche en un determinado momento. Básicamente, las colas fuerzan a realizar un acceso serializado a los objetos **Car**. Pero ésta es exactamente el tipo de trampa en la que podemos caer; podemos decir “Tratemos de optimizar el programa no sincronizando la clase **Car**, porque parece que no es necesario”. Pero posteriormente, al conectar este sistema a otro que *si que necesite* que el objeto **Car** esté sincronizado, el sistema no funcionará.

Brian Goetz comenta:

Resulta mucho más fácil decir: “Car podría ser usado en múltiples hebras, así que hagamos que sea seguro de cara a las hebras de la forma más evidente”. La manera en que podemos caracterizar este

enfoque es la siguiente: en determinados lugares, podemos encontrar una serie de vallas dispuestas en lugares donde existen desniveles abruptos, y junto a ellas podemos ver señales que dicen: "No se apoye en las vallas". Por supuesto, el auténtico propósito de esta regla no es impedirnos apoyarnos en las vallas, sino impedirnos que nos caigamos por el acantilado. Pero "No se apoye en las vallas" es una regla mucho más fácil de seguir que "no se caiga por el acantilado".

- Ejercicio 37:** (2) Modifique **CarBuilder.java** para añadir otra etapa al proceso de construcción de automóviles, en la que añadiremos el sistema de escape, los asientos y los accesorios. Al igual que con la segunda etapa, suponga que estos procesos pueden ser realizados simultáneamente por robots.
- Ejercicio 38:** (3) Utilizando la técnica empleada en **CarBuilder.java**, modele el ejemplo de construcción de casas que hemos comentado en este capítulo.

Optimización del rendimiento

Muchas de las clases de la biblioteca **java.util.concurrent** de Java SE5 tienen el propósito de mejorar el rendimiento. Cuando examinamos de manera somera la biblioteca **concurrent**, puede resultar difícil discernir qué clases están pensadas para una utilización normal (como por ejemplo **BlockingQueue**) y cuáles otras se usan exclusivamente para mejorar el rendimiento. En esta sección, vamos a examinar algunos de los problemas y de las clases relativos a las técnicas de optimización del rendimiento.

Comparación de las tecnologías mutex

Ahora que Java incluye la antigua palabra clave **synchronized** junto con las nuevas clases **Lock** y **Atomic** de Java SE5, resulta interesante comparar las diferentes técnicas para poder comprender mejor las ventajas de cada una y saber cuándo emplearlas.

La técnica más simple consiste en intentar una prueba sencilla de cada técnica, como la siguiente:

```
//: concurrency/SimpleMicroBenchmark.java
// Los peligros de las micropruebas.
import java.util.concurrent.locks.*;

abstract class Incrementable {
    protected long counter = 0;
    public abstract void increment();
}

class SynchronizingTest extends Incrementable {
    public synchronized void increment() { ++counter; }
}

class LockingTest extends Incrementable {
    private Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();
        try {
            ++counter;
        } finally {
            lock.unlock();
        }
    }
}

public class SimpleMicroBenchmark {
    static long test(Incrementable incr) {
        long start = System.nanoTime();
        for(long i = 0; i < 100000000L; i++)
            incr.increment();
        return System.nanoTime() - start;
    }
}
```

```

    incr.increment();
    return System.nanoTime() - start;
}
public static void main(String[] args) {
    long synchTime = test(new SynchronizingTest());
    long lockTime = test(new LockingTest());
    System.out.printf("synchronized: %1$10d\n", synchTime);
    System.out.printf("Lock:          %1$10d\n", lockTime);
    System.out.printf("Lock/synchronized = %1$.3f",
        (double)lockTime/(double)synchTime);
}
} /* Output: (75% match)
synchronized: 244919117
Lock:          939098964
Lock/synchronized = 3.834
*///:-

```

Podemos ver, analizando la salida, que las llamadas al método **synchronized** parecen ser más rápidas que la utilización de un bloque **ReentrantLock**. ¿Qué es lo que está sucediendo?

Este ejemplo ilustra los peligros de las denominadas “micropruebas de rendimiento”.²³ Generalmente, este término se refiere a la realización de pruebas de rendimiento de una característica aislada, fuera de contexto. Por supuesto, sigue siendo necesario diseñar pruebas para verificar enunciados como “**Lock** es mucho más rápido que **synchronized**”. Pero tenemos que ser conscientes de lo que está sucediendo realmente durante la compilación y en tiempo de ejecución a la hora de escribir estos tipos de pruebas.

Existen diversos problemas en el ejemplo anterior. En primer lugar, sólo podremos ver la verdadera diferencia de rendimiento si los mutex *están conteniendo*, así que tiene que haber múltiples tareas intentando acceder a las secciones de código protegidas por el mutex. En el ejemplo anterior, cada mutex se comprueba mediante la única hebra **main()** aislada.

En segundo lugar, es posible que el compilador realice optimizaciones especiales al ver la palabra clave **synchronized**, y que incluso se percate de que este programa tiene una sola hebra. El compilador podría incluso identificar que el contador **counter** simplemente se está incrementando un número fijo de veces, y limitarse a precalcular el resultado. Existen muchas variaciones entre los distintos compiladores y sistemas de ejecución, así que resulta difícil saber exactamente qué es lo que sucederá, pero necesitamos impedir que el compilador pueda llegar a predecir el resultado de los cálculos.

Para diseñar una prueba válida, debemos hacer el programa más complejo. En primer lugar, necesitamos múltiples tareas, y no sólo tareas que modifiquen valores internos, sino también tareas que lean esos valores (en caso contrario, el optimizador podría darse cuenta de que los valores no están siendo utilizados nunca). Además, el cálculo debe ser complejo y lo suficientemente impredecible como para que el compilador no tenga la posibilidad de realizar optimizaciones agresivas. Conseguiremos esto precargando una matriz de gran tamaño con valores enteros aleatorios (la precarga reduce el impacto de las llamadas a **Random.nextInt()** en los bucles principales) y utilizando esos valores en un sumatorio:

```

//: concurrency/SynchronizationComparisons.java
// Comparación del rendimiento de objetos Lock y Atomic
// explícitos y la palabra clave synchronized.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

abstract class Accumulator {
    public static long cycles = 50000L;
    // Número de modificadores y lectores durante cada prueba:
    private static final int N = 4;
    public static ExecutorService exec =

```

²³ Brian Goetz me ayudó mucho, explicándome estos problemas. Consulte su artículo en www128.ibm.com/developerworks/library/j-jtp12214 para conocer más detalles acerca de las medidas de rendimiento.

```

        Executors.newFixedThreadPool(N*2);
    private static CyclicBarrier barrier =
        new CyclicBarrier(N*2 + 1);
    protected volatile int index = 0;
    protected volatile long value = 0;
    protected long duration = 0;
    protected String id = "error";
    protected final static int SIZE = 100000;
    protected static int[] preLoaded = new int[SIZE];
    static {
        // Cargar la matriz con números aleatorios:
        Random rand = new Random(47);
        for(int i = 0; i < SIZE; i++)
            preLoaded[i] = rand.nextInt();
    }
    public abstract void accumulate();
    public abstract long read();
    private class Modifier implements Runnable {
        public void run() {
            for(long i = 0; i < cycles; i++)
                accumulate();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    private class Reader implements Runnable {
        private volatile long value;
        public void run() {
            for(long i = 0; i < cycles; i++)
                value = read();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public void timedTest() {
        long start = System.nanoTime();
        for(int i = 0; i < N; i++) {
            exec.execute(new Modifier());
            exec.execute(new Reader());
        }
        try {
            barrier.await();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        duration = System.nanoTime() - start;
        printf("%-13s: %13d\n", id, duration);
    }
    public static void
    report(Accumulator acc1, Accumulator acc2) {
        printf("%-22s: %.2f\n", acc1.id + "/" + acc2.id,
            (double)acc1.duration/(double)acc2.duration);
    }
}

```

```

class BaseLine extends Accumulator {
    { id = "BaseLine"; }
    public void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public long read() { return value; }
}

class SynchronizedTest extends Accumulator {
    { id = "synchronized"; }
    public synchronized void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public synchronized long read() {
        return value;
    }
}

class LockTest extends Accumulator {
    { id = "Lock"; }
    private Lock lock = new ReentrantLock();
    public void accumulate() {
        lock.lock();
        try {
            value += preLoaded[index++];
            if(index >= SIZE) index = 0;
        } finally {
            lock.unlock();
        }
    }
    public long read() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}

class AtomicTest extends Accumulator {
    { id = "Atomic"; }
    private AtomicInteger index = new AtomicInteger(0);
    private AtomicLong value = new AtomicLong(0);
    public void accumulate() {
        // ¡Vaya! Depender de más de un objeto Atomic a la vez
        // no funciona. Pero sigue dándonos un indicador de
        // rendimiento:
        int i = index.getAndIncrement();
        value.getAndAdd(preLoaded[i]);
        if(++i >= SIZE)
            index.set(0);
    }
    public long read() { return value.get(); }
}

public class SynchronizationComparisons {
    static BaseLine baseLine = new BaseLine();
    static SynchronizedTest synch = new SynchronizedTest();
}

```

```

static LockTest lock = new LockTest();
static AtomicTest atomic = new AtomicTest();
static void test() {
    print("=====");
    printf("%-12s : %13d\n", "Cycles", Accumulator.cycles);
    baseLine.timedTest();
    synch.timedTest();
    lock.timedTest();
    atomic.timedTest();
    Accumulator.report(synch, baseLine);
    Accumulator.report(lock, baseLine);
    Accumulator.report(atomic, baseLine);
    Accumulator.report(synch, lock);
    Accumulator.report(synch, atomic);
    Accumulator.report(lock, atomic);
}
public static void main(String[] args) {
    int iterations = 5; // Predeterminado
    if(args.length > 0) // Cambiar opcionalmente las iteraciones
        iterations = new Integer(args[0]);
    // La primera vez rellena el conjunto compartido de hebras:
    print("Warmup");
    baseLine.timedTest();
    // Ahora la prueba inicial no incluye el coste de
    // iniciar las hebras por primera vez.
    // Generar múltiples puntos de datos:
    for(int i = 0; i < iterations; i++) {
        test();
        Accumulator.cycles *= 2;
    }
    Accumulator.exec.shutdown();
}
/* Output: (Sample)
Warmup
BaseLine      :      34237033
=====
Cycles      :      50000
BaseLine      :      20966632
synchronized :      24326555
Lock         :      53669950
Atomic        :      30552487
synchronized/BaseLine : 1.16
Lock/BaseLine : 2.56
Atomic/BaseLine : 1.46
synchronized/Lock : 0.45
synchronized/Atomic : 0.79
Lock/Atomic   : 1.76
=====
Cycles      :     100000
BaseLine      :      41512818
synchronized :      43843003
Lock         :      87430386
Atomic        :      51892350
synchronized/BaseLine : 1.06
Lock/BaseLine : 2.11
Atomic/BaseLine : 1.25
synchronized/Lock : 0.50
synchronized/Atomic : 0.84
Lock/Atomic   : 1.68
=====
```

```

Cycles      :      200000
Baseline    :      80176670
synchronized : 5455046661
Lock        :      177686829
Atomic      :      101789194
synchronized/BaseLine : 68.04
Lock/BaseLine       : 2.22
Atomic/BaseLine     : 1.27
synchronized/Lock   : 30.70
synchronized/Atomic  : 53.59
Lock/Atomic         : 1.75
=====
Cycles      :      400000
Baseline    :      160383513
synchronized : 780052493
Lock        :      362187652
Atomic      :      202030984
synchronized/BaseLine : 4.86
Lock/BaseLine       : 2.26
Atomic/BaseLine     : 1.26
synchronized/Lock   : 2.15
synchronized/Atomic  : 3.86
Lock/Atomic         : 1.79
=====
Cycles      :      800000
Baseline    :      322064955
synchronized : 336155014
Lock        :      704615531
Atomic      :      393231542
synchronized/BaseLine : 1.04
Lock/BaseLine       : 2.19
Atomic/BaseLine     : 1.22
synchronized/Lock   : 0.47
synchronized/Atomic  : 0.85
Lock/Atomic         : 1.79
=====
Cycles      :      1600000
Baseline    :      650004120
synchronized : 52235762925
Lock        :      1419602771
Atomic      :      796950171
synchronized/BaseLine : 80.36
Lock/BaseLine       : 2.18
Atomic/BaseLine     : 1.23
synchronized/Lock   : 36.80
synchronized/Atomic  : 65.54
Lock/Atomic         : 1.78
=====
Cycles      :      3200000
Baseline    :      1285664519
synchronized : 96336767661
Lock        :      2846988654
Atomic      :      1590545726
synchronized/BaseLine : 74.93
Lock/BaseLine       : 2.21
Atomic/BaseLine     : 1.24
synchronized/Lock   : 33.84
synchronized/Atomic  : 60.57
Lock/Atomic         : 1.79
*///:-
```

Este programa utiliza el patrón de diseño *basado en plantillas*²⁴ para poner todo el código común en la clase base y aislar todo el código variante en las implementaciones de `accumulate()` y `read()` en las clases derivadas. En cada una de las clases derivadas `SynchronizedTest`, `LockTest` y `AtomicTest`, podemos ver cómo `accumulate()` y `read()` expresan diferentes formas de implementar la exclusión mutua.

En este programa, las tareas se ejecutan mediante un conjunto compartido `FixedThreadPool` en un intento de realizar toda la creación de hebras al principio, impidiendo así que se pague un coste adicional durante las pruebas. Simplemente para asegurarse, la prueba inicial se duplica y el primer resultado se descarta, porque incluye el coste de la creación inicial de hebras.

Es necesaria una barrera `CyclicBarrier` porque queremos asegurarnos de que todas las tareas se hayan completado antes de declarar completa cada prueba.

Se utiliza una cláusula `static` para precargar la matriz de números aleatorios, antes de que den comienzo las pruebas. De esta forma, si existe cualquier coste asociado con la generación de los números aleatorios, este coste no se reflejará durante la prueba.

Cada vez que se invoca `accumulate()`, nos movemos a la siguiente posición de la matriz `preLoaded` (volviendo al principio cuando se alcanza el final de la matriz) y se añade otro número generado aleatoriamente a `value`. Las múltiples tareas `Modifier` y `Reader` hacen que aparezca el fenómeno de la contienda para el objeto `Accumulator`.

Observe que, en `AtomicTest`, la situación es demasiado compleja como para tratar de utilizar objetos `Atomic`, básicamente, si hay más de un objeto `Atomic` implicado, probablemente nos tengamos que dar por vencidos y utilizar mutex más convencionales (la documentación del JDK indica específicamente que la utilización de objetos `Atomic` sólo funciona cuando las actualizaciones críticas de un objeto están limitadas a una única variable). Sin embargo, hemos dejado la prueba dentro del ejemplo para poder seguir teniendo una idea de la mejora de prestaciones que se pueden tener con los objetos `Atomic`.

En `main()`, se ejecuta la prueba repetidamente y tenemos una opción de pedir que se ejecuten más de cinco repeticiones, que es el valor predeterminado. Para cada repetición, se dobla el número de ciclos de prueba, de manera que podemos ver cómo se comportan los diferentes mutex a medida que el tiempo de ejecución crece. Analizando la salida, vemos que los resultados son bastante sorprendentes. En las primeras cuatro iteraciones, la palabra clave `synchronized` parece ser más eficiente que la utilización de `Lock` o `Atomic`. Pero repentinamente, se cruza un umbral y `synchronized` parece ser bastante ineficiente, mientras que `Lock` y `Atomic` parecen mantener aproximadamente las prestaciones relativas a la prueba inicial `BaseLine`, llegando a ser así mucho más eficientes que `synchronized`.

Recuerde que este programa sólo nos proporciona una indicación de las diferencias entre las diferentes técnicas de mutex, y que la salida del ejemplo anterior sólo indica esas diferencias en mi máquina concreta y en mis circunstancias concretas. Como podrá ver si experimenta con el programa, encontrará significativos cambios de comportamiento cuando se utiliza un número diferente de hebras y cuando se ejecuta el programa durante períodos de tiempo más largos. Algunas optimizaciones de tiempo de ejecución no se invocan hasta que un programa ha estado ejecutándose durante varios minutos, y en el caso de los programas servidores, algunas horas.

Dicho esto, resulta bastante claro que la utilización de `Lock` suele ser bastante más eficiente que la de `synchronized`, y también resulta que el coste de `synchronized` varía ampliamente, mientras que el de `Lock` es relativamente estable.

¿Quiere esto decir que nunca deberíamos utilizar la palabra clave `synchronized`? Hay que considerar dos factores: en primer lugar, en `SynchronizationComparisons.java`, el cuerpo de los métodos protegidos por mutex es muy pequeño. En general, ésta es una buena práctica: proteja sólo con mutex las secciones que sean imprescindibles. Sin embargo, en la práctica, las secciones protegidas con mutex pueden tener un tamaño mayor que en el ejemplo anterior, por lo que el porcentaje de tiempo que se invertirá dentro del cuerpo de los métodos, será probablemente significativamente mayor que el coste de entrar y salir del mutex, lo que podría anular cualquier beneficio derivado de los intentos de acelerar el mutex. Por supuesto, la única forma de saberlo es (y sólo en el momento en que estemos realizando las actividades de rendimiento, no antes) probar las distintas técnicas y ver el impacto que tienen.

En segundo lugar, está claro, al leer el código contenido en este capítulo, que la palabra clave `synchronized` permite un código mucho más legible que la sintaxis `lock-try/finally-unlock` que `Lock` requiere, y esa es la razón por la que en este capítulo hemos utilizado principalmente la palabra clave `synchronized`. Como hemos dicho en otros lugares del libro, resulta

²⁴ Véase *Thinking in Patterns* en www.MindView.net.

mucho más normal leer código que escribirlo (al programar resulta más importante comunicarse con otros seres humanos que comunicarse con la computadora), por lo que la legibilidad del código es crítica. Como resultado, tiene bastante sentido comenzar utilizando la palabra clave **synchronized** y cambiar únicamente a objetos **Lock** si la optimización del rendimiento lo requiere.

Finalmente, resulta bastante atractivo poder utilizar las clases **Atomic** en nuestros programas concurrentes, pero tenga en cuenta, como hemos visto en **SynchronizationComparisons.java**, que los objetos **Atomic** sólo son útiles en casos muy simples, generalmente cuando sólo hay un objeto **Atomic** que esté siendo modificado y cuando dicho objeto sea independiente de todos los demás objetos. Resulta más seguro comenzar con otras técnicas más tradicionales de mutex y sólo tratar de cambiar a **Atomic** posteriormente, si así lo exige la optimización del rendimiento.

Contenedores libres de bloqueos

Como hemos indicado en el Capítulo 11, *Almacenamiento de objetos*, los contenedores son una herramienta fundamental en el campo de la programación y esto incluye, por supuesto, la programación concurrente. Por esta razón, contenedores primitivos como **Vector** y **Hashtable** tenían muchos métodos **synchronized**, que hacían que se incurriera en un coste inaceptable cuando no se los estaba utilizando en aplicaciones multihebra. En Java 1.2, la nueva biblioteca de contenedores no estaba sincronizada, y a la clase **Collections** se le añadieron varios métodos de decoración estáticos “sincronizados” para sincronizar los distintos tipos de contenedores. Aunque esto representaba una mejora, porque nos daba la posibilidad de usar o no la sincronización dentro de nuestros contenedores, el coste asociado sigue estando basado en los bloqueos de tipo **synchronized**. Java SE5 ha añadido nuevos contenedores específicamente para incrementar el rendimiento en aplicaciones que sean seguras con respecto a las hebras, utilizando inteligentes técnicas para eliminar el bloqueo.

La estrategia general que subyace a estos contenedores libres de bloqueo es la siguiente: las modificaciones de los contenedores pueden tener lugar al mismo tiempo que las lecturas, siempre y cuando los lectores sólo puedan ver el resultado de las modificaciones *completadas*. Cada modificación se realiza en una copia separada de la estructura de datos (o en ocasiones en una copia separada de toda la estructura) y esta copia es invisible durante el proceso de modificación. Sólo cuando la modificación se haya completado se intercambia atómicamente la estructura modificada por la estructura de datos “principal”, después de lo cual los lectores podrán ver la modificación.

En **CopyOnWriteArrayList**, una escritura hará que se cree una copia de toda la matriz subyacente. La matriz original sigue existiendo para que puedan realizarse lecturas seguras mientras se está modificando la matriz copiada. Una vez que se ha completado la modificación, una operación atómica intercambia la nueva matriz por la antigua de modo que las nuevas lecturas podrán ver la información. Una de las ventajas de **CopyOnWriteArrayList** es que no genera excepciones **ConcurrentModificationException** cuando hay múltiples iteradores recorriendo y modificando la lista, así que no hace falta escribir código especial para protegerse frente a tales excepciones, a diferencia de lo que ocurría en el pasado.

CopyOnWriteArraySet utiliza **CopyOnWriteArrayList** para conseguir un comportamiento libre de bloqueos.

ConcurrentHashMap y **ConcurrentLinkedQueue** emplean técnicas similares para permitir lecturas y escrituras concurrentes, pero sólo se modifican partes del contenedor en lugar del contenedor completo. Sin embargo, los lectores seguirán sin ver ninguna modificación antes de que éstas estén completadas. **ConcurrentHashMap** no genera la excepción **ConcurrentModificationException**.

Problemas de rendimiento

Mientras que estemos principalmente leyendo de un contenedor libre de bloqueos, éste será mucho más rápido que otro basado en **synchronized**, porque se elimina el coste de adquirir y liberar los bloqueos. Esto sigue siendo cierto en cuanto se realiza un pequeño número de escrituras en un contenedor libre de bloqueos, aunque resultaría interesante poder hacerse una idea de qué quiere decir eso de “un número pequeño”. En esta sección trataremos de hacernos una idea aproximada de las diferencias de rendimiento de estos contenedores bajo distintas condiciones.

Comenzaremos con un sistema genérico para la realización de pruebas sobre cualquier tipo de contenedor, incluyendo los mapas. El parámetro genérico **C** representa el tipo de contenedor:

```
//: concurrency/Tester.java
// Sistema para probar el rendimiento de los contenedores concurrentes.
import java.util.concurrent.*;
import net.mindview.util.*;
```

```

public abstract class Tester<C> {
    static int testReps = 10;
    static int testCycles = 1000;
    static int containerSize = 1000;
    abstract C containerInitializer();
    abstract void startReadersAndWriters();
    C testContainer;
    String testId;
    int nReaders;
    int nWriters;
    volatile long readResult = 0;
    volatile long readTime = 0;
    volatile long writeTime = 0;
    CountDownLatch endLatch;
    static ExecutorService exec =
        Executors.newCachedThreadPool();
    Integer[] writeData;
    Tester(String testId, int nReaders, int nWriters) {
        this.testId = testId + " " +
            nReaders + "r " + nWriters + "w";
        this.nReaders = nReaders;
        this.nWriters = nWriters;
        writeData = Generated.array(Integer.class,
            new RandomGenerator.Integer(), containerSize);
        for(int i = 0; i < testReps; i++) {
            runTest();
            readTime = 0;
            writeTime = 0;
        }
    }
    void runTest() {
        endLatch = new CountDownLatch(nReaders + nWriters);
        testContainer = containerInitializer();
        startReadersAndWriters();
        try {
            endLatch.await();
        } catch(InterruptedException ex) {
            System.out.println("endLatch interrupted");
        }
        System.out.printf("%-27s %14d %14d\n",
            testId, readTime, writeTime);
        if(readTime != 0 && writeTime != 0)
            System.out.printf("%-27s %14d\n",
                "readTime + writeTime =", readTime + writeTime);
    }
    abstract class TestTask implements Runnable {
        abstract void test();
        abstract void putResults();
        long duration;
        public void run() {
            long startTime = System.nanoTime();
            test();
            duration = System.nanoTime() - startTime;
            synchronized(Tester.this) {
                putResults();
            }
            endLatch.countDown();
        }
    }
}

```

```

public static void initMain(String[] args) {
    if(args.length > 0)
        testReps = new Integer(args[0]);
    if(args.length > 1)
        testCycles = new Integer(args[1]);
    if(args.length > 2)
        containerSize = new Integer(args[2]);
    System.out.printf("%-27s %14s %14s\n",
                      "Type", "Read time", "Write time");
}
} //:~

```

El método abstracto **containerInitializer()** devuelve el contenedor inicializado que hay que probar, que se almacena en el campo **testContainer**. El otro método abstracto **startReadersAndWriters()**, inicia las tareas lectora y escritora que leerán y modificarán el contenedor que estemos probando. Se ejecutan diferentes pruebas con un número diferente de lectores y escritores para ver los efectos de la contienda de bloqueo (para los contenedores basados en **synchronized**) y de las escrituras (para los contenedores libres de bloqueos).

Al constructor se le proporciona diversa información acerca de la prueba (los identificadores del argumento deben ser auto-explicativos), después de lo cual invoca el método **runTest()** un número de veces igual al valor **repetitions**. **runTest()** crea un contador **CountDownLatch** (para que la prueba pueda saber cuándo se han completado todas las tareas), inicializa el contenedor, llama a **startReadersAndWriters()** y espera hasta que todas las tareas se hayan completado.

Cada clase lectora o escritora está basada en **TestTask**, que mide la duración de su método abstracto **test()**, y luego llama a **putResults()** dentro de un bloque de tipo **synchronized** para almacenar los resultados.

Para usar este sistema (en el que podemos reconocer el patrón de diseño basado en el método de plantillas), debemos heredar de **Tester** para el tipo de contenedor concreto que queramos probar, y proporcionar las apropiadas clases **Reader** y **Writer**:

```

//: concurrency/ListComparisons.java
// {Args: 1 10 10} (Prueba rápida de verificación durante la construcción)
// Comparación aproximada del rendimiento de listas compatibles con hebras.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class ListTest extends Tester<List<Integer>> {
    ListTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    testContainer.set(index, writeData[index]);
        }
        void putResults() {
            writeTime += duration;
        }
    }
}

```

```

        }
    }

    void startReadersAndWriters() {
        for(int i = 0; i < nReaders; i++)
            exec.execute(new Reader());
        for(int i = 0; i < nWriters; i++)
            exec.execute(new Writer());
    }
}

class SynchronizedArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return Collections.synchronizedList(
            new ArrayList<Integer>(
                new CountingIntegerList(containerSize)));
    }
    SynchronizedArrayListTest(int nReaders, int nWriters) {
        super("Synched ArrayList", nReaders, nWriters);
    }
}

class CopyOnWriteArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return new CopyOnWriteArrayList<Integer>(
            new CountingIntegerList(containerSize));
    }
    CopyOnWriteArrayListTest(int nReaders, int nWriters) {
        super("CopyOnWriteArrayList", nReaders, nWriters);
    }
}

public class ListComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedArrayListTest(10, 0);
        new SynchronizedArrayListTest(9, 1);
        new SynchronizedArrayListTest(5, 5);
        new CopyOnWriteArrayListTest(10, 0);
        new CopyOnWriteArrayListTest(9, 1);
        new CopyOnWriteArrayListTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Sample)
Type           Read time      Write time
Synched ArrayList 10r 0w     232158294700          0
Synched ArrayList 9r 1w      198947618203     24918613399
readTime + writeTime =
Synched ArrayList 5r 5w      117367305062     132176613508
readTime + writeTime =
CopyOnWriteArrayList 10r 0w   758386889          0
CopyOnWriteArrayList 9r 1w   741305671     136145237
readTime + writeTime =
CopyOnWriteArrayList 5r 5w   212763075     67967464300
readTime + writeTime =
*///:~

```

En **ListTest**, las clases **Reader** y **Writer** realizan las acciones específicas para un contenedor **List<Integer>**. En **Reader.putResults()**, la duración (**duration**) se almacena, al igual que el resultado (**result**), para impedir que los campos sean optimizados por el compilador. **startReadersAndWriters()** se define a continuación para crear y ejecutar los objetos lectores y escritores específicos.

Una vez creada la lista **ListTest**, hay que heredar de ella para sustituir **containerInitializer()** con el fin de crear e inicializar los contenedores específicos de prueba.

En **main()**, podemos ver variantes de las pruebas, con diferentes números de lectores y escritores. Podemos cambiar las variables de prueba usando argumentos de la línea de comandos gracias a la llamada a **Tester.initMain(args)**.

El comportamiento predeterminado consiste en ejecutar cada prueba 10 veces, esto ayuda a estabilizar la salida, que puede variar debido a actividades propias de la máquina JVM, como la optimización y la depuración de memoria.²⁵ La salida de ejemplo que podemos ver ha sido editada para mostrar únicamente la última iteración de cada prueba. Analizando la salida, podemos ver que un contenedor **ArrayList** sincronizado tiene aproximadamente el mismo rendimiento independientemente del número de lectores y escritores: los lectores contienen con otros lectores para la obtención de dos bloqueos, al igual que hacen los escritores. Sin embargo, el contenedor **CopyOnWriteArrayList** es mucho más rápido cuando no hay escritores y sigue siendo significativamente más rápido cuando hay cinco escritores. Parece que podemos utilizar de manera bastante libre **CopyOnWriteArrayList**; el impacto de escribir en la lista no parece superar al impacto de sincronizar la lista completa. Por supuesto, es necesario probar las dos técnicas en cada aplicación específica para cerciorarse de cuál es la mejor.

De nuevo, observe que este programa no constituye una verdadera prueba de rendimiento en lo que respecta a los números absolutos, y los resultados que obtenga en su máquina serán diferentes casi con toda seguridad. El objetivo del programa es simplemente hacerse una idea del comportamiento relativo de los dos tipos de contenedor.

Puesto que **CopyOnWriteArrayList** utiliza **CopyOnWriteArrayList**, su comportamiento será similar y no es necesario que hagamos aquí una prueba separada.

Comparación de las implementaciones de mapas

Podemos utilizar el mismo sistema para obtener una idea aproximada del rendimiento de un **HashMap** de tipo **synchronized** comparado con un **ConcurrentHashMap**:

```
//: concurrency/MapComparisons.java
// {Args: 1 10 10} (Prueba rápida de verificación durante la construcción)
// Comparación aproximada del rendimiento de los mapas
// compatibles con hebras.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class MapTest
extends Tester<Map<Integer, Integer>> {
    MapTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
```

²⁵ Para ver una introducción a la realización de pruebas comparativas de rendimiento bajo la influencia de la compilación dinámica de Java, consulte www-128.ibm.com/developerworks/library/j-jtp12214.

```

        for(int index = 0; index < containerSize; index++)
            testContainer.put(index, writeData[index]);
    }
    void putResults() {
        writeTime += duration;
    }
}
void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
        exec.execute(new Reader());
    for(int i = 0; i < nWriters; i++)
        exec.execute(new Writer());
}
}

class SynchronizedHashMapTest extends MapTest {
    Map<Integer, Integer> containerInitializer() {
        return Collections.synchronizedMap(
            new HashMap<Integer, Integer>(
                MapData.map(
                    new CountingGenerator.Integer(),
                    new CountingGenerator.Integer(),
                    containerSize)));
    }
    SynchronizedHashMapTest(int nReaders, int nWriters) {
        super("Synched HashMap", nReaders, nWriters);
    }
}

class ConcurrentHashMapTest extends MapTest {
    Map<Integer, Integer> containerInitializer() {
        return new ConcurrentHashMap<Integer, Integer>(
            MapData.map(
                new CountingGenerator.Integer(),
                new CountingGenerator.Integer(), containerSize));
    }
    ConcurrentHashMapTest(int nReaders, int nWriters) {
        super("ConcurrentHashMap", nReaders, nWriters);
    }
}

public class MapComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedHashMapTest(10, 0);
        new SynchronizedHashMapTest(9, 1);
        new SynchronizedHashMapTest(5, 5);
        new ConcurrentHashMapTest(10, 0);
        new ConcurrentHashMapTest(9, 1);
        new ConcurrentHashMapTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Sample)
Type                      Read time      Write time
Synched HashMap 10r 0w      306052025049          0
Synched HashMap 9r 1w      428319156207      47697347568
readTime + writeTime =      476016503775
Synched HashMap 5r 5w      243956877760      244012003202
readTime + writeTime =      487968880962

```

```

ConcurrentHashMap 10r 0w      23352654318      0
ConcurrentHashMap 9r 1w      18833089400      1541853224
readTime + writeTime =      20374942624
ConcurrentHashMap 5r 5w      12037625732      11850489099
readTime + writeTime =      23888114831
*///:-

```

El impacto de añadir escritores a un mapa **ConcurrentHashMap** es todavía menos evidente que para **CopyOnWriteArrayList**, pero eso es porque **ConcurrentHashMap** emplea una técnica distinta que minimiza claramente el impacto de las escrituras.

Bloqueo optimista

Aunque los objetos **Atomic** realizan operaciones atómicas como **decrementAndGet()**, algunas clases **Atomic** también nos permiten realizar lo que se denomina “bloqueo optimista”. Esto quiere decir que no se utiliza realmente un mutex cuando se está realizando un cálculo, sino que, después de que el cálculo ha acabado y estamos listos para actualizar el objeto **Atomic**, se usa un método denominado **compareAndSet()**. Lo que se hace es entregar a este método el antiguo valor y el nuevo valor, y si el antiguo valor no concuerda con el que está almacenado en el objeto **Atomic**, la operación falla: esto significa que alguna otra tarea ha modificado el objeto mientras tanto. Recuerde que normalmente lo que haríamos es utilizar un mutex (**synchronized** o **Lock**) para impedir que más de una tarea pudiera modificar un objeto al mismo tiempo, pero lo que estamos haciendo aquí es ser “optimistas” dejando los datos desbloqueados y esperando que ninguna otra tarea llegue mientras tanto y nos modifique. De nuevo, todo esto se hace en aras del rendimiento: utilizando **Atomic** en lugar de **synchronized** o **Lock**, podemos aumentar las prestaciones.

¿Qué ocurre si falla la operación **compareAndSet()**? Aquí es donde el asunto se complica, y sólo podemos aplicar esta técnica a aquellos problemas que puedan adaptarse a una serie de requisitos. Si fallara **compareAndSet()**, tenemos que decidir qué hay que hacer; este aspecto es muy importante, porque si no hay nada que podamos hacer para recuperarnos de este hecho, entonces no se puede emplear esta técnica y es preciso utilizar en su lugar mutex convencionales. Quizás podamos reintentar la operación y no pase nada si ésta tiene éxito a la segunda. O quizás sea perfectamente adecuado ignorar el fallo: en algunas simulaciones, si se pierde un punto de datos, esto no tiene ninguna importancia dentro del esquema general de las cosas (por supuesto, debemos entender nuestro modelo lo suficientemente bien como para saber si esto es cierto).

Considere una simulación ficticia, compuesta de 100.000 “genes” de longitud 30; quizás pudiera tratarse del principio de alguna especie de algoritmo genético. Suponga que para cada “evolución” del algoritmo genético se realizan algunos cálculos muy costosos, por lo que decidimos emplear una máquina multiprocesador con el fin de distribuir las tareas y mejorar las prestaciones. Además, utilizamos objetos **Atomic** en lugar de objetos **Lock** para evitar el coste asociado de los mutex (naturalmente, sólo habremos llegado a esta solución después de escribir el código de la forma más simple posible, utilizando la palabra clave **synchronized**; una vez que tenemos el programa ejecutándose, descubrimos que es demasiado lento y empezamos a usar técnicas de optimización). Debido a la naturaleza de nuestro modelo, si se produce una colisión durante un cálculo, la tarea que descubra la colisión puede limitarse a ignorarla, sin actualizar el valor. He aquí el aspecto que tendría la solución:

```

//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
    static final int N_ELEMENTS = 100000;
    static final int N_GENES = 30;
    static final int N_EVOLVERS = 50;
    static final AtomicInteger[][] GRID =
        new AtomicInteger[N_ELEMENTS][N_GENES];
    static Random rand = new Random(47);
    static class Evolver implements Runnable {
        public void run() {
            while(!Thread.interrupted()) {

```

```

// Seleccionar aleatoriamente un elemento con el que trabajar;
int element = rand.nextInt(N_ELEMENTS);
for(int i = 0; i < N_GENES; i++) {
    int previous = element - 1;
    if(previous < 0) previous = N_ELEMENTS - 1;
    int next = element + 1;
    if(next >= N_ELEMENTS) next = 0;
    int oldvalue = GRID[element][i].get();
    // Realizar algún tipo de cálculo de modelado:
    int newvalue = oldvalue +
        GRID[previous][i].get() + GRID[next][i].get();
    newvalue /= 3; // Promediar los tres valores
    if(!GRID[element][i]
        .compareAndSet(oldvalue, newvalue)) {
        // Aquí una política para tratar los fallos. En este caso,
        // nos limitaremos a informar del fallo y a ignorarlo;
        // nuestro modelo se encargará de tratar con él.
        print("Old value changed from " + oldvalue);
    }
}
}
}

public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENTS; i++)
        for(int j = 0; j < N_GENES; j++)
            GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
    for(int i = 0; i < N_EVOLVERS; i++)
        exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
}
} /* (Ejecutar para ver la salida) *///:~

```

Todos los elementos se insertan en una matriz, en la suposición de que esto ayudará a incrementar la velocidad (esta suposición será comprobada en un ejercicio). Cada objeto **Evolver** promedia su valor con los valores contenidos antes y después suyo, y si se detecta un fallo cuando se intenta hacer la actualización, simplemente se imprime el valor y se continúa. Observe que no aparece ningún mutex en el programa.

Ejercicio 39: (6) ¿Son razonables las suposiciones realizadas en **FastSimulation.java**? Pruebe a cambiar la matriz, sustituyendo los valores **AtomicInteger** por valores **int** ordinarios y utilizando mutex de tipo **Lock**. Compare el rendimiento de las dos versiones del programa.

ReadWriteLock

Los bloqueos **ReadWriteLock** optimizan aquellas situaciones en las que escribimos en una estructura de datos de manera relativamente infrecuente, pero tenemos múltiples tareas leyendo a menudo de la misma. **ReadWriteLock** permite tener múltiples lectores simultáneamente siempre y cuando ninguno de ellos esté intentando realizar una escritura. Si alguien adquiere el bloqueo de escritura, no se permite ninguna lectura hasta que el bloqueo de escritura se libere.

Es bastante incierto si **ReadWriteLock** permite mejorar la velocidad del programa, ya que esto depende de cuestiones como la frecuencia con que se leen los datos, comparada con la frecuencia con que se modifican; la duración de las operaciones de lectura y escritura (el bloqueo es más complejo, por lo que con operaciones de corta duración no se percibiría ninguna ventaja); la frecuencia con que se producen contiendas entre las hebras; y el hecho de si estamos trabajando con una máquina multiprocesador o no. En último término, la única forma de saber si podemos obtener alguna ventaja con **ReadWriteLock** consiste en comprobarlo.

He aquí un ejemplo que muestra únicamente el uso más básico de los bloqueos **ReadWriteLock**:

```

//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
    private ArrayList<T> lockedList;
    // Realizar una ordenación equitativa:
    private ReentrantReadWriteLock lock =
        new ReentrantReadWriteLock(true);
    public ReaderWriterList(int size, T initialValue) {
        lockedList = new ArrayList<T>(
            Collections.nCopies(size, initialValue));
    }
    public T set(int index, T element) {
        Lock wlock = lock.writeLock();
        wlock.lock();
        try {
            return lockedList.set(index, element);
        } finally {
            wlock.unlock();
        }
    }
    public T get(int index) {
        Lock rlock = lock.readLock();
        rlock.lock();
        try {
            // Mostrar que múltiples lectores pueden
            // adquirir el bloqueo de lectura:
            if(lock.getReadLockCount() > 1)
                print(lock.getReadLockCount());
            return lockedList.get(index);
        } finally {
            rlock.unlock();
        }
    }
    public static void main(String[] args) throws Exception {
        new ReaderWriterListTest(30, 1);
    }
}

class ReaderWriterListTest {
    ExecutorService exec = Executors.newCachedThreadPool();
    private final static int SIZE = 100;
    private static Random rand = new Random(47);
    private ReaderWriterList<Integer> list =
        new ReaderWriterList<Integer>(SIZE, 0);
    private class Writer implements Runnable {
        public void run() {
            try {
                for(int i = 0; i < 20; i++) { // Prueba de 2 segundos
                    list.set(i, rand.nextInt());
                    TimeUnit.MILLISECONDS.sleep(100);
                }
            } catch(InterruptedException e) {
                // Forma aceptable de salir
            }
            print("Writer finished, shutting down");
        }
    }
}

```

```

        exec.shutdownNow();
    }
}

private class Reader implements Runnable {
    public void run() {
        try {
            while(!Thread.interrupted()) {
                for(int i = 0; i < SIZE; i++) {
                    list.get(i);
                    TimeUnit.MILLISECONDS.sleep(1);
                }
            }
        } catch(InterruptedException e) {
            // Forma aceptable de salir
        }
    }
}

public ReaderWriterListTest(int readers, int writers) {
    for(int i = 0; i < readers; i++)
        exec.execute(new Reader());
    for(int i = 0; i < writers; i++)
        exec.execute(new Writer());
}
} /* (Ejecutar para ver la salida) *///:-
```

Una lista **ReaderWriterList** puede almacenar un número fijo de objetos de cualquier tipo. Debemos indicar al constructor el tamaño deseado de la lista y un objeto inicial con el que llenar la lista. El método **set()** adquiere el bloqueo de escritura para poder invocar el método **ArrayList.set()** subyacente, mientras que el método **get()** adquiere el bloqueo de lectura para poder invocar **ArrayList.get()**. Además, **get()** comprueba si hay más de un lector que haya adquirido el bloqueo de lectura y, en caso afirmativo, muestra dicho número para demostrar que puede haber múltiples lectores que adquieran el bloqueo de lectura.

Para probar la lista **ReaderWriterList**, **ReaderWriterListTest** crea tareas tanto lectoras como escritoras para una lista **ReaderWriterList<Integer>**. Observe que hay muchas menos escrituras que lecturas.

Si examina la documentación del JDK para **ReentrantReadWriteLock**, verá que hay varios otros métodos disponibles, así como cuestiones relativas a la equidad y a las “decisiones de política”. Se trata de una herramienta bastante sofisticada, y que sólo debemos usar cuando estemos buscando formas de aumentar las prestaciones. El primer prototipo de un programa debería emplear una sincronización directa, debiéndose introducir **ReadWriteLock** sólo si es necesario.

Ejercicio 40: (6) Siguiendo el ejemplo de **ReaderWriterList.java**, cree un mapa **ReaderWriterMap** utilizando un mapa **HashMap**. Investigue su rendimiento modificando **MapComparisons.java**. ¿Cómo se compara con un contenedor **HashMap** sincronizado y con un contenedor **ConcurrentHashMap**?

Objetos activos

Después de estudiar este capítulo, habrá observado que el mecanismo de gestión de hebras en Java parece bastante complejo y difícil de usar correctamente. Además, puede parecer que es un mecanismo que reduce la productividad: aunque las tareas funcionan en paralelo, es necesario hacer un gran esfuerzo para implementar técnicas que impidan que dichas tareas interfieran entre sí.

Si alguna vez ha escrito programas en lenguaje ensamblador, la escritura de programas multihebra produce una sensación similar: todos los detalles importan, nosotros somos los responsables de todo y no existe ninguna red de seguridad en forma de comprobaciones realizadas por el compilador.

¿Podría ser que existiera un problema con el propio modelo de hebras? Después de todo, este modelo proviene, con pocas modificaciones del mundo de la programación procedimental. Quizá exista un modelo diferente de concurrencia que encaje mejor con la programación orientada a objetos.

Una técnica alternativa es la basada en los denominados *objetos activos* o *actores*.²⁶ La razón de que dichos objetos se denominen “activos” es que cada objeto mantiene su propia hebra funcional y su propia cola de mensajes, y todas las solicitudes a cada objeto se ponen en cola para procesarlas de una en una. Por tanto, con los objetos activos, *serializamos los mensajes en lugar de los métodos*, lo que significa que no necesitamos protegernos frente a aquellos problemas que surgen cuando se interrumpe una tarea en mitad de su bucle.

Cuando enviamos un mensaje a un objeto activo, dicho mensaje se transforma en una tarea que se inserta en la cola del objeto, para ejecutarla en algún instante posterior. La clase **Future** de Java SE5 resulta útil para implementar este esquema. He aquí un ejemplo simple que dispone de dos métodos que ponen en cola las llamadas a método:

```
//: concurrency/ActiveObjectDemo.java
// Sólo se pueden pasar constantes, immutables, "objetos
// desconectados", u otros objetos activos como argumentos
// a métodos asíncronos.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ActiveObjectDemo {
    private ExecutorService ex =
        Executors.newSingleThreadExecutor();
    private Random rand = new Random(47);
    // Insertar un retardo aleatorio para producir el efecto
    // de un tiempo de cálculo:
    private void pause(int factor) {
        try {
            TimeUnit.MILLISECONDS.sleep(
                100 + rand.nextInt(factor));
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public Future<Integer>
    calculateInt(final int x, final int y) {
        return ex.submit(new Callable<Integer>() {
            public Integer call() {
                print("starting " + x + " + " + y);
                pause(500);
                return x + y;
            }
        });
    }
    public Future<Float>
    calculateFloat(final float x, final float y) {
        return ex.submit(new Callable<Float>() {
            public Float call() {
                print("starting " + x + " + " + y);
                pause(2000);
                return x + y;
            }
        });
    }
    public void shutdown() { ex.shutdown(); }
    public static void main(String[] args) {
        ActiveObjectDemo d1 = new ActiveObjectDemo();
        // Evita ConcurrentModificationException:
        List<Future<?>> results =
```

²⁶ Gracias a Allen Holub por dedicarme el tiempo necesario para explicarme este tema.

```

        new CopyOnWriteArrayList<Future<?>>();
    for(float f = 0.0f; f < 1.0f; f += 0.2f)
        results.add(d1.calculateFloat(f, f));
    for(int i = 0; i < 5; i++)
        results.add(d1.calculateInt(i, i));
    print("All asynch calls made");
    while(results.size() > 0) {
        for(Future<?> f : results)
            if(f.isDone()) {
                try {
                    print(f.get());
                } catch(Exception e) {
                    throw new RuntimeException(e);
                }
                results.remove(f);
            }
    }
    d1.shutdown();
}
} /* Output: (85% match)
All asynch calls made
starting 0.0 + 0.0
starting 0.2 + 0.2
0.0
starting 0.4 + 0.4
0.4
starting 0.6 + 0.6
0.8
starting 0.8 + 0.8
1.2
starting 0 + 0
1.6
starting 1 + 1
0
starting 2 + 2
2
starting 3 + 3
4
starting 4 + 4
6
8
*///:-
```

El “ejecutor monohebra” producido por la llamada a `Executors.newSingleThreadExecutor()` mantiene su propia cola bloqueante no limitada, y tiene una única hebra que extrae tareas de la cola y las ejecuta hasta completarlas. Todo lo que necesitamos hacer en `calculateInt()` y `calculateFloat()` es enviar con `submit()` un nuevo objeto `Callable` en respuesta a una llamada a método, transformando así las llamadas a métodos en mensajes. El cuerpo del método está contenido dentro del método `call()` en la clase interna anónima. Observe que el valor de retorno de cada método de objeto activo es un objeto `Future` con un parámetro genérico que es el tipo de retorno real del método. De esta forma, la llamada a método vuelve casi inmediatamente, y el llamante utiliza el objeto `Future` para descubrir cuándo se completa la tarea y para extraer el valor de retorno real. Esto permite gestionar el caso más complejo, pero si la llamada no tiene valor de retorno el proceso se simplifica.

En `main()`, se crea una lista `List<Future<?>>` para capturar los objetos `Future` devueltos por los mensajes `calculateFloat()` y `calculateInt()` enviados al objeto activo. Esta lista se sondea utilizando `isDone()` para cada objeto `Future`, extrayéndose el objeto de la lista cuando se completa el objeto y sus resultados se procesan. Observe que el uso de `CopyOnWriteArrayList` elimina la necesidad de copiar la lista para evitar la excepción `ConcurrentModificationException`.

Para impedir un acoplamiento inadvertido entre hebras, los argumentos que se pasen a una llamada a método de objeto activo deben ser bien de sólo lectura, bien objetos activos, o bien *objetos desconectados* (esto es terminología del autor), que son objetos que no tienen ninguna conexión con ninguna otra tarea (resulta difícil imponer esta regla, porque no hay ninguna estructura para soportarla).

Con los objetos activos:

1. Cada objeto tiene su propia hebra funcional.
2. Cada objeto mantiene un control total sobre sus propios campos (lo que es algo más riguroso que las clases normales, que no sólo disponen de la *opción* de proteger sus campos).
3. Toda la comunicación entre objetos activos toma la forma de mensajes intercambiados entre esos objetos.
4. Todos los mensajes entre objetos activos se ponen en cola.

Los resultados son muy atractivos, puesto que un mensaje de un objeto activo a otro sólo puede ser bloqueado por el retardo a la hora de ponerlo en cola, y puesto que dicho retardo es siempre muy corto y no depende de ningún otro objeto, el envío de un mensaje no es bloqueable en la práctica (lo peor que puede pasar es un corto retardo). Puesto que un sistema basado en objetos activos sólo se comunica a través de mensajes, no puede suceder que dos sucesos se bloqueen mientras contienen por invocar un objeto de otro método, y esto significa que puede llegar a producirse un interbloqueo, lo cual ya es un gran paso adelante. Puesto que la hebra funcional dentro de un objeto activo sólo ejecuta un mensaje cada vez, no hay contienda por los recursos y no tenemos que preocuparnos por sincronizar los métodos. La sincronización se sigue produciendo, pero tiene lugar en el nivel de mensajes, poniendo en cola las llamadas a métodos de modo que todas se procesen de una en una.

Lamentablemente, sin un soporte directo del compilador, la técnica de codificación mostrada anteriormente resulta demasiado engorrosa. Sin embargo, hay que resaltar que se está progresando mucho en el campo de los objetos activos y actores y, lo que es más interesante, en el campo de la denominada *programación basada en agentes*. Los agentes son, en la práctica, objetos activos, pero los sistemas de agentes también soportan mecanismos de transparencia entre redes y máquinas. No me sorprendería nada que la programación basada en agentes llegara a convertirse en el sucesor de la programación orientada a objetos, porque combina los objetos con una solución de concurrencia relativamente sencilla.

Puede encontrar más información sobre los objetos activos, los actores y los agentes buscando en la Web. En particular, algunas de las ideas subyacentes a los objetos activos provienen de la Teoría de Procesos Secuenciales Comunicantes (CSP, *Communicating Sequential Processes*) de C.A.R. Hoare.

Ejercicio 41: (6) Añada una rutina de tratamiento de mensajes a **ActiveObjectDemo.java** que no tenga ningún valor de retorno e invóquela desde **main()**.

Ejercicio 42: (7) Modifique **WaxOMatic.java** para implementar objetos activos.

Proyecto:²⁷ Utilice anotaciones y Javassist para crear una anotación de clase **@Active** que transforme la clase indicada en un objeto activo.

Resumen

El objetivo de este capítulo era proporcionar las bases de la programación concurrente con hebras en Java, de modo que el lector entendiera que:

1. Pueden ejecutarse múltiples tareas independientes.
2. Hay que considerar todos los posibles problemas que pueden producirse cuando estas tareas terminan.
3. Las tareas pueden interferir entre sí por el uso de recursos compartidos. El mutex (bloqueo) es la herramienta básica utilizada para impedir estas colisiones.
4. Las tareas pueden interbloquearse si no se diseñan cuidadosamente.

²⁷ Los proyectos son sugerencias que pueden utilizarse, por ejemplo, como proyectos de fin de curso. Las soluciones a los proyectos no se incluyen en la *Guía de soluciones*.

Resulta vital aprender cuándo hay que utilizar la concurrencia y cuándo hay que evitarla. Las principales razones para emplearla son:

- Gestionar una serie de tareas que al entremezclarlas utilizarán la computadora de manera más eficiente (incluyendo la capacidad de distribuir transparentemente las tareas entre múltiples procesadores).
- Permitir una mejor organización del código.
- Aumentar la comodidad para el programador.

El ejemplo clásico de equilibrado de recursos consiste en utilizar el procesador durante la espera de E/S. La mejora en la organización de código suele manifestarse de forma especialmente clara en las simulaciones. El ejemplo clásico de mejora de la comodidad para los programadores es el de monitorizar un botón de “parada” durante las descargas de larga duración a través de la red.

Una ventaja adicional de las hebras es que proporcionan cambios de contexto de ejecución “ligeros” (del orden de 100 instrucciones) en lugar de cambios de contexto de proceso “pesados” (miles de instrucciones). Puesto que todas las hebras de un proceso dado comparten el mismo espacio de memoria, un cambio de contexto ligero sólo modifica el punto de ejecución del programa y las variables locales. Un cambio de proceso (el cambio de contexto pesado) debe intercambiar el espacio de memoria completo.

Las principales desventajas de los mecanismos multihebra son:

1. Se produce una ralentización mientras las hebras están esperando a utilizar los recursos compartidos.
2. Se requiere un gasto adicional de procesador para gestionar las hebras.
3. Las decisiones de diseño inadecuadas conducen a un incremento de la complejidad sin que se obtenga a cambio ninguna ventaja.
4. Se abre la puerta a patologías tales como la inanición de hebras, las condiciones de carrera, los interbloqueos y los bloqueos activos (múltiples hebras están realizando tareas individuales que el sistema no es capaz de terminar).
5. Pueden aparecer incoherencias entre las distintas plataformas. Por ejemplo, al desarrollar algunos de los ejemplos de este libro, descubrí condiciones de carrera que aparecían rápidamente en algunas computadoras, pero que sin embargo no aparecían en otras. Si desarrolla un programa en estas últimas, podría encontrarse con desagradables sorpresas a la hora de distribuir el programa.

Una de las mayores dificultades con las hebras se produce cuando hay más de una tarea compartiendo un recurso (como por ejemplo, la memoria de un objeto) y es necesario asegurarse de que no haya múltiples tareas intentando leer y modificar el recurso al mismo tiempo. Esto requiere un uso juicioso de los mecanismos disponibles de bloqueo (por ejemplo, la palabra clave **synchronized**). Estos mecanismos son herramientas esenciales, pero es necesario comprenderlos en profundidad, porque podrían introducirse inadvertidamente en situaciones de posible interbloqueo.

Además, la aplicación de hebras es todo un arte. Java está diseñado para permitirnos crear tantos objetos como necesitemos para resolver nuestro problema, al menos en teoría (la creación de millones de objetos para un análisis de elementos finitos en Bioingeniería, por ejemplo, podría no resultar práctica en Java sin utilizar el patrón de diseño *Peso mosca*). Sin embargo, parece que existe un límite superior al número de hebras que pueden crearse, porque llegados a un cierto número las hebras ralentizan el sistema. Este punto crítico puede ser difícil de detectar y dependerá a menudo del sistema operativo y de la máquina JVM; puede ser menor de cien o encontrarse en el rango de los miles. Puesto que muy a menudo sólo creamos un puñado de hebras para resolver un problema, este límite no suele ser una restricción, pero en problemas de diseño más generales si que esa restricción puede obligarnos a agregar un esquema de concurrencia cooperativo.

Independientemente de lo simple que pueda parecer la programación multihebra empleando una biblioteca o un lenguaje concretos, considere ese tipo de programación como una tecnología realmente avanzada. Siempre hay algo que puede estallarnos en la cara cuando menos lo esperemos. La razón de que la cena de los filósofos resulte interesante es que se puede ajustar de manera que el interbloqueo aparezca muy raramente, dándonos la impresión de que todo está bien diseñado.

En general, utilice las hebras con cuidado y con medida. Si sus programas multihebra son muy grandes y complejos, valore utilizar un lenguaje como *Erlang*. Éste es uno de los varios lenguajes *funcionales* especializados en el tema de hebras. Puede que sea posible emplear uno de esos lenguajes para aquellas partes del programa que necesiten soporte multihebra,

si es que está escribiendo muchos de estos programas y su nivel de complicación es lo suficientemente alto como para justificar esta solución.

Lecturas adicionales

Lamentablemente, hay mucha información errónea acerca de la concurrencia; esto constituye una constatación de lo confuso que puede llegar a ser este tema, y lo fácil que resulta pensar que se comprenden adecuadamente los problemas (y sé de lo que hablo porque ya he pensado muchas veces anteriormente que había comprendido perfectamente el tema de las hebras, y no tengo ninguna duda de que volveré a tener esa misma errónea sensación en el futuro). Siempre es necesario abordar con una cierta mirada crítica cada nuevo documento que encontramos acerca del tema de concurrencia, para intentar entender hasta qué punto la persona que ha escrito el documento comprende este tema adecuadamente. He aquí algunos libros que podemos considerar como fiables:

Java Concurrency in Practice, por Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes y Doug Lea (Addison-Wesley, 2006). Básicamente, es el “quién es quién” dentro del mundo de la programación multihebra en Java.

Concurrent Programming in Java, Second Edition, por Doug Lea (Addison-Wesley, 2000). Aunque este libro se basa significativamente en Java SE5, buena parte del trabajo de Doug se utilizó para crear las nuevas bibliotecas `java.util.concurrent`, así que este libro resulta esencial para comprender a fondo los problemas de concurrencia. Va más allá de la concurrencia en Java y analiza el estado actual de la técnica en lo que respecta a lenguajes y tecnologías. Aunque algunas de las partes pueden resultar un tanto obtusas, merece la pena leerlo varias veces (dejando preferiblemente unos meses entre una lectura y otra, para poder interiorizar la información). Doug es una de las pocas personas del mundo que comprende realmente la concurrencia, así que merece la pena abordar la lectura de esta obra.

The Java Language Specification, Third Edition (Capítulo 17), por Gosling, Joy, Steele y Bracha (Addison-Wesley, 2005). La especificación técnica, está disponible como documento electrónico en <http://java.sun.com/docs/books/jls>.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Interfaces gráficas de usuario

Una directriz de diseño fundamental es: “Hacer fáciles las cosas simples y hacer posibles las cosas difíciles”.¹

El objetivo de diseño original de la biblioteca de interfaces gráficas de usuario (GUI, *graphical user interface*) de Java 1.0 era permitir al programador construir una interfaz GUI que tuviera un buen aspecto en todas las plataformas. Ese objetivo no se consiguió. En su lugar, la herramienta AWT (*Abstract Windowing Toolkit, herramienta abstracta de ventanas*) de Java producía una GUI con un aspecto igualmente mediocre en todos los sistemas. Además, era bastante restrictiva; sólo se utilizaban cuatro tipos de fuentes y no se podía acceder a ninguno de los elementos GUI sofisticados que existen en el sistema operativo. El modelo de programación AWT de Java era asimismo bastante obstuso y no estaba orientado a objetos. Un estudiante de uno de mis seminarios (que había estado en Sun durante la creación de Java) nos explicó por qué: la herramienta AWT original había sido concebida, diseñada e implementada en un solo mes. Se trata, ciertamente, de una productividad maravillosa, y también de una lección instructiva sobre por qué es importante el diseño.

La situación mejoró con el modelo de sucesos AWT de Java 1.1, que adopta un enfoque orientado a objetos, mucho más claro, junto con la adición de JavaBeans, un modelo de programación con componentes que está orientado a facilitar la creación de entornos de programación visuales. Java 2 (JDK 1.2) completó la transformación desde el modelo AWT antiguo de Java 1.0 sustituyendo esencialmente casi todo con las clases JFC (*Java Foundation Classes*), cuya parte para diseño de interfaces GUI se denomina “Swing”. Se trata de un rico conjunto de componentes JavaBeans fáciles de usar y de entender que pueden arrastrarse y colocarse (además de programarse de forma manual) para crear una GUI razonable. La regla de la “tercera revisión” aplicable al sector del software (un producto no es bueno hasta la tercera revisión) parece tener vigencia también en el campo de los lenguajes de programación.

Este capítulo presenta la moderna biblioteca Swing de Java y se basa en la razonable suposición de que Swing es la biblioteca GUI que Sun tiene prevista como destino final para Java.² Si por alguna razón necesita utilizar el modelo AWT “antiguo” (porque tiene que soportar código heredado o debido a limitaciones del explorador web), puede encontrar una introducción a dicho modelo en la primera edición de este libro, descargable en www.MindView.net. Observe que algunos componentes AWT continúan estando presentes en Java y en algunas situaciones es preciso utilizarlos.

Tenga en cuenta que este capítulo no constituye un glosario completo ni de todos los componentes Swing ni de todos los métodos de las clases descritas. Lo que pretendemos es proporcionar una introducción sencilla. La biblioteca Swing es enorme y el objetivo de este capítulo sólo es familiarizar al lector con los aspectos esenciales y hacer que se sienta cómodo con los conceptos. Si necesita ir más allá de lo que aquí se cuenta, probablemente Swing le permita resolver el problema que tenga entre manos, aunque tendrá que investigar un poco acerca del tema.

Presuponemos aquí que el lector ha descargado e instalado la documentación del JDK desde <http://java.sun.com> y que tiene la posibilidad de examinar las clases **javax.swing** contenidas en la documentación para conocer los detalles concretos y los métodos de la biblioteca Swing. También puede buscar información en la Web, aunque el mejor lugar para comenzar es el propio tutorial de Swing elaborado por Sun disponible en <http://java.sun.com/docs/books/tutorial/uiswing>.

¹ Una variante de esta regla es la que se denomina “el principio de menor estupefacción” que esencialmente dice: “No sorprendas al usuario”.

² Observe que IBM ha creado una nueva biblioteca GUI de código abierto para su editor Eclipse (www.Eclipse.org), que podemos evaluar como alternativa de Swing. Presentaremos esta alternativa posteriormente en el capítulo.

Existen numerosos (y voluminosos) libros dedicados específicamente a Swing, y puede consultar alguno de ellos si necesita analizar los temas con mayor profundidad o si quiere modificar el comportamiento predeterminado de Swing.

A medida que estudiemos Swing, verá que:

1. Swing es un modelo de programación enormemente mejorado, si se compara con muchos otros lenguajes y entornos de desarrollo (no queremos decir que sea perfecto, pero sí que representa un gran paso adelante). JavaBeans (del que hablaremos hacia el final del capítulo) es el marco de trabajo para dicha biblioteca.
2. Los "constructores de interfaces GUI" (entornos de programación visual) forman una parte fundamental de cualquier entorno completo de desarrollo en Java. JavaBeans y Swing permiten al constructor de interfaces GUI escribir código automáticamente a medida que colocamos los componentes sobre formularios empleando herramientas gráficas. Esto permite acelerar el desarrollo enormemente durante la construcción de interfaces GUI, y permite una mayor dosis de experimentación y, por tanto, la posibilidad de probar más diseños y, presumiblemente, terminar adoptando mejores diseños.
3. Puesto que Swing es razonablemente sencillo, aún cuando utilicemos un constructor de interfaces GUI en lugar de realizar la codificación a mano, el código resultante debería seguir siendo comprensible. Esto resuelve uno de los mayores problemas que los constructores de interfaces GUI presentaban en el pasado, que era que generaban fácilmente código ciertamente ilegible.

Swing contiene todos los componentes que cabría esperar ver en una interfaz de usuario moderna: desde botones con imágenes a árboles y tablas. Se trata de una biblioteca enormemente grande, pero ha sido diseñada de tal forma que su complejidad resulta apropiada para la tarea que tengamos entre manos; si esa tarea es simple no es necesario escribir código, pero a medida que tratamos de hacer cosas más complejas, la complejidad del código se incrementa proporcionalmente.

Uno de los aspectos más atractivos de Swing es lo que podríamos denominar "ortogonalidad de uso". Este término quiere decir que, una vez que entendemos las ideas generales acerca de la biblioteca, usualmente podemos aplicarlas en todas partes. Debido principalmente a los convenios estándar de denominación, mientras escribimos ejemplos de este capítulo comprobé que normalmente se podía adivinar con precisión qué es lo que cada método hacia basándose en su nombre. Ciertamente, se trata de todo un hito en lo que respecta al diseño adecuado de bibliotecas de programación. Además, podemos generalmente complementar unos componentes con otros y las cosas funcionarán correctamente.

La navegación mediante el teclado es automática; podemos ejecutar una aplicación Swing utilizando el ratón y esto no requiere ningún esfuerzo de programación adicional. El soporte de desplazamiento de pantalla es muy sencillo; basta con insertar el componente en un panel `JScrollPane` en el momento de añadirlo al formulario. Características tales como las sugerencias de pantalla requieren, normalmente, una única linea de código.

En aras de la portabilidad, Swing está escrito enteramente en Java.

Swing soporta también una funcionalidad bastante radical denominada "aspecto y estilo seleccionables", que quiere decir que la apariencia de la interfaz de usuario puede modificarse dinámicamente para ajustarla a las expectativas de los usuarios que están trabajando bajo diferentes plataformas y sistemas operativos. Resulta incluso posible (aunque algo complicado) inventar nuestros propios aspecto y estilo de interfaz. En la Web pueden encontrarse algunos ejemplos de modelos básicos de interfaz.³

A pesar de todos sus aspectos positivos, Swing no es para todos los programadores, ni tampoco ha podido resolver todos los problemas de interfaz de usuario que los diseñadores querían. Al final del capítulo, examinaremos dos soluciones alternativas a Swing: el modelo SWT patrocinado por IBM, desarrollado para el editor Eclipse que está disponible de manera gratuita como biblioteca GUI autónoma de código abierto y la herramienta Flex de Macromedia para el desarrollo de interfaces Flash del lado del cliente para aplicaciones web.

Applets

Cuando apareció Java, una parte de la publicidad que recibió el lenguaje se debía al concepto de *applet*, un programa que puede distribuirse a través de Internet para ejecutarlo dentro de un explorador web (dentro de un entorno de seguridad).

³ Mi ejemplo favorito es el modelo "Servilleta" de Ken Arnold que hace que las ventanas parezcan dibujadas en una servilleta. Véase <http://napkinlaf.sourceforge.net>.

La gente pensó que el *applet* Java era el paso siguiente en la evolución de Internet, y muchos de los libros originales sobre Java presuponían que la razón por la que uno podía estar interesado en el lenguaje era para escribir *applets*.

Por diversas razones, esta revolución nunca llegó a suceder. Buena parte del problema es que la mayoría de las máquinas no incluye el software Java necesario para ejecutar *applets*, y descargar e instalar un paquete de 10 MB para poder ejecutar algo que hemos encontrado casualmente en la Web no es algo que la mayoría de los usuarios estén dispuestos a hacer. Muchos usuarios se muestran incluso atemorizados ante esa idea. Los *applets* Java como sistema de distribución de aplicaciones del lado del cliente nunca consiguieron una masa crítica, y aunque ocasionalmente podemos seguir encontrándonos con *applets*, estos componentes se han visto relegados, en general, al rincón de la historia de la informática.

Esto no significa que los *applets* no sean una tecnología interesante y valiosa. Si nos encontramos en una situación en la que podamos garantizar que los usuarios tienen instalado un entorno JRE (como por ejemplo dentro de un entorno corporativo), entonces los *applets* (o JNLP/Java Web Start, que se describe posteriormente en el capítulo) pueden ser la forma perfecta de distribuir programas cliente y de actualizar automáticamente la máquina de todos los usuarios sin el coste y el esfuerzo habitualmente necesarios para distribuir e instalar un nuevo software.

Puede encontrar una introducción a la tecnología de *applets* en los suplementos (en inglés) en línea de este libro en www.MindView.net.

Fundamentos de Swing

La mayoría de las aplicaciones Swing se construyen dentro de un marco **JFrame**, que crea la ventana en el sistema operativo que estemos empleando. El título de la ventana se puede fijar utilizando el constructor **JFrame** de la forma siguiente:

```
//: gui>HelloSwing.java
import javax.swing.*;

public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
    }
} //:-
```

setDefaultCloseOperation() le dice a **JFrame** lo que debe hacer cuando el usuario ejecute una operación de cierre. La constante **EXIT_ON_CLOSE** le dice que salga del programa. Sin esta llamada, el comportamiento predeterminado consiste en no hacer nada, por lo que la aplicación no se cerraría.

setSize() establece el tamaño de la ventana en píxeles.

Observe la última línea:

```
frame.setVisible(true);
```

Sin ella, no podríamos ver nada en la pantalla.

Podemos hacer las cosas algo más interesantes añadiendo una **JLabel** al marco **JFrame**:

```
//: gui>HelloLabel.java
import javax.swing.*;
import java.util.concurrent.*;

public class HelloLabel {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame("Hello Swing");
        JLabel label = new JLabel("A Label");
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
    }
}
```

```

        TimeUnit.SECONDS.sleep(1);
        label.setText("Hey! This is Different!");
    }
} //:-

```

Después de un segundo, el texto de **JLabel** cambia. Aunque esto resulta entretenido y seguro, teniendo en cuenta la trivialidad del programa, no resulta en realidad una buena idea que la hebra **main()** escriba directamente en los componentes GUI. Swing dispone de su propia hebra dedicada a recibir sucesos de la interfaz de usuario y a actualizar la pantalla. Si comenzamos a manipular la pantalla con otras hebras, podemos encontrarnos con las colisiones e interbloqueos descritos en el Capítulo 21, *Concurrencia*.

En lugar de ello, otras hebras (como aquí **main()**) deben enviar las tareas para que sean ejecutadas por la *hebra de despacho de sucesos* de Swing.⁴ Para hacer esto, entregamos una tarea a **SwingUtilities.invokeLater()**, que la coloca en la *cola de sucesos* para que la ejecute (en algún momento) la hebra de despacho de sucesos. Si hacemos esto con el ejemplo anterior, lo que obtendríamos es:

```

//: gui/SubmitLabelManipulationTask.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitLabelManipulationTask {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame("Hello Swing");
        final JLabel label = new JLabel("A Label");
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
        TimeUnit.SECONDS.sleep(1);
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                label.setText("Hey! This is Different!");
            }
        });
    }
} //:-

```

Ahora ya no estamos manipulando directamente la etiqueta **JLabel**. En lugar de ello, enviamos un objeto **Runnable**, y la hebra de despacho de sucesos se encarga de realizar la manipulación real, cuando le toque procesar esa tarea dentro de la cola de sucesos. Y cuando esté ejecutando este objeto **Runnable**, no estará haciendo ninguna otra cosa, así que no se puede producir ninguna colisión (*siempre y cuando* todo el código del programa se ajuste a esta técnica de enviar las solicitudes de manipulación a través de **SwingUtilities.invokeLater()**). Esto incluye el propio arranque del programa: **main()** no debe invocar los métodos Swing como lo hace en el programa anterior, sino que en su lugar debe enviar una tarea a la cola de sucesos.⁵ Por tanto, el problema escrito correctamente tendría el siguiente aspecto:

```

//: gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
    JLabel label;
    public SubmitSwingProgram() {
        super("Hello Swing");
        label = new JLabel("A Label");
        add(label);
    }
}

```

⁴ Técnicamente, la hebra de despacho de sucesos proviene de la biblioteca AWT.

⁵ Esta práctica fue añadida en Java SE5, por lo que podrá encontrar multitud de programas más antiguos que no se ajustan a ella. Eso no quiere decir que los autores de esos programas fueran ignorantes. Las prácticas sugeridas parecen estar en constante evolución.

```

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 100);
        setVisible(true);
    }
    static SubmitSwingProgram ssp;
    public static void main(String[] args) throws Exception {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() { ssp = new SubmitSwingProgram(); }
        });
        TimeUnit.SECONDS.sleep(1);
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                ssp.label.setText("Hey! This is Different!");
            }
        });
    }
} //:-~

```

Observe que la llamada a `sleep()` no se encuentra dentro del constructor. Si la ponemos ahí, el texto de la etiqueta `JLabel` original nunca aparecerá, debido a que el constructor no se completa hasta que la llamada a `sleep()` finaliza y se inserta la nueva etiqueta. Además, si la llamada a `sleep()` se encuentra dentro del constructor, o dentro de cualquier operación de la interfaz de usuario, quiere decir que estaremos suspendiendo la hebra de despacho de sucesos durante la ejecución de `sleep()`, lo que generalmente no es una buena idea.

Ejercicio 1: (1) Modifique `HelloSwing.java` para demostrar que la aplicación no se cerrará sin la llamada a `setDefaultCloseOperation()`.

Ejercicio 2: (2) Modifique `HelloLabel.java` para demostrar que la adición de etiquetas es dinámica, añadiendo un número aleatorio de etiquetas.

Un entorno de visualización

Podemos combinar las ideas anteriores y reducir la redundancia del código creando un entorno de visualización para emplearlo en los ejemplos Swing del resto del capítulo:

```

//: net/mindview/util/SwingConsole.java
// Herramienta para ejecutar ejemplos de Swing desde
// la consola, tanto applets como marcos JFrame.
package net.mindview.util;
import javax.swing.*;

public class SwingConsole {
    public static void
    run(final JFrame f, final int width, final int height) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                f.setTitle(f.getClass().getSimpleName());
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setSize(width, height);
                f.setVisible(true);
            }
        });
    }
} //:-~

```

Dado que el lector podría querer utilizar esta herramienta en otras situaciones, la hemos incluido en la biblioteca `net.mindview.util`. Para utilizarla, la aplicación debe estar dentro de un marco `JFrame` (como lo están todos los ejemplos del libro). El método estático `run()` establece el título de la ventana asignándole el nombre simple de la clase del marco `JFrame`.

Ejercicio 3: (3) Modifique `SubmitSwingProgram.java` para utilizar `SwingConsole`.

Definición de un botón

Definir un botón es bastante simple: basta con invocar el constructor **JButton** con la etiqueta que queramos incluir en el botón. Veremos posteriormente que se pueden hacer otras cosas más atractivas, como por ejemplo incluir imágenes gráficas en los botones.

Usualmente, lo que haremos será crear un campo para el botón dentro de nuestra clase, con el fin de poder hacer referencia al mismo posteriormente.

JButton es un componente (con su propia ventana de pequeño tamaño) que se volverá a pintar automáticamente como parte de cada actualización. Esto quiere decir que no hace falta pintar explícitamente los botones ni ningún otro tipo de control; basta con incluirlos en el formulario y dejar que ellos mismos se encarguen de pintarse. Normalmente, para insertar un botón en un formulario, lo haremos dentro del constructor:

```
//: gui/Button1.java
// Inclusión de botones en una aplicación Swing.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Button1 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public Button1() {
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new Button1(), 200, 100);
    }
} //:~
```

Aquí hemos añadido algo nuevo: antes de colocar ningún elemento en el marco **JFrame**, definimos un “gestor de disposición” de tipo **FlowLayout**. El gestor de disposición es la forma en que el panel decide implicitamente dónde colocar los controles en un formulario. El comportamiento normal de un marco **JFrame** consiste en utilizar el gestor **BorderLayout**, pero esa solución no funcionaría aquí porque (como veremos posteriormente en el capítulo) su comportamiento predeterminado consiste en cubrir cada control completamente con los nuevos controles que se vayan añadiendo. Por el contrario, **FlowLayout** hace que los controles fluyan equitativamente en el formulario de izquierda a derecha y de arriba a abajo.

Ejercicio 4: (1) Verifique que sin la llamada a **setLayout()** en **Button1.java**, sólo aparece un botón en el programa resultante.

Captura de un suceso

Si compilamos y ejecutamos el programa anterior, no sucede nada cuando apretamos los botones. Aquí es donde debemos intervenir y escribir algo de código para determinar qué es lo que tiene que suceder. La base fundamental de la programación dirigida por sucesos, que están muy relacionados con el comportamiento de una interfaz GUI, consiste en conectar los sucesos con el código que debe responder a esos sucesos.

La manera para llevar esto a cabo en Swing consiste en separar limpiamente la interfaz (los componentes gráficos) de la implementación (el código que queramos ejecutar cuando un suceso afecte a un cierto componente). Cada componente Swing puede informar de todos los sucesos que le ocurran, pudiendo informar de cada tipo de suceso individualmente. Por tanto, si no estamos interesados en, por ejemplo, si se ha desplazado el ratón por encima del botón, no registraremos nuestro interés en dicho suceso. Se trata de una forma muy sencilla y elegante de gestionar la programación dirigida por sucesos, y una vez que se entiendan los conceptos básicos, pueden emplearse fácilmente componentes Swing que no se hayan visto anteriormente; de hecho, este modelo puede abarcar cualquier cosa que se pueda clasificar como un componente JavaBean (de los que hablaremos más adelante en el capítulo).

Al empezar, vamos a centrarnos solamente en el suceso de interés principal para los componentes que estemos utilizando. En el caso de un botón **JButton**, este “suceso de interés” es que se pulse el botón. Para registrar nuestro interés en la pulsación de un botón, invocamos el método **addActionListener()** de **JButton**. Este método espera un argumento que es un objeto que implemente la interfaz **ActionListener**. Dicha interfaz contiene un único método denominado **actionPerformed()**. Por tanto, para asociar código con un botón **JButton**, implemente la interfaz **ActionListener** en una clase y registre un objeto de dicha clase ante **JButton** mediante **addActionListener()**. Entonces, se invocará el método **actionPerformed()** cada vez que se presione el botón (normalmente esto se denomina *retrollamada*).

Pero, ¿cuál debe ser el resultado de presionar ese botón? Nos gustaría que algo cambiara en la pantalla, así que introduciremos un nuevo componente Swing: el campo de texto **JTextField**. Este campo es un lugar en el que el usuario final puede escribir texto o, en este caso, en el que el programa puede insertar texto. Aunque hay varias formas de crear un campo **JTextField**, la más simple consiste en informar al constructor sobre cuál es la anchura que queremos que tenga ese campo. Una vez colocado el campo **JTextField** en el formulario, podemos modificar su contenido utilizando el método **setText()** (existen muchos otros métodos en **JTextField**, y puede encontrar la información correspondiente en la documentación del JDK disponible en <http://java.sun.com>). He aquí un ejemplo:

```
//: gui/Button2.java
// Respuesta a la pulsación de un botón.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    }
    private ButtonListener bl = new ButtonListener();
    public Button2() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2(), 200, 150);
    }
} //:~
```

Para crear un campo **JTextField** y colocarlo en el formulario hay que realizar los mismos pasos que para **JButton** o para cualquier otro componente Swing. La diferencia en el programa anterior radica en la creación de la clase **ButtonListener** que implementa la interfaz **ActionListener** antes mencionada. El argumento de **actionPerformed()** es de tipo **ActionEvent**, que contiene toda la información acerca del suceso y del lugar en que se ha producido. En este caso, queríamos describir el botón que ha sido presionado; **getSource()** devuelve el objeto en el que se ha originado el suceso y hemos supuesto (utilizando una proyección de tipos) que el objeto es de tipo **JButton**. El método **getText()** devuelve el texto asociado al botón, el cual se coloca en el campo **JTextField** para demostrar que verdaderamente se ha invocado el código en el momento de pulsar el botón.

En el constructor, se utiliza **addActionListener()** para registrar el objeto **ButtonListener** ante ambos botones.

A menudo resulta más cómodo programar la clase **ActionListener** como una clase interna anónima, especialmente, dado que lo normal es utilizar una única instancia de cada una de estas clases. Podemos modificar **Button2.java** para utilizar una clase interna anónima de la forma siguiente:

```
//: gui/Button2b.java
// Utilización de clases internas anónimas.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2b extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    private ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    };
    public Button2b() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2b(), 200, 150);
    }
} //:~
```

Esta técnica de emplear una clase interna anónima será la que utilizaremos con preferencia (siempre que sea posible) en los ejemplos del libro.

Ejercicio 5: (4) Cree una aplicación utilizando la clase **SwingConsole**. Incluya un campo de texto y tres botones. Cuando pulse cada botón, haga que aparezca un texto distinto en el campo de texto.

Áreas de texto

Un área **JTextArea** se parece a un campo **JTextField** salvo porque puede tener múltiples líneas y tiene una mayor funcionalidad. Un método particularmente útil es **append()**; con él podemos ir colocando fácilmente la salida de datos del componente **JTextArea**. Como podemos desplazar la pantalla hacia atrás, se trata de una mejora con respecto a los programas de línea de comandos que imprimen en la salida estándar. Por ejemplo, el siguiente programa rellena un área **JTextArea** con la salida del generador **geography** que hemos presentado en el Capítulo 17, *Análisis detallado de los contenedores*:

```
//: gui/TextArea.java
// Utilización del control JTextArea.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
```

```

private JButton
    b = new JButton("Add Data"),
    c = new JButton("Clear Data");
private JTextArea t = new JTextArea(20, 40);
private Map<String, String> m =
    new HashMap<String, String>();
public TextArea() {
    // Utilizar todos los datos:
    m.putAll(Countries.capitals());
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            for(Map.Entry me : m.entrySet())
                t.append(me.getKey() + ":" + me.getValue() + "\n");
        }
    });
    c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("");
        }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(b);
    add(c);
}
public static void main(String[] args) {
    run(new TextArea(), 475, 425);
}
} //:-

```

En el constructor, el mapa se rellena con todos los países y sus capitales. Observe que, para ambos botones, se crea el objeto **ActionListener** y se añade sin definir una variable intermedia, dado que no necesitamos volver a referirnos a dicho objeto durante el programa. El botón “Add Data” (añadir datos) formatea y añade todos los datos, y el botón “Clear Data” (borrar datos) utiliza **setText()** para eliminar todo el texto de **JTextArea**.

Al añadir el control **JTextArea** al marco **JFrame**, lo envolvemos en un panel **JScrollPane** para controlar el desplazamiento de pantalla cuando se inserta demasiado texto en el control. Es lo único que tenemos que hacer para obtener capacidades completas de desplazamiento de pantalla. Habiendo intentado averiguar cómo hacer algo equivalente en algunos otros entornos de programación GUI, he de confesar que me impresionan bastante la simplicidad y el adecuado diseño de componentes tales como **JScrollPane**.

Ejercicio 6: (7) Transforme **strings/TestRegularExpression.java** en un programa Swing interactivo que permita insertar una cadena de caracteres de entrada en un área **JTextArea** y una expresión regular en un campo **JTextField**. Los resultados deben mostrarse en un segundo control **JTextArea**.

Ejercicio 7: (5) Cree una aplicación usando **SwingConsole**, y añada todos los componentes Swing que dispongan de un método **addActionListener()** (búsquelos en la documentación del JDK disponible en <http://java.sun.com>. Consejo: busque **addActionListener()** utilizando el índice). Capture sus sucesos y muestre un mensaje apropiado para cada uno dentro de un campo de texto.

Ejercicio 8: (6) Casi todos los componentes Swing derivan de **Component**, que dispone de un método **setCursor()**. Busque este método en la documentación del JDK. Cree una aplicación y cambie el cursor por uno de los cursores definidos en la clase **Cursor**.

Control de la disposición

La forma de colocar los componentes en un formulario en Java difiere, probablemente, de cualquier otro sistema GUI que haya utilizado. En primer lugar, todo se fija en el código, no hay ningún “recurso” que controle la colocación de los componentes. En segundo lugar, la forma en la que se colocan los componentes en un formulario está controlada no por la posi-

ción absoluta, sino por un “gestor de diseño o de disposición” (*layout manager*), que decide cómo se disponen los componentes, basándose en el orden con el que los agreguemos mediante el método `add()`. El tamaño, la forma y la colocación de los componentes difieren enormemente de un gestor de diseño a otro. Además, los gestores de diseño se adaptan a las dimensiones del *applet* o de la ventana de aplicación, por lo que si se cambian las dimensiones de la ventana, el tamaño, la forma y la colocación de los componentes pueden cambiar como resultado.

JApplet, JFrame, JWindow, JDialog, JPanel, etc., pueden contener y visualizar objetos **Component**. En **Container**, existe un método denominado `setLayout()` que permite elegir un gestor de diseño diferente. En esta sección, vamos a analizar los diversos gestores de diseño, colocando botones en ellos (ya que ésa es la cosa más simple que podemos hacer). En estos ejemplos no vamos a capturar los sucesos de botón, ya que sólo queremos mostrar cómo se disponen los botones.

BorderLayout

A menos que indiquemos otra cosa, un marco **JFrame** utilizará un **BorderLayout** como su esquema de disposición predeterminado. En ausencia de instrucciones adicionales, este gestor toma todo aquello que agreguemos con `add()` y lo coloca en el centro, estirando el objeto hasta alcanzar los bordes.

BorderLayout se basa en la existencia de cuatro regiones de borde y un área central. Cuando añadimos algo a un panel que esté utilizando **BorderLayout**, podemos emplear el método sobrecargado `add()` que toma un valor constante como primer argumento. Este valor puede ser uno de los siguientes:

| | |
|----------------------------|--|
| BorderLayout.NORTH | Arriba |
| BorderLayout.SOUTH | Abajo |
| BorderLayout.EAST | Derecha |
| BorderLayout.WEST | Izquierda |
| BorderLayout.CENTER | Rellenar la parte central, hasta alcanzar a otros componentes o hasta alcanzar los bordes. |

Si no especificamos un área en la que colocar el objeto, el área predeterminada será **CENTER**.

En este ejemplo, utilizamos el gestor de disposición predeterminado, ya que **JFrame** toma como opción predeterminada **BorderLayout**:

```
//: gui/BorderLayout1.java
// Ejemplo de BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
    public BorderLayout1() {
        add(BorderLayout.NORTH, new JButton("North"));
        add(BorderLayout.SOUTH, new JButton("South"));
        add(BorderLayout.EAST, new JButton("East"));
        add(BorderLayout.WEST, new JButton("West"));
        add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String[] args) {
        run(new BorderLayout1(), 300, 250);
    }
} //:~
```

Para todas las colocaciones salvo **CENTER**, el elemento que añadamos se comprime para que quepa en la cantidad de espacio más pequeña posible a lo largo de una dimensión, mientras que se estira para ocupar el máximo espacio sobre la otra dimensión. Sin embargo, **CENTER** se estira según ambas dimensiones para ocupar toda la parte central.

FlowLayout

Este gestor hace simplemente “fluir” los componentes en el formulario de izquierda a derecha, hasta que se llena la parte superior; a continuación, se desplaza una fila hacia abajo y continúa con el flujo de los componentes.

He aquí un ejemplo donde se selecciona **FlowLayout** como gestor de disposición y luego se colocan botones en el formulario. Observará que, con **FlowLayout**, los componentes se muestran con su tamaño “natural”. Un control **JButton**, por ejemplo, tendrá el tamaño de su cadena de caracteres asociada.

```
//: gui/FlowLayout1.java
// Ejemplo de FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
    public FlowLayout1() {
        setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new FlowLayout1(), 300, 300);
    }
} //:-
```

Todos los componentes se compactarán para tener el tamaño más pequeño posible cuando se emplea un gestor **FlowLayout**, por lo que el comportamiento que se obtiene puede resultar algo sorprendente. Por ejemplo, como una etiqueta **JLabel** tendrá el tamaño de su cadena de caracteres asociada, si tratamos de justificar a la derecha su texto, la visualización no se modificará.

Observe que si cambiamos el tamaño de la ventana, el gestor de disposición hará que los componentes vuelvan a fluir en consecuencia.

GridLayout

GridLayout permite construir una tabla de componentes, y a medida que los añadimos, esos componentes se colocan de izquierda a derecha y de arriba a abajo dentro de la cuadrícula. En el constructor, especificamos el número de filas y columnas necesarias y dichas filas y columnas se disponen en iguales proporciones.

```
//: gui/GridLayout1.java
// Ejemplo de GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
    public GridLayout1() {
        setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new GridLayout1(), 300, 300);
    }
} //:-
```

En este caso, hay 21 casillas pero sólo 20 botones. La última casilla se deja vacía porque **GridLayout** no efectúa ningún proceso de “equilibrado”.

GridLayout

GridLayout proporciona una gran cantidad de control a la hora de decidir cómo disponer exactamente las regiones de la ventana y cómo éstas deben reformatearse cuando el tamaño de la ventana cambie. Sin embargo, también es el gestor de disposición más complicado, y resulta bastante difícil de comprender. Está pensado principalmente para la generación automática de código por parte de un constructor de interfaces GUI (los constructores de interfaces GUI pueden usar **GridLayout** en lugar de un sistema de posicionamiento absoluto). Si el diseño es tan complicado que piensa que puede necesitar **GridLayout**, es mejor que emplee una herramienta de construcción de interfaces GUI para generar ese diseño. Si, por alguna razón quiere conocer todos los detalles acerca de este gestor de disposición, debe consultar para empezar, alguno de los libros dedicados específicamente al tema de Swing.

Como alternativa, puede evaluar la utilización de **TableLayout**, que *no* forma parte de la biblioteca Swing pero puede descargarse de <http://java.sun.com>. Este componente está apilado sobre **GridLayout** y oculta la mayor parte de su complejidad, así que permite simplificar enormemente el diseño.

Posicionamiento absoluto

También resulta posible establecer la posición absoluta de los componentes gráficos:

1. Asigne el valor **null** al gestor de disposición del objeto **Container**: **setLayout(null)**.
2. Invoque **setBounds()** o **reshape()** (dependiendo de la versión del lenguaje) para cada componente, pasando como parámetro un rectángulo de contorno en coordenadas de píxel. Puede hacer esto en el constructor o en **paint()**, dependiendo del efecto que quiera conseguir.

Algunos constructores de interfaces GUI utilizan esta técnica de manera intensiva, pero normalmente ésta no es la mejor forma de generar código.

BoxLayout

Debido a que los programadores tenían muchas dificultades a la hora de comprender y utilizar **GridLayout**, Swing también incluye **BoxLayout**, que proporciona muchos de los beneficios de **GridLayout** pero sin la complejidad asociada. A menudo podemos utilizar este gestor de disposición cuando necesitemos colocar manualmente los componentes (de nuevo, si el diseño se hace demasiado complejo, utilice una herramienta de construcción de interfaces GUI que se encargue de generar automáticamente la disposición de componentes). **BoxLayout** permite controlar la colocación de los componentes en sentido vertical u horizontal, así como el espaciado entre los componentes. Podrá encontrar algunos ejemplos básicos de **BoxLayout** en los suplementos en línea (en inglés) del libro, disponibles en www.MindView.net.

¿Cuál es la mejor solución?

Swing es bastante potente; pueden hacerse una gran cantidad de cosas con sólo unas líneas de código. Los ejemplos mostrados son bastante simples y, de cara a aprender a utilizar la biblioteca tiene sentido escribirlos manualmente. De hecho, podemos conseguir una gran cantidad de funcionalidad combinando gestores de disposición simples. Llegados a un punto, sin embargo, deja de tener sentido escribir de forma manual los formularios GUI; la tarea se hace demasiado complicada y no es una buena forma de invertir nuestro tiempo de programación. Los diseñadores de Java y de Swing orientaron el lenguaje y las bibliotecas para soportar las herramientas de construcción de interfaces GUI, que han sido creadas con el expreso propósito de facilitar la experiencia de programación. Mientras que comprendamos el tema de los gestores de disposición y sepamos cómo tratar los sucesos (tal como se describe a continuación) no resulta particularmente importante conocer los detalles acerca de cómo disponer los componentes de forma manual; deje que la herramienta apropiada se encargue de hacer su tarea por usted (Java está diseñado, después de todo, para incrementar la productividad de los programadores).

El modelo de sucesos de Swing

En el modelo de sucesos de Swing, un componente puede iniciar (“disparar”) un suceso. Cada tipo de sucesos está representado por una clase diferente. Cuando se dispara un suceso, es recibido por uno o más “escuchas” que actúan de acuerdo

con ese suceso. Por tanto, el origen de un suceso y el lugar donde ese suceso se trata pueden estar separados. Puesto que normalmente emplearemos los componentes Swing tal como son, pero necesitaremos escribir código personalizado que se invoque cuando los componentes reciban un suceso, éste es un ejemplo excelente de la separación que existe entre interfaz e implementación.

Cada escucha de sucesos es un objeto de una clase que implementa un tipo concreto de interfaz de escucha. Por tanto, como programador, todo lo que tenemos que hacer es crear un objeto escucha y registrarlo ante el componente que está disparando el suceso. Este registro se realiza invocando el método **addXXXListener()** en el componente encargado de disparar el suceso, donde “XXX” representa el tipo de suceso para el que estamos a la escucha. Podemos determinar fácilmente los tipos de sucesos que pueden tratarse fijándonos en los nombres de los métodos “addListener” y si intentamos detectar los sucesos incorrectos, descubriremos un error en tiempo de compilación. Veremos más adelante en el capítulo que JavaBeans también utiliza los nombres de los métodos “addListener” para determinar los sucesos que un componente Bean puede tratar.

Por tanto, toda la lógica de sucesos estará contenida dentro de la clase escucha. Cuando creamos una clase escucha, la única restricción es que ésta tiene que implementar la interfaz adecuada. Podemos crear una clase escucha global, pero ésta es una situación en la que las clases internas tienden a ser muy útiles, no sólo porque proporcionan un agrupamiento lógico de las clases escucha dentro de la interfaz del usuario o de las clases de la lógica del negocio a las que están prestando servicio, sino también, porque un objeto de una clase interna mantiene una referencia a su objeto padre, lo que proporciona una forma muy adecuada para realizar invocaciones a través de las fronteras de clase y del subsistema.

Todos los ejemplos que hemos presentado hasta ahora en este capítulo han estado empleando el modelo de sucesos de Swing, pero en el resto de esta sección vamos a proporcionar el resto de los detalles que describen dicho modelo.

Tipos de sucesos y de escuchas

Todos los componentes Swing incluyen métodos **addXXXListener()** y **removeXXXListener()** de tal forma que se pueden agregar y eliminar los tipos apropiados de escuchas para cada componente. Observará que “XXX” en cada caso también representa el argumento del método, como por ejemplo en **addMyListener(MyListener m)**. La siguiente tabla presenta los sucesos, escuchas y métodos básicos asociados, junto con los componentes básicos que soportan esos sucesos concretos, proporcionando los métodos **addXXXListener()** y **removeXXXListener()**. Recuerde que el modelo de sucesos está diseñado para ser ampliable, así que puede que se encuentre con otros tipos de sucesos y de escuchas que no están incluidos en esta tabla.

| Suceso, interfaz escucha y métodos de adición y eliminación | Componentes que soportan este suceso |
|--|--|
| ActionEvent ActionListener addActionListener() removeActionListener() | JButton , JList , JTextField , JMenuItem y sus derivados, incluyendo JCheckBoxMenuItem , JMenu y JPopupMenu |
| AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener() | Jscrollbar y cualquier cosa que creemos que implemente la interfaz Adjustable |
| ComponentEvent ComponentListener addComponentListener() removeComponentListener() | * Component y sus derivados, incluyendo JButton , JCheckBox , JComboBox , Container , JPanel , JApplet , JScrollPane , Window , JDialog , JFileDialog , JFrame , JLabel , JList , JScrollbar , JTextArea y JTextField |
| ContainerEvent ContainerListener addContainerListener() removeContainerListener() | Container y sus derivados incluyendo JPanel , JApplet , JScrollPane , Window , JDialog , JFileDialog y JFrame |

| Suceso, interfaz escucha y métodos de adición y eliminación | Componentes que soportan este suceso |
|--|---|
| FocusEvent FocusListener addFocusListener() removeFocusListener() | Component y sus derivados* |
| KeyEvent KeyListener addKeyListener() removeKeyListener() | Component y sus derivados* |
| MouseEvent (tanto para los clics como para el movimiento del ratón) MouseListener addMouseListener() removeMouseListener() | Component y sus derivados* |
| MouseEvent ⁶ (tanto para los clics como para el movimiento del ratón) MouseMotionListener addMouseMotionListener() removeMouseMotionListener() | Component y sus derivados* |
| WindowEvent WindowListener addWindowListener() removeWindowListener() | Window y sus derivados, incluyendo JDialog , JFileDialog y JFrame |
| ItemEvent ItemListener addItemListener() removeItemListener() | JCheckBox , JCheckBoxMenuItem , JComboBox , JList y cualquier cosa que implemente la interfaz ItemSelectable |
| TextEvent TextListener addTextListener() removeTextListener() | Cualquier cosa derivada de JTextComponent , incluyendo JTextArea y JTextField |

Puede ver que cada tipo de componente soporta sólo ciertos tipos de sucesos. Resulta bastante tedioso buscar todos los sucesos soportados por cada componente. Una solución más sencilla consiste en modificar el programa **ShowMethods.java** del Capítulo 14, *Información de tipos*, para que muestre todos los escuchas de sucesos soportados por cualquier componente Swing que introduzcamos.

En el Capítulo 14, *Información de tipos*, se presentó el mecanismo de *reflexión* y dicha funcionalidad se empleó para buscar los métodos de una clase concreta, bien la lista completa de todos o un subconjunto de los nombres que se correspondan con una palabra clave que proporcionemos. Uno de los aspectos más atractivos del mecanismo de reflexión es que permite mostrar automáticamente *todos* los métodos de una clase, sin tener que recorrer la jerarquía de herencia y tener que examinar las clases base de cada nivel. Así, proporciona una valiosa herramienta de ahorro de tiempo de programación, puesto que los nombres de la mayoría de los métodos Java son suficientemente descriptivos, podemos buscar los nombres de método que contengan una palabra en concreto. Cuando haya encontrado lo que crea que está buscando, consulte la documentación del JDK.

He aquí la versión GUI más útil de **ShowMethods.java**, especializada para buscar los métodos “*addListener*” de los componentes Swing:

⁶ No hay ningún suceso **MouseMotionEvent**, aún cuando parezca que debería haberlo. Los clics y el movimiento de ratón están combinados en el suceso **MouseEvent**, por lo que esta segunda aparición de **MouseEvent** en la tabla no es un error.

```

//: gui>ShowAddListeners.java
// Muestra los métodos "addXXXListener" de cualquier clase Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
    private JTextField name = new JTextField(25);
    private JTextArea results = new JTextArea(40, 65);
    private static Pattern addListener =
        Pattern.compile("(add\\w+Listener\\(.*?\\))");
    private static Pattern qualifier =
        Pattern.compile("\\w+\\.");
    class NameL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String nm = name.getText().trim();
            if(nm.length() == 0) {
                results.setText("No match");
                return;
            }
            Class<?> kind;
            try {
                kind = Class.forName("javax.swing." + nm);
            } catch(ClassNotFoundException ex) {
                results.setText("No match");
                return;
            }
            Method[] methods = kind.getMethods();
            results.setText("");
            for(Method m : methods) {
                Matcher matcher =
                    addListener.matcher(m.toString());
                if(matcher.find())
                    results.append(qualifier.matcher(
                        matcher.group(1)).replaceAll("") + "\n");
            }
        }
    }
    public ShowAddListeners() {
        NameL nameListener = new NameL();
        name.addActionListener(nameListener);
        JPanel top = new JPanel();
        top.add(new JLabel("Swing class name (press Enter):"));
        top.add(name);
        add(BorderLayout.NORTH, top);
        add(new JScrollPane(results));
        // Datos iniciales y prueba:
        name.setText("JTextArea");
        nameListener.actionPerformed(
            new ActionEvent("", 0, ""));
    }
    public static void main(String[] args) {
        run(new ShowAddListeners(), 500, 400);
    }
} //:-

```

Introducimos el nombre de la clase Swing que queramos buscar en el campo **JtextField name**. Los resultados se extraen utilizando expresiones regulares y se muestran en un control **JTextArea**.

Observará que no hay botones ni otros componentes para indicar que dé comienzo la búsqueda. Eso se debe a que el campo **JTextField** está monitorizado por un escucha **ActionListener**. Cada vez que realizamos un cambio y pulsamos Intro, la lista se actualiza inmediatamente. Si el campo de texto no está vacío, se utiliza en **Class.forName()** para tratar de buscar la clase. Si el nombre es incorrecto, **Class.forName()** fallará, lo que quiere decir que generará una excepción. Esta excepción se captura y en el control **JTextArea** se muestra el mensaje “No match” (no hay correspondencia). Pero si escribimos un nombre correcto (teniendo en cuenta las mayúsculas y las minúsculas), **Class.forName()** tendrá éxito y **getMethods()** devolverá una matriz de objetos **Method**.

Aquí se emplean dos expresiones regulares. La primera, **addListener**, busca la cadena “add” seguida por cualquier combinación de caracteres alfabéticos y seguida de “Listener” y de la lista de argumentos entre paréntesis. Observe que esta expresión regular está encerrada entre paréntesis carácter de escape, lo que significa que será accesible como “grupo” de la expresión regular, cuando se detecte una correspondencia. Dentro de **NameL.ActionPerformed()** se crea un objeto **Matcher** pasando cada objeto **Method** al método **Pattern.matcher()**. Cuando se invoca **find()** para el objeto **Matcher**, devuelve **true** sólo si se detecta una correspondencia, y en este caso podemos seleccionar el primer grupo de correspondencia entre paréntesis invocando **group(1)**. Esta cadena de caracteres sigue conteniendo cualificadores, así que para quitarlos se emplea el objeto **Pattern** de tipo **qualifier**, al igual que se hacia en **ShowMethods.java**.

Al final del constructor, se coloca un valor inicial en **name** y se ejecuta el suceso de acción para realizar una prueba con datos iniciales.

Este programa es una forma cómoda de investigar las capacidades de un componente Swing. Una vez que conocemos los sucesos soportados por un componente concreto, no necesitamos buscar información adicional para ver reaccionar a dicho suceso. Simplemente:

1. Tomamos del nombre de la clase de suceso y eliminamos la palabra “Event”. Añadimos la palabra “Listener” a lo que nos haya quedado. Ésta será la interfaz escucha que habrá que implementar en la clase interna.
2. Implementamos la interfaz anterior y escribimos los métodos para los sucesos que queramos capturar. Por ejemplo, podríamos estar interesados en detectar movimientos del ratón, así que escribiríamos código para el método **mouseMoved()** de la interfaz **MouseMotionListener** (hay que implementar, por supuesto, los otros métodos, pero a menudo existe un atajo para esta tarea, como veremos más adelante).
3. Creamos un objeto de la clase escucha del paso 2. Lo registramos ante el componente de interés utilizando para ello el método producido al agregar como prefijo “add” al nombre del escucha. Por ejemplo, **addMouseMotionListener()**.

He aquí algunas de las interfaces escucha:

| Interfaz escucha / adaptador | Métodos de la interfaz |
|---|---|
| ActionListener | actionPerformed(ActionEvent) |
| AdjustmentListener | adjustmentValueChanged(AdjustmentEvent) |
| ComponentListener ComponentAdapter | componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) |
| ContainerListener ContainerAdapter | componentAdded(ContainerEvent) componentRemoved(ContainerEvent) |
| FocusListener FocusAdapter | focusGained(FocusEvent) focusLost(FocusEvent) |
| KeyListener KeyAdapter | keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent) |

| Interfaz escucha / adaptador | Métodos de la interfaz |
|---|---|
| MouseListener MouseAdapter | <code>mouseClicked(MouseEvent)</code> <code>mouseEntered(MouseEvent)</code> <code>mouseExited(MouseEvent)</code> <code>mousePressed(MouseEvent)</code> <code>mouseReleased(MouseEvent)</code> |
| MouseMotionListener MouseMotionAdapter | <code>mouseDragged(MouseEvent)</code> <code>mouseMoved(MouseEvent)</code> |
| WindowListener WindowAdapter | <code>windowOpened(WindowEvent)</code> <code>windowClosing(WindowEvent)</code> <code>windowClosed(WindowEvent)</code> <code>windowActivated(WindowEvent)</code> <code>windowDeactivated(WindowEvent)</code> <code>windowIconified(WindowEvent)</code> <code>windowDeiconified(WindowEvent)</code> |
| ItemListener | <code>itemStateChanged(ItemEvent)</code> |

No se trata de un listado exhaustivo, en parte porque el modelo de sucesos permite crear nuestros propios tipos de sucesos junto a sus escuchas asociados. Por tanto, probablemente se encuentre con bibliotecas en las que el programador habrá inventado sus propios sucesos y los conocimientos obtenidos en este capítulo le permitirán figurarse cómo hay que usar esos sucesos.

Utilización de adaptadores de escucha por simplicidad

En la tabla anterior, podemos ver que algunas interfaces escucha sólo tienen un método. Implementar estas interfaces es trivial. Sin embargo, las interfaces escucha que tienen múltiples métodos pueden ser más cómodas de emplear. Por ejemplo, si queremos capturar un clic de ratón (que no haya sido ya capturado por nosotros, por ejemplo, mediante un botón), necesitaremos escribir un método para `mouseClicked()`. Pero como **MouseListener** es una interfaz, hay que implementar todos los demás métodos, incluso aunque no hagan nada. Esto puede resultar bastante tedioso.

Para resolver el problema, algunas (pero no todas) de las interfaces escucha que disponen de más de un método se proporcionan con *adaptadores*, cuyos nombres podemos ver en la tabla anterior. Cada adaptador proporciona métodos predeterminados vacíos para cada uno de los métodos de la interfaz. Cuando heredamos del adaptador, basta con sustituir sólo los métodos que tengamos que modificar. Por ejemplo, el escucha **MouseListener** típico que utilizaremos sería similar al siguiente:

```
class MyMouseListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        // Responder al clic del ratón...
    }
}
```

El objetivo de los adaptadores es facilitar la creación de clases escucha.

Sin embargo, los adaptadores presentan una desventaja. Suponga que escribimos un adaptador **MouseAdapter** como el anterior:

```
class MyMouseListener extends MouseAdapter {
    public void MouseClicked(MouseEvent e) {
        // Responder al clic del ratón...
    }
}
```

Esto funciona, pero el programador puede volverse loco tratando de ver por qué, ya que todo se compilará y ejecutará correctamente, salvo porque el método no será invocado cuando se haga un clic de ratón. ¿Puede ver el problema? El problema se encuentra en el nombre del método: **MouseClicked()** en lugar de **mouseClicked()**. Un simple error en el uso de mayúsculas y minúsculas ha dado como resultado la adición de un método completamente nuevo. Sin embargo, este método nuevo no es el que se invoca cuando se hace clic sobre el ratón, así que no se obtienen los resultados deseados. A pesar de la inco-

modidad, una interfaz garantiza que los métodos se implementen adecuadamente, porque el compilador avisará, en caso de cometer algún error con el uso de mayúsculas y de que falte por implementar un método.

Una mejor alternativa para garantizar que estamos sustituyendo efectivamente un método consiste en emplear la anotación `@Override` predefinida en el código anterior.

Ejercicio 9: (5) Partiendo de `ShowAddListeners.java`, cree un programa con la funcionalidad completa de `typeinfo.ShowMethods.java`.

Control de múltiples sucesos

Para demostrar que estos sucesos se están realmente disparando, merece la pena crear un programa que controle el comportamiento de un botón `JButton`, y que no se limite a ver si ha sido pulsado. En este ejemplo también nos muestra cómo heredar nuestro propio botón de `JButton`.⁷

En el código que se muestra a continuación, la clase `MyButton` es una clase interna de `TrackEvent`, por lo que `MyButton` puede acceder a la ventana padre y manipular sus campos de texto, lo cual es necesario para poder escribir la información de estado en los campos de la ventana padre. Por supuesto, se trata de una solución limitada, ya que `MyButton` sólo puede emplearse en conjunción con `TrackEvent`. Este tipo de código se denomina en ocasiones “código altamente acoplado”:

```
//: gui/TrackEvent.java
// Mostrar los sucesos a medida que tienen lugar.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class TrackEvent extends JFrame {
    private HashMap<String,JTextField> h =
        new HashMap<String,JTextField>();
    private String[] event = {
        "focusGained", "focusLost", "keyPressed",
        "keyReleased", "keyTyped", "mouseClicked",
        "mouseEntered", "mouseExited", "mousePressed",
        "mouseReleased", "mouseDragged", "mouseMoved"
    };
    private MyButton
        b1 = new MyButton(Color.BLUE, "test1"),
        b2 = new MyButton(Color.RED, "test2");
    class MyButton extends JButton {
        void report(String field, String msg) {
            h.get(field).setText(msg);
        }
        FocusListener fl = new FocusListener() {
            public void focusGained(FocusEvent e) {
                report("focusGained", e paramString());
            }
            public void focusLost(FocusEvent e) {
                report("focusLost", e paramString());
            }
        };
        KeyListener kl = new KeyListener() {
            public void keyPressed(KeyEvent e) {
                report("keyPressed", e paramString());
            }
            public void keyReleased(KeyEvent e) {
                report("keyReleased", e paramString());
            }
        };
    }
}
```

⁷ En Java 1.0/1.1 *no se podía* heredar el objeto botón ni ninguna clase útil. Éste era uno de los numerosos fallos de diseño fundamentales.

```

        report("keyReleased", e paramString());
    }
    public void keyTyped(KeyEvent e) {
        report("keyTyped", e paramString());
    }
}
MouseListener ml = new MouseListener() {
    public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e paramString());
    }
    public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e paramString());
    }
    public void mouseExited(MouseEvent e) {
        report("mouseExited", e paramString());
    }
    public void mousePressed(MouseEvent e) {
        report("mousePressed", e paramString());
    }
    public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e paramString());
    }
};
MouseMotionListener mmml = new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e paramString());
    }
    public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e paramString());
    }
},
public MyButton(Color color, String label) {
    super(label);
    setBackground(color);
    addFocusListener(f1);
    addKeyListener(kl);
    addMouseListener(ml);
    addMouseMotionListener(mmml);
}
}
public TrackEvent() {
    setLayout(new GridLayout(event.length + 1, 2));
    for(String evt : event) {
        JTextField t = new JTextField();
        t.setEditable(false);
        add(new JLabel(evt, JLabel.RIGHT));
        add(t);
        h.put(evt, t);
    }
    add(b1);
    add(b2);
}
public static void main(String[] args) {
    run(new TrackEvent(), 700, 500);
}
} //:-
}

```

En el constructor de **MyButton**, el color del botón se fija mediante una llamada a **SetBackground()**. Todos los escuchas se instalan con simples llamadas a métodos.

La clase **TrackEvent** contiene un **HashMap** para almacenar las cadenas de caracteres que representan el tipo de suceso, y campos **JtextField** donde se almacena la información acerca de dicho suceso. Por supuesto, podríamos haber creado esta información estáticamente en lugar de incluirla en un contenedor **HashMap**, pero estoy convencido de que el lector estará de acuerdo en que el mapa es mucho más fácil de utilizar y modificar. En particular, si necesitamos agregar o eliminar un nuevo tipo de suceso en **TrackEvent**, simplemente basta con añadir o borrar una cadena de caracteres en la matriz **event**; todo lo demás se hace automáticamente.

Cuando se invoca **report()**, le proporcionamos al método el nombre del suceso y la cadena de parámetro del suceso. Ese método utiliza el mapa **HashMap h** de la clase externa para buscar el campo **JTextField** asociado con ese nombre de suceso y luego coloca la cadena de parámetro dentro de dicho campo.

Resulta bastante entretenido este programa, porque con él podemos ver qué sucesos se están esperando dentro del programa.

Ejercicio 10: (6) Cree una aplicación utilizando **SwingConsole**, con un control **JButton** y otro control **JTextField**. Escriba y asocie los escuchas apropiados, de modo que si botón tiene el foco, los caracteres escritos aparezcan en el campo **JTextField**.

Ejercicio 11: (4) Herede un nuevo tipo de botón de **JButton**. Cada vez que pulse este botón, debe cambiar su color a otro color elegido aleatoriamente. Consulte **ColorBoxes.java** (más adelante en el capítulo) para ver cómo generar un valor aleatorio de color.

Ejercicio 12: (4) Monitorice un nuevo tipo de suceso en **TrackEvent.java** añadiendo el nuevo código de tratamiento de sucesos. Deberá descubrir por sí mismo el tipo de suceso que quiera monitorizar.

Una selección de componentes Swing

Ahora que comprendemos los gestores de disposición y el modelo de sucesos, estamos preparados para ver cómo pueden utilizarse los componentes Swing. Esta sección es un recorrido no exhaustivo de los componentes Swing y de las características que probablemente vaya a utilizar la mayor parte de las veces. Hemos intentado que cada ejemplo sea razonablemente pequeño, para poder tomar fácilmente ese código y aplicarlo en otros programas.

Tenga en cuenta que:

1. Podemos ver fácilmente qué aspecto tendría la ejecución de estos ejemplos compilando y ejecutando el código fuente descargable correspondiente a este capítulo (www.MindView.net).
2. La documentación del JDK disponible en <http://java.sun.com> contiene todas las clases y los métodos de Swing (aquí sólo mostramos algunos).
3. Debido al convenio de denominación empleado a los sucesos Swing, resulta bastante fácil adivinar cómo escribir e instalar una nueva rutina de tratamiento para un tipo de suceso concreto. Utilice el programa de búsqueda **ShowAddListeners.java**, presentado anteriormente en el capítulo, como ayuda a la hora de investigar un componente concreto.
4. Cuando las cosas comienzan a complicarse demasiado, piense en utilizar un constructor de interfaces GUI.

Botones

Swing incluye distintos tipos de botones. Todos los botones, casillas de verificación, botones de opción e incluso elementos de menú heredan de **AbstractButton** (que en realidad, ya que están incluidos los elementos de menú, debería, probablemente, haberse llamado "AbstractSelector" o algún otro nombre igualmente genérico). Más adelante veremos la utilización de los elementos de menú, pero el siguiente ejemplo ilustra los distintos tipos de botones disponibles:

```
//: gui/Buttons.java
// Diversos botones Swing.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
```

```

import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
    private JButton jb = new JButton("JButton");
    private BasicArrowButton
        up = new BasicArrowButton(BasicArrowButton.NORTH),
        down = new BasicArrowButton(BasicArrowButton.SOUTH),
        right = new BasicArrowButton(BasicArrowButton.EAST),
        left = new BasicArrowButton(BasicArrowButton.WEST);
    public Buttons() {
        setLayout(new FlowLayout());
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        add(jp);
    }
    public static void main(String[] args) {
        run(new Buttons(), 350, 200);
    }
} //:~

```

El ejemplo comienza con el botón **BasicArrowButton** de `javax.swing.plaf.basic`, y luego continúa con los diversos tipos específicos de botones. Cuando se ejecuta el ejemplo, vemos que el botón conmutador mantiene su última posición, pulsado o no pulsado. Pero las casillas de verificación y los botones de opción se comportan de forma idéntica, permitiendo simplemente hacer clic para activarlo o desactivarlo (heredan de **JToggleButton**).

Grupos de botones

Si queremos que una serie de botones de opción se comporte de forma “or exclusiva”, debemos añadirlos a un “grupo de botones”. Sin embargo, como el siguiente ejemplo demuestra, cualquier botón de tipo **AbstractButton** puede añadirse a un grupo **ButtonGroup**.

Para evitar repetir una gran cantidad de código, este ejemplo utiliza el mecanismo de reflexión para generar los grupos de diferentes tipos de botones. Esto se ve en `makeBPanel()`, que crea un grupo de botones en un control **JPanel**. El segundo argumento de `makeBPanel()` es una matriz de objetos `String`. Para cada objeto `String`, se añade al control **JPanel** un botón de la clase representada por el primer argumento:

```

//: gui/ButtonGroups.java
// Uso del mecanismo de reflexión para crear grupos
// de diferentes tipos de botones AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
    private static String[] ids = {
        "June", "Ward", "Beaver", "Wally", "Eddie", "Lumpy"
    };
    static JPanel makeBPanel(
        Class<? extends AbstractButton> kind, String[] ids) {
        ButtonGroup bg = new ButtonGroup();

```

```

 JPanel jp = new JPanel();
 String title = kind.getName();
 title = title.substring(title.lastIndexOf('.') + 1);
 jp.setBorder(new TitledBorder(title));
 for(String id : ids) {
    AbstractButton ab = new JButton("failed");
    try {
        // Obtener el método constructor dinámico
        // que toma un argumento de tipo String:
        Constructor ctor =
            kind.getConstructor(String.class);
        // Crear un nuevo objeto:
        ab = (AbstractButton)ctor.newInstance(id);
    } catch(Exception ex) {
        System.err.println("can't create " + kind);
    }
    bg.add(ab);
    jp.add(ab);
}
return jp;
}
public ButtonGroups() {
    setLayout(new FlowLayout());
    add(makeBPanel(JButton.class, ids));
    add(makeBPanel(JToggleButton.class, ids));
    add(makeBPanel(JCheckBox.class, ids));
    add(makeBPanel(JRadioButton.class, ids));
}
public static void main(String[] args) {
    run(new ButtonGroups(), 500, 350);
}
} //:-

```

El título del borde está tomado del nombre de la clase, una vez eliminada la información de ruta. El botón **AbstractButton** se inicializa con un control **JButton** que tiene la etiqueta “Failed”, de modo que si ignoramos el mensaje de excepción, seguiremos viendo el botón en la pantalla. El método **getConstructor()** produce un objeto **Constructor** que toma la matriz de argumentos de los tipos contenidos en la lista de objetos **Class** que se pasan a **getConstructor()**. Entonces todo lo que tenemos que hacer es invocar **newInstance()**, pasándole la lista de argumentos, que en este caso es simplemente el objeto **String** de la matriz **ids**.

Para conseguir un comportamiento de tipo “or exclusivo” con los botones, creamos un grupo de botones y añadimos cada uno de los botones deseados al grupo. Al ejecutar el programa, veremos que todos los botones excepto **JButton** exhiben este comportamiento de tipo “or exclusivo”.

Iconos

Podemos utilizar un objeto **Icon** dentro de un control **JLabel** o de cualquier control que herede de **AbstractButton** (incluyendo **JButton**, **JCheckBox**, **JRadioButton** y los diferentes tipos de **JMenuItem**). Utilizar **Icon** con **JLabel** resulta bastante sencillo (veremos un ejemplo posteriormente). El siguiente ejemplo explora todas las formas adicionales en las que podemos emplear iconos con los botones y con sus descendientes.

Podemos utilizar cualquier archivo **GIF** que deseemos, pero los empleados en este ejemplo forman parte de la distribución de código del libro, disponible en www.MindView.net. Para abrir un archivo y tomar la imagen, cree simplemente un control **ImageIcon** y pasarle el nombre del archivo. A partir de ahí, podrá emplear el ícono resultante en su programa.

```

//: gui/Faces.java
// Comportamiento de los iconos en controles JButton.
import javax.swing.*;
import java.awt.*;

```

```

import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Faces extends JFrame {
    private static Icon[] faces;
    private JButton jb, jb2 = new JButton("Disable");
    private boolean mad = false;
    public Faces() {
        faces = new Icon[]{
            new ImageIcon(getClass().getResource("Face0.gif")),
            new ImageIcon(getClass().getResource("Face1.gif")),
            new ImageIcon(getClass().getResource("Face2.gif")),
            new ImageIcon(getClass().getResource("Face3.gif")),
            new ImageIcon(getClass().getResource("Face4.gif"))
        };
        jb = new JButton("JButton", faces[3]);
        setLayout(new FlowLayout());
        jb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(mad) {
                    jb.setIcon(faces[3]);
                    mad = false;
                } else {
                    jb.setIcon(faces[0]);
                    mad = true;
                }
                jb.setVerticalAlignment(JButton.TOP);
                jb.setHorizontalAlignment(JButton.LEFT);
            }
        });
        jb.setRolloverEnabled(true);
        jb.setRolloverIcon(faces[1]);
        jb.setPressedIcon(faces[2]);
        jb.setDisabledIcon(faces[4]);
        jb.setToolTipText("Yow!");
        add(jb);
        jb2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(jb.isEnabled()) {
                    jb.setEnabled(false);
                    jb2.setText("Enable");
                } else {
                    jb.setEnabled(true);
                    jb2.setText("Disable");
                }
            }
        });
        add(jb2);
    }
    public static void main(String[] args) {
        run(new Faces(), 250, 125);
    }
} //:-

```

Podemos emplear un objeto **Icon** como argumento para muchos constructores diferentes de componentes Swing, pero también podemos usar **setIcon()** para añadir o modificar un objeto **Icon**. Este ejemplo también muestra cómo un control **JButton** (o cualquier control **AbstractButton**) puede fijar los distintos tipos de iconos que aparecen cuando suceden cosas con un botón: cuando se pulsa, cuando se desactiva o cuando se pasa el ratón por encima del botón sin hacer clic. Podrá comprobar que estos efectos proporcionan a los botones un aspecto bastante animado.

Sugerencias

En el ejemplo anterior hemos añadido una “sugerencia” a un botón. Casi todas las clases que utilizaremos para crear nuestras interfaces de usuario derivan de **JComponent**, que contiene un método denominado **setToolTipText(String)**, que sirve para definir una sugerencia (*tool tip*). Por tanto, para casi todos los componentes que coloquemos en el formulario, lo único que hace falta es decir (para un objeto jc de cualquier clase derivada de **JComponent**):

```
jc.setToolTipText("Mi sugerencia");
```

Cuando el ratón permanezca sobre dicho objeto **JComponent** durante un período predeterminado de tiempo, aparecerá un pequeño recuadro al lado del puntero del ratón en el que se mostrará la sugerencia.

Campos de texto

Este ejemplo muestra lo que se puede hacer con los controles **JTextField**:

```
//: gui/TextFields.java
// Campos de texto y sucesos Java.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TextFields extends JFrame {
    private JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    private JTextField
        t1 = new JTextField(30),
        t2 = new JTextField(30),
        t3 = new JTextField(30);
    private String s = "";
    privateUpperCaseDocument ucd = newUpperCaseDocument();
    public TextFields() {
        t1.setDocument(ucd);
        ucd.addDocumentListener(new T1());
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        t1.addActionListener(new T1A());
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(t1);
        add(t2);
        add(t3);
    }
    class T1 implements DocumentListener {
        public void changedUpdate(DocumentEvent e) {}
        public void insertUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
            t3.setText("Text: " + t1.getText());
        }
        public void removeUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
        }
    }
    class T1A implements ActionListener {
        private int count = 0;
```

```

public void actionPerformed(ActionEvent e) {
    t3.setText("t1 Action Event " + count++);
}
}
class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(t1.getSelectedText() == null)
            s = t1.getText();
        else
            s = t1.getSelectedText();
        t1.setEditable(true);
    }
}
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        t1.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        t1.setEditable(false);
    }
}
public static void main(String[] args) {
    run(new TextFields(), 375, 200);
}
}

class UpperCaseDocument extends PlainDocument {
    private boolean upperCase = true;
    public void setUpperCase(boolean flag) {
        upperCase = flag;
    }
    public void
    insertString(int offset, String str, AttributeSet attSet)
    throws BadLocationException {
        if(upperCase) str = str.toUpperCase();
        super.insertString(offset, str, attSet);
    }
} //:-_

```

El control **t3** de tipo **JTextField** se incluye para disponer de un lugar en el que informar cada vez que se dispare el escucha de acción para el control **t1** de tipo **JTextField** **t1**. Podrá comprobar que el escucha de acción de un control **JTextField** sólo se dispara cuando se pulsa la tecla Intro.

El control **t1** de tipo **JTextField** tiene asociados varios escuchas. El escucha **T1** es un objeto **DocumentListener** que responde a cualquier cambio que se produzca en el “documento” (el contenido del control **JTextField**, en este caso). Ese escucha copia automáticamente todo el texto de **t1** en **t2**. Además, el documento de **t1** ha sido definido como una clase derivada de **PlainDocument**, denominada **UpperCaseDocument**, que fuerza a todos los caracteres a aparecer en mayúsculas. Esta clase detecta automáticamente los caracteres de retroceso y se encarga de realizar el borrado, ajustando la posición del cursor de texto y gestionando toda la operación de la manera que cabría esperar.

Ejercicio 13: (3) Modifique **TextFields.java** para que los caracteres en **t2** retengán su condición original de mayúsculas o minúsculas con la que fueron escritos, en lugar de transformar automáticamente todo a mayúsculas.

Bordes

JComponent contiene un método denominado **setBorder()**, que permite añadir diversos bordes interesantes a cualquier componente visible. El siguiente ejemplo ilustra varios de los diferentes bordes disponibles utilizando un método denominado **showBorder()** que crea un control **JPanel** y agrega en cada caso el borde correspondiente. Asimismo, el ejemplo utiliza el mecanismo RTTI para averiguar el nombre del borde que se esté usando (quitando la información de ruta) y luego inserta dicho nombre en una etiqueta **JLabel** situada en la zona central del panel:

```

//: gui/Borders.java
// Diferentes bordes Swing.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
               BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public Borders() {
        setLayout(new GridLayout(2,4));
        add(showBorder(new TitledBorder("Title")));
        add(showBorder(new EtchedBorder()));
        add(showBorder(new LineBorder(Color.BLUE)));
        add(showBorder(
            new MatteBorder(5,5,30,30,Color.GREEN)));
        add(showBorder(
            new BevelBorder(BevelBorder.RAISED)));
        add(showBorder(
            new SoftBevelBorder(BevelBorder.LOWERED)));
        add(showBorder(new CompoundBorder(
            new EtchedBorder(),
            new LineBorder(Color.RED))));
    }
    public static void main(String[] args) {
        run(new Borders(), 500, 300);
    }
} //:-

```

También podemos crear nuestros propios bordes y colocarlos en botones, etiquetas, etc., es decir, en cualquier cosa derivada de **JComponent**.

Un mini-editor

El control **JTextPane** proporciona un gran soporte de edición de texto, sin demasiado esfuerzo. El siguiente ejemplo hace un uso muy simple de este componente, ignorando gran parte de su funcionalidad:

```

//: gui/TextPane.java
// El control JTextPane es un pequeño editor.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextPane extends JFrame {
    private JButton b = new JButton("Add Text");
    private JTextPane tp = new JTextPane();
    private static Generator sg =
        new RandomGenerator.String(7);
    public TextPane() {

```

```

        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(int i = 1; i < 10; i++)
                    tp.setText(tp.getText() + sg.next() + "\n");
            }
        });
        add(new JScrollPane(tp));
        add(BorderLayout.SOUTH, b);
    }
    public static void main(String[] args) {
        run(new TextPane(), 475, 425);
    }
} //:-

```

El botón añade texto generado aleatoriamente. La intención del control **JTextPane** es permitir editar el texto en pantalla, por lo que verá que no existe un método **append()**. En este ejemplo (que admite que constituye un uso muy pobre de las capacidades de **JTextPane**), el texto tiene que capturarse, modificarse y volverse a colocar en el panel utilizando **setText()**.

Los elementos se añaden al control **JFrame** utilizando su gestor predeterminado **BorderLayout**. El control **JTextPane** se añade (dentro de un panel **JScrollPane**) sin especificar una región, por lo que el gestor rellena con él, el centro del panel, hasta los bordes del mismo. El botón **JButton** se añade a la región **SOUTH**, por lo que el componente encará en dicha región, en este caso, el botón se mostrará en la parte inferior de la pantalla.

Observe la funcionalidad integrada de **JTextPane**, como por ejemplo, el salto automático de línea. Este control tiene otras muchas funcionalidades que puede examinar en la documentación del JDK.

Ejercicio 14: (2) Modifique **TextPane.java** para usar un control **JTextArea** en lugar de **JTextPane**.

Casillas de verificación

Una casilla de verificación proporciona una forma de efectuar una única elección binaria. Está formada por un pequeño recuadro y una etiqueta. El recuadro almacena normalmente una pequeña "x" minúscula (o alguna otra indicación de que la casilla está activada) o está vacía, dependiendo de si el elemento ha sido seleccionado o no.

Normalmente, creamos un control **JCheckBox** utilizando un constructor que tome la etiqueta como argumento. Podemos establecer y consultar el estado y también establecer y consultar la etiqueta, si es que queremos leerla o modificarla después de haber creado el control **JCheckBox**.

Cada vez que se activa o desactiva un control **JCheckBox** se produce un suceso que se puede capturar de la misma forma que para un botón: utilizando un objeto **ActionListener**. El siguiente ejemplo emplea un control **JTextArea** para enumerar todas las casillas de verificación que hayan sido activadas:

```

//: gui/CheckBoxes.java
// Utilización de controles JCheckBox.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
    private JTextArea t = new JTextArea(6, 15);
    private JCheckBox
        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");
    public CheckBoxes() {
        cb1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                trace("1", cb1);
            }
        });
    }
}

```

```

cb2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        trace("2", cb2);
    }
});
cb3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        trace("3", cb3);
    }
));
setLayout(new FlowLayout());
add(new JScrollPane(t));
add(cb1);
add(cb2);
add(cb3);
}
private void trace(String b, JCheckBox cb) {
    if(cb.isSelected())
        t.append("Box " + b + " Set\n");
    else
        t.append("Box " + b + " Cleared\n");
}
public static void main(String[] args) {
    run(new CheckBoxes(), 200, 300);
}
} //:-

```

El método **trace()** envía el nombre de la casilla **JCheckBox** seleccionada, junto con su estado actual, al control **JTextArea** utilizando el método **append()**, de manera que podremos ver una lista acumulada de las casillas de verificación que hayan sido seleccionadas, junto con su estado.

Ejercicio 15: (5) Añada una casilla de verificación a la aplicación creada en el Ejercicio 5, capture el suceso e inserte diferentes textos en el campo de texto.

Botones de opción

El concepto de botones de opción (o de radio) en la programación de interfaces GUI proviene de las radios de automóvil anteriores a la aparición de los circuitos electrónicos que estaban equipadas con botones mecánicos. Cuando se pulsaba uno de ellos, todos los demás saltaban hacia arriba. Así, este tipo de controles nos permite imponer que se efectúe una única elección entre varias disponibles.

Para definir un grupo asociado de botones **JRadioButton**, los añadimos a un grupo **ButtonGroup** (en un formulario puede haber varios grupos **ButtonGroup**). Uno de los botones puede opcionalmente configurarse con el valor **true** (utilizando el segundo argumento del constructor). Si tratamos de asignar el valor **true** a más de un botón de opción, sólo el último de los botones configurados adoptará el valor **true**.

He aquí un ejemplo simple del uso de los botones de opción, donde se ilustra la captura de sucesos utilizando **ActionListener**:

```

//: gui/RadioButton.java
// Utilización de controles JRadioButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
public class RadioButtons extends JFrame {
    private JTextField t = new JTextField(15);
    private ButtonGroup g = new ButtonGroup();
    private JRadioButton

```

```

rb1 = new JRadioButton("one", false),
rb2 = new JRadioButton("two", false),
rb3 = new JRadioButton("three", false);
private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        t.setText("Radio button " +
            ((JRadioButton)e.getSource()).getText());
    }
};
public RadioButtons() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    setLayout(new FlowLayout());
    add(t);
    add(rb1);
    add(rb2);
    add(rb3);
}
public static void main(String[] args) {
    run(new RadioButtons(), 200, 125);
}
} //:-

```

Para mostrar el estado, se emplea un campo de texto. Este campo se define como no editable, ya que sólo se utiliza para mostrar datos, no para aceptarlos como entrada. Así, se trata de una alternativa a **JLabel**.

Cuadros combinados (listas desplegables)

Al igual que los grupos de botones de opción, una lista desplegable es una forma de obligar al usuario a seleccionar un elemento de entre un grupo de posibilidades. Sin embargo, es una forma más compacta de conseguir este efecto y una forma más fácil de cambiar los elementos de la lista sin sorprender al usuario (también podríamos cambiar los botones de opción dinámicamente, pero el efecto visual resulta bastante molesto).

De manera predeterminada, el recuadro **JComboBox** no se parece al cuadro combinado de Windows, que permite seleccionar un elemento de una lista o escribir nuestra propia selección. Para conseguir este comportamiento debemos invocar el método **setEditable()**. Con un recuadro **JComboBox**, se selecciona un único elemento de entre una lista. En el siguiente ejemplo, el recuadro **JComboBox** comienza con un cierto número de entradas y luego se añaden nuevas entradas al recuadro cuando se pulsa un botón.

```

//: gui/ComboBoxes.java
// Utilización de listas desplegables.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class ComboBoxes extends JFrame {
    private String[] description = {
        "Ebullient", "Obtuse", "Recalcitrant", "Brilliant",
        "Sommescent", "Timorous", "Florid", "Putrescent"
    };
    private JTextField t = new JTextField(15);
    private JComboBox c = new JComboBox();
    private JButton b = new JButton("Add items");
    private int count = 0;
    public ComboBoxes() {
        for(int i = 0; i < 4; i++)

```

```

        c.addItem(description[count++]);
        t.setEditable(false);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(count < description.length)
                    c.addItem(description[count++]);
            }
        });
        c.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("index: " + c.getSelectedIndex() + "    " +
                          ((JComboBox)e.getSource()).getSelectedItem());
            }
        });
        setLayout(new FlowLayout());
        add(t);
        add(c);
        add(b);
    }
    public static void main(String[] args) {
        run(new Comboboxes(), 200, 175);
    }
} //:~
}

```

El control **JTextField** muestra el “índice seleccionado”, que es el número de secuencia del elemento actualmente seleccionado, junto con el texto que ese elemento tiene en el recuadro combinado.

Cuadros de lista

Los cuadros de lista difieren significativamente de los cuadros **JComboBox**, y no sólo por su distinta apariencia visual. Mientras que un recuadro **JComboBox** se despliega cuando los activamos, un control **JList** ocupa un número fijo de líneas dentro de una pantalla durante todo el tiempo, y no se modifica. Si queremos ver los elementos de una lista, basta con invocar **getSelectedValues()**, que produce una matriz de objetos **String** de los elementos que hayan sido seleccionados.

Un control **JList** permite efectuar selecciones múltiples; si hacemos control-clic sobre más de un elemento (manteniendo pulsada la tecla Control mientras realizamos clics adicionales con el ratón), el elemento original continuará estando resaltado y podemos seleccionar tantos elementos como queramos. Si seleccionamos un elemento, y luego pulsamos Mayús-clic sobre otro elemento, todos los elementos pertenecientes al rango comprendido entre esos dos elementos quedarán seleccionados. Para eliminar un elemento de un grupo, podemos hacer Control-clic sobre él.

```

//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private DefaultListModel lItems = new DefaultListModel();
    private JList lst = new JList(lItems);
    private JTextArea t =
        new JTextArea(flavors.length, 20);
    private JButton b = new JButton("Add Item");

```

```

private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(count < flavors.length) {
            lItems.addElement(flavors[count++]);
        } else {
            // Desactivar, ya que no quedan más
            // sabores por añadir a la lista.
            b.setEnabled(false);
        }
    }
};

private ListSelectionListener ll =
    new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        if(e.getValueIsAdjusting()) return;
        t.setText("");
        for(Object item : lst.getSelectedValues())
            t.append(item + "\n");
    }
};

private int count = 0;
public List() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Crear bordes para los componentes:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Añadir los primeros cuatro elementos a la lista
    for(int i = 0; i < 4; i++)
        lItems.addElement(flavors[count++]);
    add(t);
    add(lst);
    add(b);
    // Registrar escuchas de sucesos
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
}
public static void main(String[] args) {
    run(new List(), 250, 375);
}
} //:-

```

Como podrá observar también se han añadido bordes a las listas.

Si simplemente queremos incluir una matriz de objetos **String** en un control **JList**, existe una solución mucho más sencilla: basta con pasar la matriz al constructor **JList**, y éste construye la lista automáticamente. La única razón para usar el “modelo de lista” en el ejemplo precedente es para poder manipular la lista durante la ejecución del programa.

JList no proporciona un soporte directo automático para el desplazamiento de pantalla. Por supuesto, nos bastaría con insertar el control **JList** en un panel **JScrollPane**, que se encargaría de gestionar todos los detalles por nosotros.

Ejercicio 16: (5) Simplifique **List.java** pasando la matriz al constructor y eliminando la posibilidad de adición dinámica de elementos a la lista.

Tableros con fichas

El control **JTabbedPane** permite crear un “cuadro de diálogo con fichas”, que tiene una serie de pestañas de archivador a lo largo de uno de sus bordes. Cuando se pulsa en una de las fichas, aparece un cuadro de diálogo diferente.

```

//: gui/TabbedPane.java
// Ejemplo de panel con fichas.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;
public class TabbedPane extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTabbedPane tabs = new JTabbedPane();
    private JTextField txt = new JTextField(20);
    public TabbedPane() {
        int i = 0;
        for(String flavor : flavors) {
            tabs.addTab(flavors[i],
                new JButton("Tabbed pane " + i++));
            tabs.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                    txt.setText("Tab selected: " +
                        tabs.getSelectedIndex());
                }
            });
        }
        add(BorderLayout.SOUTH, txt);
        add(tabs);
    }
    public static void main(String[] args) {
        run(new TabbedPane(), 400, 250);
    }
} //:-

```

Al ejecutar el programa, vemos que el control **JTabbedPane** apila automáticamente las fichas si hay demasiadas como para que todas quepan en una misma fila. Podemos comprobar que esta funcionalidad cambia el tamaño de la ventana al ejecutar el programa desde la línea de comandos de la consola.

Recuadros de mensaje

Los entornos de ventanas suelen contener un conjunto estándar de recuadros de mensaje que permite presentar rápidamente información al usuario o capturar información del usuario. En Swing, estos recuadros de mensajes están contenidos en el control **JOptionPane**. Disponemos de muchas posibilidades distintas (algunas bastante sofisticadas), y las que usaremos más comúnmente, con toda probabilidad, son el cuadro de diálogo de mensajes y el cuadro de diálogo de confirmación, que se invocan con los métodos estáticos **JOptionPane.showMessageDialog()** y **JOptionPane.showConfirmDialog()**. El siguiente ejemplo muestra un subconjunto de los recuadros de mensajes disponibles con **JOptionPane**:

```

//: gui/MessageBoxes.java
// Ejemplo de JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
    private JButton[] b = {
        new JButton("Alert"), new JButton("Yes/No"),
        new JButton("Color"), new JButton("Input"),
        new JButton("3 Vals")
    }
}

```

```

};

private JTextField txt = new JTextField(15);
private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String id = ((JButton)e.getSource()).getText();
        if(id.equals("Alert"))
            JOptionPane.showMessageDialog(null,
                "There's a bug on you!", "Hey!",
                JOptionPane.ERROR_MESSAGE);
        else if(id.equals("Yes/No"))
            JOptionPane.showConfirmDialog(null,
                "or no", "choose yes",
                JOptionPane.YES_NO_OPTION);
        else if(id.equals("Color")) {
            Object[] options = { "Red", "Green" };
            int sel = JOptionPane.showOptionDialog(
                null, "Choose a Color!", "Warning",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.WARNING_MESSAGE, null,
                options, options[0]);
            if(sel != JOptionPane.CLOSED_OPTION)
                txt.setText("Color Selected: " + options[sel]);
        } else if(id.equals("Input")) {
            String val = JOptionPane.showInputDialog(
                "How many fingers do you see?");
            txt.setText(val);
        } else if(id.equals("3 Vals")) {
            Object[] selections = {"First", "Second", "Third"};
            Object val = JOptionPane.showInputDialog(
                null, "Choose one", "Input",
                JOptionPane.INFORMATION_MESSAGE,
                null, selections, selections[0]);
            if(val != null)
                txt.setText(val.toString());
        }
    }
};

public MessageBoxes() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        add(b[i]);
    }
    add(txt);
}

public static void main(String[] args) {
    run(new MessageBoxes(), 200, 200);
}
} //:-
}

```

Para escribir un único escucha **ActionListener**, hemos adoptado en el ejemplo la solución, algo arriesgada, de comprobar las etiquetas de tipo **String** de los botones. El problema con este enfoque es que resulta fácil obtener la etiqueta de manera errónea, normalmente en lo que respecta al uso de mayúsculas y minúsculas, y estos errores pueden ser difíciles de detectar.

Observe que **showOptionDialog()** y **showInputDialog()** proporcionan objetos que contienen el valor introducido por el usuario.

Ejercicio 17: (5) Cree una aplicación utilizando **SwingConsole**. En la documentación del JDK disponible en <http://java.sun.com>, localice el control **JPasswordField** y añádalo al programa. Si el usuario escribe la contraseña correcta, utilice **JOptionPane** para proporcionar un mensaje de confirmación al usuario.

Ejercicio 18: (4) Modifique **MessageBoxes.java** para que tenga un escucha **ActionListener** para cada botón (en lugar de buscar la correspondencia por el texto del botón).

Menús

Cada componente capaz de almacenar un menú, incluyendo **JApplet**, **JFrame**, **JDialog** y sus descendientes, dispone de un método **setJMenuBar()** que acepta un objeto **JMenuBar** que representa una barra de menú (sólo puede haber un componente **JMenuBar** en cada componente concreto). Lo que hacemos es añadir menús **JMenu** a la barra de menús **JMenuBar** y elementos de menú **JMenuItem** a los menús **JMenu**. Cada objeto **JMenuItem** puede tener asociado un escucha **ActionListener** que se seleccione cuando se seleccione ese elemento de menú.

Con Java y Swing es necesario agrupar manualmente todos los menús en el código fuente. He aquí un ejemplo simple de menú:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
    private JTextField t = new JTextField(15);
    private ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText(((JMenuItem)e.getSource()).getText());
        }
    };
    private JMenu[] menus = {
        new JMenu("Winken"), new JMenu("Blinken"),
        new JMenu("Nod")
    };
    private JMenuItem[] items = {
        new JMenuItem("Fee"), new JMenuItem("Fi"),
        new JMenuItem("Fo"), new JMenuItem("Zip"),
        new JMenuItem("Zap"), new JMenuItem("Zot"),
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
        new JMenuItem("Free")
    };
    public SimpleMenus() {
        for(int i = 0; i < items.length; i++) {
            items[i].addActionListener(al);
            menus[i % 3].add(items[i]);
        }
        JMenuBar mb = new JMenuBar();
        for(JMenu jm : menus)
            mb.add(jm);
        setJMenuBar(mb);
        setLayout(new FlowLayout());
        add(t);
    }
    public static void main(String[] args) {
        run(new SimpleMenus(), 200, 150);
    }
} //:-
```

El uso del operador módulo en “*i%3*” distribuye los elementos de menú entre los tres controles **JMenu**. Cada objeto **JMenuItem** puede tener asociado un escucha **ActionListener**; aquí, se utiliza el mismo escucha **ActionListener** en todas partes, pero lo normal es que necesitemos un escucha distinto para cada elemento **JMenuItem**.

JMenuItem hereda de **AbstractButton**, así que tiene algunos comportamientos similares a los de los botones. Por sí mismo, proporciona un elemento que puede situarse en un menú desplegable. Existen también tres tipos heredados de **JMenuItem**: **JMenu**, para almacenar otros elementos **JMenuItem** (de modo que pueda haber menús en cascada); **JCheckBoxMenuItem**, que produce una casilla de verificación para indicar si dicho elemento de menú está seleccionado y **JRadioButtonMenuItem**, que contiene un botón de opción.

Como ejemplo más sofisticado, he aquí de nuevo el ejemplo de los sabores de helados para crear menús. Este ejemplo también ilustra la definición de menús en cascada, de atajos de teclado, de elementos **JCheckBoxMenuItem**, y también muestra la forma de cambiar dinámicamente los menús.

```
//: gui/Menus.java
// Submenús, elementos de menú con casillas de verificación, menús
// intercambiables (atajos de teclado) y comandos de acción.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Menus extends JFrame {
    private String[] flavors = {
        "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
        "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie"
    };
    private JTextField t = new JTextField("No flavor", 30);
    private JMenuBar mb1 = new JMenuBar();
    private JMenu
        f = new JMenu("File"),
        m = new JMenu("Flavors"),
        s = new JMenu("Safety");
    // Solución alternativa:
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };
    private JMenuItem[] file = { new JMenuItem("Open") };
    // Una segunda barra de menú para efectuar un intercambio:
    private JMenuBar mb2 = new JMenuBar();
    private JMenu fooBar = new JMenu("fooBar");
    private JMenuItem[] other = {
        // Adición de un atajo de teclado es muy simple,
        // pero sólo pueden incluirse los atajos en los constructores
        // en el caso de los elementos JMenuItem:
        new JMenuItem("Foo", KeyEvent.VK_F),
        new JMenuItem("Bar", KeyEvent.VK_A),
        // Sin atajo de teclado:
        new JMenuItem("Baz"),
    };
    private JButton b = new JButton("Swap Menus");
    class BL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuBar m = getJMenuBar();
            setJMenuBar(m == mb1 ? mb2 : mb1);
            validate(); // Refrescar el marco
        }
    }
    class ML implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuItem target = (JMenuItem)e.getSource();
            if (target instanceof JCheckBoxMenuItem)
                target.setSelected(!target.isSelected());
            else if (target instanceof JMenuItem)
                target.setAccelerator(new KeyStroke(KeyEvent.VK_S, Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
        }
    }
}
```

```

        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(String flavor : flavors)
                if(s.equals(flavor))
                    chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening " + s + ". Mmm, mmm!");
        }
    }
}

class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
    }
}
// Alternativamente, podemos crear una clase diferente
// para cada elemento MenuItem. Entonces, no tenemos
// por qué tratar de figurarnos cuál es:
class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}
class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
    }
}
class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Baz selected");
    }
}
class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
            (JCheckBoxMenuItem)e.getSource();
        String actionCommand = target.getActionCommand();
        if(actionCommand.equals("Guard"))
            t.setText("Guard the Ice Cream! " +
                     "Guarding is " + target.getState());
        else if(actionCommand.equals("Hide"))
            t.setText("Hide the Ice Cream! " +
                     "Is it hidden? " + target.getState());
    }
}
public Menus() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[1].setMnemonic(KeyEvent.VK_H);
}

```

```

safety[1].addItemListener(cmil);
other[0].addActionListener(new FooL());
other[1].addActionListener(new BarL());
other[2].addActionListener(new BazL());
FL f1 = new FL();
int n = 0;
for(String flavor : flavors) {
    JMenuItem mi = new JMenuItem(flavor);
    mi.addActionListener(f1);
    m.add(mi);
    // Añadir separadores a intervalos:
    if((n++ + 1) % 3 == 0)
        m.addSeparator();
}
for(JCheckBoxMenuItem sfty : safety)
    s.add(sfty);
s.setMnemonic(KeyEvent.VK_A);
f.add(s);
f.setMnemonic(KeyEvent.VK_F);
for(int i = 0; i < file.length; i++) {
    file[i].addActionListener(f1);
    f.add(file[i]);
}
mbl.add(f);
mbl.add(m);
setJMenuBar(mbl);
t.setEditable(false);
add(t, BorderLayout.CENTER);
// Preparar un sistema para intercambiar menús:
b.addActionListener(new BL());
b.setMnemonic(KeyEvent.VK_S);
add(b, BorderLayout.NORTH);
for(JMenuItem oth : other)
    fooBar.add(oth);
fooBar.setMnemonic(KeyEvent.VK_B);
mb2.add(fooBar);
}
public static void main(String[] args) {
    run(new Menus(), 300, 200);
}
} //:~

```

En este programa, hemos colocado los elementos de menú en matrices y luego hemos recorrido cada matriz invocando **add()** para cada elemento **JMenuItem**. Esto hace que la adición o eliminación de un elemento de menú sea algo menos tedioso.

Este programa crea dos controles **JMenuBar** para demostrar que las barras de menú pueden intercambiarse de manera activa mientras el programa se está ejecutando. Podemos ver cómo los patrones **JMenuBar** están compuestos de elementos **JMenu**, y que cada control **JMenu** está formado por elementos **JMenuItem**, **JCheckBoxMenuItem**, o incluso por otros elementos **JMenu** (lo que permite obtener submenús). Cuando se define un control **JMenuBar**, se puede instalar en el programa actual mediante el método **setJMenuBar()**. Observe que, cuando se pulsa el botón se comprueba qué menú está instalado actualmente invocando **getJMenuBar()** y luego se sustituye por el otro menú.

A la hora de comprobar la correspondencia con la cadena de caracteres “Open”, observe que la ortografía y la utilización de mayúsculas y minúsculas son críticas, pero que Java no proporciona ninguna señal de error si no se detecta ninguna correspondencia con “Open”. Este tipo de comparación entre cadenas de caracteres constituye una fuente de errores de programación.

La activación y desactivación de los elementos de menú se gestiona automáticamente. El código que gestiona los elementos **JCheckBoxMenuItem** muestra dos formas distintas de determinar qué es lo que se ha activado: comparación de cade-

na de caracteres (la técnica menos segura, aunque también se suele utilizar) y la detección del objeto de destino del suceso. Tal como se muestra, podemos utilizar el método `getState()` para determinar el estado. También podemos cambiar el estado de un objeto `JCheckBoxMenuItem` con `setState()`.

Los sucesos relacionados con los menús son algo incoherentes y pueden conducir a confusión: los elementos `JMenuItem` utilizan escuchas `ActionListener`, mientras que los elementos `JCheckBoxMenuItem` utilizan escuchas `ItemListener`. Los objetos `JMenu` también pueden soportar los escuchas `ActionListener`, aunque eso no suele resultar útil. En general, asociaremos escuchas a cada elemento `JMenuItem`, `JCheckBoxMenuItem` o `JRadioButtonMenuItem`, pero el ejemplo muestra escuchas de tipo `ItemListener` y `ActionListener` asociados a los diversos componentes de menú.

Swing soporta el uso de “atajos de teclado”, así que podemos seleccionar cualquier cosa derivada de `AbstractButton` (botones, elementos de menú, etc.) utilizando el teclado en lugar del ratón. Estos atajos funcionan de forma muy simple; para `JMenuItem`, podemos utilizar el constructor sobrecargado que admite como segundo argumento el identificador de la tecla. Sin embargo, la mayoría de las clases `AbstractButton` no tiene constructores como éste, por lo que la forma más general de resolver el problema consiste en utilizar el método `setMnemonic()`. En el ejemplo anterior se añaden atajos al botón y a algunos elementos de menú; al hacerlo, aparecen automáticamente indicadores de los componentes que informan del atajo de teclado.

También podemos ver cómo se utiliza el método `setActionCommand()`. Este método parece algo extraño, porque en cada caso el “comando de acción”, definido por el método coincide exactamente con la etiqueta del menú. ¿Por qué no utilizar simplemente la etiqueta en lugar de esta cadena de caracteres alternativa? El problema estriba en la internacionalización. Si rehiciéramos este programa para otro idioma lo que queríamos es cambiar simplemente la etiqueta del menú, sin tener que cambiar el código (porque sin duda ese proceso de modificación podría introducir nuevos errores). Utilizando `setActionCommand()`, el “comando de acción” puede ser inmutable, mientras que la etiqueta de menú se puede modificar. Todo el código funciona con el “comando de acción”, así no se ve afectado por los cambios que efectuemos en las etiquetas de menú. Observe que en este programa, no se examinan todos los componentes de menú para determinados comandos de acción, de modo que no hemos configurado aquellos componentes de menú donde ese examen no se realiza.

El grueso del trabajo tiene lugar dentro de los escuchas. `BL` se encarga de realizar el intercambio de los objetos `JMenuBar`. En `ML`, se adopta la solución de “adivinar quién ha llamado” obteniendo el origen del suceso `ActionEvent` y proyectándolo sobre `JMenuItem`, después de lo cual se extrae el comando de acción para pasarlo a través de una cascada de instrucciones `if`.

El escucha `FL` es lo bastante simple, aún cuando se encarga de gestionar todos los diferentes sabores del menú de sabores. Esta técnica resulta útil si la lógica que estamos utilizando es suficientemente simple, pero en general, conviene emplear la solución por `Fool`, `BarL` y `BazL`, en la que cada escucha se asocia a un único componente de menú, lo que hace innecesario utilizar una lógica adicional de detección y nos permite saber exactamente quién ha invocado el escucha. Incluso con la profusión de clases que de esta manera se genera, el código tiende a ser más pequeño, y el proceso está más libre de errores.

Como puede ver, el código de especificación de menús puede llegar rápidamente a ser muy largo y confuso. Éste es un caso en el que la solución apropiada consistiría en emplear una herramienta de construcción de interfaces GUI. Si se tiene una buena herramienta, ésta se encargará también del mantenimiento de los menús.

Ejercicio 19: (3) Modifique `Menus.java` para utilizar botones de opción en lugar de casillas de verificación en los menús.

Ejercicio 20: (6) Cree un programa que descomponga un archivo texto en sus palabras componentes. Distribuya dichas palabras como etiquetas en una serie de menús y submenús.

Menús emergentes

La forma más directa de implementar un menú emergente `JPopupMenu` consiste en crear una clase interna que amplíe `MouseAdapter`, y luego agregar un objeto de dicha clase interna a cada componente al que queramos añadir ese comportamiento emergente:

```
//: gui/Popup.java
// Creación de menús emergentes con Swing.
import javax.swing.*;
```

```

import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
    private JPopupMenu popup = new JPopupMenu();
    private JTextField t = new JTextField(10);
    public Popup() {
        setLayout(new FlowLayout());
        add(t);
        ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText(((JMenuItem)e.getSource()).getText());
            }
        };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
        popup.add(m);
        PopupListener pl = new PopupListener();
        addMouseListener(pl);
        t.addMouseListener(pl);
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }
        private void maybeShowPopup(MouseEvent e) {
            if(e.isPopupTrigger())
                popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
    public static void main(String[] args) {
        run(new Popup(), 300, 200);
    }
} //:-/

```

En el ejemplo se añade el mismo escucha **ActionListener** a cada elemento **JMenuItem**. El escucha extrae el texto de la etiqueta del menú y lo inserta en el campo **JTextField**.

Dibujo

En un buen entorno GUI, las tareas de dibujo deberían resultar razonablemente sencillas, y así sucede en la biblioteca Swing. El problema con cualquier ejemplo de dibujo es que los cálculos que determinan dónde hay que colocar cada elemento suelen ser bastante más complicados que las llamadas a las rutinas de dibujo, y estos cálculos están a menudo mezclados con las llamadas a esas rutinas, lo que puede hacer que parezca que la interfaz es más complicada de lo que realmente es.

En aras de la simplicidad, considere el problema de representar una serie de datos en la pantalla; aquí, los datos estarán proporcionados por el método predefinido **Math.sin()**, que genera una función matemática seno. Para hacer las cosas algo más interesantes y para demostrar lo fácil que es utilizar los componentes Swing, colocaremos un deslizador en la parte inferior del formulario, para controlar dinámicamente el número de ciclos de la onda senoidal que se van mostrando. Además, si cambiamos el tamaño de la ventana, veremos que la onda seno se reajusta automáticamente en la nueva ventana.

Aunque cualquier control **JComponent** puede ser pintado y ser utilizado como el lienzo, si queremos disponer de una superficie de dibujo sencillo, lo que haremos normalmente será heredar de **JPanel**. El único método que hay que sustituir es **paintComponent()**, que se invoca cada vez que hay que repintar dicho componente (normalmente no hace falta preocuparse por esto, porque Swing toma la decisión). Cuando se invoca el método, Swing le pasa un objeto de tipo **Graphics**, pudiendo nosotros pasar dicho objeto para dibujar o pintar en la superficie.

En el siguiente ejemplo, toda la inteligencia relativa al proceso de dibujo se encuentra dentro de la clase **SineDraw**; la clase **SineWave** simplemente configura el programa y el control deslizador. Dentro de **SineDraw**, el método **setCycles()** proporciona un mecanismo para que otro objeto (en este caso, el control deslizador) controle el número de ciclos.

```
//: gui/SineWave.java
// Dibujo con Swing, utilizando un control JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw() { setCycles(5); }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int maxWidth = getWidth();
        double hstep = (double)maxWidth / (double)points;
        int maxHeight = getHeight();
        pts = new int[points];
        for(int i = 0; i < points; i++)
            pts[i] =
                (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);
        g.setColor(Color.RED);
        for(int i = 1; i < points; i++) {
            int x1 = (int)((i - 1) * hstep);
            int x2 = (int)(i * hstep);
            int y1 = pts[i-1];
            int y2 = pts[i];
            g.drawLine(x1, y1, x2, y2);
        }
    }
    public void setCycles(int newCycles) {
        cycles = newCycles;
        points = SCALEFACTOR * cycles * 2;
        sines = new double[points];
        for(int i = 0; i < points; i++) {
            double radians = (Math.PI / SCALEFACTOR) * i;
            sines[i] = Math.sin(radians);
        }
        repaint();
    }
}
```

```

public class SineWave extends JFrame {
    private SineDraw sines = new SineDraw();
    private JSlider adjustCycles = new JSlider(1, 30, 5);
    public SineWave() {
        add(sines);
        adjustCycles.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                sines.setCycles(
                    ((JSlider)e.getSource()).getValue());
            }
        });
        add(BorderLayout.SOUTH, adjustCycles);
    }
    public static void main(String[] args) {
        run(new SineWave(), 700, 400);
    }
} //:-

```

Todos los campos y matrices se utilizan en el cálculo de los puntos que definen la onda senoidal; **cycles** indica el número de ondas senoidales completas que deseamos, **points** contiene el número total de puntos que se dibujarán, **sines** contiene los valores de la función seno y **pts** contiene las coordenadas de los puntos que se dibujarán sobre el panel **JPanel**. El método **setCycles()** crea las matrices de acuerdo con el número de puntos necesarios y rellena la matriz **sines** con una serie de valores. Invocando **repaint()**, **setCycles()** fuerza a que se llame a **paintComponent()**, con lo que se producirá el resto de los cálculos y el proceso de redibujo.

Lo primero que hay que hacer cuando se sustituye **paintComponent()** es invocar la versión de la clase base del método. Después, somos libres de hacer lo que queramos; normalmente, esto significa emplear los métodos gráficos que podemos encontrar en la documentación correspondiente a **java.awt.Graphics** (en la documentación del JDK disponible en <http://java.sun.com>), para dibujar y pintar píxeles en el control **JPanel**. Podemos ver que casi todo el código está dedicado a la realización de los cálculos; las únicas dos llamadas a método que se encargan de manipular de hecho la pantalla son **setColor()** y **drawLine()**. Probablemente se encuentre, al hacer sus propios programas que muestren datos gráficos, con una experiencia similar: invertirá la mayor parte del tiempo tratando de determinar qué es lo que quiere dibujar, pero el propio proceso de dibujo será bastante simple.

Cuando creé este programa, la mayor parte del tiempo lo invertí en intentar que se visualizara la onda sinusoidal. Una vez resuelto eso, pensé que sería bastante atractivo poder cambiar dinámicamente el número de ciclos. Mis experiencias de programación intentando hacer cosas como ésta en otros lenguajes hacia que fuera un poco renuente a realizar esto, pero resultó ser la parte más fácil del proyecto. Creé un control **JSlider** (los argumentos el valor más a la izquierda del deslizador, el valor más a la derecha y el valor de inicio, respectivamente, pero también hay otros constructores) y lo inserté en el marco **JFrame**. Después examiné la documentación del JDK y observé que el único escucha era **addChangeListener**, que se disparaba cada vez que se cambiaba el deslizador lo suficiente como para generar un nuevo valor. El único método para este escucha era el denominado **stateChanged()**, que proporcionaba un objeto **ChangeEvent** con el que se podía determinar el origen del cambio y averiguar el nuevo valor. Invocar el método **setCycles()** del objeto **sines** permitía encontrar el nuevo valor y redibujar el panel **JPanel**.

En general, podrá encontrar que la mayor parte de los problemas basados en Swing pueden resolverse siguiendo un proceso similar, y además verá que es un proceso bastante simple, incluso si no ha utilizado un componente concreto anteriormente.

Si el problema es más complejo existen otras alternativas más sofisticadas para el tema de dibujo, incluyendo componentes JavaBeans de otras fuentes y la API 2D de Java. Estas soluciones caen fuera del alcance de este libro, pero puede informarse acerca de ellas si el código que está empleando para dibujar resulta demasiado complejo.

Ejercicio 21: (5) Modifique **SineWave.java** para transformar **SineDraw** en un componente JavaBean añadiendo métodos “getter” y “setter”.

Ejercicio 22: (7) Cree una aplicación utilizando **SwingConsole**. La aplicación debe tener tres deslizadores, que permitan ajustar los valores rojo, verde y azul en **java.awt.Color**. El resto del formulario debe ser un control **JPanel** que muestre el color determinado por los tres deslizadores. Incluya también campos de texto no editables que muestren los valores RGB actuales.

- Ejercicio 23:** (8) Utilizando **SineWave.java** como punto de partida, cree un programa que muestre un cuadrado rotatorio en la pantalla. Un deslizador debe controlar la velocidad de rotación y un segundo deslizador controlará el tamaño del recuadro.
- Ejercicio 24:** (7) ¿Recuerda ese juguete de dibujo que tenía dos mandos, uno para controlar el movimiento vertical del punto de dibujo y otro para controlar el movimiento horizontal? Cree una variante de este juguete, utilizando **SineWave.java** como punto de partida. En lugar de mandos, utilice deslizadores. Añada un botón que permita borrar todo el dibujo.
- Ejercicio 25:** (8) Comenzando con **SineWave.java**, cree un programa (una aplicación utilizando la clase **SwingConsole**) que dibuje una onda senoidal animada que parezca deslizarse a lo largo de la pantalla, como si fuera un osciloscopio, controlando la animación con un temporizador **java.util.Timer**. La velocidad de la animación debe poder regularse mediante un control **javax.swing.JSlider**.
- Ejercicio 26:** (5) Modifique el ejercicio anterior para que se creen múltiples paneles con ondas sinoidales dentro de la aplicación. El número de paneles con ondas sinoidales debe controlarse mediante parámetros de la línea de comandos.
- Ejercicio 27:** (5) Modifique el Ejercicio 25 para utilizar la clase **javax.swing.Timer** con el fin de dirigir la animación. Observe la diferencia entre esta clase y **java.util.Timer**.
- Ejercicio 28:** (7) Cree una clase que represente un dado (sólo una clase sin interfaz GUI). Cree cinco dados y láncelos repetidamente. Dibuje la curva que muestra la suma de los puntos obtenidos en cada tirada y muestre la curva evolucionando dinámicamente a medida que se hacen más tiradas.

Cuadros de diálogo

Un cuadro de diálogo es una ventana que emerge a partir de otra ventana. Su propósito es resolver algún tema específico, sin atestar la ventana original con los correspondientes detalles. Los cuadros de diálogo normalmente se emplean en entornos de programación basados en ventanas.

Para crear un cuadro de diálogo, hay que heredar de **JDialog**, que es simplemente otro tipo de objeto **Window**, como **JFrame**. Un control **JDialog** tiene un gestor de disposición (que de manera predeterminada es **BorderLayout**), y tenemos que añadir escuchas de sucesos al cuadro para poder tratar los sucesos. He aquí un ejemplo muy simple:

```
//: gui/Dialogs.java
// Creación y uso de cuadros de diálogo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);
        setLayout(new FlowLayout());
        add(new JLabel("Here is my dialog"));
        JButton ok = new JButton("OK");
        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose(); // Cierra el cuadro de diálogo
            }
        });
        add(ok);
        setSize(150,125);
    }
}

public class Dialogs extends JFrame {
    private JButton b1 = new JButton("Dialog Box");
    
```

```

private MyDialog dlg = new MyDialog(null);
public Dialogs() {
    b1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            dlg.setVisible(true);
        }
    });
    add(b1);
}
public static void main(String[] args) {
    run(new Dialogs(), 125, 75);
}
} //:-
```

Una vez creado el control **JDialog**, hay que invocar **setVisible(true)** para mostrarlo y activarlo. Cuando se cierra el cuadro de diálogo, es necesario liberar los recursos empleados por esa ventana, invocando **dispose()**.

El siguiente ejemplo es algo más complejo, el cuadro de diálogo está formado por una cuadrícula (utilizando **GridLayout**) de un tipo especial de botón que se define aquí mediante la clase **ToeButton**. Este botón dibuja un marco alrededor suyo y, dependiendo de su estado, un espacio en blanco, una "x," o una "o" en la parte central. Inicialmente, el botón está en blanco y luego, dependiendo de a quién le corresponda el turno cambia a una "x" o a una "o". Sin embargo, también alternará entre "x" y "o" cuando se haga clic en el botón, para proporcionar una interesante variante del juego de tres en raya. Además, el cuadro de diálogo puede configurarse para tener cualquier número de filas ocultas, modificando los valores en la venta de aplicación principal.

```

//: gui/TicTacToe.java
// Cuadros de diálogo y creación de sus propios componentes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
    private JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");
    private enum State { BLANK, XX, OO }
    static class ToeDialog extends JDialog {
        private State turn = State.XX; // Comenzar con el turno de x
        ToeDialog(int cellsWide, int cellsHigh) {
            setTitle("The game itself");
            setLayout(new GridLayout(cellsWide, cellsHigh));
            for(int i = 0; i < cellsWide * cellsHigh; i++)
                add(new ToeButton());
            setSize(cellsWide * 50, cellsHigh * 50);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }
        class ToeButton extends JPanel {
            private State state = State.BLANK;
            public ToeButton() { addMouseListener(new ML()); }
            public void paintComponent(Graphics g) {
                super.paintComponent(g);
                int
                    x1 = 0, y1 = 0,
                    x2 = getSize().width - 1,
                    y2 = getSize().height - 1;
                g.drawRect(x1, y1, x2, y2);
                x1 = x2/4;
                y1 = y2/4;
                int wide = x2/2, high = y2/2;
```

```

        if(state == State.XX) {
            g.drawLine(x1, y1, x1 + wide, y1 + high);
            g.drawLine(x1, y1 + high, x1 + wide, y1);
        }
        if(state == State.OO)
            g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
    }
    class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            if(state == State.BLANK) {
                state = turn;
                turn =
                    (turn == State.XX ? State.OO : State.XX);
            }
            else
                state =
                    (state == State.XX ? State.OO : State.XX);
            repaint();
        }
    }
}
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            new Integer(rows.getText()),
            new Integer(cols.getText()));
        d.setVisible(true);
    }
}
public TicTacToe() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Rows", JLabel.CENTER));
    p.add(rows);
    p.add(new JLabel("Columns", JLabel.CENTER));
    p.add(cols);
    add(p, BorderLayout.NORTH);
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    add(b, BorderLayout.SOUTH);
}
public static void main(String[] args) {
    run(new TicTacToe(), 200, 200);
}
} //:-

```

Puesto que los valores estáticos sólo se pueden encontrar en el nivel interno de la clase, las clases internas no pueden tener datos estáticos ni clases anidadas.

El método **paintComponent()** dibuja el cuadrado alrededor del panel y la indicación “x” u “o”. Este proceso está lleno de tediosos cálculos, pero resulta bastante sencillo.

Los clics de ratón se capturan mediante el escucha **MouseListener**, que primero comprueba si hay algo escrito en el panel. Si no, se consulta la ventana padre para averiguar a quién le corresponde el turno, lo que establece el estado del control **ToeButton**. Gracias al mecanismo de la clase interna, el control **ToeButton** puede acceder a los datos del parent y pasar el turno. Si el botón ya está mostrando una indicación “x” u “o”, entonces se invierte esa indicación. Podemos ver, dentro de los cálculos la comodidad que proporciona el uso de la instrucción **if-else ternaria**, descrita en el Capítulo 3, *Operadores*. Después de cada cambio de estado, se redibuja el control **ToeButton**.

El constructor para **ToeDialog** es bastante simple: añade a un gestor **GridLayout** tantos botones como solicitemos y luego cambia el tamaño para que cada botón tenga 50 píxeles de lado.

TicTacToe configura la aplicación completa creando los campos **JTextField** (para introducirlos en las filas y columnas de la cuadrícula de botones) y el botón “inicio” con su escucha **ActionListener**. Cuando se pulsa el botón, es necesario extraer los datos de **JtextField** y, como están en formato **String**, transformarlos a formato **int** utilizando el constructor de **Integer** que toma un argumento de tipo **String**.

Cuadros de diálogo de archivos

Algunos sistemas operativos tienen una serie de cuadros de diálogo predefinidos especiales para gestionar la selección de cosas tales como tipos de fuentes, colores, impresoras, etc. Casi todos los sistemas operativos gráficos permiten abrir y guardar archivos, así que un componente de Java, **JfileChooser**, encapsula estas operaciones para facilitar su realización.

La siguiente aplicación ilustra dos formas de cuadros de diálogo **JFileChooser**, una para abrir un archivo y otra para guardarlo. La mayor parte del código debería resultar familiar al lector, y todas las actividades de interés tienen lugar dentro de los escuchas de acción asociados con los dos clics de ratón distintos:

```
//: gui/FileChooserTest.java
// Ejemplo de cuadros de diálogo de archivos.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
    private JTextField
        fileName = new JTextField(),
        dir = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    public FileChooserTest() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        add(p, BorderLayout.SOUTH);
        dir.setEditable(false);
        fileName.setEditable(false);
        p = new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(fileName);
        p.add(dir);
        add(p, BorderLayout.NORTH);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Ejemplo de cuadro de diálogo "Open" (abrir):
            int rVal = c.showOpenDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                fileName.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                fileName.setText("You pressed cancel");
                dir.setText("");
            }
        }
    }
}
```

```

        }
    }

    class SaveL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Ejemplo de cuadro de diálogo "Save" (guardar):
            int rVal = c.showSaveDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                fileName.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                fileName.setText("You pressed cancel");
                dir.setText("");
            }
        }
    }

    public static void main(String[] args) {
        run(new FileChooserTest(), 250, 150);
    }
} //:-

```

Observe que hay muchas variantes que podemos aplicar a **JFileChooser**, incluyendo filtros para seleccionar los nombres de archivo permitidos.

Para un cuadro de diálogo de apertura de archivos (“open file”), invocamos **showOpenDialog()**, y para un cuadro de diálogo para guardar el archivo (“save file”), invocamos **showSaveDialog()**. Estos comandos no vuelven hasta que se cierra el cuadro de diálogo. El objeto **JFileChooser** sigue existiendo, así que podemos leer datos desde el mismo. Los métodos **getSelectedFile()** y **getCurrentDirectory()** son dos formas que permiten preguntar por el resultado de la operación. Si devuelven **null**, quiere decir que el usuario ha cancelado el cuadro de diálogo.

Ejercicio 29: (3) En la documentación del JDK correspondiente a **javax.swing**, busque el control **JColorChooser**. Escriba un programa con un botón que haga aparecer en forma de cuadro de diálogo este selector de color.

HTML en los componentes Swing

Cualquier componente que admita texto puede aceptar también texto HTML, que será reformateado de acuerdo con las reglas del lenguaje HTML. Esto quiere decir que podemos añadir muy fácilmente texto formateado a un componente Swing. Por ejemplo:

```

//: gui/HTMLButton.java
// Adición de texto HTML en componentes Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
    private JButton b = new JButton(
        "<html><b><font size=+2>" +
        "<center>Hello!<br><i>Press me now!" );
    public HTMLButton() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                add(new JLabel("<html>" +
                    "<i><font size=+4>Kapow!" ));
                // Forzar una redimensionación para incluir una nueva etiqueta:
                validate();
            }
        });
    }
}

```

```

        });
        setLayout(new FlowLayout());
        add(b);
    }
    public static void main(String[] args) {
        run(new HTMLButton(), 200, 500);
    }
} //:-
}

```

Es necesario iniciar el texto con "<html>", después de lo cual se pueden emplear marcadores HTML. Observe que no estamos obligados a incluir los marcadores normales de cierre.

ActionListener añade una nueva etiqueta **JLabel** al formulario, que también contiene texto HTML. Sin embargo, esta etiqueta no se añade durante la construcción, así que es necesario invocar el método **validate()** del contenedor para forzar una nueva disposición de los componentes (y con ello la visualización de la nueva etiqueta).

También podemos utilizar texto HTML para **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton**, y **JCheckBox**.

Ejercicio 30: (3) Escriba un programa que ilustre el uso de texto HTML en todos los elementos mencionados en el párrafo anterior.

Deslizadores y barras de progreso

Un deslizador (que ya ha sido utilizado en **SineWave.java**) permite al usuario introducir datos moviendo un punto de un lado a otro, lo cual resulta bastante intuitivo de algunas situaciones (como por ejemplo, en los controles de volumen). Una barra de progreso muestra los datos en una forma relativa, entre una posición "vacía" y una posición "llena" para que el usuario pueda hacerse una idea del estado en el que se encuentra el proceso. Mi ejemplo favorito consiste en asociar el deslizador con la barra de progreso, de modo que cuando se mueve el deslizador, la barra de progreso modifica su aspecto correspondientemente. El siguiente ejemplo también ilustra la clase **ProgressMonitor**, que es un cuadro de diálogo emergente con una funcionalidad más rica:

```

//: gui/Progress.java
// Utilización de deslizadores, barras de progreso y monitores de progreso.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
    private JProgressBar pb = new JProgressBar();
    private ProgressMonitor pm = new ProgressMonitor(
        this, "Monitoring Progress", "Test", 0, 100);
    private JSeparator sb =
        new JSeparator(JSeparator.HORIZONTAL, 0, 100, 60);
    public Progress() {
        setLayout(new GridLayout(2,1));
        add(pb);
        pm.setProgress(0);
        pm.setMillisToPopup(1000);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        pb.setModel(sb.getModel()); // Modelo compartido
        add(sb);
        sb.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                pm.setProgress(sb.getValue());
            }
        });
    }
}

```

```

        }
    });
}

public static void main(String[] args) {
    run(new Progress(), 300, 200);
}
} //:-)

```

La clave para asociar los componentes deslizador y barra de progreso estriba en compartir sus modelos, en la línea:

```
pb.setModel(sb.getModel());
```

Por supuesto, también podríamos controlar los dos utilizando un escucha, pero usar el modelo es mucho más simple en algunas situaciones sencillas. El control **ProgressMonitor** no tiene un modelo, por lo que es obligatorio utilizar el método basado en escucha. Observe que el control **ProgressMonitor** sólo se mueve hacia adelante y que se cierra automáticamente una vez que alcanza el final.

La barra de progreso **JProgressBar** es bastante sencilla, pero el control **JSlider** tiene bastantes opciones, como por ejemplo las que fijan la orientación y las marcas principales y secundarias del deslizador. Observe lo fácil que es añadir un borde con título.

Ejercicio 31: (8) Cree un “indicador asintótico de progreso” que vaya cada vez más lento a medida que se aproxime al final. Añada un comportamiento errático aleatorio, de manera que el indicador parezca estarse acelerando periódicamente.

Ejercicio 32: (6) Modifique **Progress.java** para que, en lugar de compartir los modelos, utilice un escucha para conectar el deslizador y la barra de progreso.

Selección del aspecto y del estilo

El concepto “aspecto y estilo seleccionables” permite al programa emular el aspecto y el estilo de diversos sistemas operativos. Podemos incluso cambiar dinámicamente el aspecto y el estilo mientras el programa se está ejecutando. Sin embargo, generalmente haremos una de las dos cosas: o bien seleccionar el “aspecto y estilo interplataforma” (que es el diseño “metal” de Swing), o seleccionar el aspecto y estilo del sistema en el que nos encontramos actualmente, de modo que el programa Java parezca haber sido creado específicamente para dicho sistema (está es casi siempre la mejor elección en la mayoría de los casos, para evitar confundir al usuario). El código para seleccionar uno de estos dos comportamientos es bastante simple, pero es preciso ejecutarlo *antes* de crear ningún componente visual, porque los componentes se construirán basándose en el aspecto y estilo actuales, y no se modificarán simplemente porque cambiemos el aspecto y el estilo a mitad de programa (dicho proceso es más complicado y resulta menos común, por lo que remitimos al lector a los libros dedicados específicamente a Swing).

De hecho, si queremos usar el aspecto y estilo interplataforma (“metal”), que es característico de los programas Swing, no tenemos que hacer nada, ya que se trata de la opción predeterminada. Pero si queremos en su lugar emplear el aspecto y estilo actuales del sistema operativo⁸, basta con insertar el siguiente código, normalmente al principio de **main()**, pero al menos antes de añadir ningún componente:

```

try {
    UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName());
} catch(Exception e) {
    throw new RuntimeException(e);
}

```

No hace falta incluir nada en la cláusula **catch** porque el gestor de la interfaz de usuario **UIManager** tomará como opción predeterminada el aspecto y estilo interplataforma si los intentos de seleccionar cualquiera de las otras alternativas fallan. Sin embargo, durante la depuración, la excepción puede resultar muy útil, así que podemos tratar de ver al menos algunos resultados mediante la cláusula **catch**.

⁸ Cabría discutir si las capacidades de visualización de Swing hacen justicia al sistema operativo.

He aquí un programa que acepta un argumento de la línea de comandos para seleccionar un determinado aspecto y estilo que muestra el aspecto de los distintos componentes con la opción elegida:

```

//: gui/LookAndFeel.java
// Selección del aspecto y el estilo de la aplicación.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
    private String[] choices =
        "Beny Meeny Minnie Mickey Moe Larry Curly".split(" ");
    private Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("Radio"),
        new JComboBox(choices),
        new JList(choices),
    };
    public LookAndFeel() {
        super("Look And Feel");
        setLayout(new FlowLayout());
        for(Component component : samples)
            add(component);
    }
    private static void usageError() {
        System.out.println(
            "Usage:LookAndFeel [cross|system|motif]");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals("cross")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getCrossPlatformLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals("system")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getSystemLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals("motif")) {
            try {
                UIManager.setLookAndFeel("com.sun.java.+" +
                    "swing.plaf.motif.MotifLookAndFeel");
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else usageError();
        // Observe que el aspecto y estilo deben configurarse
        // antes de crear cualquier componente.
    }
}
```

```

        run(new LookAndFeel(), 300, 300);
    }
} //:-
```

Puede ver que una opción consiste en crear explícitamente una cadena de caracteres para definir el aspecto y el estilo, como se ve con **MotifLookAndFeel**. Sin embargo, esa opción y la opción predeterminada “metal” son las únicas que pueden emplearse legalmente en todas las plataformas, aunque hay otras cadenas de caracteres que definen el aspecto y estilo para Windows y Macintosh, dichas cadenas sólo pueden usarse en sus respectivas plataformas (puede obtener estas cadenas invocando **getSystemLookAndFeelClassName()** mientras se encuentre en la plataforma deseada).

También es posible crear un paquete personalizado de aspecto y estilo, por ejemplo si estamos diseñando un sistema para una compañía que quiera disponer de una apariencia distintiva en sus aplicaciones. Se trata de una tarea compleja y que queda fuera del alcance de este libro (de hecho, verá que queda fuera del alcance de muchos libros dedicados a Swing).

Árboles, tablas y portapapeles

Podrá encontrar una breve introducción y diversos ejemplos sobre estos temas en el suplemento en línea para este capítulo (en inglés) disponible en www.MindView.net.

JNLP y Java Web Start

Resulta posible *firmar* un *applet* por razones de seguridad. Esto se muestra en el suplemento en línea de este capítulo (en inglés) disponible en www.MindView.net. Los *applets* firmados son potentes y pueden tomar el lugar de una aplicación, pero deben ejecutarse dentro de un explorador web. Esto requiere el sobrecoste adicional que el explorador se esté ejecutando en la máquina cliente, y significa también que la interfaz de usuario del *applet* está limitada y es, a menudo, visualmente confusa. El explorador web tiene su propio conjunto de menús y barras de herramientas, que aparecerán por encima del *applet*.⁹

El protocolo JNLP (*Java Network Launch Protocol*, protocolo Java de inicio a través de red) resuelve el problema sin sacrificar las ventajas de los *applets*. Con una aplicación JNLP, podemos tratar de descargar e instalar una aplicación Java autónoma en la máquina del cliente. Esta aplicación puede ejecutarse desde la línea de comandos, desde un ícono de escritorio o a través del gestor de aplicaciones instalado con la implementación JNLP. La aplicación puede incluso ejecutarse desde el sitio web desde el que fue inicialmente descargada.

Una aplicación JNLP puede descargar dinámicamente recursos de Internet en tiempo de ejecución y se puede comprobar automáticamente la versión si el usuario está conectado a Internet. Esto significa que proporciona todas las ventajas de un *applet* junto con las ventajas de las aplicaciones autónomas.

Al igual que los *applets*, las aplicaciones JNLP necesitan tratarse con cierta precaución por parte del sistema cliente. Debido a esto, las aplicaciones JNLP están sujetas a las mismas restricciones de seguridad que los *applets*. Al igual que los *applets*, pueden implantarse mediante archivos JAR firmados, dando así al usuario la opción de confiar en quien firma. A diferencia de los *applets*, si se implantan mediante un archivo JAR no firmado, pueden seguir teniendo acceso a ciertos recursos del sistema cliente por medio de diversos servicios de la API JNLP. El usuario deberá aprobar estas solicitudes durante la ejecución del programa.

JNLP describe un protocolo, no una implementación, así que hace falta una implementación para poder usarlo. Java Web Start, o JAWS, es la implementación de referencia oficial de Sun, disponible de forma gratuita y distribuida como parte de Java SE5. Si la está utilizando para desarrollo, deberá asegurarse de que el archivo JAR (**javaws.jar**) se encuentre en su ruta de clases; la solución más cómoda es añadir **javaws.jar** a la ruta de clases a partir de su ruta de instalación normal en **jre/lib**. Si está implantado la aplicación JNLP desde un servidor web, deberá comprobar que el servidor reconozca el tipo MIME **application/x-java-jnlp-file**. Si está empleando una versión reciente del servidor Tomcat (<http://jakarta.apache.org/tomcat>) este tipo MIME ya estará configurado. Consulte la guía de usuario de su servidor concreto.

Crear una aplicación JNLP no es difícil. Lo que hacemos es crear una aplicación estándar que se archiva en un archivo JAR, y luego proporcionar un archivo de arranque, que es XML simple que proporciona al sistema cliente toda la información necesaria para descargar e instalar la aplicación. Si decide no firmar el archivo JAR, entonces deberá emplear los servicios suministrados por la API JNLP para cada tipo de recurso de la máquina del usuario al que quiera acceder.

⁹ Jeremy Meyer ha desarrollado esta sección.

He aquí una variante **FileChooserTest.java** utilizando los servicios JNLP para abrir el selector de archivos, de modo que la clase pueda implantarse como una aplicación JNLP en un archivo JAR no firmado.

```
//: gui/jnlp/JnlpFileChooser.java
// Apertura de archivos en una máquina local con JNLP.
// {Requires: javax.jnlp.FileOpenService;
// Hay que tener javaws.jar en la ruta de clases}
// Para crear el archivo jnlpfilechooser.jar, haga esto:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
    private JTextField fileName = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    private JEditorPane ep = new JEditorPane();
    private JScrollPane jsp = new JScrollPane();
    private FileContents fileContents;
    public JnlpFileChooser() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        jsp.setViewportView(ep);
        add(jsp, BorderLayout.CENTER);
        add(p, BorderLayout.SOUTH);
        fileName.setEditable(false);
        p = new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(fileName);
        add(p, BorderLayout.NORTH);
        ep.setContentType("text");
        save.setEnabled(false);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            FileOpenService fs = null;
            try {
                fs = (FileOpenService)ServiceManager.lookup(
                    "javax.jnlp.FileOpenService");
            } catch(UnavailableServiceException use) {
                throw new RuntimeException(use);
            }
            if(fs != null) {
                try {
                    fileContents = fs.openFileDialog(".", 
                        new String[]{"txt", "*"});
                    if(fileContents == null)
                        return;
                    fileName.setText(fileContents.getName());
                    ep.read(fileContents.getInputStream(), null);
                }
            }
        }
    }
}
```

```

        } catch(Exception exc) {
            throw new RuntimeException(exc);
        }
        save.setEnabled(true);
    }
}

class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileSaveService fs = null;
        try {
            fs = (FileSaveService) ServiceManager.lookup(
                "javax.jnlp.FileSaveService");
        } catch(UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents = fs.saveFileDialog(".", 
                    new String[]{ "txt" },
                    new ByteArrayInputStream(
                        ep.getText().getBytes()),
                    fileContents.getName());
                if(fileContents == null)
                    return;
                fileName.setText(fileContents.getName());
            } catch(Exception exc) {
                throw new RuntimeException(exc);
            }
        }
    }
}
public static void main(String[] args) {
    JnlpFileChooser fc = new JnlpFileChooser();
    fc.setSize(400, 300);
    fc.setVisible(true);
}
} //:-
}

```

Observe que las clases **FileOpenService** y **FileCloseService** se importan del paquete **javax.jnlp** y que en ninguna parte del código se hace referencia directa al cuadro de diálogo **JFileChooser**. Los dos servicios usados aquí deben solicitarse empleando el método **ServiceManager.lookup()**, y sólo se podrá acceder a los recursos del sistema cliente a través de los objetos devueltos por este método. En este caso, los archivos del sistema de archivos del cliente están siendo escritos y leídos mediante la interfaz **FileContent**, proporcionada por JNLP. Cualquier intento de acceder a los recursos directamente utilizando, por ejemplo, un objeto **File** o un objeto **FileReader** haría que se generara una excepción **SecurityException** de la misma manera que si se intentaran emplear desde un *applet* no firmado. Si desea utilizar estas clases y no quedarse restringido a las interfaces de servicios de JNLP, tendrá que firmar el archivo JAR.

El comando comentado **jar** en **JnlpFileChooser.java** generará el archivo JAR necesario. He aquí un archivo de arranque apropiado para el ejemplo anterior.

```

//:1 gui/jnlp/filechooser.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
  codebase="file:C:/AAA-TIJ4/code/gui/jnlp"
  href="filechooser.jnlp">
<information>
  <title>FileChooser demo application</title>
  <vendor>Mindview Inc.</vendor>
  <description>

```

```

Jnlp File chooser Application
</description>
<description kind="short">
    Ilustra la apertura, lectura y escritura de un archivo de texto
</description>
<icon href="mindview.gif"/>
<offline-allowed/>
</information>
<resources>
    <j2se version="1.3+"
        href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlpfilechooser.jar" download="eager"/>
</resources>
<application-desc
    main-class="gui.jnlp.JnlpFileChooser"/>
</jnlp>
///:-

```

Puede encontrar este archivo de arranque en el código fuente descargable del libro (disponible en www.MindView.net) guardado como **filechooser.jnlp** sin la primera y la última línea, en el mismo directorio que el archivo JAR. Como puede ver, se trata de un archivo XML con un marcador **<jnlp>**. Contiene muy pocos sub-elementos, que se explican por sí mismos.

El atributo **spec** del elemento **jnlp** dice al sistema cliente con qué versión de JNLP puede ejecutarse la aplicación. El atributo **codebase** apunta a la URL donde se encuentran el archivo de arranque y los recursos. En este caso, apunta a un directorio de la máquina local, lo que constituye una buena forma de probar la aplicación. *Observe que no tendrá que cambiar esta ruta, ya que indica el directorio correcto en su máquina, para que el programa se cargue con éxito.* El atributo **href** debe especificar el nombre de este archivo.

El marcador **information** contiene varios sub-elementos que proporcionan información acerca de la aplicación, que utiliza la consola administrativa de Java Web Start o equivalente, la cual instala la aplicación JNLP y permite al usuario ejecutarla desde la línea de comandos, utilizando atajos, etc.

El marcador **resources** sirve a un propósito similar que le marcador de *applet* en un archivo HTML. El sub-elemento **j2se** especifica la versión de J2SE requerida para ejecutar la aplicación, y el sub-elemento **jar** especifica el archivo JAR en el que está archivada la clase. El elemento **jar** tiene un atributo **download**, que puede tener los valores “eager” o “lazy” que indican a la implementación de JNLP si se necesita o no descargar el archivo completo antes de poder ejecutar la aplicación.

EL atributo **application-desc** indica a la implementación JNLP qué clase es la clase ejecutable, o punto de entrada, al archivo JAR.

Otro útil sub-elemento del marcador **jnlp** es el marcador **security**, que no se ha mostrado en este ejemplo. He aquí, el aspecto de un marcador **security**:

```

<security>
    <all-permissions/>
<security/>

```

Utilice el marcador **security** cuando la aplicación esté implantada en un archivo JAR firmado. En el ejemplo anterior no era necesario porque es posible acceder a todos los recursos locales a través de los servicios JNLP.

Hay disponibles unos pocos más marcadores, cuyos detalles puede consultarlos en la especificación disponible en <http://java.sun.com/products/javawebstart/download-spec.html>.

Para ejecutar el programa, es necesaria una página de descarga que contenga un vínculo de hipertexto al archivo **.jnlp**. He aquí un ejemplo (sin la primera y última línea):

```

//!: gui/jnlp/filechooser.html
<html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then:
<a href="filechooser.jnlp">click here</a>

```

```
</html>
///:-
```

Una vez que haya descargado la aplicación, podrá configurarla utilizando la consola de administración. Si está empleando Java Web Start sobre Windows, entonces se le solicitará un acceso directo a su aplicación la segunda vez que lo use. Este comportamiento es configurable.

Aquí sólo se cubren dos de los servicios JNLP, aunque existen siete servicios en la versión actual. Cada uno de ellos está diseñado para realizar una tarea específica, como imprimir, o cortar y pegar en el portapapeles. Puede encontrar más información en <http://java.sun.com>.

Concurrencia y Swing

Al programar con Swing se emplean hebras. Hemos visto esto al principio del capítulo al estudiar que todo debería ser enviado a la hebra de despacho de sucesos de Swing a través de `SwingUtilities.invokeLater()`. Sin embargo, el hecho de que no se haya creado explícitamente un objeto **Thread** significa que los problemas del mecanismo de hebras puede aparecer por sorpresa. Debemos recordar siempre que existe una hebra de despacho de sucesos en Swing, la cual está siempre allí, gestionando todos los sucesos Swing extrayéndolos uno a uno de la cola de sucesos y ejecutándolos. Recordar que la hebra de despacho de sucesos está presente le ayudará a garantizar que la aplicación no se vea sometida a interbloqueos o condiciones de carrera.

En esta sección se analizan las cuestiones relativas al mecanismo multihebra que pueden surgir a la hora de trabajar con Swing.

Tareas de larga duración

Uno de los errores más fundamentales que podemos cometer al programar con una interfaz gráfica de usuario consiste en emplear accidentalmente la hebra de despacho de sucesos para ejecutar una tarea de larga duración. He aquí un ejemplo simple:

```
//: gui/LongRunningTask.java
// Un programa mal diseñado.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

public class LongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    public LongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch(InterruptedException e) {
                    System.out.println("Task interrupted");
                    return;
                }
                System.out.println("Task completed");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                // ¿Interrumpirse a sí mismo?
                Thread.currentThread().interrupt();
```

```

        }
    });
setLayout(new FlowLayout());
add(b1);
add(b2);
}
public static void main(String[] args) {
    run(new LongRunningTask(), 200, 150);
}
} //:-~

```

Cuando se pulsa **b1**, la hebra de despacho de sucesos se ve de repente ocupada en realizar la tarea de larga duración. Podrá ver que el botón ni siquiera vuelve a salir hacia afuera, porque la hebra de despacho de sucesos que se encarga normalmente de repintar en la pantalla está ocupada. Y no podemos hacer ninguna otra cosa, como pulsar **b2**, porque el programa no responderá hasta que se haya completado la tarea de **b1** y la hebra de despacho de sucesos vuelva a estar disponible. El código de **b2** es un intento incorrecto del problema, interrumpiendo la hebra de despacho de sucesos.

Por supuesto, la respuesta consiste en ejecutar los procesos de larga duración en hebras separadas. Aquí, se emplea el objeto ejecutor con la hebra **Executor**, que pone automáticamente las tareas pendientes en cola y las ejecuta de una en una.

```

//: gui/InterruptableLongRunningTask.java
// Tareas de larga duración dentro de hebras.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    public void run() {
        System.out.println(this + " started");
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
            System.out.println(this + " interrupted");
            return;
        }
        System.out.println(this + " completed");
    }
    public String toString() { return "Task " + id; }
    public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    ExecutorService executor =
        Executors.newSingleThreadExecutor();
    public InterruptableLongRunningTask() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Task task = new Task();
                executor.execute(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {

```

```

        public void actionPerformed(ActionEvent e) {
            executor.shutdownNow(); // Solución drástica
        }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
}
public static void main(String[] args) {
    run(new InterruptableLongRunningTask(), 200, 150);
}
} //:-/

```

Esta versión es mejor, pero cuando pulsamos **b2**, se llama a **shutdownNow()** sobre el objeto **ExecutorService**, con lo que se desactiva. Si intentamos añadir más tareas, se genera una excepción. Por tanto, pulsar **b2** hace que el programa deje de funcionar correctamente. Lo que nos gustaría es poder terminar la tarea actual (y cancelar las tareas pendientes) sin detener nada. El mecanismo de **Callable/Future** de Java SE5 descrito en el Capítulo 21, *Concurrencia*, es justo lo que necesitamos. Definiremos una nueva clase denominada **TaskManager**, que contienen *tuplas* que almacenan el objeto **Callable** que representa la tarea y el objeto **Future** devuelto por el objeto **Callable**. La razón de que sea necesaria la tupla es que nos permite mantener el control de la tarea original, con lo cual podemos tener información adicional que no esté disponible en el objeto **Future**. He aquí cómo se haría:

```

//: net/mindview/util/TaskItem.java
// Un objeto Future y el objeto Callable que lo produce.
package net.mindview.util;
import java.util.concurrent.*;

public class TaskItem<R,C extends Callable<R>> {
    public final Future<R> future;
    public final C task;
    public TaskItem(Future<R> future, C task) {
        this.future = future;
        this.task = task;
    }
} //:-/

```

En la biblioteca **java.util.concurrent**, la tarea no está disponible a través del objeto **Future** de manera predeterminada, porque la tarea no tendría por qué seguir necesariamente existiendo cuando obtengamos el resultado del objeto **Future**. Aquí, obligamos a que la tarea siga existiendo por el procedimiento de almacenarla.

TaskManager ha sido incluida en **net.mindview.util** para que esté disponible como utilidad de propósito general:

```

//: net/mindview/util/TaskManager.java
// Gestión y ejecución de una cola de tareas.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
extends ArrayList<TaskItem<R,C>> {
    private ExecutorService exec =
        Executors.newSingleThreadExecutor();
    public void add(C task) {
        add(new TaskItem<R,C>(exec.submit(task),task));
    }
    public List<R> getResults() {
        Iterator<TaskItem<R,C>> items = iterator();
        List<R> results = new ArrayList<R>();
        while(items.hasNext()) {
            TaskItem<R,C> item = items.next();
            if(item.future.isDone())

```

```

        try {
            results.add(item.future.get());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        items.remove();
    }
}
return results;
}
public List<String> purge() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<String> results = new ArrayList<String>();
    while(items.hasNext()) {
        TaskItem<R,C> item = items.next();
        // Dejar las tareas completadas para insertar los resultados:
        if(!item.future.isDone()) {
            results.add("Cancelling " + item.task);
            item.future.cancel(true); // Puede interrumpir
            items.remove();
        }
    }
    return results;
}
} //:-

```

TaskManager es un contenedor **ArrayList** de elementos **TaskItem**. También contiene un ejecutor monohebra **Executor**, de modo que cuando invocamos **add()** con un objeto **Callable**, se ejecuta el objeto **Callable** y se almacena el objeto **Future** resultante junto con la tarea original. De esta forma, si necesitamos hacer algo con la tarea, disponemos de una referencia a la misma. Como ejemplo simple, en **purge()** se utiliza el método **toString()** de la tarea.

Ahora podemos utilizar este mecanismo para gestionar las tareas de larga duración de nuestro ejemplo:

```

//: gui/InterruptableLongRunningCallable.java
// Utilización de objetos Callable para tareas de larga duración.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
implements Callable<String> {
    public String call() {
        run();
        return "Return value of " + this;
    }
}

public class
InterruptableLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,CallableTask> manager =
        new TaskManager<String,CallableTask>();
    public InterruptableLongRunningCallable() {
        b1.addActionListener(new ActionListener() {

```

```

        public void actionPerformed(ActionEvent e) {
            CallableTask task = new CallableTask();
            manager.add(task);
            System.out.println(task + " added to the queue");
        }
    });
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
            System.out.println(result);
    }
});
b3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Llamada de ejemplo a un método de una tarea:
        for(TaskItem<String,CallableTask> tt :
            manager)
            tt.task.id(); // No se requiere proyección
        for(String result : manager.getResults())
            System.out.println(result);
    }
});
setLayout(new FlowLayout());
add(b1);
add(b2);
add(b3);
}
public static void main(String[] args) {
    run(new InterruptibleLongRunningCallable(), 200, 150);
}
} //:-/
}

```

Como podemos ver, **CallableTask** hace exactamente lo mismo que **Task** excepto en que devuelve un resultado (en este caso, un objeto **String** que identifica la tarea).

Se han creado utilidades no Swing (que no forman parte de la distribución estándar de Java) denominadas **SwingWorker** (disponibles en el sitio web de Sun) y *Foxtrot* (disponibles en <http://foxtrot.sourceforge.net>) para resolver un problema similar, pero en el momento de escribir estas líneas, dichas utilidades no han sido modificadas para aprovechar el mecanismo **Callable/Future** de Java SE5.

A menudo, es importante proporcionar al usuario final algún tipo de indicación visual de que una tarea se está ejecutando, y de su progreso. Normalmente, esto se hace mediante una barra de progreso **JProgressBar** o un monitor de progreso **ProgressMonitor**. Este ejemplo utiliza un monitor de progreso **ProgressMonitor**:

```

//: gui/MonitoredLongRunningCallable.java
// Visualización del progreso de la tarea con ProgressMonitors.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
    private static int counter = 0;
    private final int id = counter++;
    private final ProgressMonitor monitor;
    private final static int MAX = 8;
    public MonitoredCallable(ProgressMonitor monitor) {
        this.monitor = monitor;
        monitor.setNote(toString());
    }
    public String call() {
        for(int i = 0; i < MAX; i++) {
            if(monitor.isCanceled())
                return null;
            monitor.setProgress(i);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
            }
        }
        return "Completed";
    }
}

```

```

        monitor.setMaximum(MAX - 1);
        monitor.setMillisToPopup(500);
    }
    public String call() {
        System.out.println(this + " started");
        try {
            for(int i = 0; i < MAX; i++) {
                TimeUnit.MILLISECONDS.sleep(500);
                if(monitor.isCanceled())
                    Thread.currentThread().interrupt();
                final int progress = i;
                SwingUtilities.invokeLater(
                    new Runnable() {
                        public void run() {
                            monitor.setProgress(progress);
                        }
                    });
            }
        } catch(InterruptedException e) {
            monitor.close();
            System.out.println(this + " interrupted");
            return "Result: " + this + " interrupted";
        }
        System.out.println(this + " completed");
        return "Result: " + this + " completed";
    }
    public String toString() { return "Task " + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,MonitoredCallable> manager =
        new TaskManager<String,MonitoredCallable>();
    public MonitoredLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MonitoredCallable task = new MonitoredCallable(
                    new ProgressMonitor(
                        MonitoredLongRunningCallable.this,
                        "Long-Running Task", "", 0, 0)
                );
                manager.add(task);
                System.out.println(task + " added to the queue");
            }
        });
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(String result : manager.purge())
                    System.out.println(result);
            }
        });
        b3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                for(String result : manager.getResults())
                    System.out.println(result);
            }
        });
    }
}

```

```

        }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(b3);
}
public static void main(String[] args) {
    run(new MonitoredLongRunningCallable(), 200, 500);
}
} //:-)

```

El constructor **MonitoredCallable** toma un objeto **ProgressMonitor** como argumento, y su método **call()** actualiza ese objeto cada medio segundo. Observe que un objeto **MonitoredCallable** es una tarea separada y no debe, por tanto, intentar controlar la interfaz de usuario directamente, así que se utiliza **SwingUtilities.invokeLater()** para enviar los cambios en la información de progreso al **monitor**. El tutorial de Swing elaborado por Sun (disponible en <http://java.sun.com>) muestra una técnica alternativa, consistente en utilizar el temporizado **Timer** de Swing, el cual comprueba el estado de la tarea y actualiza el monitor.

Si se pulsa el botón de “cancelación” para el monitor, **monitor.isCanceled()** devolverá **true**. Aquí, la tarea se limita a invocar **interrupt()** sobre su propia hebra, lo que ahora hace que entre en la cláusula **catch** donde se termina el **monitor** con el método **close()**.

El resto del código es, en la práctica, igual que antes salvo por la creación del objeto **ProgressMonitor** como parte del constructor **MonitoredLongRunningCallable**.

Ejercicio 33: (6) Modifique **InterruptibleLongRunningCallable.java** para que se ejecuten todas las tareas en paralelo en lugar de secuencialmente.

Hebras visuales

El siguiente ejemplo define una clase **JPanel** de tipo **Runnable** que dibuja diferentes colores en el panel. Esta aplicación está preparada para tomar valores de la línea de comandos con el fin de determinar el tamaño de la cuadrícula de colores y cuánto tiempo hay que dormir (con **sleep()**) entre los sucesivos cambios de color. Jugando con estos valores, podemos descubrir algunas características interesantes y posiblemente inexplicables en la implementación del mecanismo multihebra en nuestra plataforma:

```

//: gui/ColorBoxes.java
// Demostración visual del mecanismo multihebra.
// {Args: 12 50}
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

```

```

class CBox extends JPanel implements Runnable {
    private int pause;
    private static Random rand = new Random();
    private Color color = new Color(0);
    public void paintComponent(Graphics g) {
        g.setColor(color);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) { this.pause = pause; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                color = new Color(rand.nextInt(0xFFFFFF));

```

```

        repaint(); // Solicitar asincronamente el redibujo
        TimeUnit.MILLISECONDS.sleep(pause);
    }
} catch(InterruptedException e) {
    // Forma aceptable de salir
}
}

public class ColorBoxes extends JFrame {
    private int grid = 12;
    private int pause = 50;
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    public ColorBoxes() {
        setLayout(new GridLayout(grid, grid));
        for(int i = 0; i < grid * grid; i++) {
            CBox cb = new CBox(pause);
            add(cb);
            exec.execute(cb);
        }
    }
    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
            boxes.pause = new Integer(args[1]);
        run(boxes, 500, 400);
    }
} //:~
```

ColorBoxes configura un objeto **GridLayout** de tal manera que tenga una serie de celdas **grid** en cada dimensión. A continuación añade el número apropiado de objetos **CBox** para llenar la cuadricula, pasando el valor **pause** a cada uno de esos objetos. En **main()** podemos ver que **pause** y **grid** tienen valores predeterminados que pueden modificarse proporcionando los correspondientes argumentos a través de la línea de comandos.

Todo el trabajo se realiza en **CBox**. Esta clase hereda de **JPanel** e implementa la interfaz **Runnable**, de tal forma que cada panel **JPanel** también puede ser una tarea independiente. Estas tareas están dirigidas por un conjunto de hebras **ExecutorService**.

El color de la celda actual es **cColor**. Los colores se crean utilizando un constructor **Color** que admite números de 24 bits, que en este caso se crean aleatoriamente.

paintComponent() es bastante simple; sólo asigna un color a **cColor** y rellena el **JPanel** completo con dicho color.

En **run()**, podemos ver el bucle infinito que asigna un nuevo color aleatorio a **cColor** y luego invoca **repaint()** para mostrarlo. A continuación, la hebra pasa a dormir (con **sleep()**) durante el intervalo de tiempo especificado en la línea de comandos.

La llamada a **repaint()** en **run()** merece una cierta atención. A primera vista, pudiera parecer que estamos creando una gran cantidad de hebras, cada una de las cuales está forzando a que se produzca una operación de dibujo. Pudiera parecer que esto viola el principio de que sólo debemos enviar hebras a la cola de sucesos. Sin embargo, estas hebras no están en realidad modificando el recurso compartido. Cuando llaman a **repaint()**, eso no fuerza un redibujo en dicho instante, sino que simplemente fija un “indicador de modificación” para especificar que la siguiente vez que la hebra de despacho de sucesos esté lista para dibujar las cosas, ese área es un candidato para el redibujo. Por tanto, el programa no provoca ningún problema de gestión de hebras con Swing.

Cuando la hebra de despacho de sucesos realiza verdaderamente un redibujo con **paint()**, llama primero a **paintComponent()**, luego a **paintBorder()** y **paintChildren()**. Si necesitáramos sustituir **paint()** en un componente derivado, tenemos que acordarnos de llamar a la versión de la clase base de **paint()**, de modo que se sigan realizando las acciones apropiadas.

Precisamente porque este diseño es flexible y el mecanismo de hebras está asociado con cada elemento **JPanel**, podemos experimentar creando tantas hebras como queramos (en realidad, hay una restricción impuesta por el número de hebras que la máquina JVM sea capaz de gestionar).

Este programa también constituye una interesante prueba comparativa de rendimiento, ya que puede mostrar enormes diferencias entre prestaciones y comportamiento entre una implementación multihebra de la JVM y otra, así como entre unas plataformas y otras.

Ejercicio 34: (4) Modifique **ColorBoxes.java** de modo que comience distribuyendo puntos (“estrellas”) por el lienzo, y luego cambiando aleatoriamente el color de esas “estrellas”.

Programación visual y componentes JavaBean

Hasta ahora, hemos visto en el libro lo útil que resulta el lenguaje Java para la creación de fragmentos de código reutilizable. La unidad de código “más reutilizable” ha sido la clase, ya que comprende una unidad cohesionada de características (campos) y comportamientos (métodos) que pueden reutilizarse bien directamente, mediante composición, o bien mediante herencia.

La herencia y el polimorfismo son partes esenciales de la programación orientada a objetos, pero en la mayoría de los casos, a la hora de construir una aplicación, lo que realmente queremos es *componentes* que hagan exactamente lo que necesitamos. Lo que nos gustaría es colocar estos componentes en nuestro diseño como si fueran los circuitos integrados que un ingeniero electrónico dispone sobre una placa de circuito impreso. Parece que debería haber alguna forma de acelerar este estilo de programación basado en la “construcción modular”.

La “programación visual” consiguió su primer éxito (un *gran éxito*) con Visual BASIC (VB) de Microsoft, seguido de un diseño de segunda generación representado por Delphi de Borland (que fue la principal inspiración para el diseño de JavaBeans). Con estas herramientas de programación, los componentes se representan visualmente, lo que tiene bastante sentido, ya que normalmente mostrará algún tipo de componente visual como un botón o un campo de texto. De hecho, la representación visual coincide a menudo con la forma exacta que el componente tendrá cuando el programa se ejecuta. Por tanto, parte del proceso de programación visual implica arrastrar un componente desde una paleta y depositarlo sobre el formulario. El entorno integrado de desarrollo (IDE, *Integrated Development Environment*) Application Builder escribe automáticamente el código a medida que realizamos estas operaciones y dicho código hará que se cree el componente dentro del programa.

Normalmente, para completar el programa no es suficiente con colocar el componente sobre un formulario. A menudo, es preciso cambiar las características de un componente, como por ejemplo el color, el texto que se muestra, la base de datos a la que está conectado, etc. Las características que se pueden cambiar en tiempo de diseño se denominan *propiedades*. Podemos manipular las propiedades de nuestro componente dentro del entorno IDE y, cuando se crea el programa, estos datos de configuración se guardan para poder ser recuperados cuando el programa se ejecute.

El lector debería tener clara a estas alturas la idea de que un objeto es más que una serie de características: también es un conjunto de comportamientos. En tiempo de diseño, los comportamientos de un componente visual están parcialmente representados por *sucesos*, cada uno de los cuales constituye una declaración del tipo: “He aquí algo que puedes hacer a este componente”. Normalmente, somos nosotros los que decidimos qué es lo que queremos que ocurra, asociando a ese suceso un cierto código.

La parte crítica es la siguiente: el entorno IDE utiliza el mecanismo de reflexión para interrogar dinámicamente al componente y averiguar qué propiedades de sucesos soporta. Una vez que sabe cuáles son, puede mostrar las propiedades y permitirnos modificarlas (guardando el estado cuando construyamos el programa), y también muestra los sucesos. En general, lo que nosotros hacemos es un doble clic sobre un suceso y el entorno IDE crea un cuerpo de código y lo asocia con el suceso particular. Lo único que hace falta en dicho punto es escribir el código que tenga que ejecutarse cuando ocurra el suceso.

Todo esto significa que el entorno IDE hace una gran cantidad de trabajo por nosotros. Como resultado, podemos centrarnos en el aspecto del programa y en lo que se supone que el programa debe hacer, delegando en IDE la gestión de todos los detalles de conexión. La razón de que las herramientas de programación visual hayan tenido tanto éxito es que permiten acelerar enormemente el proceso de construcción de una aplicación, por supuesto, la interfaz de usuario, pero a menudo también se acelera la construcción de otras partes de la aplicación.

¿Qué es un componente JavaBean?

Un componente es, en definitiva, simplemente un bloque de código que normalmente está encerrado dentro de una clase. La cuestión clave es la capacidad del IDE para descubrir las propiedades y sucesos de cada componente. Para crear un componente VB, el programador tenía originalmente que escribir un fragmento bastante complicado de código, que tenía que respetar ciertos convenios para exponer las propiedades y sucesos (que se hizo más fácil a medida que transcurrieron los años). Delphi era una herramienta de programación visual de segunda generación, y el lenguaje estaba diseñado específicamente centrándose en la programación visual, con lo que era mucho más fácil crear un componente visual. Sin embargo, Java ha llevado la creación de componentes visuales a su estado más avanzado con JavaBeans, ya que un componente Bean es simplemente una clase. No hace falta escribir ningún código adicional ni utilizar extensiones del lenguaje especiales para poder transformar algo en un componente Bean. Lo único necesario es, de hecho, modificar ligeramente la forma de denominar los métodos. Es el nombre del método lo que le dice al entorno IDE si se trata de una propiedad, de un suceso o de un método normal.

En la documentación del JDK, este convenio de denominación se llama, de manera confusa, "patrón de diseño". Resulta bastante desafortunado que esto sea así, ya que los patrones de diseño (consulte *Thinking in Patterns* en www.MindView.net) ya son lo suficientemente complejos sin necesidad de que se introduzca confusión adicional. El convenio de denominación no es un patrón de diseño, y resulta bastante sencillo:

1. Para una propiedad denominada **xxx**, normalmente creamos dos métodos: **getXXX()** y **setXXX()**. La primera letra después de "get" o "set" será puesta automáticamente en minúscula por las herramientas que examinen los métodos, con el fin de generar el nombre de la propiedad. El tipo producido por el método "get" coincide con el tipo del argumento del método "set". El nombre de la propiedad y el tipo de los métodos "get" y "set" no están relacionados.
2. Para una propiedad de tipo **boolean**, podemos emplear la técnica anterior basada en "get" y "set", pero también podemos usar "is" en lugar de "get."
3. Los métodos normales del componente Bean no se adaptan al convenio de denominación anterior, pero son de tipo **public**.
4. Para los sucesos, se utiliza la solución Swing basada en escuchas. Es exactamente el mismo convenio que hemos visto anteriormente: **addBounceListener(BounceListener)** y **removeBounceListener(BounceListener)** para gestionar un suceso **BounceEvent**. La mayor parte de las veces, los sucesos y escuchas predefinidos satisfarán completamente nuestras necesidades, pero también podemos crear nuestros propios sucesos e interfaces escucha.

Para crear un componente Bean simple, podemos emplear estas directrices:

```
//: frogbean/Frog.java
// Un componente JavaBean trivial.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
}
```

```

public void setSpots(Spots newSpots) {
    spots = newSpots;
}
public boolean isJumper() { return jmpr; }
public void setJumper(boolean j) { jmpr = j; }
public void addActionListener(ActionListener l) {
    //...
}
public void removeActionListener(ActionListener l) {
    // ...
}
public void addKeyListener(KeyListener l) {
    // ...
}
public void removeKeyListener(KeyListener l) {
    // ...
}
// Un método público "normal":
public void croak() {
    System.out.println("Ribbet!");
}
} //:~
```

En primer lugar, podemos ver que se trata simplemente de una clase. Normalmente, todos los campos serán privados y sólo se podrá acceder a ellos a través de sus métodos y propiedades. Siguiendo el convenio de denominación, las propiedades son **jumps**, **color**, **spots** y **jumper** (observe el cambio de mayúsculas a minúsculas en la primera letra del nombre de la propiedad). Aunque el nombre del identificador interno es el mismo que el nombre de la propiedad en los tres primeros casos, en **jumper** podemos ver que el nombre de la propiedad no nos obliga a utilizar ningún identificador concreto para las variables internas (ni tampoco nos obliga, de hecho, ni siquiera a *tener* ninguna variable interna para dicha propiedad).

Los sucesos gestionados por este componente Bean son **ActionEvent** y **KeyEvent**, basados en la denominación de los métodos “add” “remove” para el escucha asociado. Finalmente, podemos ver que el método normal **croak()** sigue siendo parte de la Bean simplemente porque se trata de un método público, no porque se adapte a ningún esquema de denominación.

Extracción de la información BeanInfo con Introspector

Una de las partes más críticas del esquema JavaBean tiene lugar cuando arrastramos una Bean de una paleta y la colocamos en un formulario. El entorno IDE debe ser capaz de crear la Bean (lo cual puede hacerse si existe un constructor predeterminado) y luego, sin tener ningún acceso al código fuente de la Bean, extraer toda la información necesaria para crear la hoja de propiedades y las rutinas de tratamiento de sucesos.

Parte de la solución resulta evidente a partir de lo comentado en el Capítulo 14, *Información de tipos*: el mecanismo de **reflexión** de Java permite descubrir todos los métodos de una clase desconocida. Esto resulta perfecto para resolver el problema del diseño JavaBean sin requerir palabras clave del lenguaje como las que existen en otros lenguajes de programación visual. De hecho, una de las principales razones de que se añadiera el mecanismo de reflexión de Java fue para soportar JavaBeans (aunque el mecanismo de reflexión también soporta la serialización de objetos y la invocación remota de métodos, RMI, que resulta útil para la programación normal). Por tanto, podríamos esperar que el creador del entorno IDE usara el mecanismo de reflexión con cada Bean y analizara sus métodos para encontrar las propiedades y sucesos correspondientes.

Ciertamente, esto resulta posible, pero los diseñadores de Java querían proporcionar una herramienta estándar, no sólo para hacer que los componentes Bean sean simples de usar, sino también para proporcionar una pasarela estándar para la creación de otros componentes Bean más complejos. Esta herramienta es la clase **Introspector**, y el método más importante de esta clase es el método estático **getBeanInfo()**. Basta con pasar una referencia **Class** a este método y el método se encargará de interrogar a fondo a dicha clase y de devolver un objeto **BeanInfo** que se puede diseccionar para determinar las propiedades, los métodos y los sucesos.

Normalmente, nosotros no tenemos que preguntarnos nada acerca de esto; lo más probable es que obtengamos la mayoría de nuestros componentes Bean ya diseñados, y no tenemos por qué conocer todos los detalles subyacentes. Basta con arrastrar los componentes sobre el formulario, configurar sus propiedades y escribir las rutinas de tratamiento para los sucesos

de interés. Sin embargo, la utilización de **Introspector** para mostrar información acerca de una Bean constituye un ejercicio educativo excelente. He aquí una herramienta que hace precisamente esto:

```
//: gui/BeanDumper.java
// Obtención de la información acerca de una Bean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class BeanDumper extends JFrame {
    private JTextField query = new JTextField(20);
    private JTextArea results = new JTextArea();
    public void print(String s) { results.append(s + "\n"); }
    public void dump(Class<?> bean) {
        results.setText("");
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(bean, Object.class);
        } catch(IntrospectionException e) {
            print("Couldn't introspect " + bean.getName());
            return;
        }
        for(PropertyDescriptor d: bi.getPropertyDescriptors()) {
            Class<?> p = d.getPropertyType();
            if(p == null) continue;
            print("Property type:\n" + p.getName() +
                "Property name:\n" + d.getName());
            Method readMethod = d.getReadMethod();
            if(readMethod != null)
                print("Read method:\n" + readMethod);
            Method writeMethod = d.getWriteMethod();
            if(writeMethod != null)
                print("Write method:\n" + writeMethod);
            print("=====");
        }
        print("Public methods:");
        for(MethodDescriptor m : bi.getMethodDescriptors())
            print(m.getMethod().toString());
        print("=====");
        print("Event support:");
        for(EventSetDescriptor e: bi.getEventSetDescriptors()) {
            print("Listener type:\n" +
                e.getListenerType().getName());
            for(Method lm : e.getListenerMethods())
                print("Listener method:\n" + lm.getName());
            for(MethodDescriptor lmd :
                e.getListenerMethodDescriptors())
                print("Method descriptor:\n" + lmd.getMethod());
            Method addListener= e.getAddListenerMethod();
            print("Add Listener Method:\n" + addListener);
            Method removeListener = e.getRemoveListenerMethod();
            print("Remove Listener Method:\n" + removeListener);
            print("=====");
        }
    }
    class Dumper implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```

```

        String name = query.getText();
        Class<?> c = null;
        try {
            c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
            results.setText("Couldn't find " + name);
            return;
        }
        dump(c);
    }
}

public BeanDumper() {
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Qualified bean name:"));
    p.add(query);
    add(BorderLayout.NORTH, p);
    add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Forzar la evaluación
    dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
}
public static void main(String[] args) {
    run(new BeanDumper(), 600, 500);
}
} //:-~

```

BeanDumper.dump() se encarga de realizar todo el trabajo. Primero trata de crear un objeto **BeanInfo**, y si tiene éxito, invoca los métodos de **BeanInfo** que generan la información acerca de las propiedades, métodos y sucesos. En **Introspector.getBeanInfo()**, podemos ver que hay un segundo argumento que le dice a **Introspector** dónde detenerse dentro de la jerarquía de herencia. En este ejemplo, se detiene antes de analizar todos los métodos de **Object**, ya que no estamos interesados en ellos.

Para las propiedades, **getPropertyDescriptors()** devuelve una matriz de objetos **PropertyDescriptor**. Para cada objeto **PropertyDescriptor**, podemos invocar **getPropertyType()** para encontrar la clase del objeto que se pasa a los métodos de propiedad o que estos métodos devuelven. A continuación, para cada propiedad, podemos obtener su seudónimo (extraído de los nombres de los métodos) con **getName()**, el método de lectura con **getReadMethod()** y el método de escritura con **getWriteMethod()**. Estos dos últimos métodos devuelven un objeto **Method** que puede de hecho utilizarse para invocar el método correspondiente sobre el objeto (esto es parte del mecanismo de reflexión).

Para los métodos públicos (incluyendo los métodos de propiedad), **getMethodDescriptors()** devuelve una matriz de objetos **MethodDescriptor**s. Para cada uno, podemos obtener el objeto **Method** asociado e imprimir su nombre.

Para los sucesos, **getEventSetDescriptors()** devuelve una matriz de objetos **EventSetDescriptor**. Cada uno de estos objetos puede consultarse para averiguar la clase a la que pertenece el escucha, los métodos de dicho escucha y los métodos para agregar y eliminar escuchas. El programa **BeanDumper** visualiza toda esta información.

En el arranque, el programa fuerza la evaluación de **frogbean.Frog**. La salida, después de eliminar los detalles innecesarios, es:

```

Property type:
Color
Property name:
color
Read method:
    public Color getColor()
Write method:
    public void setColor(Color)
=====

```

```

Property type:
  boolean
Property name:
  jumper
Read method:
  public boolean isJumper()
Write method:
  public void setJumper(boolean)
=====
Property type:
  int
Property name:
  jumps
Read method:
  public int getJumps()
Write method:
  public void setJumps(int)
=====
Property type:
  frogbean.Spots
Property name:
  spots
Read method:
  public frogbean.Spots getSpots()
Write method:
  public void setSpots(frogbean.Spots)
=====
Public methods:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)
=====
Event support:
Listener type:
  KeyListener
Listener method:
  keyPressed
Listener method:
  keyReleased
Listener method:
  keyTyped
Method descriptor:
  public abstract void keyPressed(KeyEvent)
Method descriptor:
  public abstract void keyReleased(KeyEvent)
Method descriptor:
  public abstract void keyTyped(KeyEvent)
Add Listener Method:
  public void addKeyListener(KeyListener)

```

```

Remove Listener Method:
public void removeKeyListener(KeyListener)
=====
Listener type:
ActionListener
Listener method:
actionPerformed
Method descriptor:
public abstract void actionPerformed(ActionEvent)
Add Listener Method:
public void addActionListener(ActionListener)
Remove Listener Method:
public void removeActionListener(ActionListener)
=====

```

Esta salida nos revela la mayor parte de la información que **Introspector** ve a medida que genera un objeto **BeanInfo** a partir de la Bean. Podemos ver que el tipo de la propiedad y su nombre son independientes. Observe el uso de minúscula en el nombre de la propiedad (la única vez que esto no sucede cuando el nombre de la propiedad comienza con más de dos letras mayúsculas seguidas). Y recuerde que los nombres de métodos que podemos ver aquí (como por ejemplo los de lectura y escritura) son producidos por un objeto **Method** que puede emplearse para invocar el método asociado sobre el objeto.

La lista de los métodos públicos incluye los métodos que no están asociados con una propiedad o un suceso, como por ejemplo **croak()**, así como los que sí están asociados. Se trata de todos los métodos que se pueden invocar mediante programa para una Bean, y el entorno IDE puede facilitarnos la tarea de programación enumerando todos esos métodos mientras realizamos llamadas a métodos.

Por último, podemos ver que los sucesos se analizan completamente, extrayendo la información acerca del escucha, de sus métodos y de los métodos para agregar y eliminar escuchas. Básicamente, una vez que disponemos del objeto **BeanInfo**, podemos determinar toda la información importante acerca de la Bean. También podemos invocar los métodos para dicha Bean, a pesar de no disponer de ninguna otra información, salvo el propio objeto (de nuevo, ésta es una funcionalidad ofrecida por el mecanismo de reflexión).

Una Bean más sofisticada

El ejemplo siguiente es ligeramente más sofisticado aunque un poco frívolo. Se trata de un control **JPanel** que dibuja un círculo alrededor del ratón cada vez que éste se mueve. Cuando apretamos el botón del ratón, aparece la palabra "Bang!" en mitad de la pantalla y se dispara un escucha de acción.

Las propiedades que podemos modificar son el tamaño del círculo y el color, el tamaño y el texto de la palabra que se muestra cuando se pulsa el botón del ratón. El componente **BangBean** también tiene sus propios métodos **addActionListener()** y **removeActionListener()**, de modo que podemos asociar nuestro propio escucha que se disparará cuando el usuario haga clic sobre el componente **BangBean**. Resulta sencillo identificar en el ejemplo el soporte y propiedades del suceso:

```

//: bangbean/BangBean.java
// Una Bean gráfica.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class BangBean extends JPanel implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Tamaño del círculo
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ActionListener actionListener;

```

```

public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
}
public int getCircleSize() { return cSize; }
public void setCircleSize(int newSize) {
    cSize = newSize;
}
public String getBangText() { return text; }
public void setBangText(String newText) {
    text = newText;
}
public int getFontSize() { return fontSize; }
public void setFontSize(int newSize) {
    fontSize = newSize;
}
public Color getTextColor() { return tColor; }
public void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
}
// Se trata de un escucha unidifusión, que es la
// forma más simple de gestionar los escuchas:
public void addActionListener(ActionListener l)
throws TooManyListenersException {
    if(actionListener != null)
        throw new TooManyListenersException();
    actionListener = l;
}
public void removeActionListener(ActionListener l) {
    actionListener = null;
}
class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Graphics g = getGraphics();
        g.setColor(tColor);
        g.setFont(
            new Font("TimesRoman", Font.BOLD, fontSize));
        int width = g.getFontMetrics().stringWidth(text);
        g.drawString(text, (getSize().width - width) /2,
                    getSize().height/2);
        g.dispose();
        // Invocar el método del escucha:
        if(actionListener != null)
            actionListener.actionPerformed(
                new ActionEvent(BangBean.this,
                               ActionEvent.ACTION_PERFORMED, null));
    }
}
class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

```

```

    public Dimension getPreferredSize() {
        return new Dimension(200, 200);
    }
} //:-

```

Lo primero que podemos observar es que **BangBean** implementa la interfaz **Serializable**. Esto quiere decir que el entorno IDE puede extraer toda la información de **BangBean** utilizando serialización después de que el diseñador haya ajustado los valores de las propiedades. Cuando se crea la Bean como parte de la aplicación que se está ejecutando, estas propiedades extraídas se restauran de modo que podamos obtener exactamente lo que deseamos.

Si examinamos la firma de **addActionListener()**, vemos que puede generar una excepción **TooManyListenersException**. Esto indica que se trata de un escucha de *unidifusión*, lo que quiere decir que envía una notificación a un único escucha en el momento de producirse el suceso. Normalmente, lo que usamos son sucesos de *multidifusión*, de modo que muchos escuchas puedan ser notificados por un suceso. Sin embargo, eso nos haría tropezar con cuestiones de programación multihebra, por lo que volveremos sobre el tema en la siguiente sección, "Sincronización en JavaBeans". Mientras tanto, un suceso de unidifusión nos permite resolver momentáneamente el problema.

Cuando hacemos clic con el ratón, el texto aparece en la parte central del componente **BangBean**, y si el campo **actionListener** es distinto de **null**, se invoca su método **actionPerformed()** creando un nuevo objeto **ActionEvent** en el proceso. Cada vez que se mueve el ratón se capturan sus nuevas coordenadas y se vuelve a dibujar el lienzo (borrando el texto que hubiera en el lienzo, como se verá al ejecutar el programa).

He aquí la clase **BangBeanTest** para probar la Bean:

```

//: bangbean/BangBeanTest.java
// {Timeout: 5} Abortar después de 5 segundos durante las pruebas
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
    private JTextField txt = new JTextField(20);
    // Durante las pruebas, informar de las acciones:
    class BBL implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            txt.setText("BangBean action "+ count++);
        }
    }
    public BangBeanTest() {
        BangBean bb = new BangBean();
        try {
            bb.addActionListener(new BBL());
        } catch(TooManyListenersException e) {
            txt.setText("Too many listeners");
        }
        add(bb);
        add(BorderLayout.SOUTH, txt);
    }
    public static void main(String[] args) {
        run(new BangBeanTest(), 400, 500);
    }
} //:-

```

Cuando se emplea la Bean en un entorno IDE, esta clase no se usará, pero es útil proporcionar un método de prueba rápido para cada Bean que diseñemos. **BangBeanTest** coloca un componente **BangBean** dentro del marco **JFrame**, asociando un escucha **ActionListener** simple con el componente **BangBean** con el fin de imprimir un recuento de sucesos en el campo

JTextField cada vez que se produzca un suceso **ActionEvent**. Por supuesto, normalmente el IDE crearía la mayor parte del código que utilice la Bean.

Cuando ejecute el componente **BangBean** a través de **BeanDumper** o lo incluya dentro de un entorno de desarrollo preparado para componentes Bean, observará que hay muchas más propiedades y acciones de las que el código precedente permite incluir. Eso se debe a que **BangBean** hereda de **JPanel**, y **JPanel** también es un componente Bean, así que se mostrarán también sus propiedades y sucesos.

Ejercicio 35: (6) Localice y descargue uno o más de los entornos gratuitos para el desarrollo de interfaces GUI disponibles en Internet, o utilice algún producto comercial que posea. Descubra qué es lo que hace falta para añadir un **BangBean** a ese entorno y añádalo.

Sincronización en JavaBeans

Siempre que creamos una Bean, deberemos asumir que ésta será ejecutada dentro de un entorno multihebra. Esto quiere decir que:

1. Siempre que sea posible, todos los métodos públicos de una Bean deben ser sincronizados. Por supuesto, esto implica que tendremos que pagar el coste de la sincronización en tiempo de ejecución (que se ha reducido significativamente en las versiones recientes del JDK). Si esto es un problema, podemos dejar sin sincronizar los métodos que no vayan a provocar problemas en las secciones críticas, pero recuerde que dichos métodos no resultan siempre obvios. Los métodos que podrían caer dentro de esta categoría tienden a ser pequeños (tal como **getCircleSize()** en el ejemplo siguiente) y/o “atómicos”; es decir, la llamada al método se ejecuta utilizando una cantidad de código tan pequeña que el objeto no puede modificarse durante la ejecución (pero recuerde del Capítulo 21, *Concurrencia*, que aquello que pensemos que es atómico puede en realidad no serlo). Dejar sin sincronizar dichos métodos puede no tener un efecto significativo sobre la velocidad de ejecución del programa. Lo mejor es que todos los métodos públicos de la Bean sean sincronizados y eliminar la palabra clave **synchronized** de un método únicamente cuando estemos completamente seguros de que eso va a aumentar la velocidad y podemos eliminar la palabra con seguridad.
2. Cuando se dispara un suceso multidifusión para notificar a un conjunto de escuchas interesados en dicho suceso, debemos asumir que se pueden añadir o eliminar escuchas mientras estemos recorriendo la lista.

El primer punto es bastante sencillo de entender, pero el segundo requiere algo más de reflexión. En **BangBean.java** evitamos los problemas de concurrencia ignorando la palabra clave **synchronized** y haciendo que el suceso fuera de unidifusión. He aquí una versión modificada que trabaja en un entorno multihebra y utiliza el mecanismo de multidifusión para los sucesos:

```
//: gui/BangBean2.java
// Debería escribir sus componentes Beans de esta forma para
// poder ejecutarlos en un entorno multihebra.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Tamaño del círculo
    private String text = "Bang!";
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ArrayList<ActionListener> actionListeners =
        new ArrayList<ActionListener>();
    public BangBean2() {
        addMouseListener(new ML());
    }
}
```

```

        addMouseMotionListener(new MM());
    }
    public synchronized int getCircleSize() { return cSize; }
    public synchronized void setCircleSize(int newSize) {
        cSize = newSize;
    }
    public synchronized String getBangText() { return text; }
    public synchronized void setBangText(String newText) {
        text = newText;
    }
    public synchronized int getFontSize(){ return fontSize; }
    public synchronized void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public synchronized Color getTextColor(){ return tColor;}
    public synchronized void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // Éste es un escucha de multidifusión, que se usa más
    // a menudo que la solución unidifusión empleada en BangBean.java:
    public synchronized void
    addActionListener(ActionListener l) {
        actionListeners.add(l);
    }
    public synchronized void
    removeActionListener(ActionListener l) {
        actionListeners.remove(l);
    }
    // Observe que esto no está sincronizado:
    public void notifyListeners() {
        ActionEvent a = new ActionEvent(BangBean2.this,
            ActionEvent.ACTION_PERFORMED, null);
        ArrayList<ActionListener> lv = null;
        // Hacer una copia somera de la lista por si alguien
        // añade un escucha mientras estamos
        // invocando los escuchas:
        synchronized(this) {
            lv = new ArrayList<ActionListener>(actionListeners);
        }
        // Invocar todos los métodos escucha:
        for(ActionListener al : lv)
            al.actionPerformed(a);
    }
    class MI extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(tColor);
            g.setFont(
                new Font("TimesRoman", Font.BOLD, fontSize));
            int width = g.getFontMetrics().stringWidth(text);
            g.drawString(text, (getSize().width - width) /2,
                getSize().height/2);
            g.dispose();
            notifyListeners();
        }
    }
}

```

```

}

class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
        xm = e.getX();
        ym = e.getY();
        repaint();
    }
}

public static void main(String[] args) {
    BangBean2 bb2 = new BangBean2();
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("ActionEvent" + e);
        }
    });
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("BangBean2 action");
        }
    });
    bb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("More action");
        }
    });
    JFrame frame = new JFrame();
    frame.add(bb2);
    run(frame, 300, 300);
}
} //:-
```

Añadir la palabra clave **synchronized** a los métodos es un cambio sencillo. Sin embargo, observe en **addActionListener()** y **removeActionListener()** que los escuchas **ActionListener** se añaden y eliminan ahora en un contenedor **ArrayList**, por lo que podemos tener tantos como queramos.

Podemos ver que el método **notifyListeners()** no está sincronizado. Este método puede ser invocado desde más de una hebra a la vez. También resulta posible invocar **addActionListener()** o **removeActionListener()** en mitad de una llamada a **notifyListeners()**, lo que constituye un problema, porque este último método recorre el contenedor **actionListeners** de tipo **ArrayList**. Para aliviar el problema, el contenedor **ArrayList** se clona dentro de una cláusula **synchronized**, y lo que se hace es recorrer el clon (consulte los suplementos (en inglés) en línea de este libro para conocer más detalles sobre la clonación). De esta forma, podemos manipular el contenedor **ArrayList** original sin que ello suponga ningún impacto sobre **notifyListeners()**.

El método **paintComponent()** tampoco está sincronizado. Decidir si sincronizar los métodos sustituidos no resulta tan claro como cuando estamos añadiendo nuestros propios métodos. En este ejemplo, resulta que **paintComponent()** parece funcionar correctamente independientemente de si se sincroniza o no. Sin embargo, las cuestiones que hay que tener en cuenta son las siguientes:

1. ¿Modifica el método del estado de las variables “críticas” dentro del objeto? Para descubrir si las variables son “críticas”, hay que determinar si esas variables serán leídas o escritas por otras hebras del programa (en este caso, la lectura o escritura se hace casi siempre a través de métodos sincronizados, por lo que nos podemos limitar a examinar esos métodos). En el caso de **paintComponent()**, no se realiza ninguna modificación.
2. ¿Depende el método del estado de estas variable “críticas”? Si un método sincronizado modifica una variable que nuestro método utilice, entonces conviene hacer que nuestro método también esté sincronizado. Basándonos en esto, podemos observar que **eSize** es modificado por métodos sincronizados y, por tanto, **paintComponent()** debería estar sincronizado. Sin embargo, en este caso, podemos preguntarnos: “¿Qué es lo peor que puede suceder si se cambia **eSize** durante la ejecución de **paintComponent()**?” Si la respuesta a esta pregunta es que no puede suceder nada catastrófico, y que no sucede más que un efecto transitorio, debemos decidir dejar sin sincronizar **paintComponent()** para evitar el coste adicional asociado a la llamada al método sincronizado.

3. Una tercera clave consiste en analizar si la versión de la clase base de `paintComponent()` está sincronizada, lo que no es así. Este elemento no tiene mucho peso, pero sí que nos proporciona una pista. En este caso, por ejemplo, un campo que *sí* que se cambia mediante métodos sincronizados (`cSize`) se ha mezclado en la fórmula de `paintComponent()` y podría haber modificado nuestras conclusiones. Sin embargo, observe que el carácter de sincronizado no se hereda, es decir, si un método está sincronizado en la clase base, *no* se sincroniza automáticamente en la versión sustituida de la clase derivada.
4. `paint()` y `paintComponent()` son métodos que deben ser lo más rápidos posible. Cualquier cosa que permita reducir el coste de procesamiento de estos métodos resultará altamente recomendable, por lo que si cree que necesita sincronizar estos métodos, eso será un indicador de que el diseño no es demasiado bueno.

El código de prueba en `main()` ha sido modificado con respecto al que se muestra en `BangBeanTest` para ilustrar las capacidades de multidifusión de `BangBean2` añadiendo escuchas adicionales.

Empaquetado de una Bean

Antes de poder incluir componentes JavaBean en un entorno IDE preparado para ese tipo de componentes, es necesario incluirlo en un contenedor Bean, que es un archivo JAR que incluye todas las clases Bean, junto con un archivo de “manifiesto” que dice: “Esto es una Bean”. Un archivo de manifiesto es simplemente un archivo de texto que se ajusta a un formato concreto. Para el componente `BangBean`, el archivo de manifiesto tendría el aspecto siguiente:

```
Manifest-Version: 1.0
Name: bangbean/BangBean.class
Java-Bean: True
```

La primera linea indica la versión del esquema de manifiesto, que será la 1.0 en tanto que no se produzca una modificación de Sun en sentido contrario. La segunda línea (las líneas vacías se ignoran) proporciona el nombre del archivo `BangBean.class`, y la tercera dice: “Esto es una Bean”. Sin la tercera línea, la herramienta de construcción de programas no podría reconocer esa clase como una Bean.

La única parte complicada es que tenemos que asegurarnos de incluir la ruta adecuada en el campo “Name:”. Si volvemos a examinar `BangBean.java`, veremos que se encuentra en el paquete `bangbean` (y por tanto en un subdirectorio denominado `bangbean` que está fuera de la ruta de clases), y el nombre del archivo de manifiesto deberá incluir esta información de paquete. Además, es necesario colocar el archivo de manifiesto en el directorio situado *encima* de la raíz de la ruta del paquete, lo que en este caso significa colocar el archivo en el directorio situado encima del subdirectorio “bangbean”. Entonces, deberemos invocar `jar` desde el mismo directorio en el que se encuentre el archivo de manifiesto, de la forma siguiente:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Esto presupone que queremos que el archivo JAR resultante se denomine `BangBean.jar`, y que hemos colocado el archivo de manifiesto en un archivo denominado `BangBean.mf`.

Podríamos preguntarnos, “¿Qué sucede con las demás clases que se generaron en el momento de compilar `BangBean.java`?”. Todas las clases han terminado incluidas dentro del subdirectorío `bangbean`, y como puede ver, el último argumento de la línea de comandos `jar` anterior es un subdirectorío `bangbean`. Cuando se da a `jar` el nombre de un subdirectorío, la herramienta empaqueta dicho subdirectorío completo dentro del archivo JAR (incluyendo, en este caso, el archivo de código fuente original `BangBean.java`; no podemos incluir el código fuente con nuestros propios componentes Beans). Además, si invertimos el proceso y desempaquetamos un archivo JAR recién creado, descubriremos que el archivo de manifiesto no se encuentra en su interior, sino que `jar` ha creado su propio archivo de manifiesto (basado parcialmente en el nuestro) denominado `MANIFEST.MF` y colocado dentro del subdirectorío `META-INF` (“meta-information”). Si abre este archivo de manifiesto, verá también que `jar` ha añadido información de firma digital para cada archivo, de la forma:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/00=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

En general, no tenemos que preocuparnos por nada de esto, y si realizamos modificaciones, basta con cambiar nuestro archivo de manifiesto original y volver a invocar `jar` para crear un nuevo archivo JAR para nuestra Bean. También podemos añadir otros componentes Bean al archivo JAR simplemente añadiendo la información correspondiente al manifiesto.

Un aspecto que hay que resaltar es que normalmente conviene colocar cada Bean en su propio subdirectorio, ya que en el momento de crear un archivo JAR le entregamos a la utilidad **jar** el nombre de un subdirectorío y esta utilidad se encarga de colocar todo lo que ese subdirectorío contenga dentro del archivo JAR. Podrá observar que tanto **Frog** como **BangBean** se encuentran en sus propios subdirectorios.

Una vez que hemos incluido adecuadamente nuestra Bean dentro de un archivo JAR, podemos integrarla dentro de un entorno IDE de desarrollo con componentes Bean. La forma de hacerlo varía de una herramienta a otra, pero Sun proporciona una herramienta de prueba gratuita para JavaBeans, denominada "Bean Builder" (puede descargarla en <http://java.sun.com/beans>). Podemos insertar una Bean dentro de Bean Builder simplemente copiando el archivo JAR en el subdirectorío correcto.

Ejercicio 36: (4) Añada **Frog.class** al archivo de manifiesto de esta sección y ejecute **jar** para crear un archivo JAR que contenga tanto **Frog** como **BangBean**. Ahora, descargue e instale la herramienta Bean Builder de Sun, o utilice su propia herramienta de construcción de programas con Beans y añada el archivo JAR al entorno, para poder probar los dos componentes Bean.

Ejercicio 37: (5) Cree su propia JavaBean denominada **Valve** que contenga dos propiedades: un valor de tipo **boolean** denominado "on" y un valor **int** denominado "level". Cree un archivo de manifiesto, utilice **jar** para empaquetar la Bean, y luego cargue Bean Builder o alguna herramienta de construcción de programas basada en Bean para poder probar el componente.

Soporte avanzado de componentes Bean

Podemos ver lo fácil que resulta construir una Bean, pero en realidad no estamos limitados a las operaciones que se han descrito aquí. La arquitectura JavaBeans proporciona un punto de entrada muy simple para poder aprender los fundamentos, pero también permite escalar las soluciones para adaptarlas para situaciones más complejas. Dichas situaciones quedan fuera del alcance de este libro, pero las vamos a presentar aquí de forma breve. Puede encontrar más información en <http://java.sun.com/beans>.

Un aspecto en el que se puede añadir sofisticación es el relativo a las propiedades. Los ejemplos que hemos visto hasta ahora mostraban únicamente propiedades simples, pero también es posible representar múltiples propiedades mediante una matriz. Esto es lo que se denomina *propiedades indexadas*. Simplemente basta con proporcionar los métodos apropiados (que de nuevo deberán ajustarse a un convenio de denominación para los nombres de métodos) e **Introspector** reconocerá las propiedades indexadas para que el entorno IDE pueda responder adecuadamente.

Las propiedades puedan estar *acopladas*, lo que significa que enviarán notificaciones a otros objetos mediante un suceso **PropertyChangeEvent**. Los otros objetos pueden entonces decidir modificarse a sí mismos basándose en el cambio sufrido por la Bean.

Las propiedades pueden estar *restringidas*, lo que significa que otros objetos pueden vetar una cierta modificación de la propiedad si ésta resulta inaceptable. Los restantes objetos reciben una notificación utilizando un suceso **PropertyChangeEvent**, y pueden generar una excepción **PropertyVetoException** para impedir que ese cambio tenga lugar y para restaurar los antiguos valores.

También se puede modificar la forma en que la Bean está representada en tiempo de diseño:

1. Podemos proporcionar una hoja de propiedades personalizada para nuestra Bean concreta. La hoja de propiedades normal se empleará para todos los restantes componentes Bean, pero cuando se seleccione nuestra Bean se invocará automáticamente la hoja personalizada.
2. Podemos crear un editor personalizado para una propiedad concreta, de modo que se utilice la hoja de propiedades normal, pero que cuando se edite la propiedad especial, se invoque automáticamente el editor especificado.
3. Podemos proporcionar una clase **BeanInfo** personalizada para nuestra Bean que genere información distinta de la clase predeterminada creada por **Introspector**.
4. También es posible activar y desactivar el modo "experto" en todos los descriptores **FeatureDescriptor**, para distinguir entre características básicas y otras más complicadas.

Más información sobre componentes Bean

Hay disponibles varios libros acerca de JavaBeans; por ejemplo, *JavaBeans* de Elliotte Rusty Harold (IDG, 1998).

Alternativas a Swing

Aunque la biblioteca Swing es la GUI recomendada por Sun, no es en modo alguno la única forma de crear interfaces gráficas de usuario. Dos alternativas importantes son *Macromedia Flash*, que usa el sistema de programación *Flex*, para interfaces GUI del lado del cliente a través de la Web, y la biblioteca de código abierto SWT (*Standard Widget Toolkit*) de Eclipse para aplicaciones de escritorio.

¿Por qué merecería la pena considerar otras alternativas? Para los clientes web, podemos sostener con cierta rotundidad que los *applets* han fallado. Considerando el tiempo que ha transcurrido desde que aparecieron (desde el principio de Java) y todas las promesas y expectativas que levantaron, y sigue siendo una sorpresa encontrarse con una aplicación web basada en *applets*, ni siquiera Sun usa *applets* en todas partes. He aquí un ejemplo:

<http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html>

Un mapa interactivo de las características de Java en el sitio de Sun parecería un candidato bastante apropiado para construir un *applet* Java, a pesar de lo cual han construido este papel interactivo en Flash. Esto parece ser un reconocimiento tácito de que las *applets* no han sido precisamente un éxito. Además, el reproductor Flash Player está instalado en más del 98 por ciento de las plataformas informáticas, por lo que cabe considerarlo como un estándar de facto. Como veremos, el sistema Flex proporciona un entorno de programación del lado del cliente muy potente, ciertamente más potente que JavaScript y con un aspecto y estilo que resultan a menudo preferibles a los de un *applet*. Si queremos emplear *applets*, debemos seguir convenciendo al cliente de que descargue el entorno JRE, mientras que Flash Player es de pequeño tamaño y se descarga muy rápidamente.

Para aplicaciones de escritorio, un problema con Swing es que los usuarios *se dan cuenta* de que están utilizando un tipo de aplicación distinto, porque el aspecto y estilo de las aplicaciones Swing es diferente del que tiene el escritorio normal. Los usuarios no están, generalmente, interesados en nuevos aspectos y estilos de aplicaciones, lo que quieren es realizar su trabajo y prefieren que el aspecto de una aplicación se asemeje al de las restantes aplicaciones. SWT crea aplicaciones que se asemejan a las aplicaciones nativas, y como la biblioteca utiliza componentes nativos siempre que es posible, las aplicaciones tienden a ejecutarse más rápidamente que las aplicaciones equivalentes Swing.

Construcción de clientes web Flash con Flex

Debido a la ubicuidad de la máquina virtual ligera Flash de Macromedia, la mayor parte de las personas podrán utilizar una interfaz basada en Flash sin tener que instalar nada, y esa interfaz tendrá el mismo aspecto y se comportará de la misma forma en todos los sistemas y plataformas.¹⁰

Con *Macromedia Flex*, podemos desarrollar interfaces de usuario Flash para aplicaciones Java. Flex consiste en un modelo de programación basado en XML y en *scripts*, similar a los modelos de programación basados en HTML y JavaScript, junto con una robusta biblioteca de componentes. Se emplea la sintaxis MXML para declarar la gestión de disposición y los controles de *widget* (componentes), y se usan *scripts* dinámicos para añadir mecanismos de tratamiento de sucesos y código de invocación de servicios que enlazan la interfaz de usuario con clases Java, modelos de datos, servicios web, etc. El compilador toma los archivos MXML y de *script* y los compila para generar código intermedio. La máquina virtual Flash en el cliente opera como la máquina virtual Java, en el sentido de que interpreta código intermedio compilado. El formato del código intermedio se conoce como SWF, y el compilador Flex genera archivos SWF.

Observe que existe una alternativa de código abierto a Flex disponible en <http://openlaszlo.org>; esta alternativa tiene una estructura similar a la de Flex, pero puede resultar preferible para algunas personas. Existen también otras herramientas para crear aplicaciones Flash de diferentes formas.

Hello, Flex

Considere el siguiente fragmento de código MXML, que define una interfaz de usuario (observe que la primera y la última líneas no aparecerán dentro del código que descargue como parte del código fuente de este libro):

¹⁰ Sean Neville ha creado una parte fundamental del material de esta sección.

```
//:! gui/flex/helloflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#ffffff">
    <mx:Label id="output" text="Hello, Flex!" />
</mx:Application>
//:-
```

Los archivos MXML son documentos XML, por lo que comienzan con una directiva XML de versión/codificación. El elemento MXML más externo es el elemento **Application**, que es el contenedor visual y lógico de mayor nivel para una interfaz de usuario Flex. Podemos declarar marcadores que presenten controles visuales, como por ejemplo la etiqueta **Label** del ejemplo anterior, dentro del elemento **Application**. Los controles se incluyen siempre dentro de un contenedor, y los contenedores encapsulan gestores de disposición, entre otros mecanismos para poder gestionar la disposición de los controles incluidos en ellos. En el caso más simple, como en el ejemplo anterior, **Application** actúa como el contenedor. El gestor de dispositivo predeterminado de **Application** se limita a colocar los controles verticalmente en la interfaz en el orden que hayan sido declarados.

ActionScript es una versión de ECMAScript, o JavaScript, que parece muy similar a Java y soporta clases y mecanismos fuertes de tipado, además de mecanismos de *script* dinámico. Añadiendo un *script* al ejemplo, podemos introducir un cierto comportamiento. Aquí, se utiliza el control MXML **Script** para incluir código ActionScript directamente dentro del archivo MXML:

```
//:! gui/flex/helloflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#ffffff">
    <mx:Script>
        <![CDATA[
            function updateOutput() {
                output.text = "Hello! " + input.text;
            }
        ]]>
    </mx:Script>
    <mx:TextInput id="input" width="200"
        change="updateOutput()" />
    <mx:Label id="output" text="Hello!" />
</mx:Application>
//:-
```

Un control **TextInput** acepta la entrada de usuario y un control **Label** muestra los datos a medida que se los escribe. Observe que el atributo **id** de cada control está accesible dentro del *script* en forma de nombre de variable, por lo que el *script* puede hacer referencias a instancias de los marcadores MXML. En el campo **TextInput**, podemos ver que el atributo **change** está conectado a la función **updateOutput()** de modo que se invoca la función cada vez que se produce cualquier tipo de cambio.

Compilación de MXML

La forma más fácil de comenzar a trabajar con Flex es con la versión gratuita de prueba, que se puede descargar en www.macromedia.com/software/flex/trial.¹¹ El producto está empaquetado en una serie de ediciones, desde versiones de prueba gratuitas hasta versiones de servidor empresarial, y Macromedia ofrece herramientas adicionales para el desarrollo de aplicaciones Flex. El empaquetado exacto de las distintas versiones está sujeto a cambios, por lo que deberá comprobar el sitio de Macromedia para conocer los detalles específicos. Observe también que puede que necesite modificar el archivo **jvm.config** situado en el directorio **bin** de la instalación de Flex.

¹¹ Observe que es preciso descargar Flex y no FlexBuilder. Esta última es una herramienta de diseño IDE.

Para compilar el código MXML y generar código intermedio Flash, tenemos dos opciones:

1. Podemos insertar el archivo MXML en una aplicación web Java, junto con páginas JSP y HTML en un archivo WAR, y hacer que las solicitudes del archivo **.mxml** se compilen en tiempo de ejecución cada vez que un explorador solicite la URL del documento MXML.
2. Podemos compilar el archivo MXML utilizando el compilador de la línea de comandos Flex, **mxmlc**.

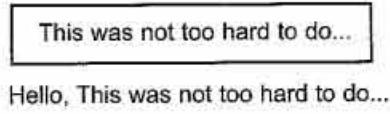
La primera opción, la de la compilación en tiempo de ejecución basada en la Web, requiere un contenedor de *servlet* (como Apache Tomcat) además de Flex. El archivo WAR del contenedor *servlet* debe actualizarse con la información de configuración de Flex, como por ejemplo los mapeos de *servlet* que se añaden al descriptor **web.xml**, y debe incluir los archivos JAR de Flex, todos estos pasos se gestionan automáticamente cuando se instala Flex. Después de configurado el archivo WAR, podemos insertar los archivos MXML en la aplicación Web y solicitar la URL del documento mediante cualquier explorador. Flex compilará la aplicación cuando se reciba la primera solicitud, de forma similar a la que sucede en el modelo JSP, y en lo sucesivo suministrará el código SWF compilado y almacenado en caché dentro de un envoltorio HTML.

La segunda opción no requiere un servidor. Cuando se invoca el compilador **mxmlc** de Flex en la línea de comandos, se generan archivos SWF. Podemos implantar estos archivos de la forma que deseemos. El ejecutable **mxmlc** está ubicado en el directorio **bin** de la instalación Flex, y si lo invocamos sin ningún argumento nos proporcionará una lista de las opciones válidas de línea de comandos. Normalmente, lo que haremos será especificar la ubicación de la biblioteca de componentes de cliente Flex como valor de la opción de la línea de comandos **flexlib**, pero en algunos ejemplos muy simples como los dos que hemos visto hasta ahora, el compilador Flex presupondrá la ubicación de la biblioteca de componentes. Por tanto, podemos compilar los dos primeros ejemplos de la forma siguiente:

```
mxmlc.exe helloflex1.mxml  
mxmlc.exe helloflex2.mxml
```

Esto genera un archivo **helloflex2.swf** que se puede ejecutar en Flash, o que se puede insertar junto con código HTML en cualquier servidor HTTP (una vez que Flash haya sido cargado en el explorador web, a menudo basta con hacer doble clic sobre el archivo SWF para que éste se inicie en el explorador).

Para **helloflex2.swf**, veremos la siguiente interfaz de usuario ejecutándose en Flash Player:



En aplicaciones más complejas, podemos separar el código MXML y ActionScript haciendo referencia a funciones en archivos ActionScript externos. Desde MXML, se utiliza la siguiente sintaxis para el control **Script**:

```
<mx:Script source="MyExternalScript.as" />
```

Este código permite a los controles MXML hacer referencia a funciones ubicadas en un archivo denominado **MyExternalScript.as** como si se encontraran en el propio archivo MXML.

MXML y ActionScript

MXML es una especie de abreviatura declarativa para las clases ActionScript. Cada vez que vemos un marcador MXML, existe una clase ActionScript del mismo nombre. Cuando el compilador Flex analiza el código MXML, primero transforma el código XML en código ActionScript y carga las clases ActionScript referenciadas, después de lo cual compila y monta el código ActionScript para crear un archivo SWF.

Podemos escribir una aplicación Flex completa utilizando exclusivamente ActionScript, sin usar nada de MXML. Por tanto, MXML es simplemente un lenguaje de utilidad. Los componentes de interfaz de usuario como los contenedores y controles, se declaran típicamente utilizando MXML, mientras que la lógica asociada, como por ejemplo las rutinas de tratamiento de sucesos y el resto de la lógica de cliente se gestionan mediante ActionScript y Java.

Podemos crear nuestros propios controles MXML y hacer referencia a ellos utilizando MXML, escribiendo clases ActionScript. También podemos combinar contenedores y controles MXML existentes en un nuevo documento MXML al que luego podamos hacer referencia mediante un marcador en otro documento MXML. El sitio web de Macromedia contiene más información acerca de cómo hacer esto.

Contenedores y controles

El núcleo visual de la biblioteca de componentes Flex es un conjunto de contenedores que gestionan la disposición de los elementos y una matriz de controles que se inserta en esos contenedores. Entre los contenedores se incluyen paneles, recuadros verticales y horizontales, mosaicos, acordeones, recuadros divididos, cuadriculas y otros. Los controles son elementos de la interfaz de usuario, como por ejemplo, botones, áreas de texto, deslizadores, calendarios, cuadriculas de datos, etc.

En el resto de esta sección vamos a mostrar una aplicación Flex que muestra y ordena una lista de archivos de audio. Esta aplicación ilustra el uso de contenedores y de controles y muestra cómo conectar Java desde Flash.

Comenzamos el archivo MXML colocando un control **DataGrid** (uno de los controles más sofisticados de Flex) dentro de un contenedor de tipo **Panel**:

```
//:! gui/flex/songs.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.macromedia.com/2003/mxml"
    backgroundColor="#B9CAD2" pageTitle="Flex Song Manager"
    initialize="getSongs()"
    >
    <mx:Script source="songScript.as" />
    <mx:Style source="songStyles.css"/>
    <mx:Panel id="songListPanel"
        titleStyleDeclaration="headerText"
        title="Flex MP3 Library">
        <mx:HBox verticalAlign="bottom">
            <mx:DataGrid id="songGrid"
                cellPress="selectSong(event)" rowCount="8">
                <mx:columns>
                    <mx:Array>
                        <mx:DataGridColumn columnName="name"
                            headerText="Song Name" width="120" />
                        <mx:DataGridColumn columnName="artist"
                            headerText="Artist" width="180" />
                        <mx:DataGridColumn columnName="album"
                            headerText="Album" width="160" />
                    </mx:Array>
                </mx:columns>
            </mx:DataGrid>
            <mx:VBox>
                <mx:HBox height="100" >
                    <mx:Image id="albumImage" source=""
                        height="80" width="100"
                        mouseOverEffect="resizeBig"
                        mouseOutEffect="resizeSmall" />
                    <mx:TextArea id="songInfo"
                        styleName="boldText" height="100%" width="120"
                        vScrollPolicy="off" borderStyle="none" />
                </mx:HBox>
                <mx:MediaPlayback id="songPlayer"
                    contentPath=""
                    mediaType="MP3"
                    height="70"
                    width="230"
                    controllerPolicy="on"
                    autoPlay="false"
                    visible="false" />
            </mx:VBox>
        </mx:HBox>
        <mx:ControlBar horizontalAlign="right">
            <mx:Button id="refreshSongsButton"

```

```

        label="Refresh Songs" width="100"
        toolTip="Refresh Song List"
        click="songService.getSongs()" />
    </mx:ControlBar>
</mx:Panel>
<mx:Effect>
    <mx:Resize name="resizeBig" heightTo="100"
        duration="500"/>
    <mx:Resize name="resizeSmall" heightTo="80"
        duration="500"/>
</mx:Effect>
<mx:RemoteObject id="songService"
    source="gui.flex.SongService"
    result="onSongs(event.result)"
    fault="alert(event.fault.faultstring, 'Error')">
    <mx:method name="getSongs"/>
</mx:RemoteObject>
</mx:Application>
//:-

```

El control **DataGrid** contiene marcadores anidados para su matriz de columnas. Cuando vemos un atributo o un elemento anidado de un control, sabemos que se corresponde con alguna propiedad, sucesos u objeto encapsulado dentro de la clase ActionScript subyacente. El control **DataGrid** tiene un atributo **id** con el valor **songGrid**, por lo que ActionScript y los marcadores MXML pueden hacer referencia a esa cuadrícula de datos mediante programa empleando **songGrid** como nombre de variable. La cuadrícula de datos **DataGrid** expone muchas más propiedades que las que se muestran aquí; puede encontrar la API completa para los controles contenidos MXML en la dirección http://livedocs.macromedia.com/flex/15/asdocs_en/index.html.

El control **DataGrid** está seguido de un control **VBox** que contiene una imagen **Image** para mostrar la carátula del álbum junto con la información acerca de la canción, y un control **MediaPlayback** que permite reproducir archivos MP3. En este ejemplo se descarga el flujo de contenido con el fin de reducir el tamaño del archivo SWF compilado. Cuando se incluyen imágenes o archivos de audio y vídeo en una aplicación Flex, en lugar de descargar el flujo de datos correspondiente, los archivos pasan a formar parte del archivo SWF compilado y se suministran junto con la interfaz de usuario, en lugar de descargarse bajo demanda en tiempo de ejecución.

El reproductor Flash Player contiene *codecs* integrados para reproducir y descargar audio y vídeo en una diversidad de formatos. Flash y Flex soportan el uso de los formatos de imagen más comunes de la Web, y Flex tiene también la posibilidad de traducir archivos SVG (*scalable vector graphics*) a recursos SWF que pueden integrarse en los clientes Flex.

Efectos y estilos

El reproductor Flash Player muestra los gráficos utilizando tecnología vectorial, así que puede realizar transformaciones altamente expresivas en tiempo de ejecución. Los *efectos* Flex proporcionan una pequeña muestra de este tipo de animaciones. Los efectos son transformaciones que pueden aplicarse a los controles y contenidos utilizando sintaxis MXML.

El marcador **Effect** mostrado en el código MXML produce dos resultados: el primer marcador anidado hace crecer dinámicamente una imagen cuando se desplaza el ratón sobre él, mientras que el segundo contrae dinámicamente dicha imagen cuando el ratón se aleja. Estos efectos se aplican a los sucesos de ratón disponibles en el control **Image** para **albumImage**.

Flex también proporciona efectos para animaciones comunes como transiciones, cortinillas y canales alfa modulados. Además de los efectos predefinidos, Flex soporta la API de dibujo de Flash para la definición de animaciones verdaderamente innovadoras. Una exploración detallada de este tema implicaría muchos conceptos de diseño gráfico y animación, y esto queda fuera del alcance de esta sección.

La utilización de estilos estándar es posible gracias al soporte que Flex proporciona para la especificación CSS (*Cascading Style Sheets*, hojas de estilo en cascada). Si asociamos un archivo CSS a un archivo MXML, los controles Flex se adaptarán a esos estilos. Para este ejemplo, **songStyles.css** contiene la siguiente declaración CSS:

```

//:! gui/flex/songStyles.css
.headerText {

```

```

font-family: Arial, "_sans";
font-size: 16;
font-weight: bold;
}

.boldText {
font-family: Arial, "_sans";
font-size: 11;
font-weight: bold;
}
//:-

```

Este archivo se importa y se emplea en la aplicación de la biblioteca de canciones a través del marcador **Style** en el archivo MXML. Después de importada la hoja de estilo, sus declaraciones pueden aplicarse a los controles Flex en el archivo MXML. Como ejemplo, el control **TextArea** utiliza la declaración **boldText** de la hoja de estilo con **songInfo id**.

Sucesos

Una interfaz de usuario es una máquina de estados; realiza diversas acciones a medida que se producen cambios de estado. En Flex, estos cambios se gestionan mediante sucesos. La biblioteca de clases Flex contiene una amplia variedad de controles con numerosos sucesos que cubren todos los aspectos de movimiento del ratón y de la utilización del teclado.

El atributo **click** de un botón **Button**, por ejemplo, representa uno de los sucesos disponibles en dicho control. El valor asignado a **click** puede ser una función o un pequeño *script* integrado. En el archivo MXML, por ejemplo, el control **ControlBar** incluye el botón **refreshSongsButton** para refrescar la lista de canciones. Puede ver, analizando el marcador, que cuando se produce el suceso **click** se invoca **songService.getSongs()**. En este ejemplo, el suceso **click** del control **Button** hace referencia al objeto **RemoteObject** que se corresponde con el método Java.

Conexión con Java

El marcador **RemoteObject** situado al final del archivo MXML establece la conexión con la clase Java externa **gui.flex.SongService**. El cliente Flex utilizará el método **getSongs()** de la clase Java para recuperar los datos con los que rellenar la cuadrícula **DataGrid**. Para hacer esto, es necesario que la clase externa aparezca como un *servicio*, es decir, como un interlocutor con el que los clientes puedan intercambiar mensajes. El servicio definido en el marcador **RemoteObject** tiene un atributo **source** que indica la clase Java del objeto **RemoteObject**, y especifica una función de retrollamada ActionScript, **onSongs()**, que hay que invocar cuando se vuelve del método Java. El marcador **method** anidado declara el método **getSongs()**, que hace que ese método Java esté accesible para el resto de la aplicación Flex.

Todas las invocaciones de servicios en Flex vuelven asincrónamente mediante sucesos disparados hacia esas funciones de retrollamada. El objeto **RemoteObject** hace que se muestre un control de cuadro de diálogo y alerta, en caso de que se produzca un error.

Ahora podemos invocar el método **getSongs()** desde Flash utilizando ActionScript:

```
songService.getSongs();
```

Debido a la configuración de MXML, esto hará que se invoque **getSongs()** en la clase **SongService**:

```

//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
    private List<Song> songs = new ArrayList<Song>();
    public SongService() { fillTestData(); }
    public List<Song> getSongs() { return songs; }
    public void addSong(Song song) { songs.add(song); }
    public void removeSong(Song song) { songs.remove(song); }
    private void fillTestData() {
        addSong(new Song("Chocolate", "Snow Patrol",

```

```

        "Final Straw", "sp-final-straw.jpg",
        "chocolate.mp3"));
    addSong(new Song("Concerto No. 2 in E", "Hilary Hahn",
        "Bach: Violin Concertos", "hahn.jpg",
        "bachviolin2.mp3"));
    addSong(new Song("Round Midnight", "Wes Montgomery",
        "The Artistry of Wes Montgomery",
        "wesmontgomery.jpg", "roundmidnight.mp3"));
}
} //:-
```

Cada objeto **Song** es simplemente un contenedor de datos:

```

//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
    private String name;
    private String artist;
    private String album;
    private String albumImageUrl;
    private String songMediaUrl;
    public Song() {}
    public Song(String name, String artist, String album,
    String albumImageUrl, String songMediaUrl) {
        this.name = name;
        this.artist = artist;
        this.album = album;
        this.albumImageUrl = albumImageUrl;
        this.songMediaUrl = songMediaUrl;
    }
    public void setAlbum(String album) { this.album = album; }
    public String getAlbum() { return album; }
    public void setAlbumImageUrl(String albumImageUrl) {
        this.albumImageUrl = albumImageUrl;
    }
    public String getAlbumImageUrl() { return albumImageUrl; }
    public void setArtist(String artist) {
        this.artist = artist;
    }
    public String getArtist() { return artist; }
    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setSongMediaUrl(String songMediaUrl) {
        this.songMediaUrl = songMediaUrl;
    }
    public String getSongMediaUrl() { return songMediaUrl; }
} //:-
```

Cuando se inicializa la aplicación o se pulsa un botón **refreshSongsButton**, se invoca **getSongs()** y, al volver del método, se llama al método **onSongs(event.result)** de ActionScript para llenar la cuadrícula **songGrid**.

He aquí el listado de código ActionScript, que está incluido dentro del control **Script** del archivo MXML:

```

//: gui/flex/songScript.as
function getSongs() {
    songService.getSongs();
}

function selectSong(event) {
    var song = songGrid.getItemAt(event.itemIndex);
    showSongInfo(song);
```

```

}

function showSongInfo(song) {
    songInfo.text = song.name + newline;
    songInfo.text += song.artist + newline;
    songInfo.text += song.album + newline;
    albumImage.source = song.albumImageUrl;
    songPlayer.contentPath = song.songMediaUrl;
    songPlayer.visible = true;
}

function onSongs(songs) {
    songGrid.dataProvider = songs;
} //:-

```

Para gestionar la selección de celdas de la cuadricula de datos **DataGrid**, añadimos el atributo de sucesos **cellPress** a la declaración **DataGrid** en el archivo MXML:

```
cellPress="selectSong(event)"
```

Cuando el usuario hace clic sobre una canción en la cuadricula de datos **DataGrid**, se invoca **selectSong()** en el código ActionScript anterior.

Modelos de datos y acoplamiento de datos

Los controles pueden invocar directamente servicios, y las retrolllamadas de suceso de ActionScript nos dan la posibilidad de actualizar mediante programa los controles visuales cada vez que los servicios devuelven datos. Aunque el *script* que actualiza los controles es bastante sencillo, puede resultar ser bastante largo y engorroso, y su funcionalidad es tan común que Flex gestiona el comportamiento automáticamente, con acoplamiento de datos.

En su forma más simple, el acoplamiento de datos permite a los controles hacer referencia directa a los datos, en lugar de requerir código de conexión para copiar los datos en un control. Cuando se actualizan los datos, también se actualiza automáticamente que hace referencia a los mismos sin necesidad de intervención del programador. La infraestructura de Flex responde correctamente a los sucesos de cambio de los datos y actualiza todos los controles que estén acoplados a ellos.

He aquí un ejemplo simple de sintaxis de acoplamiento de datos:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}">
```

Para realizar el acoplamiento de los datos, incluimos las referencias dentro de llaves: {}. Todo lo que se encuentre dentro de esas llaves se considera una expresión que Flex debe evaluar.

El valor del primer control, que es un *widget* de tipo **Slider**, se visualiza mediante el segundo control, que es un campo **Text**. A medida que el control **Slider** cambia, la propiedad **text** del campo **Text** se actualiza automáticamente. De esta forma, el desarrollador no necesita gestionar los sucesos de cambio del control **Slider** para actualizar el campo **Text**.

Algunos controles, como el control **Tree** y el control **DataGrid** de la aplicación de la biblioteca de canciones, son más sofisticados. Estos controles tienen una propiedad **dataProvider** para facilitar el acoplamiento a colecciones de datos. La función ActionScript **onSongs()** muestra cómo está acoplado el método **SongService.getSongs()** al proveedor de datos **dataProvider** del control Flex **DataGrid**. Tal como se declara en el marcador **RemoteObject** del archivo MXML, esta función es la retrollamada que ActionScript invoca cada vez que vuelve el método Java.

Una aplicación más sofisticada con un modelado de datos más complejo, como por ejemplo una aplicación empresarial que hiciera uso del estándar DTO (*Data Transfer Objects*) de objetos de transferencia de datos o una aplicación de mensajería con datos que tuvieran que adaptarse a esquemas complejos, podría desacoplar todavía más el origen de los datos de los controles. En el desarrollo con Flex, realizamos este desacoplamiento declarando un objeto “Modelo”, que es un contenedor MXML genérico para los datos. El modelo no contiene ninguna lógica. Se asemeja a los objetos DTO que podemos encontrar en las actividades de desarrollo empresarial y a las estructuras de otros lenguajes de programación. Utilizando este modelo, podemos acoplar nuestros controles al modelo, y hacer al mismo tiempo que el modelo acople sus propiedades con entradas y salidas de servicio. Esto hace que se desacoplen los orígenes de datos, los servicios, de los consumidores visua-

les de los datos, facilitando el uso del patrón *Modelo-Vista-Controlador* (MVC). En aplicaciones más sofisticadas y de mayor envergadura, la complejidad inicial provocada por la inserción de un modelo constituye una desventaja si la comparamos con el valor que nos proporciona una aplicación MVC limpiamente desacoplada.

Además de acceder a objetos Java, Flex también puede acceder a servicios web basados en SOAP y a servicios HTTP utilizando los controles **WebService** y **HttpService**, respectivamente. El acceso a todos los servicios está sujeto a restricciones de autorización por razones de seguridad.

Construcción e implantación de aplicaciones

Con los ejemplos anteriores, podíamos prescindir del indicador **-flexlib** en la línea de comandos, pero para compilar este programa, tenemos que especificar la ubicación del archivo **flex-config.xml** mediante el indicador **-flexlib**. En mi instalación, el comando que hay que utilizar es el siguiente, pero el lector tendrá que modificarlo de acuerdo con su propia configuración (el comando es una única línea, que se presenta en varias líneas debido a la limitación de la anchura del libro):

```
//:1 gui/flex/build-command.txt
mxmlc -flexlib C:/Program Files/Macromedia/Flex/jrun4/servers/default/flex/WEB-INF/flex
songs.mxml
///:-
```

Este comando construirá la aplicación generando un archivo SWF que podemos ver en nuestro explorador, pero el archivo de distribución de código del libro no contiene ningún archivo MP3 ni JPG, por lo que no verá nada salvo el marco contenido cuando ejecute la aplicación.

Además, deberá configurar un servidor para poder comunicarse adecuadamente con los archivos Java desde la aplicación Flex. El paquete de prueba de Flex incluye el servidor JRun, y podemos arrancar este servidor desde los menús de la computadora después de instalar Flex, o a través de la línea de comandos:

```
jrun -start default
```

Puede verificar que el servidor ha arrancado adecuadamente abriendo <http://localhost:8700/samples> en un explorador web y visualizando los diversos ejemplos (ésta es también una forma de familiarizarse con las capacidades de Flex).

En lugar de compilar la aplicación en la linea de comandos, podemos compilarla mediante el servidor. Para hacer esto, inserte los archivos fuente del ejemplo de las canciones, la hoja de CSS, etc., en el directorio **jrun4/servers/default/flex** y acceda a ellos desde un explorador abriendo <http://localhost:8700/flex/songs.mxml>.

Para ejecutar adecuadamente la aplicación, deberá configurar tanto el lado de Java como el lado de Flex.

Java: los archivos compilados **Song.java** y **SongService.java** deben colocarse en el directorio **WEB-INF/classes**. Ahí es donde deben incluirse las clases WAR de acuerdo con la especificación J2EE. Alternativamente, puede comprimir en un archivo JAR los archivos y colocar el resultado en **WEB-INF/lib**. Debe estar en un directorio que se ajuste a su estructura de paquetes Java. Si está usando, los archivos se colocarán en **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/Song.class** y **jrun4/servers/default/flex/WEB-INF/classes/gui/flex/SongService.class**. También necesitará los archivos de imagen y MP3 disponibles en la aplicación web (para JRun, **jrun4/servers/default/flex** es la raíz de la aplicación web).

Flex: por razones de seguridad, Flex no puede acceder a objetos Java a menos que le demos permiso modificando el **flex-config.xml**. Para JRun, este archivo se encuentra en **jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml**. Localice la entrada **<remote-objects>** en dicho archivo, y examine la sección **<whitelist>** incluida en ella y vea la siguiente nota:

```
<!--
For security, the whitelist is locked down by default. Uncomment the source element below to enable
access to all classes during development.
```

*We strongly recommend not allowing access to all source files in production, since this exposes Java
and Flex system classes.*

```
<source>*</source>
-->
```

Elimine el comentario de esa entrada **<source>** para permitir el acceso, de modo que quede **<source>*</source>**. El significado de ésta y otras entradas se describe en los documentos de configuración de Flex.

Ejercicio 38: (3) Construya el “ejemplo simple de sintaxis de acoplamiento de datos” mostrado anteriormente.

Ejercicio 39: (4) La descarga de código para este libro no incluye los archivos MP3 o JPG mostrados en **SongService.java**. Localice algunos archivos MP3 y JPG, modifique **SongService.java** para incluir los correspondientes nombres de archivo, descargue la versión de prueba de Flex y construya la aplicación.

Creación de aplicaciones SWT

Como hemos indicado anteriormente, Swing adoptó el enfoque de construir todos los componentes de la interfaz de usuario píxel por píxel, con el fin de proporcionar todos los componentes deseados independientemente de si el sistema operativo subyacente disponía de ellos o no. SWT adopta una postura intermedia, utilizando componentes nativos si el sistema operativo los proporciona, y sintetizando los componentes si no lo hace. El resultado es una aplicación que para el usuario se asemeja a una aplicación nativa, y que a menudo tiene la velocidad bastante superior a la del programa Swing equivalente. Además, SWT tiende a ser un modelo de programación menos complejo que Swing, lo cual puede resultar deseable en un gran número de aplicaciones.¹²

Puesto que SWT utiliza el sistema operativo nativo para realizar la mayor parte posible del trabajo, puede aprovechar automáticamente algunas de las características del sistema operativo que pueden no estar disponibles con Swing; por ejemplo, Windows tiene mecanismo de “representación subpíxel” que hace que las fuentes de caracteres parezcan más nítidas en las pantallas LCD.

Resulta incluso posible crear *applets* utilizando SWT.

Esta sección no pretende ser una introducción completa a SWT; simplemente se trata de proporcionar una panorámica de esta biblioteca y de comparar SWT con Swing. Descubrirá que existe una gran cantidad de *widgets* SWT y que todos ellos son bastante sencillos de utilizar. Puede analizar los detalles en la documentación completa y los muchos ejemplos que podrá encontrar en www.eclipse.org. También hay diversos libros de programación con SWT, y posiblemente aparezcan más en el futuro.

Instalación de SWT

Las aplicaciones SWT requieren que se descargue e instale la biblioteca SWT desarrollada en el proyecto Eclipse. Vaya a www.eclipse.org/downloads/ y seleccione uno de los sitios espejo. Siga los vínculos hasta localizar la versión Eclipse actual y localice un archivo comprimido con un nombre con “swt” e incluya el nombre de su plataforma (por ejemplo, “win32”). Dentro de este archivo encontrará **swt.jar**. La forma más fácil de instalar el archivo **swt.jar** consiste en colocarlo en el directorio **jre/lib/ext** (de esta forma, no tendrá que hacer ninguna modificación en su ruta de clases). Cuando descomprima la biblioteca SWT, puede que encuentre archivos adicionales que necesitará instalar en los lugares apropiados para su plataforma. Por ejemplo, la distribución Win32 incluye archivos DLL que tienen que incluirse en algún lugar de la ruta **java.library.path** (ésta coincide usualmente con la variable de entorno PATH, pero puede ejecutar **Object>ShowProperties.java** para descubrir el valor real de **java.library.path**). Una vez que haya hecho esto, debería poder compilar y ejecutar transparentemente la aplicación SWT como si fuera cualquier otro programa Java.

La documentación de SWT se encuentra en un archivo de descarga separado.

Una técnica alternativa consiste en limitarse a instalar el editor Eclipse, que incluye tanto SWT como la documentación de SWT que se puede visualizar a través del sistema de ayuda de Eclipse.

Hello, SWT

Comencemos con la aplicación más simple posible del estilo de la conocida aplicación “hello world”:

```
//: swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; You must
// install the SWT library from http://www.eclipse.org }
import org.eclipse.swt.widgets.*;
public class HelloSWT {
```

¹² Chris Grindstaff resultó de mucha ayuda a la hora de traducir los ejemplos a SWT y de proporcionar información acerca de SWT.

```

public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Hi there, SWT!"); // Barra de título
    shell.open();
    while(!shell.isDisposed())
        if(!display.readAndDispatch())
            display.sleep();
    display.dispose();
}
} //:-~
```

Si descarga el código fuente de este libro, descubrirá que la directiva de comentario “Requires” termina siendo incluida en el archivo **build.xml** de Ant como un pre-requisito para construir el subdirectorio **swt**; todos los archivos que importen **org.eclipse.swt** requieren que se instale la biblioteca SWT de www.eclipse.org.

La clase **Display** gestiona la conexión entre SWT y el sistema operativo subyacente; forma parte de un *Puente* entre el sistema operativo y SWT. La clase **Shell** es la ventana principal de nivel superior, dentro de la que se construyen todos los demás componentes. Cuando se invoca **setText()**, el argumento se convierte en la etiqueta que aparecerá en la barra de título de la ventana.

Para mostrar la ventana y luego la aplicación, debe invocar **open()** sobre el objeto **Shell**.

Mientras que Swing oculta a nuestros ojos el bucle de tratamiento de sucesos, SWT nos obliga a escribirlo explícitamente. En la parte superior del bucle, miramos si la *shell* ha sido eliminada; observe que esto nos proporciona la opción de insertar código para llevar a cabo actividades de limpieza. Pero esto quiere decir que la hebra **main()** es la hebra de la interfaz de usuario. En Swing, se crea de manera transparente una segunda hebra de despacho de sucesos, pero en SWT, es la hebra **main()** la que se encarga de gestionar la interfaz de usuario. Puesto que de manera predeterminada sólo existe una hebra y no dos, esto hace que sea algo menos probable que terminemos sobrecargando la interfaz de usuario con hebras.

Observe que no tenemos que preocuparnos de enviar tareas a la hebra de interfaz de usuario, a diferencia de lo que sucedía en Swing. SWT no sólo se encarga de esto por nosotros, sino que genera una excepción si tratamos de manipular un *widget* con la hebra errónea. Sin embargo, si necesitamos crear hebras para realizar operaciones de larga duración, seguimos necesitando enviar los cambios de la misma forma que se hace en Swing. Para esto, SWT proporciona tres métodos que pueden invocarse sobre el objeto **Display**: **asyncExec(Runnable)**, **syncExec(Runnable)** y **timerExec(int, Runnable)**.

La actividad de la hebra **main()** en este punto consiste en llamar a **readAndDispatch()** para el objeto **Display** (esto significa que sólo puede haber un objeto **Display** por cada aplicación). El método **readAndDispatch()** devuelve **true** si hay dos sucesos en la cola de sucesos esperando ser procesados. En dicho caso, hay que volver a invocar un método inmediatamente. Sin embargo, si no hay nada pendiente, invocamos el método **sleep()** del objeto **Display** para esperar durante un período breve de tiempo antes de volver a consultar la cola de sucesos.

Una vez completado el programa, es necesario eliminar explícitamente el objeto **Display** con **dispose()**. SWT requiere a menudo que eliminemos explícitamente los recursos no utilizados, porque se trata usualmente de recursos del sistema operativo subyacente, que podrían de otro modo agotarse.

Para demostrar que el objeto **Shell** es la ventana principal, he aquí un programa que construye una serie de objetos **Shell**:

```

//: swt/ShellsAreMainWindows.java
import org.eclipse.swt.widgets.*;

public class ShellsAreMainWindows {
    static Shell[] shells = new Shell[10];
    public static void main(String [] args) {
        Display display = new Display();
        for(int i = 0; i < shells.length; i++) {
            shells[i] = new Shell(display);
            shells[i].setText("Shell #" + i);
            shells[i].open();
        }
        while(!shellsDisposed())
```

```

        if(!display.readAndDispatch())
            display.sleep();
        display.dispose();
    }
    static boolean shellsDisposed() {
        for(int i = 0; i < shells.length; i++)
            if(shells[i].isDisposed())
                return true;
        return false;
    }
} //:~

```

Al ejecutarlo, se obtienen diez ventanas principales. De la forma en que se ha escrito el programa, si se cierra una ventana, se cerrarán todas.

SWT también emplea gestores de disposición, que son distintos a los de Swing, pero que están basados en la misma idea. He aquí un ejemplo ligeramente más complejo que toma el texto de `System.getProperties()` y lo añade a la `shell`:

```

//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
    public static void main(String [] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Display Properties");
        shell.setLayout(new FillLayout());
        Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
        StringWriter props = new StringWriter();
        System.getProperties().list(new PrintWriter(props));
        text.setText(props.toString());
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} //:~

```

En SWT, todos los *widgets* deben tener un objeto padre de tipo general **Composite**, y hay que proporcionar este padre como el primer argumento en el constructor de *widget*. Podemos ver esto en el constructor `Text`, donde `shell` es el primer argumento. Casi todos los constructores también toman un argumento indicador que permite proporcionar varias directivas de estilo, dependiendo de lo que el *widget* concreto acepte. Las diversas directivas de estilo se combinan bit a bit mediante la operación OR, como puede verse en el ejemplo.

A la hora de configurar el objeto `Text()`, he añadido indicadores de estilo para que el texto efectúe saltos de línea automáticos, y para que se añada automáticamente una barra de desplazamiento vertical en caso necesario. Cuando trabaje con este sistema, descubrirá que SWT depende bastante de los constructores; existen muchos atributos de un *widget* que son difíciles o imposibles de cambiar, excepto a través del constructor. Compruebe siempre la documentación del constructor del *widget* para ver qué indicadores acepta. Observe que algunos constructores requieren un argumento indicador, aun cuando la documentación no especifique ningún indicador “aceptado”. Esto permite una futura expansión sin necesidad de modificar la interfaz.

Eliminación del código redundante

Antes de continuar adelante, observe que hay que realizar ciertas cosas para cada aplicación SWT, de la misma forma que existían acciones duplicadas en los programas Swing. En SWT, siempre creamos un objeto `Display`, construimos un objeto

Shell a partir del objeto **Display**, creamos un bucle **readAndDispatch()**, etc. Por supuesto, en algunos casos especiales, puede que no hagamos esto, pero estas tareas son lo suficientemente comunes como para que merezca la pena intentar eliminar el código duplicado, como hicimos con **net.mindview.util.SwingConsole**.

Tendremos que obligar a cada aplicación a adaptarse a una interfaz:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
    void createContents(Composite parent);
} //:-
```

A la aplicación se le entrega un objeto **Composite** (**Shell** es una subclase) y la aplicación debe usar este objeto para crear todos sus contenidos dentro de **createContents()**. **SWTConsole.run()** invoca **createContents()** en el punto apropiado, establece el tamaño de la *shell* de acuerdo con lo que el usuario le haya pasado a **run()**, abre la *shell* y luego ejecuta el bucle de sucesos, eliminando finalmente la *shell* al salir del programa:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTConsole {
    public static void
    run(SWTApplication swtApp, int width, int height) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText(swtApp.getClass().getSimpleName());
        swtApp.createContents(shell);
        shell.setSize(width, height);
        shell.open();
        while(!shell.isDisposed()) {
            if(!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
} //:-
```

Esto establece también la barra de título, asignándole el nombre de la clase **SWTApplication**, y establece la anchura y altura de la **Shell** mediante los valores **width** y **height**.

Podemos crear una variante de **DisplayProperties.java** que muestre el entorno de la máquina, usando **SWTConsole**:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;

public class DisplayEnvironment implements SWTApplication {
    public void createContents(Composite parent) {
        parent.setLayout(new FillLayout());
        Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
        for(Map.Entry entry: System.getenv().entrySet()) {
            text.append(entry.getKey() + ": " +
            entry.getValue() + "\n");
        }
    }
    public static void main(String [] args) {
```

```

    SWTConsole.run(new DisplayEnvironment(), 800, 600);
}

} //:~
```

SWTConsole nos permite centrarnos en los aspectos más interesantes de una aplicación, en lugar de tener que escribir el código repetitivo.

Ejercicio 40: (4) Modifique **DisplayProperties.java** para utilizar **SWTConsole**.

Ejercicio 41: (4) Modifique **DisplayEnvironment.java** para *no* utilizar **SWTConsole**.

Menús

Para ilustrar los fundamentos de los menús, el siguiente ejemplo lee su propio código fuente y lo descompone en palabras, rellenando luego los menús con estas palabras:

```

//: swt/Menus.java
// Ejemplo de menús.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        Set<String> words = new TreeSet<String>(
            new TextFile("Menus.java", "\\\W+"));
        Iterator<String> it = words.iterator();
        while(it.hasNext().matches("[0-9]+"))
            ; // Desplazarse hasta pasados los números.
        MenuItem[] mItem = new MenuItem[7];
        for(int i = 0; i < mItem.length; i++) {
            mItem[i] = new MenuItem(bar, SWT.CASCADE);
            mItem[i].setText(it.next());
            Menu submenu = new Menu(shell, SWT.DROP_DOWN);
            mItem[i].setMenu(submenu);
        }
        int i = 0;
        while(it.hasNext()) {
            addItem(bar, it, mItem[i]);
            i = (i + 1) % mItem.length;
        }
    }
    static Listener listener = new Listener() {
        public void handleEvent(Event e) {
            System.out.println(e.toString());
        }
    };
    void
    addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
        MenuItem item = new MenuItem(mItem.getMenu(), SWT.PUSH);
        item.addListener(SWT.Selection, listener);
        item.setText(it.next());
    }
    public static void main(String[] args) {
```

```

        SWTConsole.run(new Menus(), 600, 200);
    }
} //:-/

```

Los objetos **Menu** deben colocarse dentro de una **Shell**, y **Composite** permite determinar la *shell* mediante **getShell()**. **TextFile** procede de **net.mindview.util** y ya lo hemos descrito antes en el libro; aquí, se rellena un conjunto **TreeSet** con palabras, de forma que aparezcan ordenadas. Los elementos iniciales son números, que se descartan. Utilizando el flujo de palabras, se asigna un nombre a los menús de nivel superior de la barra de menús y luego se crean los submenús y se llenan con palabras hasta que no quedan más palabras.

En respuesta a la selección de uno de los elementos de menú, **Listener** simplemente imprime el suceso para que podamos ver el tipo de información que contiene. Cuando se ejecute el programa, verá que parte de la información incluye la etiqueta del menú, de manera que podemos pasar la respuesta del menú en dicha información; o bien podemos proporcionar un escucha diferente para cada menú (que es un enfoque más seguro, de cara a la internacionalización).

Paneles con fichas, botones y sucesos events

SWT dispone de un rico conjunto de controles, denominados *widgets*. Examine la documentación de **org.eclipse.swt.widgets** para ver los controles básicos y **org.eclipse.swt.custom** para ver otros más sofisticados.

Para ilustrar algunos de los *widgets* básicos, este ejemplo coloca una serie de subejemplos dentro de paneles con fichas. También podrá ver cómo crear objeto **Composite** (aproximadamente equivalente a los paneles **JPanel** de Swing) para insertar elementos dentro de otros elementos:

```

//: swt/TabbedPane.java
// Colocación de componentes SWT en paneles con fichas.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;

public class TabbedPane implements SWTApplication {
    private static TabFolder folder;
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        parent.setLayout(new FillLayout());
        folder = new TabFolder(shell, SWT.BORDER);
        labelTab();
        directoryDialogTab();
        buttonTab();
        sliderTab();
        scribbleTab();
        browserTab();
    }
    public static void labelTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("A Label"); // Texto de la ficha
        tab.setToolTipText("A simple label");
        Label label = new Label(folder, SWT.CENTER);
        label.setText("Label text");
        tab.setControl(label);
    }
    public static void directoryDialogTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("Directory Dialog");
        tab.setToolTipText("Select a directory");
    }
}

```

```

final Button b = new Button(folder, SWT.PUSH);
b.setText("Select a Directory");
b.addListener(SWT.MouseDown, new Listener() {
    public void handleEvent(Event e) {
        DirectoryDialog dd = new DirectoryDialog(shell);
        String path = dd.open();
        if(path != null)
            b.setText(path);
    }
});
tab.setControl(b);
}
public static void buttonTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Buttons");
    tab.setToolTipText("Different kinds of Buttons");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(4, true));
    for(int dir : new int[]{
        SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
    }) {
        Button b = new Button(composite, SWT.ARROW | dir);
        b.addListener(SWT.MouseDown, listener);
    }
    newButton(composite, SWT.CHECK, "Check button");
    newButton(composite, SWT.PUSH, "Push button");
    newButton(composite, SWT.RADIO, "Radio button");
    newButton(composite, SWT.TOGGLE, "Toggle button");
    newButton(composite, SWT.FLAT, "Flat button");
    tab.setControl(composite);
}
private static Listener listener = new Listener() {
    public void handleEvent(Event e) {
        MessageBox m = new MessageBox(shell, SWT.OK);
        m.setMessage(e.toString());
        m.open();
    }
};
private static void newButton(Composite composite,
    int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
}
public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Sliders and Progress bars");
    tab.setToolTipText("Tied Slider to ProgressBar");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider =
        new Slider(composite, SWT.HORIZONTAL);
    final ProgressBar progress =
        new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            progress.setSelection(slider.getSelection());
        }
    });
    tab.setControl(composite);
}

```

```

    }
    public static void scribbleTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("Scribble");
        tab.setToolTipText("Simple graphics: drawing");
        final Canvas canvas = new Canvas(folder, SWT.NONE);
        ScribbleMouseListener sml= new ScribbleMouseListener();
        canvas.addMouseListener(sml);
        canvas.addMouseMoveListener(sml);
        tab.setControl(canvas);
    }
    private static class ScribbleMouseListener
        extends MouseAdapter implements MouseMoveListener {
        private Point p = new Point(0, 0);
        public void mouseMove(MouseEvent e) {
            if((e.stateMask & SWT.BUTTON1) == 0)
                return;
            GC gc = new GC((Canvas)e.widget);
            gc.drawLine(p.x, p.y, e.x, e.y);
            gc.dispose();
            updatePoint(e);
        }
        public void mouseDown(MouseEvent e) { updatePoint(e); }
        private void updatePoint(MouseEvent e) {
            p.x = e.x;
            p.y = e.y;
        }
    }
    public static void browserTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText("A Browser");
        tab.setToolTipText("A Web browser");
        Browser browser = null;
        try {
            browser = new Browser(folder, SWT.NONE);
        } catch(SWTError e) {
            Label label = new Label(folder, SWT.BORDER);
            label.setText("Could not initialize browser");
            tab.setControl(label);
        }
        if(browser != null) {
            browser.setUrl("http://www.mindview.net");
            tab.setControl(browser);
        }
    }
    public static void main(String[] args) {
        SWTConsole.run(new TabbedPane(), 800, 600);
    }
} //:-
}

```

Aquí, **createContents()** establece la disposición de los elementos y luego invoca los métodos que se encargan de crear las distintas fichas. El texto de cada ficha se establece con **setText()** (también podemos crear botones y gráficos en una ficha), y cada una de las fichas establece también el texto de sugerencia. Al final de cada método, puede ver una llamada a **setControl()**, que coloca el control que el método ha creado dentro del espacio correspondiente a cada ficha concreta.

labelTab() ilustra un etiqueta simple de texto. **directoryDialogTab()** contiene un botón que abre un objeto estándar **DirectoryDialog** para que el usuario pueda seleccionar un directorio. El resultado se asigna como texto del botón.

buttonTab() muestra los diferentes botones básicos. **sliderTab()** repite el ejemplo Swing presentado anteriormente en el capítulo, que consistía en asociar un deslizador con una barra de progreso.

scribbleTab() es un divertido ejemplo de gráficos. Se genera un programa de dibujo utilizando sólo unas pocas líneas de código.

Por último, **browserTab()** muestra la potencia del componente **Browser** de SWT, que es un explorador web completamente funcional empaquetado en un único componente.

Gráficos

He aquí el programa **SineWave.java** de Swing traducido a SWT:

```
//: swt/SineWave.java
// Traducción a SWT del programa Swing SineWave.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class SineDraw extends Canvas {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw(Composite parent, int style) {
        super(parent, style);
        addPaintListener(new PaintListener() {
            public void paintControl(PaintEvent e) {
                int maxWidth = getSize().x;
                double hstep = (double)maxWidth / (double)points;
                int maxHeight = getSize().y;
                pts = new int[points];
                for(int i = 0; i < points; i++)
                    pts[i] = (int)((sines[i] * maxHeight / 2 * .95)
                        + (maxHeight / 2));
                e.gc.setForeground(
                    e.display.getSystemColor(SWT.COLOR_RED));
                for(int i = 1; i < points; i++) {
                    int x1 = (int)((i - 1) * hstep);
                    int x2 = (int)(i * hstep);
                    int y1 = pts[i - 1];
                    int y2 = pts[i];
                    e.gc.drawLine(x1, y1, x2, y2);
                }
            }
        });
        setCycles(5);
    }
    public void setCycles(int newCycles) {
        cycles = newCycles;
        points = SCALEFACTOR * cycles * 2;
        sines = new double[points];
        for(int i = 0; i < points; i++) {
            double radians = (Math.PI / SCALEFACTOR) * i;
            sines[i] = Math.sin(radians);
        }
        redraw();
    }
}
```

```

public class SineWave implements SWTApplication {
    private SineDraw sines;
    private Slider slider;
    public void createContents(Composite parent) {
        parent.setLayout(new GridLayout(1, true));
        sines = new SineDraw(parent, SWT.NONE);
        sines.setLayoutData(
            new GridData(SWT.FILL, SWT.FILL, true, true));
        sines.setFocus();
        slider = new Slider(parent, SWT.HORIZONTAL);
        slider.setValues(5, 1, 30, 1, 1, 1);
        slider.setLayoutData(
            new GridData(SWT.FILL, SWT.DEFAULT, true, false));
        slider.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                sines.setCycles(slider.getSelection());
            }
        });
    }
    public static void main(String[] args) {
        SWTConsole.run(new SineWave(), 700, 400);
    }
} //:-.

```

En lugar de **JPanel**, la superficie de dibujo básica en SWT es **Canvas**.

Si comparamos esta versión del programa con la versión Swing, veremos que **SineDraw** es prácticamente idéntico. En SWT, obtenemos el contexto gráfico **gc** a partir del objeto suceso que se entrega al escucha **PaintListener**, y en Swing el objeto **Graphics** se entrega directamente al método **paintComponent()**. Pero las actividades realizadas con el objeto gráfico son iguales y **setCycles()** es idéntico.

createContents() requiere algo más de código que la versión Swing, para disponer los elementos y configurar el deslizador y su correspondiente escucha, pero de nuevo, las actividades básicas son aproximadamente iguales.

Concurrencia en SWT

Aunque AWT/Swing es monohebra, resulta posible violar fácilmente esa característica monohebra de manera que se obtenga un programa no determinista. Básicamente, debemos evitar tener múltiples hebras escribiendo en la pantalla, porque las unas escribirán sobre lo que hayan escrito las otras de manera sorprendente.

SWT no permite que esto suceda, puesto que genera una excepción si tratamos de escribir en la pantalla empleando más de una hebra. Eso evitará que un programador inexperto cometa accidentalmente este error e introduzca errores difíciles de localizar dentro de un programa.

He aquí la traducción a SWT del programa Swing **ColorBoxes.java**:

```

//: swt/ColorBoxes.java
// Traducción a SWT del programa Swing ColorBoxes.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
    class CBoxPaintListener implements PaintListener {
        public void paintControl(PaintEvent e) {
            Color color = new Color(e.display, cColor);

```

```

        e.gc.setBackground(color);
        Point size = getSize();
        e.gc.fillRect(0, 0, size.x, size.y);
        color.dispose();
    }
}

private static Random rand = new Random();
private static RGB newColor() {
    return new RGB(rand.nextInt(255),
                   rand.nextInt(255), rand.nextInt(255));
}
private int pause;
private RGB cColor = newColor();
public CBox(Composite parent, int pause) {
    super(parent, SWT.NONE);
    this.pause = pause;
    addPaintListener(new CBoxPaintListener());
}
public void run() {
    try {
        while(!Thread.interrupted()) {
            cColor = newColor();
            getDisplay().asyncExec(new Runnable() {
                public void run() {
                    try { redraw(); } catch(SWTException e) {} // SWTException es OK cuando el padre es
                    // terminado desde debajo de nosotros.
                }
            });
            TimeUnit.MILLISECONDS.sleep(pause);
        }
    } catch(InterruptedException e) {
        // Forma aceptable de salir
    } catch(SWTException e) {
        // Forma aceptable de salir: nuestro parent
        // ha sido terminado desde debajo de nosotros.
    }
}

public class ColorBoxes implements SWTApplication {
    private int grid = 12;
    private int pause = 50;
    public void createContents(Composite parent) {
        GridLayout gridLayout = new GridLayout(grid, true);
        gridLayout.horizontalSpacing = 0;
        gridLayout.verticalSpacing = 0;
        parent.setLayout(gridLayout);
        ExecutorService exec = new DaemonThreadPoolExecutor();
        for(int i = 0; i < (grid * grid); i++) {
            final CBox cb = new CBox(parent, pause);
            cb.setLayoutData(new GridData(GridData.FILL_BOTH));
            exec.execute(cb);
        }
    }
    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
    }
}

```

```

        boxes.pause = new Integer(args[1]);
        SWTConsole.run(boxes, 500, 400);
    }
} //:-

```

Como en el ejemplo anterior, el dibujo se controla creando un escucha **PaintListener** con un método **paintControl()** que se invoca cuando la hebra SWT está lista para pintar el componente. El escucha **PaintListener** se registra en el constructor de **CBox**.

Lo que difiere notablemente en esta versión de **CBox** es el método **run()**, que no puede limitarse a invocar **redraw()** directamente, sino que tiene que enviar la llamada a **redraw()** al método **asyncExec()** del objeto **Display**, que equivale aproximadamente al método **SwingUtilities.invokeLater()**. Si sustituimos estos por una llamada directa a **redraw()**, comprobaremos que el programa simplemente se detiene.

Al ejecutar el programa, veremos pequeños artefactos visuales: líneas horizontales que atraviesan ocasionalmente un recuadro. Esto es debido a que SWT *no utiliza* doble *buffer* de manera predeterminada, mientras que Swing sí lo utiliza. Trate de ejecutar la versión Swing al lado de la versión SWT y podrá observar la diferencia más claramente. Podemos escribir código para utilizar un doble *buffer* SWT; podrá encontrar ejemplos en el sitio web www.eclipse.org.

Ejercicio 42: (4) Modifique **swt/ColorBoxes.java** para que comience distribuyendo una serie de puntos (“estrellas”) por todo el lienzo y luego cambie aleatoriamente el color de esas “estrellas”.

¿SWT o Swing?

Resulta difícil hacerse una idea general a partir de una introducción tan corta, pero el lector debería al menos comenzar a percibir que SWT puede ser, en muchas situaciones, una forma más sencilla de escribir programas que Swing. Sin embargo, la programación de GUI en SWT puede seguir siendo compleja, por lo que los motivos para utilizar SWT deberían ser, en primer lugar, proporcionar al usuario una experiencia más transparente a la hora de usar la aplicación (porque el aspecto y estilo de la aplicación se asemejarán a los de otras aplicaciones en dicha plataforma) y segundo, si la mayor velocidad proporcionada por SWT resulta importante. En caso contrario, Swing puede ser una elección perfectamente apropiada.

Ejercicio 43: (6) Seleccione alguno de los ejemplos Swing que no haya sido traducido en esta sección y tradúzcalo a SWT. (Nota: esto puede constituir un buen ejercicio para que los estudiantes realicen en casa, ya que las soluciones no se han incluido en la guía de soluciones).

Resumen

Las bibliotecas GUI de Java han experimentado cambios bastante drásticos a lo largo del tiempo de existencia del lenguaje. La biblioteca AWT de Java 1.0 fue muy criticada por su pobre diseño, y aunque permitía crear programas portables, la GUI resultante era “igualmente mediocre en todas las plataformas”. También era bastante limitadora, abstrusa e incómoda de emplear comparada con las herramientas de desarrollo nativas disponibles en diversas plataformas.

Cuando Java 1.1 introdujo el nuevo modelo de sucesos y los componentes JavaBeans, el escenario completo ya estaba preparado: ahora era posible crear componentes GUI que se podían arrastrar y colocar fácilmente dentro de un entorno IDE visual. Además, el diseño del modelo de sucesos y de JavaBeans muestra claramente que los objetivos eran la facilidad de programación y la utilización de código fácilmente mantenible (algo que no era evidente en la biblioteca AWT de la versión 1.0). Pero la transición no se concretó hasta que aparecieron las clases JFC/Swing. Con los componentes Swing, la programación de interfaces GUI multiplataforma puede resultar sencilla.

La verdadera revolución radica en el uso de entornos IDE. Si desea adquirir un entorno IDE comercial para mejorar la programación con un lenguaje propietario, está obligado a cruzar los dedos y a esperar que el fabricante le proporcione lo que usted espera. Pero Java es un entorno abierto, por lo que no sólo permite que existan entornos IDE competidores, sino que fomenta esa existencia. Y para que estas herramientas puedan tomarse en serio, están obligadas a soportar JavaBeans. Esto significa que el campo de juegos está nivelado para todos los contendientes; si aparece un entorno IDE mejor, no estamos obligados a continuar empleando el anterior. Podemos seleccionar el nuevo e incrementar con ello nuestra productividad. Este tipo de campo de juego competitivo para entornos IDE de creación de interfaces GUI no había sido experimentado antes, y el mercado resultante permitirá generar resultados muy positivos en lo que respecta a la productividad de los programadores.

Este capítulo ha pretendido únicamente proporcionar una introducción a la potencia de la programación de interfaces GUI, y hacer que el lector comenzara a familiarizarse con material de programación de interfaces para ver lo simple que resulta emplear las correspondientes bibliotecas. Lo que hemos visto en el capítulo bastará, probablemente, para cubrir una parte de nuestras necesidades de diseño de interfaces de usuario. Sin embargo, tanto Swing como SWT y Flash/Flex tienen muchas otras características y funcionalidades adicionales, ya que se trata de herramientas de diseño de interfaces de usuario completas. Con ellas, probablemente encuentre una forma de realizar cualquier cosa que sea capaz de imaginar.

Recursos

Las presentaciones en línea de Ben Galbraith disponibles en www.galbraiths.org/presentations hacen un repaso muy adecuado tanto de Swing como de SWT.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible para la venta en www.MindView.net.

Suplementos

A

Este libro tiene varios suplementos, incluyendo los elementos, seminarios y servicios disponibles en el sitio web MindView.

Este apéndice describe estos suplementos, para que pueda decidir si le pueden ser útiles.

Observe que aunque los seminarios suelen ser públicos, también pueden impartirse como seminarios privados en sus propias oficinas.

Suplementos descargables

El código correspondiente a este libro está disponible para su descarga en www.MindView.net. Esto incluye los archivos de construcción Ant y otros archivos de soporte necesarios para construir y ejecutar correctamente todos los ejemplos del libro.

Además, algunas partes del libro sólo se ofrecen en formato electrónico. Los temas que estas partes abarcan:

- Clonación de objetos
- Paso y devolución de objetos
- Análisis y diseño
- Partes de otros capítulos procedentes de la 3^a edición de *Thinking in Java*, que no eran lo suficientemente relevantes como para incluirlos en la versión impresa de la cuarta edición del libro.

Thinking in C: fundamentos para Java

En www.MindView.net, podrá descargarse gratuitamente el seminario *Thinking in C*. Esta presentación, creada por Chuck Allison y desarrollada por MindView, es un curso Flash multimedia que proporciona una introducción a la sintaxis, los operadores y las funciones de C en las que se basa la sintaxis de Java.

Observe que es necesario instalar en nuestro sistema el reproductor Flash Player de www.Macromedia.com para poder ver la presentación *Thinking in C*.

Seminario *Thinking in Java*

Mi empresa, MindView, Inc., proporciona seminarios de formación públicos o privados de carácter práctico y de cinco días de duración basados en el material de este libro. Anteriormente denominado seminario *Hands-On Java*, se trata de nuestro principal seminario introductorio, que proporciona la base para seminarios más avanzados. Una serie de materiales seleccionados de cada capítulo representan una lección, que va seguida por un período de ejercicios monitorizado, de modo que cada estudiante recibe una atención personalizada. Puede encontrar información sobre horarios y lugares donde se imparte este seminario, junto con testimonios y otros detalles en www.MindView.net.

Seminario en CD *Hands-On Java*

El CD *Hands-On Java* contiene una versión ampliada del material del seminario *Thinking in Java* y está basado en este libro. Proporciona al menos una parte de la experiencia del seminario real, pero sin los gastos de viaje asociados. Existe una conferencia audio y una serie de presentaciones correspondientes a cada uno de los capítulos del libro. Yo he creado personalmente dicho seminario y me he encargado de narrar el material contenido en el CD. El material está en formato Flash, por lo que debería poder ejecutarse en cualquier plataforma que soporte Flash Player. El CD de *Hands-On Java CD* puede adquirirse a través de www.MindView.net, donde encontrará demostraciones de prueba del producto.

Seminario *Thinking in Objects*

Este seminario introduce las ideas de la programación orientada a objetos desde el punto de vista del diseñador. Explora el proceso de desarrollo y construcción de un sistema, centrándose principalmente en los denominados “Métodos ágiles” o “Metodologías ligeras”, y en especial en XP (*Extreme Programming*). En el seminario introduce las metodologías en general, unas pequeñas herramientas como las técnicas de planificación mediante “tarjetas índice” descritas en *Planning Extreme Programming* por Beck y Fowler (Addison-Wesley, 2001), las tarjetas CRC para diseñar objetos, la programación por pares, la planificación de iteraciones, las pruebas unitarias, la construcción automatizada, el control de código fuente y temas similares. El curso incluye un proyecto XP que se desarrolla a lo largo de una semana.

Si está iniciando un proyecto y desea comenzar a utilizar técnicas de diseño orientadas a objetos, podemos emplear su proyecto como ejemplo y disponer de un primer prototipo de diseño al final de la semana.

Visite www.MindView.net para obtener la información sobre horarios y lugares donde se imparte este seminario, así como testimonios y otros detalles.

Thinking in Enterprise Java

Este libro se emplea a partir de algunos de los capítulos más avanzados de las ediciones anteriores en *Thinking in Java*. Este libro no es un segundo volumen de *Thinking in Java*, sino que se centra más bien en cubrir los temas avanzados de la programación empresarial. Está disponible (aunque todavía en desarrollo) en forma de descarga gratuita en www.MindView.net. Pero como es un libro separado, debe expandirse para cubrir los temas necesarios. El objetivo, como el de *Thinking in Java*, es proporcionar una introducción comprensible a los fundamentos de las tecnologías de programación empresarial, para que el lector esté preparado para acometer un estudio basado en dichos temas.

La lista de temas que incluye, entre otros, es la siguiente:

- Introducción a la programación empresarial
- Programación de red con Sockets y canales
- Invocación remota de métodos (RMI)
- Conexión a bases de datos
- Servicios de denominación y directorio
- Servlets
- Java Server Pages
- Marcadores, fragmentos JSP y lenguaje de expresión
- Automatización de la creación de interfaces de usuario
- Enterprise JavaBeans
- XML
- Servicios Web
- Pruebas automáticas

Puede encontrar información sobre el estado actual (en inglés) de *Thinking in Enterprise Java* en www.MindView.net.

Thinking in Patterns (con Java)

Uno de los pasos adelante más importantes dentro del diseño orientado a objeto es el movimiento de los “patrones de diseño”, que se tratan en *Design Patterns*, de Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). Dicho libro muestra 23 clases generales de problemas junto con sus soluciones, escritos principalmente en C++. El libro *Design Patterns* es una fuente autorizada de lo que ahora se ha convertido en un vocabulario esencial, si es que no obligatorio, para la programación orientada a objetos. *Thinking in Patterns* introduce los conceptos básicos de los patrones de diseño junto con una serie de ejemplos en Java. El libro no pretende ser una simple traducción de *Design Patterns*, sino más bien una nueva perspectiva desde el punto de vista de Java. No está limitado a los 23 patrones tradicionales, sino que incluye también otras ideas y técnicas de resolución de problemas.

Este libro comenzó con el último capítulo de la primera edición de *Thinking in Java*, y a medida que las ideas continuaron desarrollándose, quedó claro que era necesario incluir ese material en su propio libro independiente. En el momento de escribir estas líneas, el libro se encuentra todavía en desarrollo, pero el material ya está muy avanzado y ha sido utilizado en numerosas presentaciones del seminario *Objects & Patterns* (que ahora se ha dividido en los seminarios *Designing Objects & Systems* y *Thinking in Patterns*).

Puede encontrar más información acerca de este libro en www.MindView.net.

Seminario Thinking in Patterns

Este seminario ha evolucionado a partir del seminario *Objects & Patterns* que Bill Venners y yo mismo hemos impartido en los últimos años. Dicho seminario llegó a estar demasiado cargado de contenido, por lo que lo dividimos en dos seminarios independientes: éste y el seminario *Designing Objects & Systems*, descrito anteriormente.

El seminario se ajusta de manera bastante fiel al material y a la presentación del libro *Thinking in Patterns*, por lo que la mejor forma de conocer los detalles sobre el seminario es consultar la información relativa al libro en www.MindView.net.

Buena parte de la presentación que se centra en el proceso de evolución del diseño, comenzando por una solución inicial y repasando la lógica y el proceso que subyace a la evolución de esa solución hacia diseños más apropiados. El último proyecto mostrado (una simulación de reciclado de residuos) ha ido evolucionando a lo largo del tiempo, y se puede examinar esa evolución como un prototipo de la manera en que un diseño real puede comenzar en forma de una solución adecuada a un problema concreto, para terminar evolucionando hasta convertirse en una solución flexible para toda una clase de problemas.

Este seminario le ayudará a:

- Aumentar enormemente la flexibilidad de sus diseños.
- Incorporar a los diseños los conceptos de ampliabilidad y reusabilidad.
- Crear mecanismos de comunicación más densos acerca de los diseños utilizando el lenguaje de patrones.
- Después de cada presentación, hay un conjunto de ejercicios sobre patrones de diseño que los asistentes deben resolver, siendo dirigidos durante la escritura del código para poder aplicar patrones concretos a la solución de los problemas de programación.

Visite www.MindView.net para obtener más información acerca de los horarios y los lugares donde se imparte este seminario, junto con testimonios y detalles adicionales.

Consultoría y revisión de diseño

Mi empresa también proporciona servicios de consultoría, de tutoría, de revisión de diseño y revisión de implementación como ayuda durante el ciclo de desarrollo de un proyecto, incluyendo los primeros proyectos Java de cualquier empresa. Visite www.MindView.net para obtener más información sobre disponibilidad y otros detalles.



Recursos

Software

El **JDK** disponible en <http://java.sun.com>. Incluso si decide emplear un entorno de desarrollo de otro fabricante, siempre es conveniente tener el JDK a mano, por si acaso se tropieza con algún posible error del compilador. El JDK es la piedra de toque del diseño Java, y si existe un error en él, hay grandes posibilidades de que dicho error esté bien documentado.

La **documentación del JDK** disponible en <http://java.sun.com>, en formato HTML. No he podido encontrar todavía un libro de referencia sobre las bibliotecas estándar de Java que no estuviera desactualizado o en el que no faltara información. Aunque la documentación del JDK de Sun tiene algunos pequeños errores y es excesivamente concisa en algunos puntos, a menos incluye todas las clases y métodos. En ocasiones, algunas personas no se sienten cómodas utilizando inicialmente un recurso en línea en lugar de un libro en papel, pero merece la pena superar esa incomodidad inicial y consultar los documentos HTML, al menos para obtener una panorámica general. Si le sigue sin gustar el método de las referencias en línea, adquiera los libros impresos.

Editores y entornos IDE

Existe una sana competición dentro de este área. Muchos de los productos son gratuitos (y los que no lo son, normalmente disponen de versiones de prueba gratuitas), por lo que lo mejor es probar los distintos productos y ver cuál se adapta mejor a sus necesidades. He aquí algunos de ellos:

JEdit, es un editor gratuito diseñado por Slava Pestov, escrito en Java, con lo que obtenemos la ventaja adicional de poder ver en acción una aplicación de escritorio desarrollada Java. Este editor está basado fuertemente en la utilización de *plugins*, muchos de los cuales han sido escritos por la comunidad Java. Puede descargarlo en www.jedit.org.

NetBeans, un entorno IDE gratuito de Sun, disponible en www.netbeans.org. Diseñado para la construcción de interfaces GUI mediante el procedimiento de arrastrar y colocar, para la creación, edición y depuración de código,etc.

Eclipse, un proyecto de código abierto respaldado por IBM, entre otros. La plataforma Eclipse también está diseñada para constituir una base ampliable de desarrollo, de modo que se pueden construir aplicaciones autónomas sobre Eclipse. Este proyecto desarrolló el lenguaje SWT descrito en el Capítulo 22, *Interfaces gráficas de usuario*. Puede descargarlo en www.eclipse.org.

IntelliJ IDEA, el entorno comercial favorito de un gran número de programadores Java, muchos de los cuales afirman que IDEA siempre está a uno o dos pasos de Eclipse, posiblemente porque IntelliJ no trata tanto de crear un entorno IDE como una plataforma de desarrollo, sino que se limita a los aspectos del entorno IDE. Puede descargar una versión gratuita en www.jetbrains.com.

Libros

Core Java™ 2, 7th Edition, Volúmenes I & II, de Horstmann & Cornell (Prentice Hall, 2005). Voluminoso, completo y es donde debe buscar en primer lugar cuando esté buscando respuestas. Recomiendo la lectura de este libro una vez que se haya completado *Thinking in Java* y se necesite comenzar a profundizar en los distintos temas.

The Java™ Class Libraries: An Annotated Reference, por Patrick Chan and Rosanna Lee (Addison-Wesley, 1997). Aunque está desactualizado, este libro es lo que la referencia JDK *debería* haber sido: incluye las suficientes descripciones

como para hacerlo utilizable. Uno de los revisores técnicos de *Thinking in Java* me dijo: "Si tuviera un único libro sobre Java, sería éste (además del tuyo, por supuesto)." Yo no estoy tan entusiasmado con este libro, como dicho revisor. Es voluminoso, resulta caro y la calidad de los ejemplos no me parece adecuada. Sin embargo, es un buen lugar en el que consultar cuando estemos bloqueados con un problema y aborda los temas con una mayor profundidad que la mayoría de los demás libros. Sin embargo, *Core Java 2* tiene un tratamiento más actualizado de muchos de los componentes de la biblioteca.

Java Network Programming, 2^a Edición, por Elliotte Rusty Harold (O'Reilly, 2000). Personalmente, no empecé a comprender los aspectos de las redes en Java (ni, en realidad, los aspectos de comunicación por red, en general) hasta que encontré este libro. También me parece que su sitio web, Café au Lait, es muy estimulante, informativo y actualizado en lo que respecta a los desarrollos Java, siendo muy independiente respecto a los distintos fabricantes de software. Las actualizaciones regulares hacen que el sitio web esté lleno de noticias acerca de la evolución de Java. Consulte www.cafeaulait.org.

Design Patterns, por Gamma, Helm, Johnson y Vlissides (Addison-Wesley, 1995). El libro original que dio comienzo al movimiento de patrones de diseño dentro del campo de la programación y que hemos mencionado en numerosos lugares a lo largo del libro.

Refactoring to Patterns, por Joshua Kerievsky (Addison-Wesley, 2005). Combina el tema de la reingeniería con el de los patrones de diseño. Lo más valioso de este libro es que muestra cómo hacer evolucionar un diseño introduciendo nuevos patrones a medida que son necesarios.

The Art of UNIX Programming, por Eric Raymond (Addison-Wesley, 2004). Aunque Java es un lenguaje interplataforma, la prevalencia de Java en el mundo de los servidores ha hecho que el conocimiento de Unix/Linux sea importante. El libro de Eric constituye una excelente introducción a la historia y filosofía de este sistema operativo, y resulta fascinante de leer aunque sólo se quieran comprender algunos aspectos de los orígenes de la informática.

Análisis y diseño

Extreme Programming Explained, 2^a Edition, por Kent Beck con Cynthia Andres. (Addison-Wesley, 2005). Siempre he pensado que debería haber un proceso de desarrollo de programas muy distinto y mucho mejor que el que se emplea actualmente, y creo que XP se acerca bastante a ese ideal. El único libro que me ha causado un impacto similar es *Peopleware* (del que hablaré más adelante), que habla principalmente acerca del entorno y de cómo tratar con la cultura corporativa. *Extreme Programming Explained* habla acerca de la programación y proporciona una nueva visión sobre los principales aspectos de esta ciencia. Los autores llegan incluso a decir que las imágenes están bien siempre que no invirtamos demasiado tiempo en ellas y estemos dispuestos a prescindir de ellas en caso necesario (observará que el libro *no tiene* la marca de aprobación "UML" en la cubierta). En mi opinión podría decidirse si trabajar para una empresa basándose exclusivamente en si utilizan XP. Es un libro pequeño, de capítulos cortos, que se leen sin esfuerzo y que resulta excitante. Uno puede imaginarse a sí mismo en ese tipo de atmósfera y le asaltan visiones de un nuevo mundo que se abre a sus pies.

UML Distilled, 2^a Edición, por Martin Fowler (Addison-Wesley, 2000). Cuando uno tropieza por primera vez con UML, resulta aterrador, dada la gran cantidad de diagramas y de detalles que existen. De acuerdo con Fowler, la mayor parte de estos detalles son innecesarios, por lo que él se centra en los aspectos esenciales. Para la mayoría de los proyectos, basta con conocer unas pocas herramientas de realización de diagramas, y el objetivo de Fowler es conseguir un buen diseño, en lugar de preocuparse acerca de todos los aditamentos que hacen falta para conseguirlo. De hecho, la mayor parte de los lectores no necesitarán la primera mitad del libro. Es un libro muy atractivo, de pequeño tamaño y mi recomendación es que lo adquiera si desea comprender UML.

Domain-Driven Design, por Eric Evans (Addison-Wesley, 2004). Este libro se centra en el *modelo de dominios* como principal herramienta del proceso de diseño. En mi opinión, se trata de un desplazamiento interesante del foco de atención que ayuda a los diseñadores a adoptar el nivel de abstracción adecuado.

The Unified Software Development Process, por Ivar Jacobson, Grady Booch y James Rumbaugh (Addison-Wesley, 1999). Cuando aborde la lectura de este libro, estaba preparado para que no me gustara. Parecía tener todos los ingredientes de un aburrido libro universitario. Sin embargo, me vi gratamente sorprendido (aunque hay algunas partes que incluyen explicaciones que dan la sensación de que los autores no tienen los conceptos claros). El conjunto del libro no sólo es muy claro, sino también muy agradable de leer. Lo mejor de todo es que el proceso descrito tiene bastante sentido práctico. No se trata de *Extreme Programming* (y no tiene la claridad que esta otra técnica presenta en lo que respecta a las pruebas), pero también forma parte del arsenal del mundo UML; incluso aquellos que no desean utilizar XP suelen estar de acuerdo en que "UML es un buen lenguaje de modelado" (independientemente del nivel experiencia *real* que tengan los que emiten estas

opiniones). Este libro constituye una verdadera referencia de UML y es lo que yo recomendaría, para conocer más detalles después de leer *UML Distilled* de Fowler.

Antes de elegir ningún método concreto, resulta útil ganar cierta perspectiva, aprendiéndola de aquellos que no tratan de vendernos ninguna en concreto. Es fácil adoptar un método sin llegar a entender realmente qué es lo que queremos obtener de él o qué es lo que nos puede proporcionar. Que otras personas usen este método, parece una razón suficiente. Sin embargo, los seres humanos tienen una tendencia psicológica muy curiosa. Si quieren creer que algo resolverá sus problemas, tratarán de usarlo (esto es experimentación, lo cual es algo bueno). Pero si no resuelve sus problemas, puede que redoble sus esfuerzos y comience a anunciar a grandes voces esa cosa tan increíble que han descubierto (esto es negación de la realidad, que no es tan bueno). El mecanismo que subyace a este comportamiento es que si podemos conseguir que otras personas se suban al mismo barco, al menos no estaremos solos, incluso aunque el barco no vaya a ninguna parte (o se esté hundiendo).

Con esto no quiero sugerir que todas las metodologías no vayan a ninguna parte, sino sólo que debemos utilizar las herramientas mentales que nos permitan permanecer en modo de experimentación ("No está funcionando, probemos alguna otra cosa"), sin entrar en el modo de negación ("No, no se trata realmente de un problema. Como todo es maravilloso, no necesitamos cambiarlo"). En mi opinión, los siguientes libros que hay que leer *antes* de elegir un método, le proporcionarán esas herramientas fundamentales.

Software Creativity, por Robert L. Glass (Prentice Hall, 1995). Éste es el mejor libro que yo he visto en donde se analiza la perspectiva de todo el tema de las metodologías. Es una colección de ensayos cortos y artículos que Glass ha escrito y en ocasiones adquirido (P.J. Plauger es uno de los contribuidores), en donde se refleja sus muchos años de reflexión y estudio sobre el tema. Son bastante entretenidos y su longitud es estrictamente la adecuada para transmitir toda la información necesaria; el autor no divaga ni aburre. Tampoco se dedica a vender humo: hay cientos de referencias a otros artículos y estudios. Todos los programadores y gestores deberían leer este libro antes de entrar en el tema de las metodologías.

Software Runaways: Monumental Software Disasters, por Robert L. Glass (Prentice Hall, 1998). Lo mejor acerca de este libro es que pone de manifiesto todas esas cosas de las que a nadie le gusta hablar: la cantidad de proyectos que no sólo fallan, sino que lo hacen de manera espectacular. La mayoría de la gente tiende a pensar: "Eso no me puede pasar a mí" (o "Eso no me puede pasar *de nuevo*"), y eso hace que juguemos con desventaja. Recordando que las cosas siempre pueden ir mal, estaremos en una posición mucho mejor para hacer que vayan bien.

Peopleware, 2^a Edición, por Tom DeMarco y Timothy Lister (Dorset House, 1999). Tiene que leer este libro. No sólo es divertido, sino que convierte todos los cimientos de nuestro esquema mental y destruye nuestras suposiciones previas. Aunque DeMarco y Lister provienen del campo de desarrollo de software, este libro trata de proyectos y equipos de trabajo en general. El libro se centra en las *personas* y en sus necesidades, más que en la tecnología y en las necesidades de la tecnología. Hablan acerca de crear un entorno en el que las personas estén contentas y sean productivas, en lugar de decidir las reglas que tienen que seguir para ser componentes adecuados de una máquina. Esta última aptitud, en mi opinión, es la que más contribuye a que los programadores se sonrían y se burlen cuando se adopta el método XYZ, después de lo cual continúan haciendo en silencio lo que siempre han estado haciendo.

Secrets of Consulting: A Guide to Giving & Getting Advice Successfully, por Gerald M. Weinberg (Dorset House, 1985). Este libro maravilloso es uno de mis favoritos. Resulta perfecto cuando alguien quiere dedicarse a la consultoría, o si ha contratado los servicios de algún consultor y desea mejorar las cosas. Está compuesto de cortos capítulos, llenos de historias y anécdotas que nos enseñan cómo llegar hasta el fondo del asunto con un esfuerzo mínimo. Lea también *More Secrets of Consulting*, publicado en 2002, o cualquier otro de los libros de Weinberg.

Complexity, por M. Mitchell Waldrop (Simon & Schuster, 1992). Este libro es la crónica de la reunión celebrada en Santa Fe, Nuevo México, por un grupo de científicos de diferentes disciplinas para analizar problemas reales que sus disciplinas individuales no pueden resolver (el mercado bursátil en Economía, la formación inicial de la visa en Biología, por qué las personas hacen lo que hacen en Sociología, etc.). Combinando la Física, la Economía, la Química, las Matemáticas, la Informática, la Sociología, y otras ciencias, se está desarrollando un enfoque multidisciplinar para tratar de abordar estos problemas. Pero lo más importante es que está emergiendo una forma diferente de pensar acerca de estos problemas ultra-complejos: una forma que se aparta del determinismo matemático y de la ilusión de que se puede escribir una ecuación que prediga todos los comportamientos, adoptando en su lugar una aptitud que trata primero de *observar* y de buscar un patrón, y luego de emular ese patrón utilizando cualquier medio posible (en el libro se narra, por ejemplo, la aparición de los algoritmos genéticos). Este tipo de pensamiento, en mi opinión, resulta útil para ver formas de gestionar proyectos de software cada vez más complejos.

Python

Learning Python, 2^a Edición, por Mark Lutz y David Ascher (O'Reilly, 2003). Una buena introducción a mi lenguaje favorito que constituye un excelente complemento a Java. El libro incluye una introducción a Jython, que permite combinar Java y Python en un único programa (el intérprete Jython compila los programas para generar código intermedio Java puro, por lo que no necesitamos añadir nada especial para conseguir esa combinación). Esta unión de lenguajes parece tener grandes posibilidades.

Mi propia lista de libros

No todos estos libros están actualmente disponibles, pero algunos de ellos pueden encontrarse en la librerías de segunda mano.

Computer Interfacing with Pascal & C (publicado por Eisys, 1988). Disponible a la venta únicamente en www.MindView.net). Una introducción a la electrónica escrita en los años en que CP/M seguía siendo el rey y DOS no pasaba de ser un advenedizo. Por aquel entonces, yo usaba lenguajes de alto nivel y a menudo el puerto paralelo de la computadora para controlar diversos proyectos electrónicos. Adaptado a partir de las columnas que yo escribía en la primera y mejor revista de aquellas en las que he contribuido, *Micro Cornucopia*. Esta revista, llamada también Micro C cerró mucho antes de que Internet apareciera. La creación de este libro fue una experiencia editorial extremadamente satisfactoria.

Using C++ (Osborne/McGraw-Hill, 1989). Uno de los primeros libros sobre C++. Está agotado y ha sido sustituido por su segunda edición, cuyo título es *C++ Inside & Out*.

C++ Inside & Out (Osborne/McGraw-Hill, 1993). Como ya digo, es la segunda edición de *Using C++*. El lenguaje C++ de este libro es razonablemente preciso, pero fue escrito en torno a 1992 y posteriormente fue sustituido por *Thinking in C++*. Puede encontrar más detalles acerca de este libro y descargar el código fuente en www.MindView.net.

Thinking in C++, 1^a Edición (Prentice Hall, 1995). Consiguió el premio Jolt de la revista *Software Development Magazine* como mejor libro del año.

Thinking in C++, 2^a Edición, Volumen 1 (Prentice Hall, 2000). Puede descargarlo de www.MindView.net. Actualizado para adaptarlo al estándar del lenguaje recién finalizado.

Thinking in C++, 2^a Edición, Volumen 2. Escrito en colaboración con Chuck Allison (Prentice Hall, 2003). Puede descargarlo en www.MindView.net.

Black Belt C++: The Master's Collection, Bruce Eckel, editor (M&T Books, 1994). Agotado. Una colección de capítulos escritos por diversos expertos en C++ y basados en sus presentaciones dentro del ciclo dedicado a C++ en la conferencia *Software Development Conference*, de la cual fui moderador. La cubierta de este libro fue la que me animó a decidir todos los futuros diseños de cubierta.

Thinking in Java, 1^a Edición (Prentice Hall, 1998). La primera edición de este libro ganó el premio a la productividad de la revista *Software Development Magazine*, el premio del editor de la revista *Java Developer's Journal* y el premio de los lectores de la revista *JavaWorld* como mejor libro. Puede descargarlo en www.MindView.net.

Thinking in Java, 2^a Edición (Prentice Hall, 2000). Esta edición ganó el premio del editor de la revista *JavaWorld* como mejor libro. Puede descargarse en www.MindView.net.

Thinking in Java, 3^a Edición, (Prentice Hall, 2003). Esta edición ganó el premio Jolt de la revista *Software Development Magazine* como mejor libro del año, junto con otros premios que se indican en la contraportada. Puede descargarlo de www.MindView.net.

Índice

| | | |
|--|---|--|
| ! | [| dinámico, tardío o en tiempo de ejecución · 165, 168 tardío · 10, 165, 168 temprano · 10 |
| != · 49 |] | ActionEvent · 894, 926 ActionListener · 865 ActionScript, para Macromedia Flex · 933 adaptador, patrón de diseño · 198, 204, 270, 402, 472, 474, 515 adaptadores de escucha · 873 add(), ArrayList · 242 addActionListener() · 924, 929 addChangeListener · 897 addListener · 869 Adler32 · 635 agotar la memoria, solución mediante References · 579 agregación · 6 agrupamientos, @Unit y JUnit · 717 alias, creación de · 45 Allison, Chuck · 4, 18, 955, 962 allocate() · 617 allocateDirect() · 617 ámbitos, clases internas en métodos y · 217 ampliabilidad · 171 AND: bit a bit · 55, 60 lógico (&&) · 51 anidamiento de interfaces · 206 anotaciones · 693 apt, herramientas de procesamiento de · 703 elementos · 694, 697 marcadora · 694 procesador basado en la reflexión · 700 procesador de · 696 valores predeterminados de los elementos · 695, 697 aplicación de un método a una secuencia · 468 Application Builder · 918 apt, herramienta de procesamiento de anotaciones · 703 archivos: características de · 594 cuadros de diálogo de · 901 de salida, errores y volcado · 606 File, clase · 587, 597, 602 |
| & | + | |
| & · 55 | + | |
| && · 51 | + | String, conversión con operador + · 44, 59, 317 |
| &= · 55 | < | |
| . | < · 49 | |
| .NET · 20 | << · 55 | |
| .new, sintaxis · 214 | <<= · 55 | |
| .this, sintaxis · 214 | <= · 49 | |
| @ | = | |
| @, símbolo para anotaciones · 693 | = · 49 | |
| @author · 38 | > | |
| @Deprecated, anotación · 693 | > · 49 | |
| @deprecated, marcador Javadoc · 39 | >= · 49 | |
| @docRoot · 38 | >> · 55 | |
| @inheritDoc · 38 | >>= · 55 | |
| @interfaces y extends, palabra clave · 700 | A | |
| @link · 37 | abstracción · 1 | |
| @Override · 693 | Abstract Window Toolkit (AWT) · 857 | |
| @param · 38 | abstract, palabra clave · 189 | |
| @Retention · 694 | AbstractButton · 876 | |
| @return · 38 | AbstractSequentialList · 558 | |
| @see · 37 | AbstractSet · 514 | |
| @since · 38 | acceso: clases internas y derechos de · 213 | |
| @SuppressWarnings · 693 | control de · 121, 136 | |
| @Target · 694 | control de, violación con la reflexión · 387 | |
| @Test · 694 | de clase · 134 | |
| @Test para @Unit · 709 | de paquete · 129 | |
| @TestObjectCleanup para @Unit · 715 | dentro de un directorio a través del paquete predeterminado · 130 | |
| @TestObjectCreate para @Unit · 713 | especificadores de · 5, 121, 128 | |
| @throws · 38 | acoplamiento: de las llamadas a métodos · 168 | |
| @Unit · 709 | dinámico · 168 | |
| @version · 38 | | |
| [| | |
| [], operador de indexación · 110 | | |
| ^ | | |
| ^ · 55 | | |
| ^= · 55 | | |

archivos (*continuación*)
 File.list() · 587
 JAR · 123 bloqueo de · 632
 mapeados en memoria · 629
 argumentos:
 constructor · 86
 covariantes · 453
 final · 158, 589
 inferencia del argumento de tipo · 403
 lista de argumentos variable · 113
 Arnold, Ken · 858
 ArrayBlockingQueue · 796
 ArrayList · 242, 249, 530
 add() · 242
 get() · 242
 size() · 242
 Arrays: asList() · 245, 272, 530
 binarySearch() · 508
 clase · 502
 asCharBuffer() · 619
 asignación · 44
 aspecto y estilo seleccionables · 904
 aspect-oriented programming (AOP) · 458
 assert y @Unit · 711
 atómica, operación · 760
 atomicidad en la programación
 concurrente · 754
 AtomicInteger · 764
 AtomicLong · 764
 AtomicReference · 764
 autodecremento, operador · 48
 autoincremento, operador · 48
 available() · 605

B

barra de progreso · 903
 base 16 · 53
 base 8 · 53
 base, clase · 131, 142, 168
 abstracta · 189
 constructor de la · 176
 interfaz · 171
 base, tipos · 7
 BASIC, Microsoft Visual BASIC · 918
 BasicArrowButton · 877
 Bean:
 application builder · 918
 archivo de manifiesto · 930
 BeanInfo · 931
 componente · 919
 convenio de denominación · 919
 empaquetado de · 930
 EventSetDescriptors · 922
 FeatureDescriptor · 931
 getBeanInfo() · 920
 getEventSetDescriptors() · 922
 getMethodDescriptors() · 922
 getName() · 922
 getPropertyDescriptors() · 922

get.PropertyType() · 922
 getReadMethod() · 922
 getWriteMethod() · 922
 hoja de propiedades personalizada · 931
 Introspector · 920
 Method · 922
 MethodDescriptors · 922
 programación visual · 918
 PropertyChangeEvent · 931
 PropertyDescriptors · 922
 propiedades · 918
 propiedades indexadas · 931
 PropertyVetoException · 931
 reflexión · 918, 920
 Serializable · 926
 sucesos · 919
 y Delphi de Borland · 918
 y Microsoft Visual BASIC · 918
 BeanInfo · 931
 Beck, Kent · 960
 biblioteca
 creador de, y programador de clientes · 121
 diseño de · 121
 uso · 121
 binarios, impresión de números · 58
 binarySearch() · 508, 576
 bit a bit, operadores 55:
 AND operador (&) · 55, 60
 EXCLUSIVE OR o XOR (^) · 55
 NOT ~ · 55
 OR operador (|) · 55, 60
 BitSet · 584
 Bloch, Joshua · 98, 659, 751, 762
 BlockingQueue · 796, 809
 bloqueo:
 contienda · 835
 de archivos · 632
 en concurrencia · 756
 en programas concurrentes · 729
 optimista · 847
 y available() · 605
 Booch, Grady · 960
 Boolean · 70:
 álgebra booleana · 55
 C y C++ · 51
 operadores que no funcionan con el
 tipo boolean · 49
 y proyección · 60
 borrado · 447
 en genéricos · 415
 botón:
 creación de nuestro propio · 874
 de opción · 884
 Swing · 862, 876
 BoxLayout · 868
 break etiquetado · 79
 break, palabra clave · 77
 Budd, Timothy · 2

buffer, nio · 616
 BufferedInputStream · 599
 BufferedOutputStream · 600
 BufferedReader · 304, 602, 603
 BufferedWriter · 601, 605
 búsqueda:
 de archivos de clases durante la carga · 124
 en una matriz · 508
 y ordenación en listas · 575
 ButtonGroup · 877, 884
 ByteArrayInputStream · 597
 ByteArrayOutputStream · 598
 ByteBuffer · 616

C

C#: lenguaje de programación · 20
 C++ · 49, 586
 plantillas · 394, 417
 tratamiento de excepciones · 310
 CachedThreadPool · 734
 Cadena de responsabilidad, patrón de
 diseño · 676
 Callable, concurrencia · 736
 cambio de contexto · 728
 campos, inicialización en interfaces · 205
 canal, nio · 616
 canalización · 597
 canalizaciones para la E/S · 800
 capacidad inicial de un HashMap o
 HashSet · 571
 .class, archivos · 124

carga:
 de clases · 162, 357
 inicialización y carga de clases · 162
 cargador de clases · 353
 cargador de clases primordial · 353
 case, instrucción · 82
 CASE_INSENSITIVE_ORDER, compa-
 rador de cadenas · 575, 588
 casilla de verificación · 883
 cast() · 361
 catch:
 palabra clave · 280
 capturar una excepción · 286
 cena de los filósofos, ejemplo de interblo-
 queo · 801
 CharArrayReader · 601
 CharArrayWriter · 601
 CharBuffer · 619
 CharSequence · 336
 Charset · 620
 checkedCollection() · 456
 CheckedInputStream · 634
 checkedList() · 456
 checkedMap() · 456
 CheckedOutputStream · 634
 checkedSet() · 456
 checkedSortedMap() · 456

- checkedSortedSet()** · 456
Checksum, clase · 635
Chiba, Shigeru, Dr. · 723, 724
cierre y clases internas · 229
clases · 3
 abstractas · 189
 acceso de · 134
 base · 131, 142, 168
 carga · 162, 357
 creadores de · 5
 datos de la clase · 32
 de colección · 241
 dentro de interfaces · 225
 diagramas de herencia · 155
 estilo de creación · 133
 explorador de · 134
 final · 161
 heredar de clases abstractas · 189
 heredar de clases internas · 236
 inicialización · 162, 357
 inicialización de campos de · 102
 inicialización de la clase base · 143
 inicialización de la clase derivada · 143
 inicialización de miembros · 140
instanceof/isInstance() y equivalencia de · 373
 instancia de una · 2
 literales de · 357, 367
 métodos de la clase · 32
 montaje · 357
 múltiplemente anidadas · 226
 orden de inicialización · 105
 públicas y unidades de compilación · 122
 subobjeto · 143
 tratamiento de excepciones y jerarquías de · 307
clases internas · 211-240
 anidadas · 224
 anidamiento dentro de un ámbito arbitrario · 218
 anónimas · 219, 589, 864
 cierre · 229
 derechos de acceso · 213
 en métodos y ámbitos · 217
 genéricos · 412
 heredar de · 236
 identificadores y archivos .class · 239
 local · 217
 motivación · 227
 privadas · 232, 377
 referencia al objeto de la clase externa · 215
 referencia oculta al objeto de la clase contenedora · 214
 retrollamada · 229
 static · 224
 y generalización · 216
 y hebras · 745
 y marcos de control · 230
 y super · 236
 y sustitución · 236
 y Swing · 869
Class · 878
 Class, objeto · 353, 651, 757
 forName() · 354, 872
 getCanonicalName() · 356
 getClass() · 287
 getConstructors() · 377
 getInterfaces() · 356
 getMethods() · 377
 getSimpleName() · 356
 getSuperclass() · 356
 isAssignableFrom() · 369
 isInstance() · 368
 isInterface() · 356
 límites y referencias · 360
 newInstance() · 356
 proceso de creación del objeto · 107
 referencias de, y comodines · 359
 referencias genéricas · 359
 RTTI con el objeto Class · 353
class, análisis de archivos · 721
class, palabra clave · 3, 6,
 ClassCastException · 186, 362
 ClassNotFoundException · 365
 classpath · 124
 clear(), nio · 618
 cliente, programador de · 5 y creador de bibliotecas · 5, 121
close() · 604
código:
 conducido por tablas · 674
 estándares de codificación · xxxi
 estilo de codificación · 39
 fuente · xxx
 libre de bloqueos, en programación concurrente · 760
 organización · 128
 reutilización · 139
 cola bidireccional o doble · 255, 539
 colas *Véase queue*
colisión:
 de nombres · 126
 mecanismos hash · 551
Collection · 241, 244, 266, 525
 lista de métodos · 525
 llenar con un generador · 406
 utilidades · 572
Collections:
 addAll() · 245
 enumeration() · 581
 fill() · 514
 unmodifiableList() · 530
coma, operador · 74
comando de acción · 894
comandos, patrón de diseño basado en · 235, 384, 672, 734
comentarios y documentación embebida · 35
comodines:
 de supertipo · 438
 en genéricos · 434
 no limitados · 440
 y referencias Class · 359
Comparable · 504, 533, 538
comparación de matrices · 503
Comparator · 505, 533
compareTo(), en java.lang.Comparable · 504, 535
compatibilidad · 419
compilación
 condicional · 128
 de un programa Java · 35
 unidad de · 122
 complemento a dos con signo · 58
 complemento a uno, operador · 55
componente JavaBean · 919
composición · 6, 139
 y campo de comportamiento dinámico · 184
 y herencia · 147, 152, 155, 539, 582
compresión, biblioteca de · 634
comprobación:
 de límites en matrices · 111
 pruebas unitarias basadas en anotaciones con @Unit · 709
 técnicas de · 226
concurrencia:
 almacenamiento local de habras · 771
 ArrayBlockingQueue · 796
 atomicidad · 754
 BlockingQueue · 796, 809
 Callable · 736
 canalizaciones para la E/S entre tareas · 800
 código libre de bloqueos · 760
 condición de carrera · 755
 Condition, objeto · 793
 constructores · 745
 CountDownLatch · 805
 CyclicBarrier · 807
 DelayQueue · 809
 desgajamiento de palabra · 760
 Exchanger · 820
 Executor · 734
 hebras demonio · 740
 hebras y tareas, terminología · 748
 interferencia de tareas · 753
 LinkedBlockingQueue · 796
 lock explícito · 758
 long y double · 760
 objetos activos · 850
 optimización del rendimiento · 834
 prioridad · 738
 PriorityBlockingQueue · 811
 productor-consumidor · 791

concurrencia (*continuación*)
 Prueba de Goetz para evitar la sincronización · 760
 ReadWriteLock · 848
 Regla de Brian de la sincronización · 757
 ScheduledExecutor · 814
 semáforo · 817
 señales perdidas · 788
 sleep() · 737
 SynchronousQueue · 826
 terminación de tareas · 772
 UncaughtExceptionHandler · 752
 y contenedores · 577
 y excepciones · 759
 y Swing · 910
 ConcurrentHashMap · 542, 841, 845
 ConcurrentLinkedQueue · 841
 ConcurrentModificationException · 578
 uso de CopyOnWriteArrayList para eliminar · 841, 852
 condición de carrera, concurrencia · 755
 condicional, compilación · 128
 condicional, operador · 58
 Condition, concurrencia · 793
 conjunto compartido de objetos · 817
 consola:
 entorno de visualización Swing en net.mindview.util.SwingConsole · 861
 paso de excepciones a la · 312
 constante:
 de tiempo de compilación · 156
 grupos de valores · 205
 implícitas y String · 317
 constructores · 85:
 argumentos · 86
 comportamiento de los métodos polimórficos dentro de · 181
 constructor, clase para reflexión · 375
 de la clase base · 176
 initialización de instancia · 220
 initialización durante la herencia y la composición · 147
 invocación de constructores de la clase base con argumentos · 144
 invocación desde otros constructores · 95
 nombre · 86
 orden de las llamadas a · 176
 predeterminado · 92
 predeterminado sintetizado · 377
 sin argumentos · 86, 92
 sobrecarga · 87
 static, cláusula · 108
 valor de retorno · 86
 y clases internas anónimas · 219
 y concurrencia · 745
 y finally · 303

 y polimorfismo · 175
 y tratamiento de excepciones · 302
 consultoría y formación proporcionada por MindView, Inc. · 955
 contenedores · 12
 comparación con matriz · 484
 impresión de · 247
 libres de bloqueos · 841
 pruebas de rendimiento · 558
 seguros respecto al tipo y genéricos · 242
 continue etiquetado · 79
 continue, palabra clave · 77
 contravarianza y genéricos · 438
 control de acceso · 5, 136
 control de procesos · 614
 conversión:
 de ensanchamiento · 61
 de estrechamiento · 61
 de unidades de tiempo · 811
 conversión automática de tipos · 261, 402
 y genéricos · 403, 446
 Coplien, Jim · 451
 CopyOnWriteArrayList · 821, 841
 CopyOnWriteArraySet · 841
 copyright, código fuente · xxx
 cortocircuitos y operadores lógicos · 52
 CountDownLatch, para concurrencia · 805
 covariante:
 argumento · 453
 matrices · 434
 tipo de retorno · 183, 371, 453
 CRC32 · 635
 CSS (Cascading Style Sheets) y Macromedia Flex · 936
 cuadro combinado · 885
 cuadros de diálogo · 898
 con fichas · 887
 de archivos · 901
 cuantificadores · 335
 CyclicBarrier, para concurrencia · 807

D

DatagramChannel · 632
 DataInput · 603
 DataInputStream · 599, 602, 604
 DataOutput · 603
 DataOutputStream · 600, 602
 datos:
 final · 156
 initialización de datos estáticos · 106
 tipos de datos primitivos y uso con operadores · 62
 decode(), conjunto de caracteres · 620
 Decorador, patrón de diseño · 461
 decremento, operador · 48
 default, palabra clave en una instrucción switch · 82
 defaultReadObject() · 648

defaultWriteObject() · 647
 DeflaterOutputStream · 634
 Delayed · 811
 DelayQueue, para concurrencia · 809
 delegación · 145, 461
 Delphi de Borland · 918
 DeMarco, Tom · 961
 demonio, hebras · 740
 depuración de memoria · 97, 98, 100
 cómo funciona el depurador de memoria · 100
 objetos alcanzables · 578
 y limpieza · 148
 desacoplamiento por polimorfismo · 10, 165
 Desarrollo rápido de aplicaciones. *Véase RAD.*
 desbordamiento y tipos primitivos · 70
 descompilador javap · 318, 389, 422
 desgajamiento de palabra · 760
 desigual, matriz · 488
 deslizador · 903
 despacho múltiple:
 con EnumMap · 690
 y enumeraciones · 684
 despacho simple · 684
 desplazamiento a la derecha, operador (>>) · 55
 desplazamiento a la izquierda, operador (<<) · 55
 desplazamiento, operadores de · 55
 destructor · 97, 98, 296 Java no tiene 148
 diagramas de herencia · 11, 155
 dibujo en un JPanel en Swing · 895
 diccionario · 244
 diferida, inicialización · 140
 Dijkstra, Edsger · 801
 directorios:
 búsqueda y creación de · 594
 listados de · 587
 y paquetes · 128
 disciplina de gestión de colas · 264
 diseño · 183
 adición de más métodos al · 137
 de bibliotecas · 121
 y composición · 183
 y errores · 137
 y herencia · 183
 dispose() · 899
 disposición, control de la · 865
 división · 46
 doble despacho · 684
 con EnumMap · 690
 documentación · 17
 comentarios y documentación embedida · 35
 double:
 valor literal (d o D) · 53
 y hebras · 760

- do-while · 73
 downcasting. *Véase* especialización
-
- E**
- East, BorderLayout · 866
 editor, creación con JTextPane · 882
 efecto colateral · 44, 49, 92
 eficiencia:
 y final · 161
 y matrices · 483
 ejecución de programas del sistema operativo desde dentro de Java · 614
 ejecución de un programa Java · 35
 else, palabra clave · 71
 encadenamiento de excepciones · 291, 313
 encapsulación · 133, 387
 encode(), conjunto de caracteres · 620
 entorno de visualización para Swing · 861
Entrada/salida:
 available() · 605
 biblioteca · 587
 biblioteca de compresión · 634
 bloqueo y available() · 605
 BufferedInputStream · 599
 BufferedOutputStream · 600
 BufferedReader · 304, 602, 603
 BufferedWriter · 602, 605
 ByteArrayInputStream · 597
 ByteArrayOutputStream · 598
 canalización · 597
 canalizaciones para la · 800
 características de archivos · 594
 CharArrayReader · 601
 CharArrayWriter · 601
 CheckedInputStream · 634
 CheckedOutputStream · 634
 close() · 604
 configuraciones típicas de · 603
 control en la serialización · 642
 creación y búsqueda de directorios · 594
 DataInput · 603
 DataInputStream · 599, 602, 605
 DataOutput · 603
 DataOutputStream · 600, 602
 DeflaterOutputStream · 634
 E/S de red · 616
 entrada · 596
 entrada estándar, lectura de la · 613
 Externalizable · 643
 File · 597, 602
 File, clase · 587
 File.list() · 587
 FileDescriptor · 597
 FileInputStream · 603
 FileInputStream · 597
 FilenameFilter · 587
 FileOutputStream · 598
 FileReader · 304, 601
- FileWriter · 601, 605
 FilterOutputStream · 598
 FilterReader · 602
 FilterWriter · 602
 flujo de datos canalizado · 609
 GZIPInputStream · 634
 GZIPOutputStream · 634
 InflaterInputStream · 634
 InputStream · 596
 InputStreamReader · 600, 601
 internacionalización · 601
 interrumpible · 778
 LineNumberInputStream · 599
 LineNumberReader · 602
 listado de directorios · 587
 mark() · 603
 mkdirs() · 596
 new nio · 616
 ObjectOutputStream · 639
 OutputStream · 596, 598
 OutputStreamWriter · 600, 601
 persistencia ligera · 639
 PipedInputStream · 597
 PipedOutputStream · 597, 598
 PipedReader · 601
 PipedWriter · 601
 PrintStream · 600
 PrintWriter · 602, 605, 606
 PushbackInputStream · 599
 PushbackReader · 602
 RandomAccessFile · 602, 603, 608
 read() · 596
 readDouble() · 608
 Reader · 596, 600, 601
 readExternal() · 643
 readLine() · 305, 602, 606, 613
 readObject() · 639
 redireccionamiento de la E/S estándar · 613
 renameTo() · 596
 reset() · 603
 salida · 596
 seek() · 602, 608
 SequenceInputStream · 597, 602
 Serializable · 643
 setErr(PrintStream) · 614
 setIn(InputStream) · 614
 setOut(PrintStream) · 614
 StreamTokenizer · 602
 StringBuffer · 597
 StringBufferInputStream · 597
 StringReader · 601, 604
 StringWriter · 601
 System.err · 613
 System.in · 613
 System.out · 613
 transient · 646
 Unicode · 601
 uso básico, ejemplos · 603
- write() · 596
 writeBytes() · 607
 writeChars() · 607
 writeDouble() · 607
 writeExternal() · 643
 writeObject() · 639
 Writer · 596, 600, 601
 ZipEntry · 637
 ZipInputStream · 634
 ZipOutputStream · 634
entrySet() en Map · 549
enum:
 adicción de métodos · 661
 e importaciones estáticas · 660
 e interfaces · 667
 grupos de valores constantes en C y C++ · 205
 métodos específicos de constante · 673,
 688
 palabra clave · 117, 659
 values() · 659, 663
 y despacho múltiple · 684
 y herencia · 665
 y máquinas de estado · 680
 y patrón de diseño Cadena de responsabilidad · 676
 y selección aleatoria · 666
 y switch · 662
enumeración *Véase* enum
 enumerados, tipos · 117
 Enumeration · 581
 EnumMap · 672
 EnumSet · 410, 585 en lugar de indicadores · 671
 envío de mensajes · 3
 equals() · 50
 condiciones que debe satisfacer · 547
 y estructuras de datos hash · 548
 y hashCode() · 533, 548, 554
 y HashMap · 547
 equivalencia de objetos · 49
 equivalente (==) · 49
 Erlang, lenguaje · 730
errores:
 del libro · xxxii
 error estándar · 282
 información de · 309
 recuperación · 277
 tratamiento de errores mediante excepciones · 277
 y diseño · 137
es-como-un, relación · 9
escucha:
 adaptadores de · 873
 interfaces · 872
 y sucesos · 869
 espacio de la solución · 2
 espacio de nombres · 122
 espacio del problema · 2

especialización · 155, 18, 362
 especificación de la excepción · 286, 309
 especificación explícita del tipo de argumento · 247, 405
 especificadores de acceso · 5, 121, 128
 espera activa, concurrencia · 784
 estándares de codificación · xxxi
 estático (static)
 bloque · 108
 comprobación de tipos · 392
 import y enum · 660
 initialización de datos · 106
 initializador · 371
 synchronized · 757
 y comprobación de tipos dinámicos 528
 y final · 156
 estilo de codificación · 39
 de creación de clases · 133
 estrategia, patrón de diseño basado en · 196, 203, 474, 494, 504, 588, 593, 676, 811
 estructural, tipo · 464, 472
 es-un, relación · 9, 153
 etiqueta (label) · 78
 EventSetDescriptors · 922
 excepciones:
 capturar una excepción · 286
 comprobadas · 286, 309
 condición excepcional · 278
 constructores · 303
 conversión de comprobadas a no comprobadas · 313
 creación de nuestras propias · 281
 detección de una excepción · 279
 directrices de uso · 315
 encadenamiento de · 291, 313
 Error, clase · 294
 especificación de · 286, 310
 Exception, clase · 294
 FileNotFoundException · 304
 fillInStackTrace() · 289
 finally · 295
 generar · 278
 genéricos · 457
 informe de excepciones mediante logger · 284
 jerarquías de clases y · 307
 localización de · 307
 no comprobadas · 294
 NullPointerException · 294
 perdida · 300
 printStackTrace() · 289
 problemas de diseño · 304
 regeneración de una excepción · 289
 región protegida · 279
 registro · 283
 restricciones · 301
 RuntimeException · 294
 rutina de tratamiento de · 278, 280

terminación y reanudación · 281
 Throwable · 287
 tratamiento de · 277
 try, bloque · 280, 297
 y concurrencia · 759
 y constructores · 302
 y herencia · 301, 307
 y la consola · 312
 Exchanger · 820
 exclusión mutua (mutex), concurrencia · 756
 Executor, concurrencia · 734
 ExecutorService · 734
 explorador de clases · 134
 expresiones regulares · 331
 extends · 131, 142, 185
 e interfaces · 202
 palabra clave · 142
 y @interfaces · 700
 extensión con ceros · 55
 extensión de signo · 55
 Externalizable · 643
 una alternativa a · 647
 Extreme Programming (XP) · 960

F

factor de carga de HashMap o HashSet · 571
 Factorías registradas, variante del patrón de diseño método de factoría · 371
 fallo rápido, contenedores de · 578
 false · 51
 FeatureDescriptor · 931
 Fibonacci · 401
 Field, para reflexión · 375
 FIFO (first-in, first out) · 263
 File, clase · 587
 FileChannel · 616
 FileDescriptor · 597
 FileReader · 603
 FileInputStream · 597
 FileLock · 632
 FilenameFilter · 587
 FileNotFoundException · 304
 FileOutputStream · 598
 FileReader · 304, 601
 FileWriter · 601, 605
 fillInStackTrace() · 289
 FilterInputStream · 597
 FilterOutputStream · 598
 FilterReader · 602
 FilterWriter · 602
 final · 192, 396
 argumentos · 158,
 clases · 161
 con referencias a objeto · 156
 datos · 156
 métodos · 159, 168, 182
 palabra clave · 156

valores final en blanco · 158, 589
 y eficiencia · 161
 y private · 159
 y static · 156
 finalize() · 97, 305
 finally · 148, 150, 295
 error · 300
 no ejecutar con hebras demonio · 743
 palabra clave · 295
 y constructores · 303
 y return · 299
 FixedThreadPool · 734
 Flex · 932
 flip(), nio · 617
 float: valor literal (F) · 53
 FlowLayout · 867
 flujo de dato de E/S · 596
 flujo de datos canalizado · 609
 for, palabra clave · 73
 foreach · 74, 78, 114, 115, 127, 232, 243, 262, 267, 345, 401, 402, 446, 659, 677
 e Iterable · 269
 y método basado en adaptadores · 270
 forma (shape): ejemplo · 7, 169
 y RTTI · 351
 format() · 324
 formato:
 anchura · 326
 de cadenas de caracteres · 324
 especificadores de · 326
 precisión · 326
 Formatter · 325
 forName() · 354, 872
 FORTRAN, lenguaje de programación · 54
 Fowler, Martin · 121, 960
 función hash perfecta · 551
 Función, objetos de · 474
 Future · 737

G

generador · 399, 406, 494, 515, 666, 681
 de propósito general · 407
 para llenar un objeto Collection · 406
 generalización · 11, 154, 165 y RTTI · 352 y clases internas · 216
 generar una excepción · 278
 Generator · 412, 446, 471, 494, 505 Véase
 Generator
 genérico curiosamente recurrente · 451
 @Unit con · 716
 genéricos:
 borrado · 415, 447
 clases internas anónimas · 412
 comodines · 434
 comodines de supertipo · 438
 comodines no limitados · 440
 contravarianza · 438

curiosamente recurrente · 451
 definición de clase más simple 256
 especificación explícita del tipo de argumento para métodos genéricos · 247, 405
 excepciones · 457
`instanceof` · 424, 447
 introducción · 242
`isInstance()` · 424
 límites · 418, 431
 marcador de tipos · 424
 matriz de objetos · 552
 métodos · 403, 515
 proyección · 447
 proyección mediante una clase genérica · 448
 reificación · 419
 sobrecarga · 449
 tipos autolimitados · 450
 varargs y métodos genéricos · 405
 y contenedores seguros respecto al tipo · 242
 gestor de invocaciones para proxy dinámico · 379
`get()`:
 `ArrayList` · 242
 `HashMap` · 261
 no hay `get()` para `Collection` · 525
`getBeanInfo()` · 920
`getBytes()` · 605
`getCanonicalName()` · 356
`getChannel()` · 617
`getClass()` · 287, 354
`getConstructor()` · 878
`getConstructors()` · 377
`getenv()` · 269
`getEventSetDescriptors()` · 922
`getInterfaces()` · 356
`getMethodDescriptors()` · 922
`getMethods()` · 377
`getName()` · 922
`getPropertyDescriptors()` · 922
`getPropertyType()` · 922
`getReadMethod()` · 922
`getSelectedValues()` · 886
`getSimpleName()` · 356
`getState()` · 894
`getSuperclass()` · 356
`getWriteMethod()` · 922
 Glass, Robert · 961
 Goetz, prueba de, para evitar la sincronización · 760
 Goetz, Brian · 757, 760, 835, 855
 goto, no existe en Java · 78
 Graphics, clase · 896
`GridBagLayout` · 868
`GridLayout` · 867, 917
 Grindstaff, Chris · 941
 grupo de hebras · 751

grupos, expresiones regulares · 338
 GUI (*graphical user interface*) · 231, 857
 constructores de interfaces · 858
`GZIPInputStream` · 634
`GZIPOutputStream` · 634

H

Harold, Elliotte Rusty · 931, 960
 XOM, biblioteca XML · 654
 hash, almacenamiento · 545, 550
 encadenamiento externo · 551
 función *hash* perfecta · 551
 y códigos *hash* · 545
hash, función · 550
`hashCode()` · 541, 545, 550, 553
 `equals()` · 533
 problemas · 553
 y estructuras de datos *hash* · 548
`HashMap` · 542, 571, 845, 876
`HashSet` · 258, 533, 567
`Hashtable` · 570, 582
`hasNext()`, `Iterator` · 253
 hebras:
 almacenamiento local de · 771
 estados de las · 775
 grupo de · 751
 `interrupt()` · 776
 `isDaemon()` · 742
 `notifyAll()` · 784
 planificador de · 732
 prioridad · 738
 resume() e interbloqueos · 775
 safety, biblioteca estándar Java · 807
 stop() e interbloqueos · 776
 suspend(), e interbloqueos · 775
 wait() · 784
 y tareas, terminología · 748
 herencia · 6, 131, 139, 142, 165
 ampliación de interfaces mediante la · 201
 de clases abstractas · 189
 de clases internas · 236
 de múltiples implementaciones · 228
 diagrama de · 11
 diagramas de · 155
 diseño de sistemas con · 183
 inicialización con · 162
 múltiple en Java · 199
 pura y extensión · 184
 sobrecarga y sustitución de métodos · 151
 y código genérico · 393
 y composición · 147, 152, 155, 539, 582
 y enum · 665
 y final · 161
 y synchronized · 929
 herramienta abstracta de ventanas (AWT) · 857

hexadecimal · 53
 hojas de estilo en cascada *Véase CSS*
 Holub, Allen · 851
 HTML en componentes Swing · 902

I

Iconos · 878
`IdentityHashMap` · 542, 570
`if-else`, instrucción · 71
`if-else`, operador ternario · 58
`IllegalAccessException` · 365
`IllegalMonitorStateException` · 785
`ImageIcon` · 878
 implementación · 4:
 e interfaz · 5, 133, 152, 192, 868
 ocultación · 121, 133, 216
 implements, palabra clave · 192
`import`, palabra clave · 122
 incremento, operador · 48
 y concurrencia · 755
 indexación, operador `[]` · 110
`indexOf()`, `String` · 377
 indicador de fin · 399
 indicadores en lugar de `EnumSet` · 671
 inferencia del argumento de tipo · 403
`InflaterInputStream` · 634
 información de tipos en tiempo de ejecución. *Véase RTTI*
 ingeniería de código intermedio · 721
 Javassist · 723
 inicialización:
 con constructor · 85
 con herencia · 162
 de campos de clases · 102
 de constructores durante la herencia y composición · 147
 de instancia · 140
 de instancia no estática · 109
 de la clase base · 143
 de matrices · 110
 de variables de métodos · 102
 diferida · 140
 orden de inicialización · 105, 182
 static · 163
 static explícita · 108
 y carga de clases · 162, 357
`InputStream` · 596
`InputStreamReader` · 600, 601
`instanceof` · 367
 `instanceof` dinámico con `isInstance()` · 368
 palabra clave · 362
 y tipos genéricos · 447
 instancia:
 de una clase · 2
 inicialización de · 220
 inicialización de instancia no estática · 109
 interbloqueo, en concurrencia 801

intercambiador *Véase* Exchanger
 interface, palabra clave · 192
 interfaces 189-210:
 anidamiento de · 206
 clases dentro de · 225
 colisiones de nombres al combinar · 202
 común · 189
 de un objeto · 3
 e implementación · 5, 133, 152 868
 generalización a una · 194
 inicialización de campos en las · 205
 y clases abstractas · 200
 y código genérico · 393
 y enum · 667
 y factorías · 208
 y herencia · 201
 interfaz común · 189
 interfaz gráfica de usuario (GUI) 231, 857
 internacionalización en la biblioteca de E/S · 601
 interrupt():
 conurrencia · 776
 hebras · 748
 Introspector · 920
 isAssignableFrom(), método de Class · 369
 isDaemon() · 742
 isInstance() · 368
 y genéricos · 424
 isInterface() · 356
 Iterable · 269, 401, 517
 y foreach · 269
 y matrices · 269
 Iterador nulo, patrón de diseño · 381
 Iterador, patrón de diseño · 214
 Iterator · 252, 253, 266
 hasNext() · 252
 next() · 252

J

Jacobsen, Ivar · 960
 JApplet · 866 menús · 890
 JAR · 930
 archivo · 123
 archivos jar y classpath · 125
 utilidad · 637
 Java:
 AWT · 857
 compilación y ejecución de un programa · 35
 Java Foundation Classes (JFC/Swing) · 857
 Java Web Start · 906
 JVM · 353
 seguridad de las hebras en la biblioteca · 807
 y lenguaje ensamblador · 319
 JavaBean. *Véase* Bean · 918
 javac · 35

javadoc · 36
 javap, descompilador · 318, 389, 422
 Javassist · 723
 JButton · 878
 Swing · 862
 JCheckBox · 878, 883
 JCheckBoxMenuItem · 891, 893
 JComboBox · 885
 JComponent · 880, 896
 JDialog · 898, 890
 JDK, descarga e instalación · 35
 JFC, Java Foundation Classes (Swing) · 857
 JFileChooser · 901
 JFrame · 866, 890
 JIT, compiladores just-in-time · 102
 JLabel · 881
 JList · 886
 JMenu · 890, 893
 JMenuBar · 890, 893
 JMenuItem · 878, 890, 893, 894, 895
 JNLP, Java Network Launch Protocol · 906
 join(), hebras · 748
 JOptionPane · 888
 Joy, Bill · 49
 JPanel · 877, 896, 917
 JPopupMenu · 894
 JProgressBar · 904
 JRadioButton · 878, 884
 JScrollPane · 865, 887
 JSlider · 904
 JTabbedPane · 887
 JTextArea · 864
 JTextField · 863, 880
 JTextPane · 882
 JToggleButton · 877
 JUnit, problemas con · 709
 JVM (Java Virtual Machine) · 353

K

keySet() · 571

L

latente, tipo · 464, 472
 lectura de la entrada estándar · 613
 length:
 miembro de matriz · 111
 para matrices · 485
 lenguajes funcionales · 730
 lexicográfica: ordenación · 260, 507
 LIFO (*last-in, first-out*) · 256
 ligero: objeto · 252
 límites: y referencias Class · 360
 en genéricos · 418, 431
 superclase y referencias Class · 361
 tipos genéricos autolimitados · 450
 limpieza:
 con finally · 296

realización · 98
 verificar la condición de terminación con finalize() · 98
 y depurador de memoria · 148
 LineNumberInputStream · 599
 LineNumberReader · 602
 LinkedBlockingQueue · 796
 LinkedHashMap · 542, 545, 571
 LinkedHashSet · 258, 533, 567, 569
 LinkedList · 249, 255, 263, 530
 List · 241, 244, 249, 530
 comparación de rendimiento · 561
 ordenaciones y búsqueda · 575
 lista:
 cuadro de · 886
 desplegable · 885
 listas variables de argumentos (varargs) · 113, 469
 y métodos genéricos · 405
 Lister, Timothy · 961
 ListIterator · 530
 literales:
 de clase · 357, 367
 double · 53
 float · 53
 long · 53
 valores · 53
 local:
 clase interna · 217
 variable · 29
 lock explícito, objeto · 758
 logaritmo natural · 54
 long: y hebras · 760 valor literal (L) · 53
 LRU (*least-recently-used*) · 545
 lvalor · 44

M

Macromedia Flex · 932
 main() · 142
 manifiesto, archivo de, para archivos JAR · 637, 930
 Map · 241, 244, 260, 540
 comparación de rendimiento · 569
 EnumMap · 672
 Map.Entry · 549
 MappedByteBuffer · 629
 maqueta, objeto · 387
 máquinas de estado con enumeraciones · 680
 marcadora, anotación · 694
 marco de trabajo de una aplicación · 231
 marcos de control y clases internas · 230
 mark() · 603
 matcher, expresión regular · 336
 matrices:
 asociativa · 244, 244, 540
 comparación con contenedor · 484
 comparación de · 503
 comprobación de límites · 111

- copia en · 502
 covariante · 434
 de objetos · 485
 de objetos genéricos · 552
 de primitivas · 485
 desigual · 488
 devolución de una matriz · 487
 inicialización · 110
 inicialización agregada dinámica · 486
 Iterable · 269
 length · 111, 485
 multidimensional · 488
 objetos de primera clase · 485
 mayor o igual que (\geq) · 49
 mayor que ($>$) · 49
 menor o igual que (\leq) · 49
 menor que ($<$) · 49
 mensaje, envío de · 3
 Mensajero · 396, 516, 559
 menús:
 JDialog, JApplet, JFrame · 890
 JPopupMenu · 894
 meta-anotaciones · 695
 metadatos · 693
 Method · 375, 922
 MethodDescriptors · 922
 método de factoría, patrón de diseño · 208,
 222, 371, 399, 604
 método de plantillas, patrón de diseño
 231, 365, 426, 558, 631, 768, 840,
 843
 métodos:
 acoplamiento de las llamadas a · 168
 adicción de más métodos al diseño · 137
 aplicación de un método a una secuencia · 469
 clases internas en ámbitos y · 217
 comportamiento de los métodos polimórficos entre de constructores · 181
 creación de alias en las llamadas a · 46
 distinción entre métodos sobrecargados · 88
 final · 159, 168, 182
 genéricos · 403
 inicialización de variables de · 102
 llamadas a métodos en línea · 159
 polimorfismo · 165
 private · 173,
 protected · 153
 recursivos · 322
 sobrecarga · 87
 static · 96
 sustitución de métodos private · 173
 Meyer, Jeremy · 693, 721, 906
 Meyers, Scott · 5
 micropuebas de rendimiento · 566, 835
 Microsoft Visual BASIC · 918
 miembro:
 de datos · 4
 inicializadores de · 177
 objeto · 6
 migración, compatibilidad de la · 419
 mixin · 458
 mkdirs() · 596
 mnemónicos (atajos de teclado) · 894
 módulo · 46
 monitor, para concurrencia · 756
 Mono · 20
 montaje de clases · 357
 multidifusión · 926
 multidimensional, matriz · 488
 multiparadigma, programación · 2
 multiplicación · 46
 multitarea · 729
 MXML, Macromedia Flex formato de
 entrada · 932
 mxmle, Macromedia Flex compilador ·
 933
-
- N**
- net.mindview.util.SwingConsole · 861
 Neville, Sean · 932
 new, E/S · 616
 new, operador · 97
 y matrices · 111
 newInstance() · 356, 878
 reflexión · 356
 next(), Iterator · 253
 nio · 616
 buffer · 616
 channel · 616
 rendimiento · 630
 no equivalente (\neq) · 49
 no modificable, colección o mapa · 576
 nombres:
 al combinar interfaces · 202
 colisiones de · 126
 cualificados · 356
 de paquetes · 31, 124
 North, BorderLayout · 866
 NOT, lógico (!) · 51
 notación exponencial · 54
 notifyAll() · 784
 notifyListeners() · 929
 null · 26
 NullPointerException · 294
-
- O**
- Object:
 clase raíz estándar · 142
 herencia · 142
 ObjectOutputStream · 639
 objeto · 2
 activo · 850
 alcanzable · 578
 asignación de objetos por copia de referencias · 45
 bloqueo, para concurrencia · 756
 Class · 353, 651, 757
 creación · 86
 creación de alias · 45
 de transferencia de datos · 396, 516,
 559
 equals() · 50
 equivalencia de objetos y de referencias
 50
 equivalente (\equiv) · 49
 final · 156
 getClass() · 354
 hashCode() · 541
 información de tipos · 351
 interfaz de un · 3
 matrices son objetos de primera clase ·
 485
 miembro · 6
 proceso de creación del · 107
 red de objetos · 639
 serialización · 639
 wait() y notifyAll() · 784
 Objeto nulo, patrón de diseño · 381
 Octal · 53
 onda sinusoidal · 896
 OpenLaszlo, alternativa a Flex · 932
 operación atómica · 760
 operaciones no soportadas en contenedores Java · 529
 operaciones opcionales en contenedores Java · 528
 + y += para String · 59, 142
 +, para String · 317
 operadores · 44
 bit a bit · 55
 coma, operador · 74
 complemento a uno · 55
 de desplazamiento · 55
 de indexación [] · 110
 de proyección · 60
 errores comunes · 60
 lógicos · 51
 lógicos y cortocircuitos · 52
 matemáticos · 46, 633
 precedencia · 44
 relacionales · 49
 sobrecarga de · 59
 sobrecarga de operador para String ·
 318
 String, conversión con operador + · 44,
 59
 ternario · 58
 unarios · 48, 55
 OR:
 bit a bit · 55, 60
 lógico (||) · 51
 orden:
 de inicialización · 105, 162, 182
 de llamadas a constructores · 176

ordenación · 504
 alfábetica · 260
 lexicográfica · 260
 y búsquedas en listas · 575
 ordinal(), para enum · 660
 organización del código · 128
 OSExecute · 615
 OutputStream · 596, 598
 OutputStreamWriter · 600, 601

P

paintComponent() · 896, 900
 paquete (package) · 122
 acceso de · 221
 acceso de, y protected · 153
 nombres únicos de · 124
 nombres, uso de mayúsculas · 31
 predeterminado · 122, 130
 y estructura de directorios · 128
 parámetro de recopilación, · 458, 478
 pato, tipos · 464, 472
 patrón de diseño:
 adaptador · 198, 204, 402, 472, 474, 515
 basado en comandos · 235, 384, 672, 734
 basado en estados · 184
 basado en estrategia · 196, 203, 474, 494, 504, 588, 593, 676, 811
 cadena de responsabilidad · 676
 Decorador · 461
 Iterador · 214, 252
 Iterador nulo · 381
 método basado en adaptadores · 270
 método de factoría · 208, 222, 371, 399, 604
 método de plantillas · 231, 365, 426, 558, 631, 768, 840, 843
 objeto de transferencia de datos (Mensajero) · 396, 516, 559
 Objeto nulo · 381
 Peso mosca · 519, 854
 Proxy · 378
 Singleton · 136
 solitario · 136
 Visitante · 706
 pattern, expresiones regulares · 333
 persistencia · 649
 ligera · 639
 Peso mosca, patrón de diseño · 519, 854
 PhantomReference · 578
 Piedra, papel, tijera · 685
 pila · 255, 256, 398, 582
 PipedInputStream · 597
 PipedOutputStream · 597, 598
 PipedReader · 601, 800
 PipedWriter · 601, 800
 planificador de hebras · 732
 plantillas C++ · 394, 417

Plauger, P.J. · 961
 polimorfismo · 9, 165-187, 351, 391
 comportamiento de los métodos polimórficos dentro de constructores · 181
 y constructores · 175
 y despacho múltiple · 684
 POO (programación orientada a objetos):
 características básicas · 2
 conceptos básicos · 1
 protocolo · 192
 Simula-67, lenguaje de programación · 3
 suptlación · 2
 posicionamiento absoluto · 868
 post-decremento · 48
 postfija · 48
 post-incremento · 48
 pre-decremento · 48
 predeterminado, constructor · 92, 144, 377
 predeterminado, paquete · 122, 130
 predicción, operadores de, 60
 preferences, API · 656
 prefija · 48
 pre-incremento · 48
 prerequisitos para este libro · 1
 primitivos:
 comparación · 50
 final · 156
 final y static · 156
 inicialización de campos de clases · 102
 tipos · 25
 tipos de datos primitivos y uso con operadores · 62
 printf() · 324
 printStackTrace() · 287, 289
 PrintStream · 600
 PrintWriter · 602, 605, 606
 constructor en Java SE5 · 610
 prioridad, en concurrencia · 738
 PriorityBlockingQueue, concurrencia · 811
 PriorityQueue · 264, 537
 private · 5, 121, 128, 130, 153, 159, 756
 clases internas · 232
 interfaces anidadas · 207
 sustitución de métodos · 173
 proceso concurrente · 729
 ProcessBuilder · 615
 ProcessFiles · 720
 productor-consumidor, concurrencia · 791
 programación basada en agentes · 853
 programación multiparadigma · 2
 programación orientada a aspectos (AOP) · 458
 programación orientada a objetos (POO),
 Véase POO
 programación visual · 918
 entornos de · 858
 programador de clientes · 5

promoción a int · 62, 70
 PropertyChangeEvent · 931
 PropertyDescriptors · 922
 propiedades:
 hoja de propiedades personalizada · 931
 indexadas · 931
 restringidas · 931
 PropertyVetoException · 931
 protected · 5, 121, 128, 131, 153
 y acceso de paquete · 131, 153
 protocol · 192
 Protocolo Java de inicio a través de red (JNLP) · 906
 Proxy, patrón de diseño · 378
 proxy: y java.lang.ref.Reference · 579
 para métodos no modificables de la clase Collections · 530
 proyección · 11
 asSubclass() · 361
 mediante una clase genérica · 448
 y tipos genéricos · 447
 y tipos primitivos · 70
 public · 5, 121, 128, 128
 clase, y unidades de compilación · 122
 e interface · 192
 puntero, exclusión de punteros en Java · 229
 PushbackInputStream · 599
 PushbackReader · 602
 Python 1, 5, 9, 18, 22, 464, 510, 729, 962
 queue · 241, 255, 263, 537, 796
 rendimiento · 561

R

RAD (Rapid Application Development) · 375
 random() · 260
 RandomAccess, interfaz de marcado para contenedores · 275
 RandomAccessFile · 602, 603, 608, 616
 read() · 596 nio · 616
 readDouble() · 608
 Reader · 596, 600, 601
 readExternal() · 643
 readLine() · 305, 602, 606, 613
 readObject() · 639
 con Serializable · 647
 ReadWriteLock · 848
 reanudación en el tratamiento de excepciones · 281
 recuadros de mensaje en Swing · 888
 recuento de referencias, depurador de memoria · 100
 recursión no intencionada con toString() · 321
 red de objetos · 639
 red, E/S de · 616
 redireccionamiento de la E/S estándar 613
 rediseño · 121

- ReentrantLock · 759, 781
 Reference de java.lang.ref · 578
 referencia:
 asignación de objetos por copia de referencias · 45
 equivalencia de referencias y equivalencia de objetos · 50
 final · 156 hallar el tipo exacto de referencia base · 352
 null · 26
 referencias Class genéricas · 359
 reflexión · 375, 387, 870, 920
 diferencia entre RTTI y · 375
 ejemplo · 877
 procesador de anotaciones · 696, 700
 tipos latentes y genéricos · 467
 y Beans · 918
 regeneración de una excepción · 289
 regex · 333
 región protegida en el tratamiento de excepciones · 279
 registro y excepciones · 283
 Regla de Brian de la sincronización · 757
 rehashing · 571
 reificación y genéricos · 419
 removeActionListener() · 924, 929
 removeXXXListener() · 869
 renameTo() · 596
 rendimiento:
 nio · 630
 optimización del · 834
 pruebas de · 558
 y final · 161
 reset() · 603
 respuesta rápida, interfaces de usuario de · 750
 resta · 46
 resume() e interbloqueos · 775
 retorno:
 sobrecarga de los valores de · 92
 tipos de retorno covariantes · 183, 453
 valor de retorno de constructor · 86
 retrollamada · 588, 863
 y clases internas · 229
 return:
 devolución de múltiples objetos · 396
 devolución de una matriz · 487
 y finally · 299
 reutilización · 6
 código reutilizable · 918
 de código · 139
 rewind() · 620
 RTTI (runtime type information) 186, 351
 Class, objeto · 353, 878
 ClassCastException · 362
 Constructor, clase · 375
 Field · 375
 getConstructor() · 878
 instanceof, palabra clave · 362
 isInstance() · 368
 Method · 375
 newInstance() · 878
 reflexión · 375
 shape, ejemplo · 351
 Rumbaugh, James · 960
 RuntimeException · 294, 313
 rutina de tratamiento de excepciones, 280
 rvalor · 44
-
- S**
- salto incondicional · 76
 ScheduledExecutor, para concurrencia · 814
 sección crítica y bloque synchronized · 765
 secuencia, aplicación de un método a una · 468
 seek() · 602, 608
 selección aleatoria y enum · 666
 semáforo contador · 817
 seminarios xxiii
 formación proporcionada por MindView, Inc. · 955
 señales perdidas, concurrencia · 788
 separación de la interfaz y la implementación · 5, 133, 869
 SequenceInputStream · 597, 602
 Serializable · 639, 643, 646, 653, 926.
 Véase también serialización
 readObject() · 647
 writeObject() · 647
 serialización:
 control en la · 642
 defaultReadObject() · 648
 defaultWriteObject() · 647
 Versionado · 649
 y almacenamiento de objeto · 649
 y transient · 646
 Set · 241, 244, 258, 533
 comparación de rendimiento · 567
 relaciones matemáticas · 409
 setActionCommand() · 894
 setBorder() · 881
 setErr(PrintStream) · 614
 setIcon() · 879
 setIn(InputStream) · 614
 setLayout() · 866
 setMnemonic() · 894
 setOut(PrintStream) · 614
 setToolTipText() · 880
 shuffle() · 576
 firma del método · 30
 Simula-67, lenguaje de programación · 3
 simulación · 821
 SingleThreadExecutor · 735
 size(), ArrayList · 242
 sizeof(), no existe en Java · 62
 sleep(), en concurrencia · 737
 Smalltalk · 2
- sobrecarga:
 de constructores · 87
 de los valores de retorno · 92
 de métodos · 87
 de operadores · 59
 distinción entre métodos sobrecargados · 88
 genéricos · 449
 ocultación de nombres durante la herencia · 151
 operadores + y += para String · 142, 318
 SocketChannel · 632
 SoftReference · 578
 Software Development Conference · xxiii
 solicitud · 3
 Solitario (singleton), patrón de diseño 136
 SortedMap · 544
 SortedSet · 536
 South, BorderLayout · 866
 split(), String · 196, 332
 sprintf() · 329
 SQL generado mediante anotaciones · 698
 stateChanged() · 897
 static: palabra clave · 32, 192
 clases internas · 224
 inicialización 163, 354
 método · 96,
 stop() e interbloqueos · 775
 StreamTokenizer · 602
 String:
 CASE_INSENSITIVE_ORDER · 575
 concatenación con el operador += · 59
 conversión con operador + · 44, 59
 expresiones regulares · 331
 format() · 329
 indexOf() · 377
 inmutabilidad · 317
 métodos · 317, 322
 operadores + y += para · 142
 ordenación lexicográfica y alfabetica · 507
 ordenación, CASE_INSENSITIVE_ORDER · 588
 split(), método · 196
 toString() · 140
 StringBuffer · 597
 StringBufferInputStream · 597
 StringBuilder, String y toString() · 318
 StringReader · 601, 604
 StringWriter · 601
 Stroustrup, Bjarne · 119
 Stub · 387
 subobjeto · 143
 sucesos:
 JavaBeans · 918
 modelo de Swing · 869
 multidifusión y JavaBeans · 927
 programación dirigida por · 862

respuesta a un suceso Swing · 862
 sistema dirigido por sucesos · 231
 y escuchas · 869
 sugerencias · 880
 suma · 46
 super:
 palabra clave · 143
 y clases internas · 236
 superclase · 143
 límites · 361
 supertipo, comodines de · 438
 suplantación en la POO · 2
suspend() e interbloqueos · 775
 sustitución 9:
 de métodos private · 173
 y clases internas · 236
 y sobrecarga · 151
 sustitución:
 herencia y extensión · 184
 principio de · 9
 pura · 9, 185
 SWF, formato de código intermedio Flash · 932
 Swing · 857
 componentes · 876
 HTML en los componentes · 902
 modelo de sucesos · 869
 y concurrencia · 910
 switch:
 palabra clave · 81
 y enum · 662
 synchronized:
 contenedores · 577
 decidir qué métodos sincronizar · 929
 estático · 757
 Regla de Brian de la sincronización 757
 sección crítica y bloque · 765
wait() y *notifyAll()* · 784
 y herencia · 929
 SynchronousQueue, concurrencia · 826
 System.arraycopy() · 502
 System.err · 282, 613
 System.in · 613
 System.out · 613
 systemNodeForPackage(), API preferences · 657

T

tabla de base de datos, SQL generado mediante anotaciones · 698
 tamaño de un HashMap o HashSet · 571
 tareas y hebras, terminología · 748
 teclado: navegación y Swing · 858 atajo de · 894
 Teoría del compromiso delegado · 751
 terminación:
 alta (big endian) · 624
 baja (little endian) · 624
 condición de, y finalize() · 98

en el tratamiento de excepciones · 280
 ternario, operador · 58
 this, palabra clave · 93
 ThreadFactory personalizada · 741
 throw, palabra clave · 280
 Throwable, clase base para Exception 287
 tiene-un, relación · 6, 153
 TimeUnit · 738, 811
 tipos:
 base · 7
 comprobación de · 309, 392
 derivado · 7
 enumerados · 117
 estructurales · 464, 472
 genéricos y contenedores seguros respecto al tipo · 242
 hallar el tipo exacto de referencia base · 352
 inferencia del argumento de · 403
 latentes · 464, 472
 marcador de, en genéricos · 424
 matrices y comprobación de · 483
 parametrizados · 393
 pato · 464, 472
 primitivos · 25
 seguridad de tipos en Java · 60
 seguridad dinámica de · 456
 tipo de datos equivalente a clase · 3
 tipos de datos primitivos y uso con operadores · 62
toArray() · 571
 TooManyListenersException · 926
toString() · 140
 directrices para usar StringBuilder · 319
transferFrom() · 618
transferTo() · 618
 transient, palabra clave · 646
 TreeMap · 542, 544, 571
 TreeSet · 258, 533, 536, 567
 true · 51
 try, bloque · 150, 280, 297
 en excepciones · 280
tryLock(), bloqueo de archivos · 632
 tupla · 396, 408, 413
 TYPE, campo para literales de clases primitivas · 357

U

UML(Unified Modeling Language) · 4, 6, 960
 unario, operador:
 más (+) · 48
 menos (-) · 48
 UncaughtExceptionHandler, clase Thread · 752
 Unicode · 601
 unidad de compilación · 122
 unidad de traducción · 122
 unidifusión · 926

Unified Modeling Language (UML) · 4, 960
unmodifiableList(), Collections · 530
 UnsupportedOperationException · 530
 upcasting. Véase generalización
userNodeForPackage(), API preferences · 657
 Utilidades de java.util.Collections · 572

V

values() para enumeraciones · 659, 663
 Varga, Ervin · 7, 780
 variable:
 definición · 73
 inicialización de variables de métodos · 102
 listas de argumentos variables · 113
 local · 29
 Vector · 566, 581
 vector de cambio · 232
 Venners, Bill · 98
 versionado, serialización · 649
 Visitante, patrón de diseño · 706
 Visual BASIC, Microsoft · 918
 volatile · 754, 760, 763

W

wait() · 784
 Waldrop, M. Mitchell · 961
 WeakHashMap · 542, 580
 WeakReference · 578
 Web Start, Java · 906
 West, BorderLayout · 866
 while · 72
windowClosing() · 899
write() · 596
 nio · 618
writeBytes() · 607
writeChars() · 607
writeDouble() · 607
writeExternal() · 643
writeObject() · 639
 con Serializable · 646
 Writer · 596, 600, 601

X

XDoclet · 693
 XML · 654
 XOM, biblioteca XML · 654
 XOR (Exclusive-OR) · 55

Z

ZipEntry · 637
 ZipInputStream · 634
 ZipOutputStream · 634