Introducción

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

No podemos ver Java como una simple colección de características -algunas de las características no tienen sentido aisladas. Se puede usar la suma de partes sólo si se está pensando en ***diseño,*** y no simplemente en codificación. Y para entender Java así, hay que entender los problemas del lenguaje y de la programación en general. Este libro habla acerca de problemas de programación, por qué son problemas y el enfoque que Java sigue para solucionarlos. Por consiguiente, algunas características que explico en cada capítulo se basan en cómo yo veo que se ha solucionado algún problema en particular con el lenguaje. Así, espero conducir poco a poco al lector, hasta el punto en que Java se convierta en lengua casi materna.

Durante todo el tiempo, estaré tomando la actitud de que el lector construya un modelo mental que le permita desarrollar un entendimiento profundo del lenguaje; si se encuentra un puzzle se podrá alimentar de éste al modelo para tratar de deducir la respuesta.

Prerrequisitos

Este libro asume que se tiene algo de familiaridad con la programación: se entiende que un programa es una colección de sentencias, la idea de una subrutina/función/macro, sentencias de control como "ir' y bucles estilo "while", etc. Sin embargo, se podría haber aprendido esto en muchos sitios, como, por ejemplo, la programación con un lenguaje de macros o el trabajo con una herramienta como Perl. A medida que se programa hasta el punto en que uno se siente cómodo con las ideas básicas de programación, se podrá ir trabajando a través de este libro. Por supuesto, el libro será más fácil para los programadores de C y aún más para los de C++, pero tampoco hay por qué excluirse a sí mismo cuando se desconocen estos lenguajes (aunque en este caso es necesario tener la voluntad de trabajar duro; además, el CD multimedia que acompaña a este texto te permitirá conocer rápidamente los conceptos de la sintaxis de C necesarios para aprender Java). Presentaré los conceptos de la programación orientada a objetos (POO) y los mecanismos de control básicos de Java, para tener conocimiento de ellos, y los primeros ejercicios implicarán las secuencias de flujo de control básicas.

Aunque a menudo aparecerán referencias a aspectos de los lenguajes C y C++, no deben tomarse como comentarios profundos, sino que tratan de ayudar a los programadores a poner Java en perspectiva con esos lenguajes, de los que, después de todo, es de los que desciende Java. Intentaré hacer que estas referencias sean lo más simples posibles, y explicar cualquier cosa que crea que una persona que no haya programado nunca en C o C++ pueda desconocer.

Aprendiendo Java

Casi a la vez que mi primer libro ***Using*** C++ (Osborne/McGraw-Hill, 1989) apareció, empecé a enseñar ese lenguaje. Enseñar lenguajes de programación se ha convertido en mi profesión; he visto cabezas dudosas, caras en blanco y expresiones de puzzle en audiencias de todo el mundo desde 1989. A medida que empecé con formación ***in situ*** a grupos de gente más pequeños, descubrí algo en los ejercicios. Incluso aquéllos que sonreían tenían pegas con muchos aspectos. Al dirigir la sesión de C++ en la ***Software Development Conference*** durante muchos años (y después la sesión de Java), descubrí que tanto yo como otros oradores tendíamos a ofrecer a la audiencia, en general, muchos temas demasiado rápido. Por tanto, a través, tanto de la variedad del nivel de audiencia como de la forma de presentar el material, siempre se acababa perdiendo parte de la audiencia. Quizás es pedir demasiado, pero dado que soy uno de ésos que se resisten a las conferencias tradicionales (y en la mayoría de casos, creo que esta 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 consiguiente, acabé aprendiendo a base de experimentación e iteración (una técnica que también funciona bien en un diseño de un programa en Java). Eventualmente, desarrollé un curso usando todo lo que había aprendido de mi experiencia en la enseñanza -algo que me gustaría hacer durante bastante tiempo. Descompone el problema de aprendizaje en pasos discretos, fáciles de digerir, y en un seminario en máquina (la situación ideal de aprendizaje) hay ejercicios seguidos cada uno de pequeñas lecciones. Ahora doy cursos en seminarios públicos de Java, que pueden encontrarse en *http://www.BruceEckel.com*. (El seminario introductorio también está disponible como un CDROM. En el sitio web se puede encontrar más información al respecto.)

La respuesta que voy obteniendo de cada seminario me ayuda a cambiar y reenfocar el material hasta que creo que funciona bien como medio docente. Pero este libro no es simplemente un conjunto de notas de los seminarios -intenté empaquetar tanta información como pude en este conjunto de páginas, estructurándola de forma que cada tema te vaya conduciendo al siguiente. Más que otra cosa, el libro está diseñado para servir al lector solitario que se está enfrentando y dando golpes con un nuevo lenguaje de programación.

Objetivos

Como en mi libro anterior *Thinking in C++,* este libro pretende estar estructurado en torno al proceso de enseñanza de un lenguaje. En particular, mi motivación es crear algo que me proporcione una forma de enseñar el lenguaje en mis propios seminarios. Cuando pienso en un capítulo del libro, lo pienso en términos de lo que constituiría una buena lección en un seminario. Mi objetivo es lograr fragmentos que puedan enseñarse en un tiempo razonable, seguidos de ejercicios que sean fáciles de llevar a cabo en clase.

Mis objetivos en este libro son:

1. Presentar el material paso a paso de forma que se pueda digerir fácilmente cada concepto antes de avanzar.
2. Utilizar ejemplos que sean tan simples y cortos como se pueda. Esto evita en ocasiones acometer problemas del "mundo real", pero he descubierto que los principiantes suelen estar más contentos cuando pueden entender todos los detalles de un ejemplo que cuando se ven impresionados por el gran rango del problema que solucionan. Además, hay una limitación severa de cara a la cantidad de código que se puede absorber en una clase. Por ello, no dudaré en recibir críticas por usar "ejemplos de juguete", sino que estoy deseoso de aceptarlas en aras de lograr algo pedagógicamente útil.
3. Secuenciar cuidadosamente la presentación de características de forma que no se esté viendo algo que aún no se ha expuesto. Por supuesto, esto no es siempre posible; en esas situaciones se dan breves descripciones introductorias.
4. Dar lo que yo considero que es importante que se entienda del lenguaje, en lugar de todo lo que sé. Creo que hay una jerarquía de importancia de la información, y que hay hechos que el 95% de los programadores nunca necesitarán saber y que simplemente confunden a la gente y añaden su percepción de la complejidad del lenguaje. Por tomar un ejemplo de C, si se memoriza la tabla de precedencia de los operadores (algo que yo nunca hice) se puede escribir un código más inteligente. Pero si se piensa en ello, también confundirá la legibilidad y mantenibilidad de ese código. Por tanto, hay que olvidarse de la precedencia, y usar paréntesis cuando las cosas no estén claras.
5. Mantener cada sección lo suficientemente enfocada de forma que el tiempo de exposición - el tiempo entre periodos de ejercicios- sea pequeño. Esto no sólo mantiene más activas las mentes de la audiencia, que están en un seminario en máquina, sino que también transmite más sensación de avanzar.
6. Proporcionar una base sólida que permita entender los aspectos lo suficientemente bien como para avanzar a cursos y libros más difíciles.

Documentación en línea

El lenguaje Java y las bibliotecas de Sun Microsystems (de descarga gratuita) vienen con su documentación en forma electrónica, legible utilizando un navegador web, y casi toda implementación de Java de un tercero tiene éste u

otro sistema de documentación equivalente. Casi todos los libros publicados de Java, incorporan esta documentación. Por tanto, o ya se tiene, o se puede descargar, y a menos que sea necesario, este libro no repetirá esa documentación pues es más rápido encontrar las descripciones de las clases en el navegador web que buscarlas en un libro (y la documentación en línea estará probablemente más actualizada). Este libro proporcionará alguna descripción extra de las clases sólo cuando sea necesario para complementar la documentación, de forma que se pueda entender algún ejemplo particular.

Ejercicios

He descubierto que los ejercicios simples son excepcionalmente útiles para completar el entendimiento de los estudiantes durante un seminario, por lo que se encontrará un conjunto de ellos al final de cada capítulo.
La mayoría de ejercicios están diseñados para ser lo suficientemente sencillos como para poder ser resueltos en un tiempo razonable en una situación de clase mientras que observa el profesor, asegurándose de que todos los alumnos asimilen el material. Algunos ejercicios son más avanzados para evitar que los alumnos experimentados se aburran. La mayoría están diseñados para ser resueltos en poco tiempo y probar y pulir el conocimiento. Algunos suponen un reto, pero ninguno presenta excesivas dificultades. (Presumiblemente, cada uno podrá encontrarlos -o más probablemente te encontrarán ellos a ti.)

En el documento electrónico The Thinking in Java Annotated Solution Guide pueden encontrarse soluciones a ejercicios seleccionados, disponibles por una pequeña tasa en *http://www.BruceEckel.com*.

Código fuente

Todo el código fuente de este libro está disponible de modo gratuito sometido a copyright, distribuido como un paquete único, visitando el sitio web *http://www.BruceEckel.com*. Para asegurarse de obtener la versión más actual, éste es el lugar oficial para distribución del código y de la versión electrónica del libro. Se pueden encontrar versiones espejo del código y del libro en otros sitios (algunos de éstos están referenciados en *http://www.BruceEckel.com*), pero habría que comprobar el sitio oficial para asegurarse de obtener la edición más reciente. El código puede distribuirse en clases y en otras situaciones con fines educativos.

La meta principal del copyright es asegurar que el código fuente se cite adecuadamente, y prevenir que el código se vuelva a publicar en medios impresos sin permiso. (Mientras se cite la fuente, utilizando los ejemplos del libro, no habrá problema en la mayoría de los medios.)

En cada fichero de código fuente, se encontrará una referencia a la siguiente nota de copyright:

```
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.BruceEckel.com (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 URL marked "feedback"
in the electronic version of the book, nearest the error
you've found.
```

El código puede usarse en proyectos y en clases (incluyendo materiales de presentación) mientras se mantenga la observación de ***copyright*** que aparece en cada archivo fuente.

Estándares de codificación

En el texto de este libro, los identificadores (nombres de funciones, variables y clases) están en **negrita.** La mayoría de palabras clave también están en negrita, excepto en aquellos casos en que las palabras se usan tanto que ponerlas en negrita podría volverse tedioso, como es el caso de la palabra "clase".

Para los ejemplos de este libro, uso un estilo de codificación bastante particular. Este estilo sigue al estilo que la propia Sun usa en prácticamente todo el código de sitio web (véase *http://java.sun.com/docs/codeconv/index.html),* y parece que esta soportado por la mayoría de entornos de desarrollo Java. Si ha leído el resto de mis trabajos, también verá que el estilo de codificación de Sun coincide con el mío -esto me alegra, aunque no tenía nada que hacer con él**.** El aspecto del estilo de formato es bueno para lograr horas de tenso debate, por lo que simplemente diré que no pretendo dictar un estilo correcto mediante mis ejemplos; tengo mi propia motivación para usar el estilo que uso. Java es un

lenguaje de programación de forma libre, se puede seguir usando cualquier estilo con el que uno esté a gusto.

Los programas de este libro son archivos incluidos por el procesador de textos, directamente sacados de archivos compilados. Por tanto, los archivos de código impresos en este libro deberían funcionar sin errores de compilador. Los errores que *deberían* causar mensajes de error en tiempo de compilación están comentados o marcados mediante **//!**, por lo que pueden ser descubiertos fácilmente, y probados utilizando medios automáticos. Los errores descubiertos de los que ya se haya informado al autor, aparecerán primero en el código fuente distribuido y posteriormente en actualizaciones del libro (que también aparecerán en el sitio web *http://www.BruceEckel.com*).

Versiones de Java

Generalmente confío en la implementación que Sun hace de Java como referencia para definir si un determinado comportamiento es o no correcto.

Con el tiempo, Sun ha lanzado tres versiones principales de Java: la 1.0, la 1.1 y la 2 (que se llama versión 2, incluso aunque las versiones del JDK de Sun siguen usando el esquema de numeración de 1.2, 1.3, 1.4, etc.). La versión 2 parece llevar finalmente a Java a la gloria, especialmente en lo que concierne a las herramientas de interfaces. Este libro se centra en, y está probado con, Java 2, aunque en ocasiones hago concesiones a las características anteriores de Java 2, de forma que el código pueda compilarse bajo Linux (vía el JDK de Linux que estaba disponible en el momento de escribir el libro).

Si se necesita aprender versiones anteriores del lenguaje no cubiertas en esta edición, la primera edición del libro puede descargarse gratuitamente de http://www.BruceEckel.corn, y también está en el CD adjunto a este libro.

Algo de lo que uno se dará cuenta es que, cuando menciono versiones anteriores del lenguaje, no uso los números de sub-revisión. En este libro me referiré sólo a Java 1.0, 1.1 y 2, para protegerme de errores tipográficos producidos por sub-revisiones posteriores de estos productos.

Errores

Sin que importe cuántos trucos utiliza un escritor para detectar errores, siempre hay alguno que se queda ahí y que algún lector encontrará.

Hay un formulario para remitir errores al principio de cada capítulo en la versión HTML del libro (y en el CD ROM unido al final de este libro, además de descargable de http://www.BruceEckel.corn) y también en el propio sitio web, en

la página correspondiente a este libro. Si se descubre algo que uno piense que puede ser un error, por favor, utilice el formulario para remitir el error junto con la corrección sugerida. Si es necesario, incluya el archivo de código fuente original y cualquier modificación que se sugiera. Su ayuda será apreciada.

Capítulos

Este libro se diseñó con una idea en la cabeza: la forma que tiene la gente de aprender Java. La realimentación de la audiencia de mis seminarios me ayudó a ver las partes difíciles que necesitaban aclaraciones. En las áreas en las que me volvía ambiguo e incluía varias características a la vez, descubrí -a través del proceso de presentar el material- que si se incluyen muchas características de golpe, hay que explicarlas todas, y esto suele conducir fácilmente a la confusión por parte del alumno. Como resultado, he tenido bastantes problemas para presentar las características agrupadas de tan pocas en pocas como me ha sido posible.

El objetivo, por tanto, es que cada capítulo enseñe una única característica, o un pequeño grupo de características asociadas, sin pasar a características adicionales. De esa forma se puede digerir cada fragmento en el contexto del conocimiento actual antes de continuar.

He aquí una breve descripción de los capítulos que contiene el libro, que corresponde a las conferencias y periodos de ejercicio en mis seminarios en máquina.

Capítulo 1: Introducción a los objetos

Este capítulo presenta un repaso de lo que es la programación orientada a objetos, incluyendo la respuesta a la cuestión básica "¿Qué es un objeto?", interfaz frente a implementación, abstracción y encapsulación, mensajes y funciones, herencia y composición, y la importancia del polimorfismo.

También se obtendrá un repaso a los aspectos de la creación de objetos como los constructores, en los que residen los objetos, dónde ponerlos una vez creados, y el mágico recolector de basura que limpia los objetos cuando dejan de ser necesarios. Se presentarán otros aspectos, incluyendo el manejo de errores con excepciones, el multihilo para interfaces de usuario con buen grado de respuesta, y las redes e Internet. Se aprenderá qué convierte a Java en especial, por qué ha tenido tanto éxito, y también algo sobre análisis y diseño orientado a objetos.

Capítulo 2: Todo es un objeto

Este capítulo te lleva al punto donde tú puedas crear el primer programa en Java, por lo que debe dar un repaso a lo esencial, incluyendo el concepto de ***referencia*** a un objeto; cómo crear un objeto; una introducción de los tipos primitivos y arrays; el alcance y la forma en que destruye los objetos el recolector de basura; cómo en Java todo es un nuevo tipo de datos (clase) y cómo crear cada uno sus propias clases; funciones, argumentos y valores de retorno; visibilidad de nombres y el uso de componentes de otras bibliotecas; la palabra clave **static;** y los comentarios y documentación embebida.

Capítulo 3: Controlando el flujo de los programas

Este capítulo comienza con todos los operadores que provienen de C y C++. Además, se descubrirán los fallos de los operadores comunes, la conversión de tipos, la promoción y la precedencia. Después se presentan las operaciones básicas de control de flujo y selección existentes en casi todos los lenguajes de programación: la opción con **if**-**else**; los bucles con **while** y **for**; cómo salir de un bucle con **break** y **continue**, además de sus versiones etiquetadas en Java (que vienen a sustituir al "**goto perdido**" en Java); la selección con **switch**. Aunque gran parte de este material tiene puntos comunes con el código de C y C++, hay algunas diferencias. Además, todos los ejemplos estarán hechos completamente en Java por lo que el lector podrá estar más a gusto con la apariencia de Java.

Capítulo 4: Inicialización y limpieza

Este capítulo comienza presentando el constructor, que garantiza una inicialización adecuada. La definición de constructor conduce al concepto de sobrecarga de funciones (puesto que puede haber varios constructores). Éste viene seguido de una discusión del proceso de limpieza, que no siempre es tan simple como parece. Normalmente, simplemente se desecha un objeto cuando se ha acabado con él y el recolector de basura suele aparecer para liberar la memoria. Este apartado explora el recolector de basura y algunas de sus idiosincrasias. El capítulo concluye con un vistazo más cercano a cómo se inicializan las cosas: inicialización automática de miembros, especificación de inicialización de miembros, el orden de inicialización, la inicialización **static** y la inicialización de arrays.

Capítulo 5: Ocultando la implementación

Este capítulo cubre la forma de empaquetar junto el código, y por qué algunas partes de una biblioteca están expuestas a la vez que otras partes están ocultas. Comienza repasando las palabras clave **package** e **import**, que llevan a cabo empaquetado a nivel de archivo y permiten construir bibliotecas de clases. Después examina el tema de las rutas de directorios y nombres de fichero. El resto del capítulo echa un vistazo a las palabras clave **public**, **private** y

protected, el concepto de acceso "**friendly**", y qué significan los distintos niveles de control de acceso cuando se usan en los distintos conceptos.

Capítulo 6: Reutilizando clases

El concepto de herencia es estándar en casi todos los lenguajes de POO. Es una forma de tomar una clase existente y añadirla a su funcionalidad (además de cambiarla, que será tema del Capítulo 7). La herencia es a menudo una forma de reutilizar código dejando igual la "clase base", y simplemente parcheando los elementos aquí y allí hasta obtener lo deseado. Sin embargo, la herencia no es la única forma de construir clases nuevas a partir de las existentes. También se puede empotrar un objeto dentro de una clase nueva con la composición. En este capítulo, se aprenderán estas dos formas de reutilizar código en Java, y cómo aplicarlas.

Capítulo 7: Polimorfismo

Cada uno por su cuenta, podría invertir varios meses para descubrir y entender el polimorfismo, claves en POO. A través de pequeños ejemplos simples, se verá cómo crear una familia de tipos con herencia y manipular objetos de esa familia a través de su clase base común. El polimorfismo de Java permite tratar los objetos de una misma familia de forma genérica, lo que significa que la mayoría del código no tiene por qué depender de un tipo de información específico. Esto hace que los programas sean extensibles, por lo que se facilita y simplifica la construcción de programas y el mantenimiento de código.

Capítulo 8: Interfaces y clases internas

Java proporciona una tercera forma de establecer una relación de reutilización a través de la ***interfaz,*** que es una abstracción pura del interfaz de un objeto. La **interfaz** es más que una simple clase abstracta llevada al extremo, puesto que te permite hacer variaciones de la "herencia múltiple" de C++, creando una clase sobre la que se puede hacer una conversión hacia arriba a más de una clase base.

A primera vista, las clases parecen un simple mecanismo de ocultación de código: se colocan clases dentro de otras clases. Se aprenderá, sin embargo, que la clase interna hace más que eso – conoce y puede comunicarse con la clase contenedora – y que el tipo de código que se puede escribir con clases internas es más elegante y limpio, aunque es un concepto nuevo para la mayoría de la gente y lleva tiempo llegar a estar cómodo utilizando el diseño clases internas.

Capítulo 9: Manejo de errores con excepciones

La filosofía básica de Java es que el código mal formado no se ejecutará. En la medida en que sea posible, el compilador detecta problemas, pero en ocasiones los problemas -debidos a errores del programador o a condiciones de error naturales que ocurren como parte de la ejecución normal del programa- pueden detectarse y ser gestionados sólo en tiempo de ejecución. Java tiene el ***manejo de excepciones*** para tratar todos los problemas que puedan surgir al ejecutar el programa. Este capítulo muestra cómo funcionan en Java las palabras clave **try**, **catch**, **throw**, **throws** y **finally**; cuándo se deberían lanzar excepciones y qué hacer al capturarlas. Además, se verán las excepciones estándar de Java, cómo crear las tuyas propias, qué ocurre con las excepciones en los constructores y cómo se ubican los gestores de excepciones.

Capítulo 10: Identificación de tipos en tiempo de ejecución

La identificación de tipos en tiempo de ejecución (RTTI) te permite averiguar el tipo exacto de un objeto cuando se tiene sólo una referencia al tipo base. Normalmente, se deseará ignorar intencionadamente el tipo exacto de un objeto y dejar que sea el mecanismo de asignación dinámico de Java (polimorfismo) el que implemente el comportamiento correcto para ese tipo. A menudo, esta información te permite llevar a cabo operaciones de casos especiales, más eficientemente. Este capítulo explica para qué existe la RTTI, cómo usarlo, y cómo librarse de él cuando sobra. Además, este capítulo presenta el mecanismo de reflectividad de Java.

Capítulo 11: Guardando tus objetos

Es un programa bastante simple que sólo tiene una cantidad fija de objetos de tiempo de vida conocido. En general, todos los programas irán creando objetos nuevos en distintos momentos, conocidos sólo cuando se está ejecutando el programa. Además, no se sabrá hasta tiempo de ejecución la cantidad o incluso el tipo exacto de objetos que se necesitan. Para solucionar el problema de programación general, es necesario crear cualquier número de objetos, en cualquier momento y en cualquier lugar. Este capítulo explora en profundidad la biblioteca de contenedores que proporciona Java 2 para almacenar objetos mientras se está trabajando con ellos: los simples arrays y contenedores más sofisticados (estructuras de datos) como **ArrayList** y **HashMap**.

Capítulo 12: El sistema de E/S de Java

Teóricamente, se puede dividir cualquier programa en tres partes: entrada,

proceso y salida. Esto implica que la E/S (entrada/salida) es una parte importante de la ecuación. En este capítulo se aprenderá las distintas clases que proporciona Java para leer y escribir ficheros, bloques de memoria y la consola. También se mostrará la distinción entre E/S "antigua" y "nueva". Además, este capítulo examina el proceso de tomar un objeto, pasarlo a una secuencia de bytes (de forma que pueda ser ubicado en el disco o enviado a través de una red) y reconstruirlo, lo que realiza automáticamente la serialización de objetos de Java. Además, se examinan las bibliotecas de compresión de Java, que se usan en el formato de archivos de Java CJAR).

Capítulo 13: Concurrencia

Java proporciona una utilidad preconstruida para el soporte de múltiples subtareas concurrentes denominadas hilos, que se ejecutan en un único programa. (A menos que se disponga de múltiples procesadores en la máquina, los múltiples hilos sólo son aparentes.) Aunque éstas pueden usarse en todas partes, los hilos son más lucidos cuando se intenta crear una interfaz de usuario con alto grado de respuesta, de forma que, por ejemplo, no se evita que un usuario pueda presionar un botón o introducir datos mientras se está llevando a cabo algún procesamiento. Este capítulo echa un vistazo a la sintaxis y la semántica del multihilo en Java.

Capítulo 14: Creación de ventanas y applets

Java viene con la biblioteca IGU Swing, que es un conjunto de clases que manejan las ventanas de forma portable. Estos programas con ventanas pueden o bien ser applets o bien aplicaciones independientes. Este capítulo es una introducción a Swing y a la creación de applets de World Wide Web. Se presenta la importante tecnología de los "JavaBeans", fundamental para la creación de herramientas de construcción de programas de Desarrollo Rápido de Aplicaciones **(RAD)**.

Capítulo 15: Descubriendo Problemas

Los mecanismos de comprobación de lenguaje nos pueden tomar sólo en lo que va de nuestra búsqueda para desarrollar un programa que trabaja correctamente. Este capítulo presenta herramientas para solucionar los problemas que el compilador no soluciona. Una de los pasos más grandes adelante es la incorporación de prueba de unidades automatizada. Para este libro, un sistema personalizado de prueba fue desarrollado para asegurar la exactitud de la salida de programa, pero el sistema de pruebas del **JUnit** del estándar de defacto es también introducido. La construcción automática es implementada con la herramienta del estándar de la fuente abierta **Ant**, y para el trabajo de equipo, los fundamentos de CVS son explicados. Para información de problema durante la

corrida, este capítulo introduce al mecanismo de aserción Java (mostrado aquí usado con Diseño por contrato), la API de registro, depuradores, los perfiladores, y hasta doclets (el cual puede ayudar a descubrir problemas en código fuente).

Capítulo 16: Computación distribuida

Todas las características y bibliotecas de Java aparecen realmente cuando se empieza a escribir programas que funcionen en red. Este capítulo explora la comunicación a través de redes e Internet, y las clases que proporciona Java para facilitar esta labor. Presenta los tan importantes conceptos de Servlets y JSP (para programación en el lado servidor), junto con Java DataBase Connectivity (JDBC) y el Remote Method Invocation (RMI). Finalmente, hay una introducción a las nuevas tecnologías de JINI, JavaSpaces, y Enterprise JavaBeans (EJBS).

Capítulo 17: Patrones de Diseño

Este capítulo introduce los acercamientos de patrones muy importantes y todavía poco tradicionales para diseño de programas. Un ejemplo del proceso de evolución del diseño es estudiado, comenzando con una solución inicial y moviéndose a través de la lógica y el proceso de desarrollar la solución para los diseños más correctos. Verás una forma en la que un diseño puede materializarse con el paso del tiempo.

Capítulo 18: Proyectos

Este capítulo incluye un conjunto de proyectos que se construyen en el material presentado en este libro, o de otra manera se adecuaría en los capítulos anteriores. Estos proyectos son significativamente más complicados que los ejemplos en el resto de libro, y a menudo demuestran usos y técnicas nuevas de librerías de clase.

Hay temas que no parecen ajustarte dentro del núcleo del libro, y todavía me encuentro con que los discuto durante los seminarios.

Capítulo 19: Análisis y Diseño

El paradigma orientado a objetos es una forma de pensar nueva y diferente acerca de programar, y muchas personas tienen problemas al principio sabiendo cómo acercarse un proyecto OOP. Una vez que entiendes el concepto de un objeto, y como aprendes a pensar más en un estilo orientado a objetos, puedes comenzar a crear "buenos" diseños que se aprovechan de todos los beneficios que OOP tiene que ofrecer. Este capítulo introduce las ideas de análisis, diseño, y algunas formas para acercarse los problemas de desarrollar buenos programas orientados a objetos en una cantidad razonable de tiempo. Los temas incluyen

diagramas *Unified Modeling Language* (UML) y metodología asociada, casos de uso, cartas de *Class-Responsibility-Collaboration* (código de redundancia cíclica), desarrollo iterativo, Programación Extrema (XP), formas a desarrollar y desarrollar código reusable, y estrategias para la transición para la programación orientada a objetos.

Apéndice A: Paso y retorno de objetos

Puesto que la única forma de hablar con los objetos en Java es mediante referencias, los conceptos de paso de objetos a una función y de devolución de un objeto de una función tienen algunas consecuencias interesantes. Este apéndice explica lo que es necesario saber para gestionar objetos cuando se está entrando y saliendo de funciones, y también muestra la clase **String,** que usa un enfoque distinto al problema.

Apéndice B: La Interfaz Nativa de Java (JNI)

Un programa Java totalmente portable tiene importantes pegas: la velocidad y la incapacidad para acceder a servicios específicos de la plataforma. Cuando se conoce la plataforma sobre la que está ejecutando, es posible incrementar dramáticamente la velocidad de ciertas operaciones construyéndolas como métodos nativos, que son funciones escritas en otro lenguaje de programación (actualmente, sólo están soportados C/C++). Este apéndice da una introducción más que satisfactoria que debería ser capaz de crear ejemplos simples que sirvan de interfaz con código no Java.

Apéndice C: Guías de programación Java

Este apéndice contiene sugerencias para guiarle durante la realización del diseño de programas de bajo nivel y la escritura de código.

Apéndice D: Comparando C++ y Java

Si eres programador de C++, ya tienes la idea básica de programación orientada a objetos, y la sintaxis de Java sin duda se verá muy familiar para ti. Esto tiene sentido porque Java fue derivada de C++. Sin embargo, hay un número sorprendente de diferencias entre C++ y Java. Estas diferencias están dirigidas a ser mejoras significativas, y si entiendes las diferencias verás el por qué Java es un lenguaje de programación tan beneficioso. Este apéndice te conduce por las características importantes que hacen a Java distinto de C++.

Apéndice E: Rendimiento

Esto te permitirá encontrar cuellos de botella y mejorar la velocidad en tu programa Java.

Apéndice F: Un poco sobre el Recolector de Basura

Este apéndice describe la operación y los acercamientos que se usan para implementar el recolector de basura.

Apéndice G: Suplementos

Las descripciones de material de aprendizaje adicional disponible de *MindView*:

1. El CD-ROM que está en la parte de atrás de este libro, que contiene los Fundamentos para el seminario Java en CD, para prepararte para este libro.
2. El CD-ROM Hands On Java, Edición 3, disponible en www.MindView.net. Un seminario en CD que está basado en el material en este libro.
3. The Thinking In Java Seminar. El MindView, Inc., principal seminario introductorio basado en el material en este libro. El horario y las páginas de inscripción pueden ser encontrados en www.MindView.net.
4. Thinking in Enterprise Java, un libro que cubre temas Java más adelantados apropiados para la programación de la empresa. Disponible en www.MindView.net.
5. The J2EE Seminar. Te inicia en el desarrollo práctico de aplicaciones del mundo real, habilitadas por medio de la Internet, distribuidas con Java. Vea www.MindView.net.
6. Designing Objects && Systems Seminar. El análisis orientado a objetos, el diseño, y las técnicas de implementación. Vea www.MindView.net.
7. Thinking in Patterns (con Java), que cubre temas Java más avanzados en patrones de diseño y técnicas de resoluciones de problemas. Disponible en www.MindView.net.
8. Thinking In Patterns Seminar. Un seminario en vivo basado en el libro de arriba. El horario y las páginas de inscripción pueden ser encontrados en www.MindView.net.
9. Design Consulting And Reviews. La asistencia para ayudar a conservar tu proyecto en buena forma.

Apéndice H: Recursos

Una lista de algunos libros sobre Java que he encontrado particularmente útil.

Prólogo

Sugerí a mi hermano Todd, que hace el salto desde el hardware a la programación, que la siguiente gran revolución estará en la ingeniería genética.

Diseñaremos a los microbios para hacer comida, combustible, y plástico; barrerán la contaminación del medio ambiente y en general nos permitirán dominar con maestría la manipulación del mundo físico por un fragmento de lo que cuesta ahora. Afirmé que eso haría a la revolución de las computadoras verse pequeña en contraste.

Luego me di cuenta de que cometía un error común a los escritores de ciencia ficción: perdiéndome en la tecnología (que es por supuesto fácil de hacer en la ciencia ficción). Un escritor experimentado sabe que la historia no se trata nunca de las cosas; se trata de la gente. La genética tendrá un impacto muy grande en nuestras vidas, pero no estoy tan seguro que dejará empequeñecida a la revolución de las computadoras (la cual posibilita la revolución genética) - o al menos la revolución de la información. La información se trata de hablar los unos con los otros: sí, los coches, los zapatos y las curas especialmente genéticas son importantes, pero al fin y al cabo son simplemente atavíos. Lo que verdaderamente tiene importancia es cómo guardamos relación con el mundo. Y muchas de estas cosas se tratan de comunicación.

Este libro es un caso en concreto. La mayoría de las personas pensó que fui muy atrevido o un poco loco para construir un libro completamente en la Web. ¿"Porqué lo compraría alguien"? preguntaron. Si yo hubiera sido de un carácter más conservador, entonces no lo habría hecho, pero realmente no quise escribir otro libro por computadora de la misma antigua forma. No supe qué ocurriría pero resultó ser la cosa más inteligente que alguna vez he hecho con un libro.

En primer lugar, las personas empezaron a enviar correcciones. Éste ha sido un proceso asombroso, porque las personas han investigado cada rincón y cada grieta y han percibido ambos, los errores especializados y gramaticales, ha sido posible eliminar errores de todos los tipos que conozco que de otra manera habrían pasado sin ser vistos. Las personas han sido simplemente fantásticas por eso, muy a menudo diciendo "Ahora Bien, no trato de decir esto en una forma crítica..." Y luego dándome una colección de errores que estoy seguro nunca habría encontrado. Siento como esto ha sido una clase de proceso grupal y eso realmente ha convertido al libro en algo especial. Por el valor de esta retroalimentación, he creado varias encarnaciones de un sistema llamado "BackTalk" para coleccionar y clasificar en categorías los comentarios.

Pero entonces empecé a oír "bien, estupendo, es bonito, usted ha adelantado una versión electrónica, pero quiero una copia impresa y comprometida de un editor real" Hice un intento muy duro para hacerlo fácil para todo el mundo de imprimir en un formato agradable pero eso no contuvo la demanda por el libro

publicado. La mayoría de la gente no quiere dar lectura al libro entero en pantalla, y transportar alrededor de una gavilla de papeles, no importa cuán agradablemente impresas, no les gustó tampoco. (Es más, pienso que no es tan barato en términos del toner de la impresora láser.) Parece que la revolución de la computadora no expulsará a los editores de negocio, después de todo. Sin embargo, un estudiante sugirió que esto puede volverse un modelo para la publicación de futuro: Los libros serán publicados en la Web primero, y sólo cuando existan suficientes garantías de interés el libro será editado. Actualmente, el grueso del pueblo de todos los libros son fracasos financieros, y quizá este método nuevo podría hacer la industria editorial más provechosa.

Este libro se convirtió en una experiencia aleccionadora para mí en otra forma. Originalmente me acerqué a Java como "simplemente otro lenguaje de programación", lo cual en muchos sentidos lo es. Pero a medida que el tiempo pasó y le estudié más profundamente, comencé a ver que la intención fundamental de este lenguaje fue diferente a otros lenguajes que había visto hasta el momento.

La programación trata de manejar complejidad: La complejidad del problema que usted quiere solucionar, situado en la complejidad de la máquina en la cual es solucionado. Por esta complejidad, la mayor parte de nuestros proyectos de programación fallan. Y todavía, de todos los lenguajes de programación de los cuales soy consciente, ninguno de ellos ha dado lo mejor y se ha decidido a que su meta principal de diseño debería conquistar la complejidad de desarrollar y el mantenimiento de los programas. Por supuesto, muchas decisiones del diseño de lenguajes fueron hechas con la complejidad en mente, pero en algún punto hubo siempre algunos otros asuntos que fueron considerados esenciales para estar añadidos en la mezcla. Inevitablemente, esos otros asuntos son lo que causan a los programadores a eventualmente "golpear la pared" con ese lenguaje. Por ejemplo, C++ tuvo que ser compatible hacia atrás con C, como así también eficiente (para permitir la migración fácil a los programadores de C). Esas son ambas metas muy útiles y dan mucha explicación del éxito de C ++, pero también exponen complejidad adicional que impide a algunos proyectos de ser finalizados. (Ciertamente, usted puede culpar a los programadores y a la gestión, pero si un lenguaje puede ayudar atrapando sus errores, ¿por qué no lo hace?). Como otro ejemplo, el Visual Basic (VB) estaba atado al BASIC, el cual no fue realmente diseñado para ser un lenguaje extensible, de tal manera todas las extensiones apiladas en VB han producido alguna sintaxis verdaderamente horrible e insostenible. Perl es compatible hacia atrás con Awk, Sed, Grep, y otras herramientas Unix que estaba supuesto a reemplazar, y como resultado es acusado de producir "código que solo se escribe" (esto es, después de unos pocos meses usted no lo puede leer). Por otra parte, C++, VB, Perl, y otros lenguajes como Smalltalk enfocaron una parte de sus esfuerzos de diseño en el tema de la complejidad y como consecuencia tienen un éxito notable en solucionar ciertos tipos de problemas.

Lo que, me impresionó más a medida que he llegado a entender Java es eso, en alguna parte en la mezcla de objetivos de diseño de Sun, parece que estaba la meta de reducir la complejidad para el programador. Como quien dice "nos

preocupamos por reducir el tiempo y la dificultad de producir código robusto" En los primeros días, esta meta resultó en código que no anduvo muy rápido pero efectivamente ha producido reducciones asombrosas en el tiempo de desarrollo (aunque ha habido muchas promesas hechas sobre qué tan rápidamente Java algún día ejecutará). La mitad o menos del tiempo que toma crear un programa equivalente en C++. Este resultado a solas puede ahorrar cantidades increíbles de tiempo y de dinero, pero Java no se detiene allí. Procede a envolver muchas de las tareas complicadas que han cobrado importancia, como el multihilado (multithreading) y la programación en red, en características del lenguaje o en bibliotecas que pueden a veces hacer esas tareas fáciles. Y finalmente, afronta algunos problemas de complejidad realmente grandes: Los programas interplataformas, los cambios dinámicos de código, y aun la seguridad, cada uno de los cuales puede ajustar su espectro de complejidad desde "el impedimento" al “show-stopper” (?). Así a pesar de los problemas de desempeño que hemos visto, la promesa de Java es tremenda: Nos puede hacer programadores significativamente más productivos.

Uno de los lugares en que veo el impacto más grande para esto es en la Web. El programar redes siempre ha sido duro, y Java lo hace fácil (y los diseñadores del lenguaje Java están trabajando en hacerlo aún más fácil). La programación de redes es cómo hablamos con los demás más eficazmente y más barato de lo que alguna vez hicimos con los teléfonos (el correo electrónico a solas ha revolucionado muchos negocios). Como hablamos mas entre nosotros, cosas asombrosas comienzan a ocurrir, posiblemente más asombrosas aun que la promesa de la ingeniería genética.

En todas las formas -creando programas, trabajando en equipos para crear programas, construyendo interfaces de usuario con las que programas puedan comunicarse con el usuario, ejecutando programas en tipos diferentes de máquinas, y escribiendo fácilmente programas que comunican a través de Internet- Java incrementa el ancho de banda de comunicación entre las personas. Pienso que los resultados de la revolución de la comunicación no pueden ser vistos a partir de los efectos de mover cantidades grandes de bits por allí; veremos la verdadera revolución porque todos nosotros podremos hablar con los demás más fácilmente: Unos con otros, pero también en grupos y, como un planeta. Me han sugerido que la siguiente revolución será la formación de una especie de mente global que resulte de una cantidad adecuada de personas y una cantidad adecuada de interconexiones. Java puede o no ser la herramienta que fomente esa revolución, pero al menos la posibilidad me ha hecho sentir que estoy haciendo algo significativo intentando enseñar el lenguaje.

Prefacio de la 3ra edición

Mucha de la motivación y el esfuerzo de esta edición es poner al día el libro con la liberación del lenguaje Java JDK 1.4. Sin embargo, también ha quedado claro que la mayoría de lectores usan el libro para obtener una comprensión sólida de los fundamentos de manera que puedan seguir adelante hacia temas

más complicados. Debido a que el lenguaje continúa creciendo, se volvió necesario (en parte a fin de que el libro no sobre estirara sus ataduras) reevaluar el significado de los "fundamentos". Esto significó, por ejemplo, reescribir completamente el capítulo de Concurrencia (Llamado antiguamente "Multithreading" (Multihilado)) a fin de dar los fundamentos básicos en las ideas centrales sobre hilos (threading). Sin ese núcleo, es difícil entender los asuntos más complejos sobre hilos.

También he tenido en cuenta la importancia del testeo de código. Sin un esqueleto (framework) de testeo integrado con tests que sean ejecutados cada vez que uno hace una construcción de su sistema, no hay forma de saber si el código es confiable o no. Para lograr esto en el libro, un esqueleto especial de testeo de unidades fue creado para mostrar y validar la salida de cada programa. Esto fue puesto en el Capítulo 15, un nuevo capítulo, junto con explicaciones de "ant", (el sistema estándar para la constitución de sistemas en Java de facto, parecido a "make") JUnit, (la unidad del esqueleto de testeo estándar de facto) y cobertura sobre el registro de actividades y afirmaciones, (nuevo en JDK 1.4) junto con una introducción para depurar y el trazado de perfiles. Para abarcar todos estos conceptos, el nuevo capítulo es denominado "Descubriendo Problemas," e introduce lo que yo ahora creo son habilidades fundamentales que todos los programadores de Java deberían tener en su caja de herramientas básica.

¿Además, he repasado cada ejemplo del libro y me he preguntado, "por qué lo hice así"? En la mayoría de los casos he hecho algunas modificaciones y mejoras, ambos para hacer los ejemplos más coherentes dentro de ellos mismos y también para demostrar lo que considero son las mejores prácticas en la codificación Java (al menos, dentro de las limitaciones de un texto introductorio). Los ejemplos que ya no tuvieron sentido para mí fueron removidos, y ejemplos nuevos se han agregado. Un número de los ejemplos existentes ha tenido un muy significativo rediseño y reimplementación.

Los 16 capítulos en este libro producen lo que pienso es una introducción fundamental para el lenguaje Java. El libro factiblemente puede ser utilizado como un curso introductorio. ¿Pero qué acerca del más material avanzado?

El plan original para el libro fue agregar un capítulo nuevo cubriendo los fundamentos de "Java 2 Edición Enterprise" (J2EE). Muchos de estos capítulos serían creados por amigos y colegas que trabajan conmigo en seminarios y otros proyectos, como Andrea Provaglio, Bill Venners, Chuck Allison, Dave Bartlett, y Jeremy Meyer. Cuando miré el progreso de estos nuevos capítulos, y la fecha límite del libro comencé a ponerme un poco nervioso. Luego observé que el tamaño de los primeros 16 capítulos era efectivamente igual que el tamaño de la segunda edición del libro. Y las personas algunas veces se quejan que ese es ya demasiado grande.

Los lectores han hecho muchos, muchos comentarios maravillosos acerca de las primeras dos ediciones de este libro, lo cual naturalmente ha sido muy agradable para mí. Sin embargo, de vez en cuando, alguien tiene quejas, y por

alguna razón una queja que surge periódicamente es "el libro es demasiado grande". En mi mente está esa débil condena con más razón si "demasiadas páginas" es la única queja. (Uno recuerda las quejas del Emperador de Austria sobre el trabajo de Mozart: ¡"demasiadas notas"! No de que estoy de cualquier manera tratando de compararme a Mozart.) Además, sólo puedo suponer que tal queja viene de alguien quién está aún tomando conocimiento de la inmensidad del mismo lenguaje Java y no ha visto el resto de los libros del tema. Debido a esto, una de las cosas que he tratado de hacer en esta edición es recortar las porciones que han quedado obsoletas, o al menos no son esenciales. En general, he tratado de pasar por encima de todo, quitar de la tercera edición lo que ya no es necesario, incluyendo cambios, y mejorando todas las cosas que podría. Me siento cómodo quitando porciones porque el material original permanece en el sitio Web (*www.BruceEckel.com*) y en el CD-ROM que acompaña este libro, en forma de primera y segunda ediciones libremente descargables del libro. Si usted quiere las cosas viejas, entonces están todavía disponibles, y éste es un alivio maravilloso para un autor. Por ejemplo, el capítulo "Diseño de Patrones" se tornó demasiado grande y ha sido movido a un propio libro: Thinking in Patterns (con Java) (también descargable en el sitio Web).

Ya había decidido que cuando la siguiente versión de Java (JDK 1.5) fuera publicada por Sun, la cual probablemente incluirá un importante tema nuevo llamado "generics" (inspirado en las plantillas de la C++), tendría que dividir el libro en dos para añadir ese nuevo capítulo. Una vocecita me dijo "¿por qué esperar?". Entonces, me decidí a hacerlo durante esta edición, y repentinamente todo tuvo sentido.

El nuevo libro no es un segundo volumen, son preferiblemente temas más avanzados. Será llamado "Thinking in Enterprise Java", y está actualmente disponible (en alguna forma) como una descarga gratuita desde www.BruceEckel.com. Debido a que es un libro separado, puede aumentar en tamaño para adecuarse a los temas necesarios. La meta, como "Pensando en Java", es producir una cobertura muy comprensible de los fundamentos de las tecnologías J2EE a fin que el lector se prepare para una cobertura más avanzada de estos temas. Se pueden encontrar más detalles en el Apéndice C.

Para aquéllos de ustedes que todavía no pueden aguantar el tamaño del libro, me disculpo. Aunque parezca mentira, he trabajado duramente para mantenerlo pequeño. A pesar del volumen, tengo la impresión de que pueden haber en el bastantes opciones para complacerlos. En primer lugar, el libro está disponible electrónicamente, así es que si usted llevara su laptop, puede poner el libro en el sin agregar peso adicional a su viaje. Si usted quiere algo realmente más liviano, hay actualmente versiones del libro para Palm Pilot dando vueltas por ahí. (Una persona me contó que leería el libro en su Palm en la cama a medialuz para evitar molestar a su esposa. Sólo espero que lo ayude a llegar al país de los sueños.) Si usted lo necesita en papel, se de personas que imprimen un capítulo a la vez y lo llevan en su portafolio para leer en el tren.

Java 2, JDK 1.4

Las ediciones del Java JDK son numeradas 1.0, 1.1, 1.2, 1.3, y para este libro, 1.4. A pesar de que estos números de versión están todavía en los "unos", la forma estándar para referirse a cualquier versión del lenguaje que sea JDK 1.2 o más grande es llamarle "Java 2". Esto indica los cambios muy significativos entre "el viejo Java" -que tuvo muchos problemas de los que me quejé en la primera edición de este libro- y esta versión más moderna y mejorada del lenguaje, la cual tiene mucha menos cantidad de problemas y muchas adiciones y lindos diseños.

Este libro está escrito para Java 2, en particular JDK 1.4 (mucho del código no compilará con versiones anteriores, y el sistema construido se quejará y se detendrá si usted hace el intento). Tengo el gran lujo de deshacerme de todas las cosas viejas y escribir solo para el lenguaje nuevo y mejorado, porque la información vieja todavía existe en las anteriores ediciones, en la Web, y en el CD-ROM. También, porque cualquiera libremente puede bajar el JDK de java.sun.com, eso significa que escribiendo para JDK 1.4, no impongo una dificultad financiera a todos obligándolos a hacer la actualización.

Las versiones previas de Java fueron lentas en llegar a Linux, pero esto parece haberse arreglado, las nuevas versiones son lanzadas al mercado para Linux al mismo tiempo que para otras plataformas -ahora hasta la Macintosh comienza a mantenerse al día con las más recientes versiones de Java (vea a www.Linux.org). Linux es un desarrollo muy importante en conjunción con Java, porque se está volviendo rápidamente la plataforma de servidores más importante allí fuera -rápido, confiable, robusto, seguro, bien mantenido, y gratis, es una verdadera revolución en la historia de computación (pienso que nunca hemos visto todas esas características antes en alguna herramienta). Y Java ha encontrado un nicho muy importante en la programación del lado del servidor (server-side) en la forma de "Servlets" y "JavaServer Pages" (JSPs), tecnologías que son mejoras enormes sobre la programación tradicional en "Common Gateway Interface" (CGI) (éstos y los temas relatados están cubiertos en "Thinking in Enterprise Java).

1: Introducción a los Objetos

"Cortamos la naturaleza dividiéndola, la organizamos en conceptos, y le atribuimos significados asimismo, en gran parte porque somos partes de un acuerdo que asentimos a través del discurso de nuestra comunidad y codificamos los patrones de nuestra lengua... no podemos hablar a todos excepto suscribiendo a la organización y clasificación de los datos que acordamos por decreto" Benjamin Lee Whorf (1897-1941)

La génesis de la revolución de la computadora se encuentra en la máquina. La génesis de nuestros lenguajes de programación tiende así a parecerse a esa máquina.

Pero las computadoras no son tanto máquinas como son herramientas que amplifican la mente ("bicicletas para la mente", como Steve Jobs está encariñado en decir) y una clase diferente de medio de expresión. Consecuentemente, las herramientas están comenzando a parecerse menos a las máquinas y más a las partes de nuestras mentes, y también como otras formas de expresión tales como escritura, pintura, escultura, animación, y filmación. La programación orientada al objeto (OOP) es parte de este movimiento para utilizar la computadora como medio de expresión.

Este capítulo lo introducirá a los conceptos básicos de OOP, incluyendo una descripción de los métodos de desarrollo. Este capítulo, y este libro, asumen que usted ha tenido experiencia en un lenguaje de programación procedural, aunque no necesariamente C. Si usted piensa que tiene necesidad de mayor preparación en la programación y la sintaxis de C antes de abordar este libro, debería trabajar a través del CDROM *Foundations for Java* que lo entrena, en el límite posterior de este libro.

Este capítulo es material de fondo y suplementario. Mucha gente no se siente cómoda vadeando la programación orientada al objeto sin entender el cuadro grande primero. Así, hay muchos conceptos que se introducen aquí para darle una descripción sólida de OOP. Sin embargo, la gente puede no comprender los conceptos del cuadro grande hasta el heber visto algo del mecanismo primero; esta gente puede empantanarse y perderse sin un cierto código para conseguir meter sus manos. Si usted forma parte de este último grupo y es impaciente en conseguir lo específico del lenguaje, siéntase libre de saltear más allá este capítulo - saltearlo en este punto no evitará que programe o aprenda la escritura del lenguaje. Sin embargo, usted deseará volver aquí para completar eventualmente su conocimiento así puede completar su conocimiento del porqué los objetos son importantes y cómo diseñar con ellos.

El Progreso de la Abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede ser discutido que la complejidad de los problemas capaz de solucionar está relacionada directamente con la clase y la calidad de la abstracción. Por la clase quiero decir "¿Qué es lo que está abstrayendo?". El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos lenguajes llamados "imperativos" que siguieron (por ejemplo FORTRAN, BASIC, y C) eran abstracciones del lenguaje ensamblador. Estos lenguajes son grandes mejoras del lenguaje ensamblador, pero su abstracción primaria todavía requiere pensar en términos de la estructura de la computadora antes que en la estructura del problema que está intentando resolver. El programador debe establecer la asociación entre el modelo de la máquina (en "el espacio de solución", donde está realizando el modelo de ese problema, tal como una computadora) y el modelo del problema que se está solucionando realmente (en "el espacio del problema", que es el lugar en donde existe el problema). El esfuerzo requerido para realizar este traspaso, y el hecho de que es extrínseco al lenguaje de programación, produce los programas que son difíciles de escribir y costosos de mantener, y como un efecto secundario crean la industria completa de los "métodos programación" .

La alternativa a modelar la máquina es modelar el problema que intenta resolver. Los primeros lenguajes tales como LISP y APL eligieron visiones particulares del mundo ("Los problemas todos son en última instancia listas" o "Los problemas todos son algorítmicos", respectivamente). PROLOG convierte todos los problemas en cadenas de decisiones. Los lenguajes han sido creados para la programación basada en limitaciones y para programar exclusivamente por la manipulación de símbolos gráficos. (El último provisto puede ser demasiado restrictivo). Cada uno de estos acercamientos son una buena solución a la clase particular aquellos problemas para los cuales se diseña la solución, pero cuando usted pasa fuera de este dominio, ellas se hacen torpes.

El acercamiento orientado al objeto va un paso más allá, proporcionando las herramientas para que el programador represente elementos en el espacio del problema. Esta representación es bastante general ya que el programador no está obligado a ningún tipo particular de problema. Nos referimos a los elementos en el espacio del problema y sus representaciones en el espacio de solución como "objetos". (Usted también necesitará otros objetos que no tengan análogos en el espacio del problema). La idea es que el programa esté habilitado para adaptarse a las continuas actualizaciones del problema agregando nuevos tipos de objetos, así cuando lea el código que describe la solución, las palabras de la lectura también expresarán el problema. Ésta es una abstracción más flexible y de mayor alcance del lenguaje las que teníamos antes. **[1]** .Así, la OOP permite que usted describa el problema en los términos del problema, antes que en los términos de la computadora en donde la solución funcionará. Todavía subyace una conexión a la computadora: cada objeto parece un poco a una computadora, que tiene un estado, operaciones que puede pedirle que se realicen. Sin embargo, esto no se parece a una mala analogía de los objetos en el mundo real - donde todo tiene características y comportamientos.

[1] Algunos diseñadores han decidido que la programación orientada al objeto por sí misma no es adecuada para resolver fácilmente todos los problemas de programación, y anuncian la combinación de aproximaciones diferentes dentro de los lenguajes de programación *multiparadigma*. Ver *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

Alan Kay resumió cinco características básicas de Smalltalk, el primer lenguaje orientado al objeto certero y una de los lenguajes sobre los cuales se basa Java. Estas características representan un acercamiento puro a la programación orientada al objeto:

1. **Todo es un objeto.** Piense en un objeto como variable de fantasía; almacena datos, pero usted puede hacerle peticiones a ese objeto, pidiéndole realizar operaciones en sí mismo. En teoría, usted puede tomar cualquier componente conceptual en el problema que intenta resolver (los perros, los edificios, los servicios, etc.) y representarlo como objeto en su programa.
2. **Un programa es un manojo de objetos que se dicen entre ellos qué hacer enviándose mensajes.** . Para hacer una petición de un objeto, usted "envía un mensaje" a ese objeto. Más concretamente, usted puede pensar en un mensaje como una petición de llamar a un método que pertenezca a un objeto particular.
3. **Cada objeto tiene su propia memoria compuesta de otros objetos.** . Puesto de otra manera, usted crea una nueva clase de objeto haciendo un paquete que contiene objetos existentes. Así, usted puede construir complejidad en un programa mientras que la oculta detrás de la simplicidad de los objetos.
4. **Cada objeto tiene un tipo.** Utilizando el discurso, cada objeto es una *instancia* de un *class (clase)*, en la cual "clase" es sinónimo de "tipo". La distinción más importante y característica de una clase es "Qué mensajes puede usted enviarle".
5. **Todos los objetos de un tipo particular pueden recibir los mismos mensajes.** Esto es realmente una declaración cargada, como verá más adelante. Porque un objeto del tipo "círculo" es también un objeto de tipo "figura", un círculo está garantizado para aceptar mensajes de tipo figura. Esto significa que usted puede escribir el código que negocia a las figuras y automáticamente manejar cualquier cosa que quepa en la descripción de una figura. Esta sustitución es uno de los conceptos de mayor alcance en OOP.

Booch ofrece una descripción aún más sucinta de un objeto:

Un objeto tiene estado, comportamiento e identidad.

Esto significa que un objeto puede tener datos internos (que le da el estado), métodos (el comportamiento producido), y cada objeto puede ser únicamente distinguido de cualquier otro objeto - para poner esto en un sentido concreto, cada objeto tiene una dirección única en la asignación de la memoria. **[2]**

[2] Este es actualmente un poco restrictivo, ya que los objetos pueden existir concebiblemente en diferentes máquinas y espacios de direcciones, y pueden también ser almacenados en disco. En estos casos, la identidad del objeto debe ser determinado por algún otro que la dirección de memoria.

Un objeto tiene una interfaz

Aristotle es probablemente el primer que comenzó un estudio cuidadoso del concepto *tipo*; él habló de "la clase del los pescados y de la clase de pájaros". La idea que todos los objetos, mientras que sean únicos, son también parte de una clase de los objetos que tienen características y comportamientos en común fue utilizada directamente en el primer lenguaje orientada al objeto, Simula-67, con su palabra clave fundamental **class** que introduce un nuevo tipo en un programa.

Simula, como su nombre lo indica, fue creado para desarrollar las simulaciones tales como el problema clásico de "la caja del banco". En este, usted tiene un manojo de cajeros, clientes, cuentas, transacciones, y unidades monetarias - un conjunto de "objetos". Los objetos que son idénticos a excepción de su estado durante una ejecución de los programas se agrupan juntos en "clases de objetos" y es de donde la palabra clave **class** (clase) viene. Crear los tipos de datos abstractos (clases) es un concepto fundamental en la programación orientada al objeto. Los tipos de datos abstractos trabajan casi exactamente como tipos incorporados: usted puede crear variables de un tipo (llamado los objetos o las instancias en el discurso orientado al objeto) y manipular esas variables (denominado *"envío de mensajes"* o *"peticiones"*; usted envía un mensaje y el objeto calcula qué hacer con él). Los miembros (elementos) de cada parte de la clase tienen cierta concordancia: cada cuenta tiene un saldo, cada caja puede aceptar un depósito, etc. Al mismo tiempo, cada miembro tiene su propio estado: cada cuenta tiene un saldo diferente, cada caja tiene un nombre. Así, las cajas, clientes, cuentas, transacciones, etc., conservan cada uno su representante con una entidad única en el programa de computadora. Esta entidad es el objeto, y cada objeto pertenece a una clase particular que defina sus características y comportamientos.

Así pues, aunque lo que realmente hacemos en la programación orientada al objeto es crear nuevos tipos de datos, virtualmente todos los lenguajes de programación orientados al objeto utilizan la palabra clave "**class**". Cuando usted ve la palabra "tipo" piense en "clase" viceversa. **[3]**

[3] Algunas personas hacen una distinción, aclarando que tipo determina la interfaz mientras clase es una implementación particular de esa interfaz.

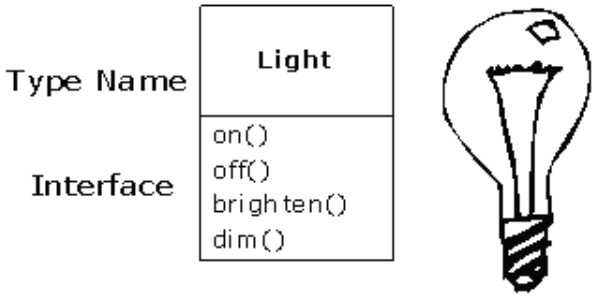
Puesto que una clase describe un sistema de objetos que tienen características idénticas (elementos de datos) y comportamientos (funcionalidad), una clase es realmente un tipo de datos porque un número de coma flotante, por ejemplo, también tiene un sistema de características y de comportamientos. La diferencia es que un programador define una clase para ajustarlo a un problema antes que ser forzado a utilizar un tipo de datos existente que fue

diseñado para representar una unidad del almacenaje en una máquina. Usted amplía el lenguaje de programación agregando nuevos tipos de datos específicos a sus necesidades. El sistema de programación da la bienvenida a las nuevas clases y les brinda todo el cuidado y la tipificación - comprobación de los tipos incorporados.

El acercamiento orientado al objeto no se limita a la construcción de simulaciones. Si o no usted convenga que cualquier programa es una simulación del sistema que diseña, el uso de las técnicas de OOP puede reducir fácilmente un sistema grande de problemas a una solución simple.

Una vez que una clase es establecida, usted poder hacer tantos objetos de esa clase como desee, y luego manipular estos objetos como si esos elemento existieran en el problema que esta intentando resolver. De hecho, uno de los desafíos de la programación orientada al objeto es crear uno a uno la relación entre elemento en el espacio del problema y objeto en el espacio de solución.

¿Pero cómo usted consigue que un objeto realice trabajo útil para usted? Debe haber una manera de hacer una petición al objeto de modo que haga algo, tal como completar una transacción, dibujar algo en la pantalla, o encender un interruptor. Y cada objeto puede satisfacer solamente ciertas peticiones. Las peticiones que usted puede hacer a un objeto son definidas por su *interfaz*, y el tipo es lo que determina la interfaz. Un ejemplo simple puede ser una representación de una bombilla:




```
Luz lt = new Luz();
lt.on();
```

La interfaz establece qué peticiones usted puede hacerle a un objeto particular. Sin embargo, debe haber código en alguna parte para satisfacer esta petición. Esto, junto con los datos ocultados, abarca la *implementación*. Desde un punto de vista de la programación procedural, no es tan complicado. Un tipo tiene un método asociado a cada petición posible, y cuando usted le hace una petición particular a un objeto, ese método es llamado. Este proceso es resumido generalmente diciendo usted está "enviando un mensaje" (haciendo una petición) a un objeto, y el objeto calcula qué hacer con ese mensaje (ejecuta código).

Aquí, el nombre del tipo/clase es **Luz**, el nombre de este objeto particular de **Luz** es **lt**, y las peticiones que usted puede hacer de un objeto **Luz** son encender, apagar, brillar, o desvanecer. Usted crea un objeto **Luz** definiendo una "referencia" (**lt**) para ese objeto y llama a **new** para solicitar un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, usted indica el nombre del objeto y lo conecta con la petición del mensaje con un punto. Desde el punto de vista del usuario de una clase predefinida, esto es mucho que todo lo que está en la programación con los objetos.

El diagrama precedente sigue el formato del *Unified Modeling Language* (UML - lenguaje unificado de modelado). Cada clase está representada por una caja, con el nombre del tipo en la parte superior de la caja, de cualquier dato de los miembros usted cuidadosamente describe en la porción central de la caja, y de los *métodos* (las funciones que pertenecen a este objeto, que reciben mensajes cualesquiera que envía a ese objeto) en la parte inferior de la caja. A menudo, solamente el nombre de la clase y los métodos públicos se muestran en diagramas de diseño de UML, así que la porción central no está. Si está interesado solamente en el nombre de la clase, entonces la parte inferior no es necesario sea mostrada.

Un objeto provee servicios

Mientras usted intenta desarrollar o entender un diseño del programa, una de las mejores maneras de pensar sobre los objetos será como "abastecedores de servicios". Su programa mismo proporcionará servicios al usuario, y logrará esto usando los servicios ofrecidos por otros objetos. Su meta es producir (o aún mejor, localizar en bibliotecas existentes de código) un sistema de los objetos que proporcionan servicios ideales para solucionar su problema.

La manera de comenzar a hacer esto es pedir "Si pudiera sacar mágicamente de un sombrero, qué objetos solucionaría mi problema inmediatamente". Por ejemplo, supongamos que está creando un programa de contabilidad. Usted puede ser que imagine algunos objetos que contienen las pantallas de entrada predefinidas de contabilidad, otro sistema de objetos que realizan cálculos de contabilidad, y un objeto que maneje la impresión de cheques y de facturas en todas las diferentes clases de impresoras. ¿Algunos de *estos* objetos existen ya?, ¿y para los que no, cómo debieran ser? ¿Qué servicios proporcionarán *estos* objetos? y ¿qué objetos necesitarán para satisfacer sus obligaciones? Si usted se mantiene haciendo esto, alcanzará eventualmente un punto donde puede decir "que cualquier objeto que parece bastante simple para sentarse y escribir" o "estoy seguro que el objeto ya existe". Esta es una manera razonable de descomponer un problema en un sistema de objetos.

El pensar en un objeto como proveedor de servicio tiene una ventaja adicional: ayuda a mejorar la cohesión del objeto. La alta cohesión es una calidad fundamental del diseño de software: Significa que varios aspectos del componente de software (tal como un objeto, a aunque esto podría también aplicarse a un método o una biblioteca de objetos) está "encajando en conjunto" muy bien. Un problema que la persona tiene cuando diseña objetos

es que está abarrotando demasiada funcionalidad en un objeto. Por ejemplo, sobre su módulo que imprime el cheque, puede decidir la necesidad de un objeto que sepa todo sobre formato e impresión. Descubre probablemente que esto es demasiado para un objeto, y que necesita tres o más objetos. Un objeto puedo ser un catálogo de todas las disposiciones posibles del cheque, al que se puede preguntar sobre la información de cómo imprimir un cheque. +Un objeto o sistema de objetos podría ser una interfaz de impresión genérico que sabe todos sobre diversas clases de impresoras (absolutamente nada sobre contabilidad - este es un candidato a comprar antes que la escritura misma). Y un tercer objeto podría utilizar los servicios de los otros dos para lograr la tarea. Así, cada objeto tiene un sistema de cohesión de servicios que ofrece. En un buen diseño orientado al objeto, cada objeto hace una cosa bien, pero no intente hacer demasiado. Según lo visto aquí, esto permite no solamente el descubrimiento de los objetos que se pudieron comprar (el objeto del interfaz de la impresora), sino también la posibilidad de producir un objeto que se puede reutilizar en otra parte (el catálogo de las disposiciones del cheque).

Tratar los objetos como proveedores de servicio son una gran herramienta de simplificación, y son muy útiles no solamente durante el proceso del diseño, sino también cuando alguien intenta entender su código o reutilizar un objeto - si pueden ver el valor del objeto basado en qué servicio proporciona, ello hace mucho más fácil hacerlo caber en el diseño.

Esconder la implementación

Es útil dividir el campo jugado por *los creadores de clases* (éstos que crean nuevos tipos de datos) y *los programadores de clientes* **[4]** (los consumidores de clase que utilizan los tipos de datos en sus aplicaciones). La meta del programador de cliente es recoger una caja de herramientas completa de clases para utilizar en el desarrollo rápido de aplicaciones. La meta del creador de clase es construir una clase que exponga solamente lo que es necesario al programador de cliente y mantener el resto ocultado. ¿Por qué? Porque si son ocultados, no puede acceder el programador de cliente a él, lo que significa que el creador de la clase puede cambiar la porción ocultada a voluntad sin la preocupación del impacto en otra persona. La porción ocultada representa generalmente los interiores blandos de un objeto que se podrían corromper fácilmente un programador de cliente descuidado o mal informado, así que ocultar la implementación reduce los errores de programación.

[4] estoy agradecido a mi amigo Scott Meyers por este término.

El concepto de ocultar la implementación no puede ser sobre acentuada. En cualquier relación es importante tener límites que son respetados por todos las partes implicadas. Cuando usted crea una biblioteca, establece una relación con el programador del cliente, quien es también programador, aunque uno que está juntando y usando su biblioteca en una aplicación, construyendo posiblemente una biblioteca más grande. Si todos los miembros de una clase están disponibles, entonces el programador de cliente puede hacer cualquier

cosa con esa clase y no hay ninguna manera de hacer cumplir las reglas. Aunque usted puede ser que realmente prefiera que el programador de cliente no manipule directamente algún miembro de su clase, sin control de acceso no hay ninguna manera de prevenirla. Todo estará descubierto al mundo.

La primera razón del control de acceso es guardar de las manos de programadores de cliente las porciones que ellos no debieran tocar - que son necesarios para la operación interna del tipo de datos pero no parte de la interfaz que los usuarios necesitan para solucionar sus problemas particulares. Esto es realmente un servicio a los usuarios porque pueden ver fácilmente qué es importantes para ellos y qué pueden ignorar.

La segunda razón del control de acceso es permitir que el diseñador de la biblioteca cambie los funcionamientos internos de la clase sin la preocupación de cómo afectará al programador de cliente. Por ejemplo, usted puede ser que ponga una clase singular en ejecución en una manera simple para facilitar el desarrollo, y después descubre necesita reescribirlo para hacer que funcione más rápidamente. Si la interfaz y la implementación se separan y se protegen claramente, usted puede lograr esto fácilmente.

Java utiliza tres palabras claves para fijar los límites en una clase: **public**, **private**, y **protected**. Su uso y significado son absolutamente directos. Estos especificadores del acceso determinan quién puede utilizar las definiciones que siguen. **public** quiere significar que el elemento siguiente está disponible para todos. La palabra clave **private**, por otra parte, significa que nadie puede tener acceso a ese elemento excepto usted, el creador del tipo, de los métodos de este tipo. El tipo **private** es una pared del ladrillo entre usted y el programador de cliente. Alguien que intenta tener acceso a un miembro **private** conseguirá un error en tiempo de compilación. La palabra clave **protected** actúa como **private**, con excepción del hecho que una clase heredera tiene el acceso a los miembros **protected**, pero no a miembros **private**. La herencia será introducida pronto.

Java también tiene un acceso por omisión, que viene en juego si usted omite usar los especificadores ya mencionados. Esto generalmente se llama *package access*, acceso de paquete, porque las clases pueden tener acceso a los miembros de otras clases en el mismo paquete, pero fuera del paquete esos mismos miembros aparecen como **private**.

Reutilizar la implementación

Una vez que se haya creado y probado una clase, debiera (idealmente) representar una unidad útil del código. Resulta que esta reutilización no es tan fácil de alcanzar como se esperaría; toma experiencia y perspicacia producir un diseño reutilizable del objeto. Pero una vez que usted tenga tal diseño, puede ser reutilizado. La reutilización de código es una de las ventajas más grandes que los lenguajes de programación orientados al objeto proporcionan.

La manera simple de reutilizar una clase es utilizar un objeto de esta clase directamente, pero también puede ubicar un objeto de esa clase dentro de una clase nueva. Llamamos a esto "creación de un objeto miembro". Su nueva clase puede ser hecha sobre cualquier número y tipo de otros objetos, en cualquier combinación que necesite para agregar la funcionalidad deseada en su nueva clase, este concepto es llamado *composición* (si la composición sucede dinámicamente, es también llamada *agregación*). La composición está siempre referida como una relación "tiene un", como en "un auto tiene un motor".

(Este diagrama de UML indica la composición con el rombo lleno, que indica que hay un coche. Utilizaré típicamente una forma más simple: apenas una línea, sin el rombo, para indicar una asociación. **[5]**)

[5] Esto es usualmente suficiente detalle para la mayoría de los diagramas, y no necesita obtener específico sobre si está usando agregación o composición.

La composición viene con mucha flexibilidad. Los objetos del miembro de su nueva clase son normalmente privados, haciéndolos inaccesibles a los programadores de cliente que están utilizando la clase. Esto permite que usted cambie a esos miembros sin interferir en el código existente del cliente. Usted puede también cambiar los objetos del miembro en el tiempo de ejecución, para cambiar dinámicamente el comportamiento de su programa. La herencia, que se describe después, no tiene esta flexibilidad puesto que el compilador debe poner restricciones en tiempo de compilación en las clases creadas con herencia.

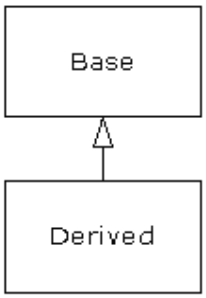
Porque la herencia es tan importante en la programación orientado al objeto a menudo se es altamente acentuada, y el nuevo programador puede conseguir la idea que la herencia se deba utilizar por todas partes. Esto puede dar lugar a diseños torpes y excesivamente complicados. En su lugar, usted debe primero mirar la composición al crear nuevas clases, puesto que es más simple y más flexible. Si usted toma este acercamiento, sus diseños estarán más clarificados. Una vez que el usted tenga cierta experiencia, verá razonablemente obvio cuando necesita la herencia.

Herencia: reutilización de la interfaz

Por sí misma, la idea de un objeto es una herramienta conveniente. Permite que usted empaquete datos y funcionalidad junta por conceptos, así puede representar una idea apropiada del espacio del problema antes que ser forzado a utilizar los idiomas de la máquina subyacente. Estos conceptos son

expresados como unidades fundamentales en el lenguaje de programación usando la palabra clave **class**

se parece una compasión, sin embargo, ir a todo el apuro a crear una clase y después ser forzado a crear una nueva forma que puede tener una funcionalidad similar. Resulta más agradable si podemos tomar la clase existente, la reproducimos, y después hacemos agregados y modificaciones a su copia. Esta es efectivamente lo que usted consigue con la *herencia*, a excepción del hecho que si la clase original (llamada *clase base* o *superclase* o *la clase padre*) se cambia, el "clon" modificado (llamada *clase derivada* o *clase heredada* o *subclase* o *clase hijo*) también refleja esos cambios.



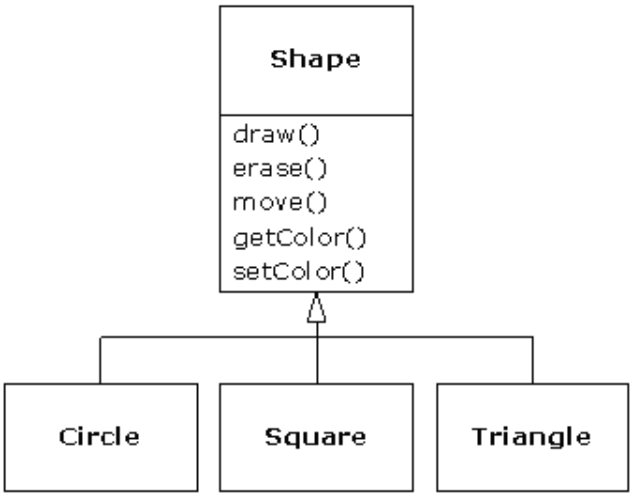
(La flecha de este diagrama UML apunta desde la clase derivada hacia la clase baja. Como usted verá, hay comúnmente más de una clase derivada.)

El tipo hace más que describir las contracciones en un sistema de objetos; también tiene relaciones con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que otro y puede también manejar más mensajes (o manejarlos diferentemente). La herencia expresa esta semejanza entre los tipos usando el concepto de tipos bases y de tipos derivados. Un tipo base contiene todas las características y comportamientos que se comparten entre los tipos derivados de él. Usted crea un tipo base para representarlo dentro de sus ideas sobre algunos objetos de su sistema. Del tipo base, usted deriva otros tipos para expresar las distintas maneras en que esta base puede ser observada.

Por ejemplo, una máquina de reciclaje de basura clasifica pedazos de basura. El tipo base es "basura" y que cada porción de basura tiene un peso, un valor, etcétera, y puede ser destruida, derretida, o descompuesta. Después de esto, tipos más específicos de basura son derivados y pueden tener características adicionales (una botella tiene un color) o comportamientos (una lata del aluminio se puede machacar, una lata de acero es magnética). Además, algunos comportamientos pueden ser diferentes (el valor del papel depende de su tipo y condición). Usando herencia, usted puede construir una jerarquía del tipo que exprese EL problema que intenta solucionar en términos de sus tipos.

Un segundo ejemplo es la "figura" clásica, quizás utilizado en un sistema de diseño automatizado o simulación de juego. El tipo base es "figura" y cada una tiene un tamaño, un color, una posición, etcétera. Cada figura se puede dibujar, borrar, mover, colorear, etc. Desde esta, los tipos específicos de

figuras son derivados (heredados) - círculo, cuadrado, triángulo, etcétera - cada uno de las cuales pueden tener características y comportamientos adicionales. Ciertas figuras se pueden estirarse, por ejemplo. Algunos comportamientos pueden ser diferentes, por ejemplo cuando usted desea calcular el área de una figura. La jerarquía del tipo incorpora las semejanzas y las diferencias entre las figuras.



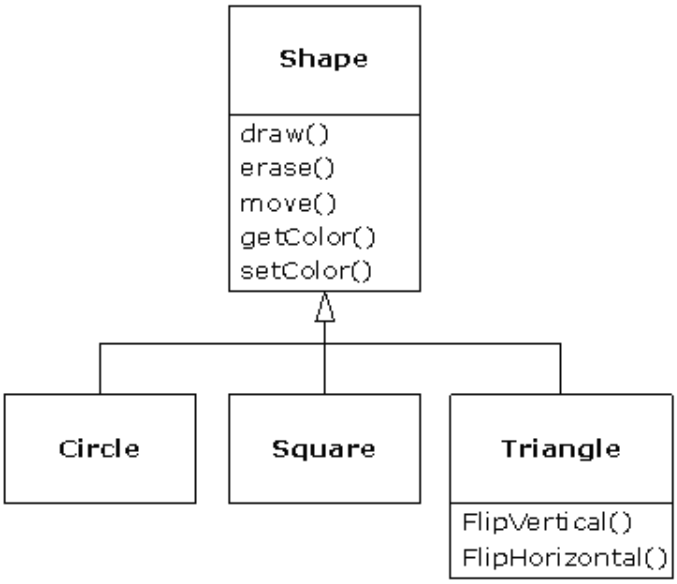
Modelar la solución en los mismos términos que el problema es enormemente beneficioso porque no tiene mucha necesidad de modelos intermedios para conseguir una descripción del problema a una descripción de la solución. Con los objetos, el tipo jerárquico es el primer modelo, así que usted va directamente de la descripción del sistema en el mundo real a la descripción del sistema en código. De hecho, una de las dificultades que la gente tiene con el diseño orientado al objeto es que es demasiado simple conseguir desde el principio al final. Una mente entrenada para buscar soluciones complejas puede inicialmente estar encantada por esta simplicidad.

Cuando hereda de un tipo existente, usted crea un nuevo tipo. Este nuevo tipo contiene no solamente a todos los miembros del tipo existente (aunque los **private** están inaccesibles y ocultados), pero lo más importante duplica la interfaz de la clase base. Es decir, todos los mensajes que usted puede enviar a los objetos de la clase base puede también enviarlos a los objetos de la clase derivada. Puesto que sabemos el tipo de una clase por los mensajes que podemos enviarle, esto significa que la clase derivada *es del mismo tipo que la clase base*. En el ejemplo anterior, "un círculo es una figura". Este tipo de equivalencia vía herencia es una de las entradas fundamentales para entender el significado de la programación orientada al objeto.

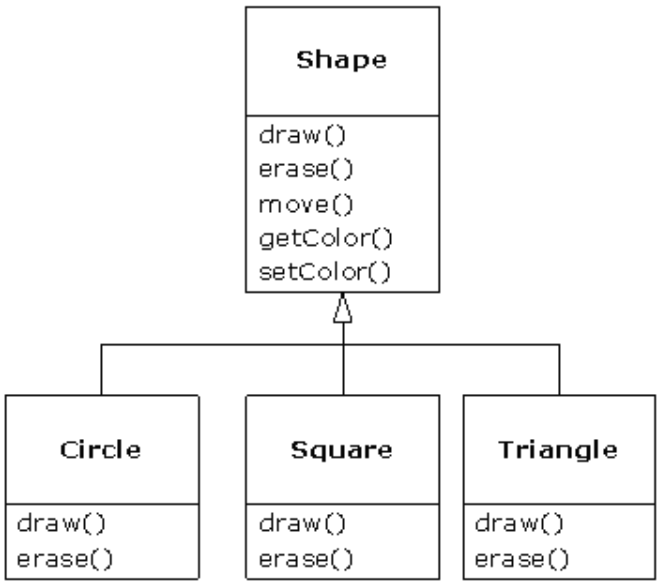
Puesto que la clase base y la clase derivada tienen la misma interfaz fundamental, debe haber una cierta implementación que vaya junto con esa interfaz. Es decir, debe haber cierto código para ejecutarse cuando un objeto recibe un mensaje particular. Si usted hereda simplemente una clase y no le hace algo, los métodos de la interfaz de la clase base vienen directamente a

instalarse en la clase derivada. Esto quiere decir que los objetos de la clase derivada tienen no solamente el mismo tipo, sino también el mismo comportamiento, que no interesa particularmente.

Usted tiene dos maneras de distinguir su nueva clase derivada de la clase base original. El primer es absolutamente directo: agrega simplemente nuevos métodos de marca a la clase derivada. Estos nuevos métodos no son parte de la interfaz de la clase base. Esto significa que la clase base no hace simplemente tanto como lo que desea, así que usted agrega más métodos. Este uso simple y primitivo de la herencia es, ocasionalmente, la solución perfecta a su problema. Sin embargo, usted debe mirar de cerca por la posibilidad de que su clase base puede también necesitar estos métodos adicionales. Este proceso de descubrimiento y de iteración de su diseño sucede regularmente en la programación orientada al objeto.



Aunque la herencia puede implicar a veces (especialmente en Java, donde está la palabra clave **extends** para herencia) que usted va a agregar los nuevos métodos a la interfaz, esto no es necesariamente verdad. La segunda y más importante forma de distinguir su nueva clase es *cambiar* el comportamiento de un método existente de la clase base. Esto es referido *sobrecarga* de aquél método.



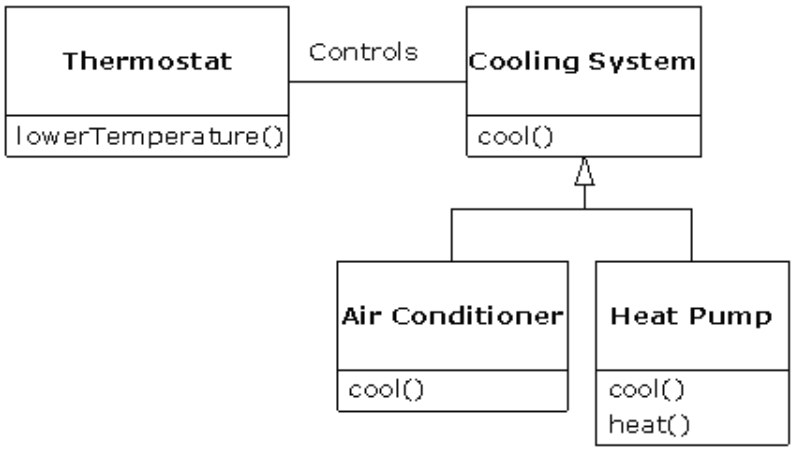
Para sobrecargar un método, usted crea simplemente una nueva definición de ese método en la clase derivada. Usted dice: "estoy usando el mismo método de la interfaz aquí, pero yo quisiera que hiciera algo diferente para mi nuevo tipo".

Relaciones Es un contra ser-como-uno

Hay cierta discusión que puede ocurrir sobre la herencia: ¿"No debiera la herencia *solamente* sobrecargar los métodos de la clase base (y no agregar nuevos métodos que no están en la clase base)"? Esto significaría que el tipo derivado es *exactamente* del mismo tipo que la clase base puesto que tiene exactamente la misma interfaz. Consecuentemente, usted puede sustituir exactamente un objeto de la clase derivada por un objeto de la clase base. Esto se puede pensar como *substitución pura*, y son designados a menudo el *principio de la substitución*. En un sentido, ésta es la manera ideal de tratar la herencia. Nos referimos a menudo a la relación entre la clase base y las clases derivadas como un caso de la relación *"es un"* , porque usted puede decir "el círculo *es una* figura". Una prueba para la herencia es determinar si puede indicar la relación "es un" sobre las clases y hacer que tenga sentido.

Hay momentos en que debe agregar nuevos elementos a la interfaz de un tipo derivado, extendiendo la interfaz y creando un nuevo tipo. El nuevo tipo se puede todavía sustituir por el tipo base, pero la substitución no es perfecta porque sus nuevos métodos no son accesibles desde el tipo base. Esto se puede describir como una relación *ser-como-un*(mi término). El nuevo tipo tiene la interfaz del tipo anterior pero también contiene otros méto dos, así que no puede decir que son realmente iguales con exactitud. Por ejemplo, considere un acondicionador de aire. Supongamos que su casa cableada con todos los controles para refrescarse; es decir, tiene una interfaz que permita que usted controle el frío. Imagínese que el acondicionador de aire se rompe y lo substituye por una bomba de calor, que puede calentar y refrescar. La

bomba de calor *es como un* acondicionador de aire, pero puede hacer mucho más. Porque el sistema de control de su casa está diseñado para controlar solamente el frío, se encuentra restringida a comunicarse con la parte que refresca del nuevo objeto. La interfaz del nuevo objeto ha sido extendida, y el sistema existente no sabe sobre otra cosa que no sea la interfaz original.



Por supuesto, una vez que usted vea este diseño llega a ver claramente que el "sistema de frío" de la clase base no es tan general, y debe ser retitulado como "sistema de control de temperatura" de modo que pueda también incluir calor - en este punto trabajará el principio de la substitución. Sin embargo, este diagrama es un ejemplo de qué puede suceder en el diseño del mundo real.

Cuando usted ve que el principio de la substitución es fácil sentirá como este acercamiento (substitución pura) es la única manera de hacer cosas, y de hecho es agradable si su diseño se resuelve de esa manera. Pero hallará que hay veces en que es igualmente claro que debe agregar nuevos métodos a la interfaz de una clase derivada. Con la inspección ambos casos deben ser razonablemente obvios.

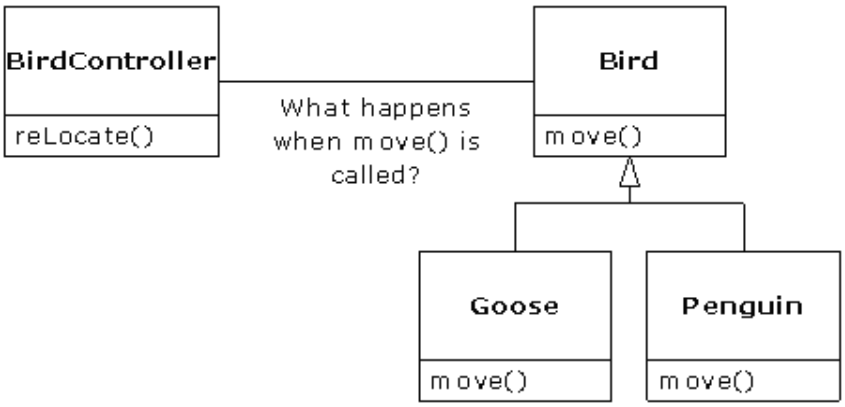
Intercambio de objetos con polimorfismo

Cuando se ocupa de jerarquías de tipo, usted desea a menudo tratar un objeto no como el tipo específico que es, sino que por el contrario como su tipo base. Esto permite que usted escriba código que no depende de un tipo específico. En el ejemplo de la figura, los métodos manipulan formas genéricas sin preocuparse si el ellos son círculos, cuadrados, triángulos, o una figura cualquiera que aún no ha sido definida todavía. Todas las figuras pueden ser dibujadas, borradas y movidas, así que estos métodos simplemente envían un mensaje a un objeto figura; sin preocuparse sobre cómo el objeto hace frente al mensaje.

Tal código no es afectado por la adición de nuevos tipos, y el agregado de nuevos tipos es la forma más común de ampliar un programa orientado al

objeto para manejar nuevas situaciones. Por ejemplo, puede derivar un nuevo subtipo de figura llamado pentágono sin modificar los métodos que se ocupan solamente de figuras genéricas. Esta capacidad de ampliar fácilmente un diseño derivando nuevos subtipos es una de las maneras fundamentales de encapsular el cambio. Esto mejora grandemente los diseños mientras que reduce el costo de mantener el software.

Hay un problema, sin embargo, con intentar tratar objetos del tipo derivados como sus tipos bases genéricos (círculos como formas, bicicletas como vehículos, cormoranes como pájaros, etc.). Si un método le dice a una figura genérica dibujarse, o un vehículo genérico encender, o a un pájaro genérico moverse, el compilador no puede saber en tiempo de compilación qué porción del código será ejecutada exactamente. Este es el punto central - cuando el mensaje es enviado, el programador no *desea* conocer qué porción del código será ejecutado; el método dibujar se puede aplicar igualmente a un círculo, a un cuadrado, o a un triángulo, y el objeto ejecutará el código apropiado dependiendo de su tipo específico. Si usted tiene que saber qué porción del código será ejecutado, entonces cuando agregue un nuevo subtipo, el código que se ejecuta puede ser diferente sin requerir cambios en la llamada al método. Por lo tanto, el compilador no puede saber exactamente qué porción del código se ejecuta, ¿así qué es lo que hace? Por ejemplo, en el diagrama siguiente el objeto de **ControlarAve** trabaja con los objetos genéricos de **Ave** y no sabe de qué tipo exacto son. Esto es conveniente desde la perspectiva de **ControlarAve** porque no tiene que escribir código especial para determinar el tipo exacto de de **Ave** con que trabaja o ese comportamiento de **Ave**. Por tanto ¿cómo sucede esto, cuando se llama a **mover()** mientras que ignora el tipo específico de **Ave**, que el comportamiento correcto ocurra. (un **Ganso** corre, vuela, o nada, y un **pingüino** corre o nada)?



La respuesta es el primer giro en la programación orientada al objeto: el compilador no puede hacer una llamada de la función en el sentido tradicional. La llamada de la función generada por un compilador de no-OOP causa lo que se llama *amarre temprano*, un término que usted pudo no haber oído antes porque nunca pensó en él de otra manera. Significa que el compilador genera una llamada a un nombre específico de la función y el linqueador resuelve esta llamada a la dirección absoluta del código que se ejecutará. En OOP, el programa no puede determinar la dirección del código hasta tiempo de

ejecución, así que cierto esquema diferente es necesario cuando un mensaje se envía a un objeto genérico.

Para solucionar el problema, los lenguajes orientados al objeto utilizan el concepto del *amarre final*. Cuando envía un mensaje a un objeto, el código que está siendo llamado no se determina hasta tiempo de ejecución. El compilador se asegura de que el método exista y realiza la comprobación del tipo sobre los argumentos y el valor de retorno (un lenguaje en el cual esto no es verdad se llama de *tipificación débil*), pero no conoce el código exacto para ejecutarse.

Para realizar el amarre final, Java utiliza un bit especial del código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo del método, usando información almacenada en el objeto (este proceso se cubre en gran detalle en el capítulo 7). Así, cada objeto puede comportarse diferentemente según el contenido de ese bit especial del código. Cuando envía un mensaje a un objeto, el objeto calcula qué hacer con ese mensaje.

En algunos lenguajes usted debe indicar explícitamente que quisiera que un método tuviera para obtener la flexibilidad de las características amarre final (en las aplicaciones de C++ la palabra clave **virtual** hace esto). En estos lenguajes, por defecto, los métodos *no* son limitados dinámicamente. En Java, el amarre dinámico es el comportamiento por defecto y usted no tiene necesidad de recordar agregar otra palabra clave adicional para conseguir polimorfismo.

Considera el ejemplo de la figura. La familia de clases (basadas todas en la misma interfaz uniforme) está diagramada al principio de este capítulo. Para demostrar polimorfismo, deseamos escribir una sola porción de código que no hace caso de los detalles específicos del tipo y se encarga solamente con la clase base. Que el código esté *des-unido* de la información específica de tipo, así es más simple de escribir y más fácil de entender. Y, si un nuevo tipo - un **Hexágono**, por ejemplo - es agregado por herencia, el código que usted escribe trabajará tan bien para el nuevo tipo de **Figura** como en los tipos existentes. Así, el programa es *extensible*.

Si usted escribe un método en Java (pronto aprenderá cómo hacerlo):

```
void
 hacerAlgo(Figura f) {
  f.borrar();
  // ...
  f.dibujar();
}
```

Este método habla a cualquier **Figura**, así que es independiente del tipo específico de objeto que dibuja y borra. Si alguna otra parte del programa utiliza el método del **hacerAlgo()**:

```
Círculo c = new Círculo();
Triángulo t = new Triángulo();
```

```
Línea l = new Línea();
hacerAlgo(c);
hacerAlgo(t);
hacerAlgo(l);
```

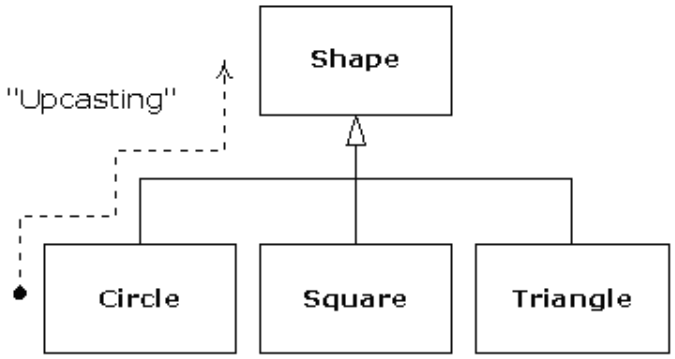
Las llamadas para **hacerAlgo()** trabajan automáticamente de manera correcta, sin importar el tipo exacto del objeto.

Este es un truco asombroso. Considere la línea:

```
hacerAlgo(c);
```

Lo que sucede aquí es que un **Círculo** se está pasando a un método que espera una **Figura**. Puesto que un **Círculo** *es* una **Figura** puede ser tratada como si fuera una por **hacerAlgo()**. Es decir, cualquier mensaje que **hacerAlgo()** pueda enviar a una **Figura**, un **Círculo** puede aceptarla. Es una cosa totalmente segura y lógica de hacer.

Llamamos este proceso de tratar un tipo derivado como si fuera su tipo base *elevación de molde*. El nombre de *molde* se utiliza en el sentido de amoldar en un molde y *elevación* viene de la manera en que el diagrama de la herencia está realizado típicamente, con el tipo base en la parte superior y las clases derivadas abajo desplegadas. Así, modelar un tipo base es elevar en el diagrama de herencia: "elevación de molde".



Un programa orientado al objeto contiene alguna elevación de moldes en alguna parte, porque esto es cómo usted se des-une de conocer sobre el tipo exacto con el que trabaja. Mire el código en **hacerAlgo()** :

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

Note que no dice "Si es un **Círculo**, haga esto, si es un **Cuadrado**, esto otro, etc." Si usted escribe aquél de manera que el código compruebe por todos los tipos posibles de **Figura** que puede actualmente ser, será sucio y necesitará cambiar cada vez que agrega una nueva clase de **Figura**. Aquí, usted dice: "Si es una Figura, sé que puede **borrar()** y **dibujar()** a sí misma, hacer esto, y tomar con cuidado los detalles correctamente".

Lo que es impresionante del código en **hacerAlgo()** es que, de alguna manera, suceden las cosas rectamente. Llamar a **dibujar()** para el **Círculo** hace código diferente para ser ejecutada que al llamar el **dibujar()** para un **Cuadrado** o una **Línea**, pero cuando el mensaje de **dibujar()** es enviado a una **Figura** anónima, el comportamiento correcto ocurre basado en el tipo real de la **Figura**. Esto es asombroso porque, según lo mencionado anteriormente, cuando el compilador de Java está compilando el código para **hacerAlgo()**, no puede saber exactamente de qué tipos se está ocupando. De ordinario, usted espera que termine llamando la versión de **borrar()** y **dibujar()** para la clase base de **Figura**, y no específica para **Círculo**, **Cuadrado**, o **Línea**. Pero la cosa sucede bien debido al polimorfismo. El compilador y el sistema de tiempo de ejecución manejan los detalles; todo lo que necesita saber ahora es que sucede, y más importantemente, cómo diseñar con él. Cuando usted envía un mensaje a un objeto, el objeto hará las cosas correctamente, incluso cuando la elevación de molde esté implicada.

Las clases bases abstractas y las interfaces

A menudo en un diseño, usted quisiera que la clase base presentara *solamente* una interfaz para sus clases derivadas. Es decir, no quiere que cualquier persona cree realmente un objeto de la clase base, sólo para elevar el molde para poder utilizar su interfaz. Esta es logrado haciendo esa clase *abstracta* usando la palabra clave **abstract**. Si cualquier persona intenta hacer un objeto de una clase **abstract**, el compilador la prevendrá. Esto es una herramienta para hacer cumplir un diseño particular.

Usted puede también utilizar la palabra clave **abstract** para describir un método que el aún no ha sido implementado - como una porción que indica "aquí está un método interfaz para todos los tipos heredados de esta clase, pero en este punto no se ha implementado nada con él". Un método **abstract** se puede crear solamente dentro de una clase **abstract**. Cuando se hereda la clase, ese método debe ser implementado, o la clase heredada viene a ser **abstract** también. Crear un método **abstract** permite que usted ponga un método en una interfaz sin estar forzado a proporcionar un posible cuerpo de código sin sentido para ese método.

La palabra clave **interface** toma el concepto de una clase **abstract** de la un poco más lejos previniendo cualquier definición de método alguno. El **interface** es una herramienta muy práctica y comúnmente usada, pues proporciona la separación perfecta de la interfaz y de la implementación. Además, usted puede combinar muchas interfaces juntas, si usted desea, mientras que la herencia de múltiples clases regulares o de clases abstractas no es posible.

Creación de objetos, utilización y tiempos de vida

Técnicamente, la OOP está justo sobre la tipificación de datos abstractos, herencia, y polimorfismo, pero otras visiones pueden ser al menos importantes. Esta sección cubrirá estas visiones.

Uno de los factores más importantes de los objetos es la manera en que se crean y se destruyen. ¿Dónde están los datos para un objeto y cuál es el tiempo de vida del objeto se controla? Hay diferentes filosofías de trabajo aquí. C++ toma el acercamiento de que el control de la eficacia es la visión más importante, así que le da al programador una opción. Para la máxima velocidad en tiempo de ejecución, el almacenamiento y el tiempo de vida se pueden determinar mientras que se está escribiendo el programa, poniendo los objetos en la pila (éstas se llaman a veces variables *automáticos* o *scoped* variables) o en el área de almacenamiento estático. Esto pone una prioridad en la velocidad de la asignación y liberación de almacenamiento, y el control de éstos puede tener mucho valor en algunas situaciones. Sin embargo, usted sacrifica flexibilidad porque debe saber la cantidad, el tiempo de vida, y el tipo exacto de objeto mientras que está escribiendo el programa. Si usted está intentando solucionar un problema más general tal como diseño automatizado, administración de depósito, o control del tráfico aéreo, éste es demasiado restrictivo.

La segundo aproximación es crear los objetos dinámicamente en un conjunto de memoria llamado el montón. En esta aproximación, usted no sabe hasta tiempo de ejecución cuántos objetos necesitará, cuál es su tiempo de vida, o cuál es su tipo exacto. Éstos son determinados en el momento que son incentivados mientras el programa está corriendo. Si necesita un nuevo objeto, simplemente lo crea en el montón, en el punto que lo necesite. Porque el almacenamiento se administra dinámicamente, en tiempo de ejecución, la cantidad de tiempo requerida para asignar el almacenamiento en el montón puede ser perceptiblemente más extenso que cuando crea el almacenamiento en pila. (Crear el almacenamiento en la pila es a menudo una sola instrucción de ensamblador para mover el puntero de pila hacia abajo y otro para moverlo de regreso. El momento para crear el almacenamiento de montón depende del diseño del mecanismo del almacenamiento). La aproximación dinámica hace la presunción generalmente lógica que los objetos tienden a ser complicados, así que los gastos indirectos adicionales de encontrar el almacenamiento y de liberar aquél no tendrá un impacto importante en la creación de un objeto. Además, la mayor flexibilidad es esencial de solucionar para el problema general de programación.

Java utiliza la segunda aproximación, exclusivamente. **[6]** Cada vez que usted desea crear un objeto, utiliza la palabra clave **new** para construir un instancia dinámica de ese objeto.

[6] Los tipos primitivos, sobre los cuales aprenderá más tarde, son un caso especial.

Hay otra cuestión, sin embargo, y ése es el tiempo de vida de un objeto. Con los lenguajes que permiten que los objetos sean creados en la pila, el

compilador determina cuánto tiempo dura el objeto y puede destruirlo automáticamente. Sin embargo, si usted lo crea en el montón el compilador no tiene conocimiento de su tiempo de vida. En una lenguaje como C++, usted debe determinar en tiempo de programación cuando destruir el objeto, lo cual puede inducir a pérdidas de memoria si no lo hace correctamente (y éste es un problema común en programas de C++). Java proporciona una característica llamada *recolector de basura (garbage collector)* que descubre automáticamente cuando un objeto no es más usado por largo tiempo y lo destruye. Un recolector de basura es mucho más conveniente porque reduce el número de los caminos que debe seguir y del código que debe escribir. Más importante, el recolector de basura proporciona un nivel mucho más alto de seguridad contra el problema insidioso de las fugas de memoria (el cual ha traído a muchos un proyecto de C++ a sus rodillas).

Colecciones e iteradores

Si usted no conoce cuántos objetos va a necesitar para solucionar un problema particular, o cuánto tiempo durarán, tampoco conoce cómo almacenar esos objetos. ¿Cómo puede usted saber cuánto espacio necesita para crear esos objetos? Usted no puede, entonces esa información no es conocida hasta tiempo de ejecución.

La solución a la mayoría de los problemas en diseño orientado al objeto parece ligero: Usted crea otro tipo de objeto. El nuevo tipo de objeto que soluciona este problema particular mantiene referencias a otros objetos. De hecho, usted puede hacer la misma cosa con un arreglo (array), que está disponible en la mayoría de los lenguajes. Pero este nuevo objeto, generalmente llamado un *contenedor (container)* (también llamado una *colección (collection)*, pero las aplicaciones de la biblioteca de Java las llaman con diferentes sentidos así que este libro utilizará “contenedor (container)”), se ampliará siempre que sea necesario para acomodar todo lo que coloque dentro de él. Así no necesita conocer cuántos objetos va a mantener un contenedor. Apenas crea un objeto contenedor y deja que el tome cuidado de los detalles.

Afortunadamente, un buen lenguaje de POO viene con un sistema de contenedores como parte del paquete. En C++, esta es parte de la biblioteca estándar de C++ y a veces se llaman Biblioteca Estándar de Plantillas (Standard Template Library-STL). Object PASCAL tiene contenedores en su Biblioteca de Componentes Visuales (Standard Template Library-VCL). Smalltalk tiene un sistema muy completo de contenedores. Java también tiene contenedores en su biblioteca estándar. En algunas bibliotecas, un contenedor genérico se considera suficientemente bueno para todas las necesidades, y en otros (Java, por ejemplo) la biblioteca tiene diversos tipos de contenedores para distintas necesidades: diferentes formas de clases **List** (llevan a cabo secuencias), clases **Map** (también conocido como *arreglos de asociación*, para asociar objetos a otros objetos), y el clases **Set** (para mantener uno de cada tipo de objeto). Las bibliotecas de contenedores pueden también incluir colas (qeues), árboles (trees), pilas (stacks), etc.

Todos los contenedores tienen alguna manera de poner y sacar cosas; hay métodos usualmente para agregar elementos a un contenedor, y otros para extraer estos elementos de vuelta. Aunque traer elementos puede ser más problemático, porque un método de selección simple es restrictivo. ¿Pero qué si desea comparar o manipular un grupo de elementos del contenedor en vez de únicamente uno?

La solución es un *iterador (iterator)*, el cual es un objeto cuyo trabajo es seleccionar los elementos dentro de un contenedor y presentarlos al usuario del iterador. Como una clase, también provee un nivel de abstracción. Esta abstracción puede ser utilizada para separar los detalles del contenedor del código al que está accediendo ese contenedor. El contenedor, vía el iterador, es abstraído a una secuencia simple. El iterador le permite atravesar esta secuencia sin preocuparlo de la estructura implícita - esto es, ya sea una **lista de arreglos (ArrayList)**, una **lista enlazada (LinkedList)**, una **Pila (Stack)**, o algo similar. Esto le permite flexibilidad para cambiar fácilmente la estructura de datos esencial sin distorsionar el código de su programa. Java comenzó (en la versión 1.0 y 1.1) con un iterador estándar, llamado **Enumeration**, para todas sus clases contenedor. Java 2 agrega una librería contenedor mucho más completa que contiene un iterador llamado **Iterator** que hace mucho más que el viejo **Enumeration**.

Desde el punto de vista del diseño, todo lo que realmente desea un una secuencia que puede ser manipulada para resolver su problema. Si un tipo único de secuencia satisface todas sus necesidades, no hay razón para tener diferentes maneras. Hay dos razones que necesitan una elección de contenedores. Primero, los contenedores proveen diferentes tipos de interfaces y comportamientos externos. Una pila tiene una interfaz y comportamientos diferentes de una cola, la cual es diferente de un grupo o una lista. Una de estas debe proveer mayor flexibilidad a la solución de su problema que algún otro. Segundo, diferentes contenedores tienen diferentes rendimientos para ciertas operaciones. El mejor ejemplo es comparar dos tipos de **List**: un **ArrayList** y un **LinkedList**. Ambos son simples secuencias que pueden tener idénticas interfaces y comportamientos. Pero ciertas operaciones pueden tener costos radicalmente diferentes. El acceso azaroso (randomly) de elementos en un **ArrayList** es una operación de tiempo constante; toma la misma cantidad de tiempo a pesar del elemento seleccionado. Sin embargo, en un **ArrayList** es caro moverse a través de la lista para seleccionar un elemento, y toma muchísimo tiempo encontrar un elemento que está en el último lugar de la lista. De otra manera, si desea insertar un elemento en el medio de una secuencia, es más barato en una **LinkedList** que en un **ArrayList**. Estas y otras operaciones tienen diferentes rendimientos dependiendo de la estructura esencial de la secuencia. En la fase de diseño, debe empezar con un **LinkedList** y, cuando afine el funcionamiento, cambiar a un **ArrayList**. Porque vía la abstracción de la clase base **List** y los iteradores, puede cambiar de uno hacia otro con mínimo impacto en su código.

La jerarquía de raíz única

Una de las razones en que la POO viene especialmente sobresaliendo desde la introducción del C++ es donde todas las clases deben finalmente ser herederas de una única clase base. En Java (como virtualmente todos los otros lenguajes POO, a excepción de C++) la respuesta es sí, y el nombre de esta clase base primaria es simplemente **Object**. De ello resulta que los beneficios de la jerarquía de raíz única sean muchos.

Todos los objetos en una jerarquía de raíz única tienen una interfaz en común, así que ellas son finalmente del mismo tipo fundamental. La alternativa (provista por C++) es que no conozca que todo sea del mismo tipo básico. Desde un punto de vista de compatibilidad hacia atrás esto llena el modelo de C mejor y puede ser pensado de una manera menos restrictiva, pero cuando desea hacer algo en programación orientada completamente al objeto debe entonces construir su propia jerarquía para proveer la misma conveniencia que construir dentro de otros lenguajes POO. Y en cualquier nueva librería de clases que adquiera, alguna otra interfaz incompatible será usada. Ello requiere esfuerzo (y posiblemente herencia múltiple) para trabajar la interfaz nueva dentro de su diseño. ¿Es la "flexibilidad" de C++ lo buscado? Lo es si lo necesita - si tiene una gran inversión en C - es suficientemente valorable. Si está empezando desde cero, otras alternativas tales como Java pueden ofrecerle mayor productividad.

Todos los objetos en una jerarquía de raíz única (tal como Java provee) pueden garantizarle tener cierta funcionalidad. Conocerá que puede llevar a cabo operaciones básicas seguras sobre todos los objetos de su sistema. Una jerarquía de raíz única, a lo largo del tiempo mientras crea todos los objetos en el montón, simplifica mayormente el pasaje de argumentos (uno de los tópicos más complejos en C++).

Una jerarquía de raíz única hace mucho más fácil implementar un recolector de basura (el cual está apropiadamente construido dentro de Java). El soporte necesario puede ser instalado en la clase base, y el recolector de basura puede enviarles los mensajes apropiados a cada objeto en el sistema. Sin una jerarquía de raíz única y un sistema para manipular un objeto mediante referencia, es difícil implementar un recolector de basura.

Desde tiempo de ejecución el tipo de información está garantizado será en todos los casos objetos, nunca acabará en un objeto cuyo tipo no pueda determinar. Esto es especialmente importante con operaciones a nivel de sistema, tales como manejo de excepciones, y permitir gran flexibilidad en la programación.

"Modelado hacia abajo" (Downcasting) contra plantillas/genéricas

Para hacer estos contenedores reutilizables, ellos mantienen un tipo universal en Java:**Object**. La jerarquía de raíz única significa que todo es un **Objeto (Object)**, así un contenedor que se sostiene en **Object**s puede mantener todo **[7]** Esto hace a los contenedores fáciles de reutilizar.

[7] Excepto, desafortunadamente, para las primitivas. Esto es discutido en detalle más adelante en este libro.

Para usar un contenedor, simplemente agréguele referencias a objetos y a lo último pregunte por ellos de nuevo. Pero, ya que el contenedor conserva solamente **Object**s, cuando agrega su referencia a objeto en el contenedor esta "modelando hacia arriba" hacia **Object**, esto es pierde su identidad. Cuando lo trae de vuelta, obtiene una referencia a **Object**, y no una referencia al tipo que puso en él. ¿Así cuando regresa dentro de aquella interfaz utilizada para el objeto que puso en el contenedor?

Aquí el modelado es usado nuevamente, pero esta vez no es hacia arriba la heredad de la jerarquía hacia un tipo más general. En contrario, usted modela hacia abajo de la jerarquía para un tipo más específico. Esta interpretación del modelado es llamada *modelado hacia abajo (downcasting)*. Con el modelado hacia arriba conoce, por ejemplo, que un **Círculo** es un tipo de **Figura**, por lo cual está seguro del modelado hacia arriba, pero no conoce necesariamente que un **Object** es un **Círculo** o una **Figura** por tanto no está tan seguro del modelado hacia abajo a menos que conozca exactamente sobre el tratamiento que hace.

No es, sin embargo, completamente peligroso, porque si modela hacia abajo y ocurre un error obtendrá un error en tiempo de ejecución llamado *excepción (Exception)*, el cual será descrito en breve. Cuando extrae las referencias a objeto desde el contenedor, asimismo, debe tener alguna manera de recordar exactamente qué son ellos así puede realizar un modelado hacia abajo apropiadamente.

El modelado hacia abajo y el chequeo en tiempo de ejecución requieren tiempo extra para correr el programa y sobreesfuerzo para el programador. ¿No tendría sentido crear el contenedor así conoce los tipos que mantiene, eliminando la necesidad para el modelado hacia abajo y un posible error? La solución es llamada mecanismo de *tipo parametrizado (parameterized type)*. Un tipo parametrizado es una clase que el compilador puede personalizar automáticamente para trabajar con tipos particulares. Por ejemplo, con un contenedor parametrizado, el compilador debiera modelar que el contenedor deba aceptar únicamente **Figura** y extraer solamente **Figura**s.

Los tipos parametrizados son una parte importante de C++, particularmente porque C++ no tiene una jerarquía de raíz única. En C++, la palabra clave que implementa los tipos parametrizados es "template". Java actualmente no tiene tipos parametrizados ya que es posible para ello obtener por - no obstante la incomodidad - medio de la jerarquía de raíz única. Sin embargo, una propuesta reciente para los tipos parametrizados utiliza una sintaxis que está llamativamente parecida a los templates de C++, y podemos esperar ver tipos parametrizados (que serán llamados *generics* en la siguiente versión de Java).

Asegurar la limpieza justa

Cada objeto requiere recursos, memoria mayormente, en cuanto a su existencia. Cuando un objeto no es utilizado por un largo período debe ser limpiado así esos recursos están disponibles para su rehúso. En situaciones de programaciones sencillas la cuestión de cuándo un objeto está para limpiarse no se ve tan difícil: crea un objeto, lo usa mientras sea necesario, y luego lo destruye. Sin embargo, no es difícil encontrar situaciones más complejas.

Suponga, por ejemplo, que está diseñando un sistema para administrar el tráfico aéreo de un aeropuerto. (El mismo modelo debe también trabajar para manejar cajas en un depósito, o un sistema de videos rentados, o un canil en una guardería de mascotas). Al principio se verá simple: hace un contenedor para sostener los aviones, entonces crea un nuevo avión y lo ubica en el contenedor para cada avión que ingresa a la zona de control de tráfico aéreo. Para limpiar, simplemente borra el objeto del avión apropiado cuando uno deja la zona.

Pero puede que tenga otro sistema para registrar datos sobre los aviones; datos que quizás no necesiten atención inmediata como la función de control principal. Podría ser un registro de un plan de vuelo de un avión pequeño, y si crea un objeto plan también lo pone en este contenedor segundo si es un plan pequeño. Entonces algunos procesos de segundo plano realizan operaciones sobre estos objetos en este contenedor durante momentos inactivos.

Ahora el problema es más difícil: ¿Cómo puede posibilitar conocer cuando destruir los objetos? Cuando está con el objeto, pueden otras partes del sistema no estarlo. Este mismo problema puede aparecer en un número de otras situaciones, y en sistemas de programación (tales como en C++) en los cuales debe explicitar el borrado de un objeto cuando esté completado el uso de él lo cual es un tanto complejo.

Con Java, el recolector de basura está diseñado para tener cuidado del problema de liberación de memoria (aunque esto no incluye otros aspectos de limpieza de un objeto). El recolector de basura conoce cuando un objeto no está mucho tiempo en uso, y entonces libera automáticamente la memoria de aquél objeto. Esto (combinado con el hecho de que todos los objetos son heredados de la clase de raíz única **Object** y que puede crear objetos solo de una manera - en el montón) hace que el proceso de programación en Java mucho más sencillo que la programación en C++. Usted tiene pocas decisiones para hacer y saltear los obstáculos.

Recolector de basura contra eficiencia y flexibilidad

Si todo esto es una buena idea, ¿porque no hacen la misma cosa en C++? Bien, porque de hecho hay un precio a pagar por toda la conveniente programación, y ese es la sobrecarga en tiempo de ejecución. Como mencionamos antes, en C++ puede crear objetos en la pila, y en ese caso serán automáticamente limpiados (pero no tendrá la flexibilidad de crear cuantos desee en tiempo de ejecución). Crear objetos en la pila es la manera más eficiente de localizar el almacenamiento de los objetos y luego liberarlo. Crear objetos en el montón puede ser más caro en verdad. Siempre heredar

de una clase base y hacer todas las llamadas a métodos polimórficos conllevan una pequeña tasa. Pero el recolector de basura es un tema singular pro que nunca conocerá cuando está iniciando o cuánto tiempo tomará. Esto significa que hay una inconsistencia en el transcurso de ejecución de un programa Java, de tal manera que no puede usarlo en determinadas situaciones, tales como cuando la velocidad de ejecución de un programa es uniformemente crítica. (Estos son llamados generalmente programas de tiempo real, aunque no todos los problemas de programación en tiempo real son estrictos).

Los diseñadores del lenguaje C++, intentaron agradar a los programadores de C (y en muchos casos lo lograron), no deseando agregar características al lenguaje que fueran a impactar la velocidad o el uso de C++ en cualquier situación donde los programadores puedan elegir C alternativamente. Esta meta fue conseguida, pero al precio de una alta complejidad cuando se programa en C++. Java es mucho más simple que C++, pero la depreciación está en la eficiencia y cierta aplicabilidad. Para una porción significativa de problemas de programación, sin embargo, Java es una elección superior.

Manejo de excepciones: tratando con errores

Ya desde los primeros lenguajes de programación, el manejo de errores ha sido una de las cuestiones más difíciles. Porque es sumamente difícil diseñar un buen esquema de manejo de errores, algunos lenguajes simplemente ignoran la cuestión, dejando el problema a los diseñadores de la biblioteca quienes las toman a mitad de camino por lo cual las medidas trabajaran en algunas situaciones y en otras será fácilmente evitadas, generalmente porque las ignoran. Un problema mayor con la mayoría de los esquemas de manejo de error es que ellos confían en la vigilancia del programador en seguir una convención sobre acordada que no está forzada por el lenguaje. Si el programador no desea controlar - como es el caso si ellos están apresurados - estos esquemas pueden ser fácilmente olvidados.

El manejo de excepción ata el manejo de error directamente dentro del lenguaje de programación y algunas veces aún al sistema operativo. Una excepción es un objeto que es "lanzado" desde el sitio del error y puede ser "tomado" por un apropiado administrador de excepciones diseñado para manejar ese tipo singular de error. Es como si el manejo de excepción es diferente, camino paralelo de ejecución que puede ser tomado cuando las cosas van mal. Y como usa un camino de ejecución separado, no necesita interferir con la ejecución normal de código. Esto logra que el código se simplifique al escribir porque no está forzado constantemente a chequear por errores. Agregado a ello, una excepción lanzada no se parece a un valor de error que es devuelto por un método o una bandera que es fijada por un método en orden a indicar una condición de error - estos pueden ser ignorados. Una excepción no puede ser ignorada, así que está garantizada a ser tratada en algún punto. Finalmente, las excepciones proveen una manera confiable de recuperarse de una mala situación. En vez de solo salir del

programa, está habilitado a fijar las cosas correctamente y restaurar la ejecución, lo cual produce programas mucho más robustos.

El manejo de excepción de Java sobresale a lo largo de los lenguajes de programación, porque en Java, el manejo de excepción está atado desde el principio y usted se ve forzado a usarlo. Si no escribe el código apropiadamente para manejar las excepciones, obtendrá un mensaje de error en tiempo de compilación. Esto consistencia garantizada puede hacer el manejo de errores mucho más simple algunas veces.

Es de valor notar que el manejo de excepción no es una característica orientada a objeto, sino que en los lenguajes orientados a objetos la excepción es normalmente representada por un objeto. El manejo de excepción existe desde antes de los lenguajes orientados a objetos.

Concurrencia

Un concepto fundamental en la programación de computadora es la idea de manejar más de una tarea al mismo tiempo. Algunos problemas de programación requieren que el programa esté habilitado a parar cuando lo hacen, tratarlos con algún otro problema, y regresar al proceso principal. La solución ha sido aproximada de muchas maneras. Inicialmente, los programadores con conocimiento de bajo nivel de la máquina escribieron rutinas de interrupción de servicio, y la suspensión del proceso principal fue iniciada a través de una interrupción hardware. Aunque esto trabaja muy bien, es dificultoso y no portable, ya que hacer un programa para mover a un nuevo tipo de máquina es lento y caro.

Algunas veces, las interrupciones son necesarias para el manejo de tareas en tiempo crítico, pero hay grandes clases de problemas en los cuales simplemente intenta partir el problema en piezas de ejecución separada así el corazón del programa puede ser mucho más reactivos. Dentro de un programa, estas piezas de ejecución separada son llamadas hilos, y el concepto general es denominado *concurrencia (concurrency)* o *multihilo (multithreading)*. Un ejemplo común de multihilo es la interfaz de usuario. Utilizando hilos, un usuario puede presionar un botón y obtener una respuesta rápida en vez que sea forzado a esperar hasta que el programa finalice su tarea actual.

Comúnmente, los hilos son una manera de localizar el tiempo de un procesador único. Pero si el sistema operativo soporta muchos procesadores, cada hilo puede ser asignado a un procesador diferente, y pueden verdaderamente correr en paralelo. Una de las características convenientes del multihilo en el nivel de lenguaje es que el programador no necesita preocuparse si hay muchos procesadores o solo uno. El programa está dividido lógicamente dentro de hilos y si la máquina tiene más de un procesador, entonces el programa correrá rápido sin ajustes especiales.

Todo esto hace un hermoso sonido hilado simple. Hay una trampa: los recursos compartidos. Si tiene más de un hilo corriendo que está esperando para acceder al mismo recurso, tiene un problema. Por ejemplo, dos procesadores no pueden enviar simultáneamente información a una impresora. Para resolver el problema, los recursos que pueden ser compartidos, tales como una impresora, deben ser bloqueados mientras están siendo usados. De tal manera los hilos bloquean un recurso, completan su tarea, y luego liberan el bloqueo ya que alguien más puede usar el recurso.

El hilado de Java está construido dentro del lenguaje, lo cual hace un supuesto complicado mucho más simple. El hilado está soportado a nivel de objeto, así que un hilo de ejecución está representado por un objeto. Java provee también de bloqueo de recursos limitados. Puede bloquear la memoria de algún objeto (lo cual es, después de todo, una manera de compartir recursos) para que solamente un hilo pueda usarlo a la vez. Esto es conseguido con la palabra clave **synchronized**. Otros tipos de recursos deben ser bloqueados explícitamente por el programador, típicamente creando un objeto para representar el bloqueo que todos lo hilos deben chequear antes de acceder a ese recurso.

Persistencia

Cuando crea un objeto, existe tanto tiempo como sea necesario, pero bajo ninguna circunstancia existe cuando el programa finaliza. Mientras esto tiene sentido al principio, hay situaciones en las cuales sería increíblemente útil si un objeto pueda existir y mantener su información aún mientras el programa no esté corriendo. Entonces la próxima vez que inicia el programa, el objeto estará ahí y tendrá la misma información que tuvo en el momento anterior cuando el programa hubo corrido. De hecho, puede obtener un efecto similar escribiendo la información a un archivo o una base de datos, pero en el espíritu de hacer todo un objeto, sería un tanto conveniente estar habilitado para declarar un objeto persistente y tener todos los detalles que toman cuidado para usted.

Java provee soporte para "persistencia liviana", lo cual significa que puede fácilmente almacenar objetos en disco y más tarde recuperarlos. La razón de "liviana" es que está forzado a hacer llamadas explícitas para almacenar y recuperar. La persistencia liviana puede ser implementada ya sea a través de la *serialización de objeto (object serialization)* (tratada en el Capítulo 12) y en *Java Data Objects* (JDO, tratada en *Thinking in Enterprise Java*

Java y la Internet

Si Java es, de hecho, aún otro lenguaje de programación de computadora, se preguntará por qué es tan importante y está siendo promocionado como un paso revolucionario en la programación de computadora. La respuesta no es inmediatamente obvia si proviene de la perspectiva de programación tradicional. Aunque Java es muy útil para resolver problemas de programación

solitarios tradicionales, es también importante porque resolverán problemas de programación en la Amplia Telaraña Mundial.

¿Qué es la Web?

La Web puede verse con un poco de misterio al principio, con todo lo que se habla de "surfear", "presencia" y "páginas de hogar". Es útil parar, volverse y ver qué es realmente esto, pero para hacerlo debe entender los sistemas cliente/servidor, otro aspecto de la computación que está llena de cuestiones confusas.

Computación Cliente/Servidor

La idea primera de un sistema cliente/servidor es que tiene un almacén central de información - algún tipo de dato, ofrecido en una base de datos - que desea distribuir a demanda para otro grupo de personas o máquinas. Una clave para el concepto cliente/servidor es que el almacenamiento de reposición está localizado centralmente así que puede ser cambiado y que estos cambios se propaguen hacia los consumidores de información. Tomado todo junto, el almacén de información, el software que distribuye la información, y la máquina donde la información y el software residen se denomina el *servidor*. El software que reside en la máquina remota, se comunica con el servidor, pide la información, la procesa, y la muestra en la máquina remota es llamada el *cliente*.

El concepto básico de computación cliente/servidor, entonces, no es tan complicado. Los problemas surgen porque usted tiene un único servidor intentando servir a muchos clientes a la vez. Generalmente, un sistema administrador de base de datos está involucrado, así el diseñador "balancea" la capa de datos dentro de tablas para uso óptimo. Además, los sistemas pueden permitir a un cliente insertar información nueva dentro del servidor. Esto significa que debe asegurar que los datos nuevos de un cliente no pase por encima de los datos nuevos de otro cliente, o que los datos no se pierdan en el proceso de agregarlo a la base de datos (esto se denomina proceso de transacción). Como el software cliente cambia, debe ser construido, depurado e instalado en las máquinas clientes, lo cual resultará ser más complicado y caro de lo que pueda pensarse. Esto es especialmente problemático para soportar múltiples tipos de computadoras y sistemas operativos. Finalmente, hay cuestiones importantísimas de rendimiento: debe tener cientos de clientes haciendo peticiones a su servidor al mismo tiempo, así que un pequeño tiempo de espera es crucial. Para minimizar la latencia, los programadores trabajan duro para descargar las tareas de procesamiento, a veces a la máquina del cliente, pero otras a otras máquinas en el sitio del servidor, utilizando lo que se llama *middleware* (El Middleware también se usa para incrementar el mantenimiento).

La idea simple de distribuir información tiene muchas capas de complejidad que el problema central puede verse desesperadamente enigmático. Y aún es crucial: la computación cliente/servidor cuenta para endurecer la mitad de todas las actividades de programación. El es responsable para todo desde

tomar órdenes y transacciones con tarjetas de crédito hasta la distribución de cualquier tipo de datos - stock de mercado, científicos, gubernamentales, nómbrelo usted. Lo que venimos teniendo desde el pasado son soluciones individuales para problemas individuales, inventar una nueva solución cada vez. Estos son difíciles de crear y de usar, y el usuario tiene que aprender una nueva interfaz cada vez. El problema cliente/servidor necesita resolverse en una gran manera.

La Web es un servidor gigante

La Web es actualmente un sistema cliente/servidor gigante. Es un tanto desagradable entonces que, ya que tiene todos los servidores y clientes coexistiendo en una única red a la vez. Usted no necesita conocer esto, porque todo su cuidado es sobre la conexión y la interacción con un servidor a la vez (aún a pesar que puede estar saltando alrededor del mundo en la búsqueda por el servidor correcto).

Inicialmente fue un proceso simple de una sola vía. Usted hace una petición a un servidor y este le entrega un archivo, el cual su software navegador de su máquina (el cliente) debe interpretar para formatear dentro de su máquina local. Pero en corto tiempo la gente empezó a desear hacer más que solicitar páginas de un servidor. Ellos quieren capacidades completas de cliente/servidor para que el cliente pueda cargar información al servidor, por ejemplo, para hacer búsquedas en la base de datos del servidor, para agregar nueva información al servidor, o para realizar una orden (lo cual requiere más seguridad que la ofrecida en los sistemas originales). Estos son cambios que hemos estado viendo en el desarrollo de la web.

El navegador Web fue un gran paso al frente: el concepto que una pieza de información puede ser exhibida en cualquier tipo de computadora sin cambio. Sin embargo, los navegadores son algo bastante primitivos y rápidamente se empantanaron por las demandas ubicadas en ellos. Ellos no son interactivos justamente, y tienden a colgarse ambos, servidor e Internet, porque cualquier momento necesita para hacer algo que requiere programación tiene que enviar información nuevamente al servidor para ser procesados. Puede tomar algunos segundos o minutos resolver si tiene algo errado en su petición. Ya que el navegador era simplemente un visualizador no podía realizar aún simples tareas de computación. (De otra manera, es seguro, porque no puede ejecutar programas en su máquina local que pueda contener errores o virus).

Para resolver este problema, diferentes aproximaciones han sido tomadas. Para empezar, los gráficos estándar han sido mejorados para permitir mejores animaciones y videos dentro de los navegadores. Los restantes problemas pueden ser solucionados solamente incorporando la habilidad de correr programas en el cliente final, bajo el navegador. Esto es la llamada programación del lado cliente.

Programación del lado cliente

En la Web inicial de diseño servidor-navegador proveía de contenido interactivo, pero la interactividad era completamente provista por el servidor. El servidor producía páginas estáticas para el navegador cliente, quien simplemente interpretaría y las mostraría. El *Lenguaje de Marcados para Hipertextos (HyperText Markup Language* - HTML) contiene mecanismos simples para reunir los datos: cajas para ingresar textos, cajas de chequeo, cajas de radio, listas y listas desplegables, así como un botón que solamente puede ser programado para resetear los datos en el formulario o "enviar" esos datos en el formulario hacia el servidor. Esta presentación pasa a través de *Interfaz de Puente Común (Common Gateway Interface* - CGI) provista en todos los servidores Web. El texto dentro del envío indica a CGI que hacer con ellos. La acción más común es correr un programa localizado en el servidor en un directorio típicamente denominado "cgi-bin". (Si observa la barra de dirección en la parte superior de su navegador cuando pulsa un botón de una página Web, puede ver a veces el "cgi-bin" dentro de toda esa jeringaza). Estos programas pueden ser escritos en muchos lenguajes. Perl ha sido una elección común porque está diseñado para manipulación de texto e interpretación, así puede ser instalado en cualquier servidor a pesar del procesador o sistema operativo. Sin embargo, Python (mi favorito - ver www.Python.org) ha estado haciendo avances porque es más poderoso y simple.

Muchos sitios Web poderosos de hoy están construidos estrictamente en CGI, y pueden de hecho hacer casi cualquier cosa con CGI. Sin embargo, los sitios Web construidos sobre programas CGI pueden rápidamente venir a convertirse en complicados para mantener, y hay aún un problema de tiempo de respuesta. La respuesta de un programa CGI depende sobre cuántos datos deben ser enviados, así como también la carga en ambos, servidor e Internet. (Por encima de todo esto, iniciar un programa CGI tiende a ser lento). Los primeros diseñadores de la Web no predijeron cuán rápidamente este ancho de banda sería saturado para las clases de aplicaciones que la gente desarrolló. Por ejemplo, cualquier ordenamiento de gráficos dinámicos es casi imposible de realizar con consistencia porque un archivo *Formato para Intercambiar Gráficos (Graphics Interchange Format* - GIF) debe ser creado y movido desde el servidor hacia el cliente por cada versión del gráfico. Y no tenga dudas que ha dirigido la experiencia con algo como una simple validación de datos en el formulario de entrada. Presiona el botón de envío en una página; los datos son embarcados al servidor; el servidor inicia un programa CGI el cual descubre un error, formatea una página HTML informando del error, y envía la página de regreso a usted; quien debe entonces regresar a la página e intentarlo de nuevo. No solamente es lento, sino falto de elegancia.

La solución es la programación del lado cliente. La mayoría de las máquinas que corren navegadores Web son poderosas máquinas capaces de hacer amplios trabajos, y con la aproximación original de HTML estático ellas están sentadas ahí, desocupadas esperando por el servidor entregue la siguiente página. La programación del lado cliente significa que el navegador Web está aprovechando para hacer todo el trabajo que puede, y el resultado es una experiencia mucho más rápida e interactiva en su sitio Web.

El problema con las discusiones de la programación del lado cliente es que no son muy diferentes de las discusiones de programación en general. Los parámetros son mayormente los mismos, pero la plataforma es diferente; un navegador Web es como un sistema operativo limitado. En el final, debe aún programar, y esto cuenta para un arsenal embotador de problemas y soluciones producidas por la programación del lado cliente. El resto de esta sección provee una visión de situaciones y aproximación en la programación del lado cliente.

Plug-ins

Uno de los pasos más significativos en el avance de la programación de lado cliente es el desarrollo del plug-in. Esta es una manera en que un programador agrega nueva funcionalidad al navegador para bajar una porción de código que engarza dentro del lugar apropiado en el navegador. El le dirá al navegador "desde ahora puede hacer esta nueva actividad". (Necesita bajar el plug-in una sola vez). Algunos comportamientos rápidos y poderosos son agregados a los navegadores vía los plug-ins, pero escribir uno no es una tarea trivial, no es algo que desee para hacer como parte del proceso de construir un sitio particular. El valor del plug-in para la programación del lado cliente es que permita a un programador experto desarrollar un nuevo lenguaje y agregarlo al navegador sin los permisos del fabricante del navegador. Esto es, los plug-ins proveen una "puerta trasera" que permite la creación de nuevos lenguajes de programación del lado cliente (aunque no todos los lenguajes son implementados como plug-ins).

Lenguajes Scripting

Los plug-ins resultaron en una explosión de lenguajes scripting. Con un lenguaje scripting, usted embebe el código fuente para el programa del lado cliente directamente dentro de su página Web, y el plug-in interpreta el lenguaje que es automáticamente activado mientras una página HTML está siendo mostrada. Los lenguajes scripting tienden a ser razonablemente fáciles para entender y, porque son textos simplemente que están en una página HTML, se cargan rápidamente como parte del único pedido al servidor requerido para procurar la página. La ganancia conseguida es que su código está expuesto para que la vean todos (y la hurten). Generalmente, por otro lado, no hará cosas sofisticadas sorprendentes con los lenguajes scripting, por lo cual no una pérdida tan grande.

Esto apunta a que los lenguajes scripting sean usados dentro de los navegadores Web para intentar resolver tipos específicos de problemas, primariamente la creación de interfaces gráficas de usuario (GUIs) más interactivas y ricas. Sin embargo, un lenguaje scripting debe resolver el 80 por ciento de los problemas encontrados en la programación del lado cliente. Su problema puede muy bien estar completamente dentro de ese 80 por ciento, y ya que los lenguajes de scripting pueden desarrollar rápido y fácilmente, debiera probablemente considerar un lenguaje scripting antes de buscar una solución más avanzada tal como programación Java o ActiveX.

La discusión más común de los lenguajes scripting para navegadores son JavaScript (el cual no tiene nada con Java; es llamado con ese nombre para aprovechar el momento comercial de Java), VBScript (el cual se parece a Visual BASIC) y Tcl/Tk, que proviene del popular lenguaje de construcción de GUI para plataformas cruzadas. Hay otros aparte de estos, y no dude que haya más en desarrollo.

JavaScript es probablemente el más soportado comúnmente. Fue construido para sendos Netscape Navigator y Microsoft Internet Explorer (IE). Desafortunadamente, el condimentado de JavaScript en los dos navegadores puede variar ampliamente (el navegador Mozilla, liberado para bajar desde www.Mozilla.org, soporta el estándar ECMAScript, que puede un día llegar a ser soportado universalmente). Además, hay probablemente más libros de JavaScript disponibles que los que para otros lenguajes de navegadores, y algunas herramientas automáticamente crean páginas usando JavaScript. Sin embargo, si se siente cómodo hoy en Visual BASIC o Tcl/Tk, será más productivo utilizando estos lenguajes scripting en vez de aprender uno nuevo. (Tendrá sus manos llenas para tratar con los usos de la Web de hoy).

Java

Si un lenguaje scripting puede resolver el 80 por ciento de los problemas del lado cliente, ¿Qué sucede con el otro 20 por ciento - "La sección realmente difícil"? Java es la solución popular para esto. No es solamente un lenguaje de programación poderoso construido para ser seguro, plataforma cruzadas, e internacionales, sino que Java está siendo continuamente extendido para proveer características de lenguaje y librerías que elegantemente manejan problemas que son difíciles en los lenguajes de programación tradicional, tales como multihilos, acceso a base de datos, programación de red, y computación distribuida. Java permite la programación del lado cliente vía el *applet* y con *Java Web Start*.

Un applet es un programa mínimo que solamente correrá bajo un navegador Web. El applet es bajado automáticamente como parte de una página Web (como, por ejemplo, un gráfico es bajado). Cuando el applet es activado, ejecuta un programa. Esta es la parte de su maravilla - provee una manera automática de distribuir el software cliente desde el servidor en el momento que el usuario lo necesita al software cliente, y no antes. El usuario obtiene la última versión del software cliente sin fallas ni reinstalaciones complicadas. Por la manera en que Java está diseñado, el programador necesita crear solamente un programa, y este trabaja automáticamente con todas las computadoras que tienen navegadores con intérpretes Java. (Esto seguramente incluye a una amplia mayoría de máquinas). Ya que Java es un lenguaje de programación de vuelo completo, puede hacer tanto trabajo como sea posible en el cliente antes y después de hacer peticiones del servidor. Por ejemplo, no necesita enviar un formulario de petición a través de la Internet para descubrir que tiene un dato o algún parámetro erróneo, y su computadora cliente puede sencillamente hacer el trabajo de armado de los datos en vez de esperar por que el servidor haga un entramado e ingrese la imagen grafica pre-hecha. No solamente obtiene una ganancia de velocidad y

respuesta, sino que el tráfico de la red en general y la carga de los servidores puede ser reducida, previniendo a la Internet entera de un bajón de velocidad.

Una de las ventajas de un applet Java por sobre un programa script es que está en forma compilada, lo cual es el código fuente no está disponible para el cliente. De otra manera, un applet Java puede ser decompilado sin muchos problemas, pero esconder su código no es una cuestión importante. Otros dos factores pueden ser importantes. Como verá más tarde en este libro, un applet Java compilado puede requerir tiempo extra para bajarse, si es muy grande. Un programa script estará integrado dentro de la página Web como parte de su texto (y generalmente es más pequeño y reduce los llamados al servidor). Esto puede ser importante para mejorar la respuesta de su sitio Web. Otro factor es la alta importancia de la curva de aprendizaje. Sin tener encuentra de que usted sea un cabeza dura, Java no es un lenguaje trivial para aprender. Si es usted un programador Visual BASIC, pasarse a VBScript será una solución más rápida (asumiendo que puede restringir a sus consumidores a plataformas Windows), y ya que probablemente resolverá los problemas cliente/servidor más típicos, puede ser duramente presionado a justificar el aprendizaje de Java. Si experimentó con lenguaje scripting ciertamente se beneficiará de observar JavaScript y VBScript antes que meterse con Java, porque ellos pueden completar sus necesidades a mano y ser más productivo en corto tiempo.

.NET and C#

Por un tiempo, el principal competidor de los applet Java fue ActiveX de Microsoft, a pesar de requerir que el cliente esté corriendo Windows. Desde entonces, Microsoft produjo un competidor completo para Java en la forma de la plataforma **.NET** y el lenguaje de programación **C#** . La plataforma **.NET** es aproximadamente la misma que la máquina virtual Java y las librerías Java, y **C#** lleva manifiestamente similaridades con Java. Este es seguramente el mejor trabajo que Microsoft ha hecho en la arena de los lenguajes de programación y los ambientes de programación. De hecho, tienen la considerable ventaja de estar avisados qué trabaja bien y qué no tanto en Java, y construir sobre ello, pero lo construido ellos lo tienen. Esta es la primera vez que desde la aparición de Java tienen una competencia real, y si todo va bien, el resultado será que los diseñadores Java de Sun tomaran una cuidadosa vista de C# y por qué los programadores pueden desear moverse a C#, y responder haciendo mejoras fundamentales en Java.

Actualmente, la principal vulnerabilidad y cuestión importante concerniente a **.NET** es si Microsoft permitirá sea portado *completamente* hacia otras plataformas. Ellos indican que no hay problemas en hacer esto, y el proyecto Mono (www.go-mono.com) tiene una implementación parcial de **.NET** trabajando sobre Linux, pero hasta la implementación esté completa y Microsoft no decida aplastar cualquier parte de él, **.NET** tiene una solución de plataforma cruzada que aún es una apuesta arriesgada.

Para aprender más sobre **.NET** y **C#**, vea *Thinking in C#* de Larry O'Brien y Bruce Eckel, Prentice Hall 2003.

Seguridad

Bajar automáticamente programas y ejecutarlos a través de la Internet puede sonar como el sueño de los constructores de virus. Si cliquea en un sitio Web, puede automáticamente bajar cualquier número de cosas a lo largo de la página HTML: archivos GIF, código script, código Java compilado, y componentes ActiveX. Algunos de estos son benignos: los archivos GIF no pueden dañar nada, y los lenguajes scripting están limitados generalmente en lo que pueden hacer. Java también está diseñado para ejecutar sus applets dentro de un "sandbox" de seguridad, el cual previene de escribir al disco o acceder a memoria fuera del sandbox.

ActiveX de Microsoft es el oponente final en el espectro. Programar con ActiveX es como programar Windows - puede hacer lo que desee. Así si cliquea una página que baja un componente ActiveX, ese componente puede causar daños a los archivos de su disco. De hecho, los programas que carga dentro de su computadora que no están restringidos para ejecutarse dentro de un navegador Web puede hacer la misma cosa. Los virus bajados desde los Bulletin-Board Systems (BBSs) han sido largamente un problema, pero la velocidad de la Internet amplifica la dificultad.

La solución parece ser las "firmas digitales" donde el código está verificado para mostrar el autor de este. Esto está basdo en la idea de que un virus trabaja porque su creador puede ser anónimo, así si saca la anonimidad, los individuos serán forzados a ser responsables de sus acciones. Esto se ve como un buen plan porque permite a los programas ser mucho más funcionales, y sospecho que eliminarán las destrucciones maliciosas. Si, sin embargo, un programa tiene un error de destrucción sin intenciones, aún causará problemas.

La aproximación de Java es prevenir estos problemas de que ocurran, mediante el sandbox. El intérprete de Java que está en su navegador Web local examinada el applet por cualquier instrucción inesperada cuando el applet está siendo cargado. En particular, el applet no puede escribir archivos a disco ni borrarlos (uno de los pilares de los virus). Los applets están considerados generalmente para ser seguros, y ya que esto es esencial para la confianza de los sistemas cliente-servidor, cualquier error en el lenguaje Java que permita a los virus sean rápidamente reparados. (Esto es no tiene sentido que el navegador actualmente mejore estas restricciones de seguridad, y algunos navegadores permiten seleccionar diferentes niveles de seguridad para proveer varios grados de acceso a su sistema).

Puede que sea escéptico de esta restricción algo draconiana en contra de escribir archivos a su disco local. Por ejemplo, puede que desee construir una base de datos local o guardar datos para después usarlos sin conexión. La visión inicial parece ser que todo eventualmente se obtendrá en línea para hacer cualquier cosa importante, pero esto pronto se verá impráctico (aunque el bajo costo de las "aplicaciones Internet" pueden algún día satisfacer las necesidades de un segmento significativo de usuarios). La solución es la "señalización de applets" que usa encriptación de llave pública para verificar

que un applet no haga solamente aquello que indica de donde procede. Un applet señalizado puede aún destruir su disco, pero la teoría es que ya que puede ahora mantener la cantidad de creadores de applet, ellos no hagan cosas maliciosas. Java provee un marco para la firma digital así que eventualmente estará habilitado para permitir a un applet pararse fuera del sandbox si es necesario. El capítulo 14 contiene un ejemplo de cómo señalizar un applet.

Además, Java Web Start es una forma relativamente nueva para distribuir fácilmente programas únicos que no necesiten un navegador web en los cuales correr. Esta tecnología tiene el potencial para resolver muchos problemas del lado cliente asociados con programas que corren dentro de un navegador. Los programas Web Start pueden ya sea estar señalizados, o pueden preguntar al cliente para que permita cada vez que ellos estén haciendo algo potencialmente peligroso en el sistema local. El capítulo 14 tiene un ejemplo simple y explayado sobre Java Web Start.

La firma digital ha errado en una cuestión importante, la cual es la velocidad que la gente se mueve a través de la Internet. Si baja un programa con errores y hace algo inesperado, ¿cuánto tiempo estará antes de descubrir el daño?. Pueden ser días o aún semanas. Para entonces, ¿cómo rastreará hacia atrás lo que el programa ha hecho? ¿Y qué bien hará en este punto?

Internet contra intranet

La web es la solución más general para el problema cliente/servidor, así tiene sentido para utilizar la misma tecnología para resolver un subgrupo de problemas, en particular el clásico problema cliente/servidor *dentro* de una empresa. Con la aproximación tradicional cliente/servidor tiene el problema de diferentes tipos de computadoras cliente, tanto como la dificultad de instalar un nuevo software cliente, ambos son difíciles de solucionar con navegadores Web y programación del lado cliente. Cuando la tecnología Web está usada para una red de información que está restringida a una empresa particular, es referida como una intranet. Las intranets proveen mucha mayor seguridad que la Internet, ya que usted puede controlar físicamente el acceso a los servidores dentro de su compañía. En términos de capacitación, se ve que una persona entiende el concepto general de navegador es mucho más fácil para ella tratar con las diferencias en la forma de páginas y applets, así la curva de aprendizaje de una nueva forma de sistemas se ve reducida.

El problema de seguridad que nos trae para una de las divisiones que veremos ser automáticamente formada en al mundo de la programación del lado cliente. Si su programa está ejecutándose en la Internet, no conoce qué plataforma estará corriendo debajo, y querrá ser extremadamente cuidadoso que no disemine código con errores. Necesita algo de plataforma cruzada y seguridad, como un lenguaje de scripting o Java.

Si está corriendo en una intranet, debe tener un grupo de consideraciones diferentes. No es poco común que sus máquinas puedan ser todas plataformas Intel/Windows. En una intranet, es responsable por la calidad de su propio

código y poder reparar errores cuando sean descubiertos. Además, puede que tenga hoy un cuerpo de código legal que ha sido usado en la aproximación cliente/servidor más tradicional, donde debe instalar físicamente los programas cliente cada vez que hace una actualización. El tiempo gastado en instalar actualizaciones es la razón más importante a moverse a los navegadores, por que las actualizaciones son invisibles y automáticas (Java Web Start es también una solución a este problema). Si está involucrado dentro de una intranet, la aproximación más sensible para tomar el camino más corto que permita usar su código base existente, en vez de intentar recodificar sus programas en un nuevo lenguaje.

Cuando encara con este desconcertante arreglo de soluciones para el problema de programación del lado cliente, el mejor plan de ataque el un análisis de costo-beneficio. Considere las restricciones de su problema y cuál sería el camino más corto para su solución. Ya que la programación del lado cliente es aún programación, es todavía una buena idea tomar la aproximación de desarrollo más rápida para su situación particular. Esta es una postura agresiva para preparar encuentros inevitables con los problemas de desarrollo de programas.

Programación del lado servidor

Esta discusión primordial ha ignorado el caso de la programación del lado servidor. ¿Qué sucede cuando hace una petición al servidor? La mayor de las veces es una petición del estilo "envíame este archivo". Su navegador interpreta entonces el archivo de manera apropiada: como una página HTML, una imagen gráfica, un Applet Java, un programa script, etc. Una petición más compleja generalmente involucra una transacción con base de datos. Un escenario común envuelve una petición para una búsqueda compleja en una base de datos, la cual el servidor da forma luego dentro de una página HTML y le envía como resultado. (De hecho, si el cliente tiene más inteligencia vía lenguaje Java o scripting, el dato en crudo puede ser enviado y formados en el cliente final, lo cual será más rápido y menor carga para el servidor). O puede desear registrar su nombre en una base de datos cuando se une a un grupo o ubicar un orden, lo cual incluye cambios en aquella base de datos. Estas peticiones deben ser procesadas mediante algún código en el lado servidor, el cual es referido normalmente como programación del lado servidor. Tradicionalmente, la programación del lado servidor ha sido realizada utilizando Perl, Python, C++, o algún otro lenguaje, parra crear programas CGI, pero sistemas más sofisticados han ido apareciendo. Estos incluyen servidores Web basados en Java que permiten realizar programación del lado servidor en Java para escribir lo que se denomina servlets. Estos y sus consecutivos, JSPs, son dos de las más importantes razones por las cuales las compañías que desarrollan sitios Web están cambiando a Java, especialmente porque eliminan el problema de tratar con diferentes navegadores (estos tópicos están tratados en *Thinking in Enterprise Java*).

Aplicaciones

Mucho de lo que se habla sobre Java ha sido sobre los applets. Java es actualmente un lenguaje de programación de propósitos generales - al menos en teoría. Y como se apuntó previamente, esta debe ser la manera más efectiva de resolver la mayoría de los problemas cliente/servidor. Cuando se mueve por fuera del terreno del applet (y simultáneamente libera las restricciones, tales como el peliagudo escritura de disco) ingresa el mundo de las aplicaciones de propósito general que corren en solitario, sin un navegador Web, como cualquier programa ordinario hace. Aquí, Java es fuerte no solamente en portabilidad, sino también en programabilidad. Como verá a través de este libro, Java tiene muchas características que permiten crear programas robustos en períodos más cortos que con los lenguajes de programación anteriores.

Estamos conscientes que esto es una bendición mezclada. Paga por la mejoras a través de velocidad de ejecución más lenta (aunque hay un trabajo significativo avanzando en esta área - en particular, la mejora de rendimiento llamada "hotspot" en versiones recientes de Java). Como algunos lenguajes, Java tiene limitaciones de construcción que pueden hacerlo inapropiado para solucionar ciertos tipos de problemas de programación. Java es un lenguaje de rápida evolución, sin embargo, y con cada nuevo lanzamiento viene con más y más atractivos para resolver grandes grupos de problemas.

¿Por qué Java satisface?

La razón que Java haya sido tan satisfactorio es que la meta fue solucionar muchos problemas que los desarrolladores encaran actualmente. Una meta fundamental de Java es incrementar la productividad. Esta productividad viene de muchas maneras, pero el lenguaje está diseñado para ser significativamente mejor que sus antecesores, y proveer importantes beneficios para el programador.

Los sistemas son fáciles de expresar y entender

Las clases diseñadas para completar el problema tienden a expresarlo mejor. Esto significa que cuando escribe código, está describiendo su solución en términos del espacio del problema ("Ponga el grommet en el bin") en vez de los términos de la computadora, el cual es el espacio de solución ("Fije el bit en el chip que significa que el relevado se cerrará"). Trata con conceptos de alto nivel y puede hacer mucho más con una simple línea de código.

El otro beneficio de esta facilidad de expresión es la mantención, el cual (si los reportes pueden ser creídos) es una porción importante del sobre costo en el tiempo de vida del programa. Si los programas son fáciles de entender, son fáciles de mantener. Esto puede también reducir el costo de crear y mantener la documentación.

Influencia máxima con las librerías

La forma más rápida para crear un programa es usar el código que ya está escrito: una librería. Una meta superlativa en Java es hacer una librería de fácil uso. Esto es cumplimentado mediante el modelado de librerías dentro de nuevos tipos de datos (clases), así que el acrecentamiento en una librería significa agregar nuevos tipos al lenguaje. Porque el compilador Java toma cuidadosamente como es utilizada la librería - garantizando la inicialización apropiada y limpieza, y asegurar que los métodos sean llamados apropiadamente - puede focalizarse en qué desea que haga la librería, y no cómo tiene que hacerlo.

Manejo de error

El manejo de errores en C es un problema notorio, y uno que es siempre ignorado - el cruce de dedos está involucrado usualmente. Si está construyendo un grande, complejo problema, no hay nada tan preocupante como tener un error que no se tiene pistas de donde viene. El *manejo de excepciones* es una forma de garantizar que un error sea informado, y que algo suceda como respuesta.

Programando en grande

Muchos programas tradicionales tienen limitaciones en la construcción para tamaños de programa y complejidad. BASIC, por ejemplo, puede ser grandioso par conseguir soluciones rápidas en ciertas clases de problemas, pero si el programa obtiene más que unas pocas páginas de largo, es como intentar nadar en un fluido por demás viscoso. No hay línea clara que indique cuando su programa está fallándole, y si lo hay, usted lo ignora. Usted no dice "Mi programa BASIC está siendo demasiado grande, tendré que rescribirlo en C". En vez de ello, intenta calzar unas pocas líneas más para agregar una nueva característica. Así los costos extras vienen a acrecentarlo.

Java está diseñado para ayudarlo a *programar en grande* - esto es, borrar estos límites de complejidad creciente entre un programa pequeño y uno grande. Seguramente no necesita usar POO cuando escriba un programa del estilo "Hola, mundo", pero las características están allí cuando las necesite. Y el compilador es agresivo para descubrir los errores producidos en programas tanto pequeños como grandes.

¿Java contra C++?

Java se parece muchísimo a C++, así que naturalmente debiera parecer que C++ será remplazada por Java. Pero estoy empezando a discernir de esta lógica. Por un lado, C++ aún tiene características que Java no tiene, y aunque en este punto han sido un montón de promesas sobre que Java algún día será tan rápido o más rápido que C++, estamos viendo algunas mejoras pero sin progresos alarmantes. También, ahí parece estar continuamente interesados en C++, así que no pienso que el lenguaje está yendo al olvido en corto tiempo. Los lenguajes parecen estar siempre ahí.

Estoy empezando a pensar que la fortaleza de Java miente en un terreno ligeramente diferente que C++, el cual es un lenguaje que no intenta llenar un molde. Ciertamente ha sido adaptado en un número de formas para resolver los problemas particulares. Algunas herramientas de C++ combinan librerías, modelos componentes y herramientas de generación de código para solucionar el problema de desarrollar aplicaciones de ventana para usuarios de aplicaciones (para Microsoft Windows). ¿Y aún así, cuál usan la amplia mayoría de los desarrolladores Windows? Microsoft Visual BASIC (VB). Esto desconcierta por el hecho que VB produce el tipo de código que se hace inmanejable cuando el programa es solamente unas pocas páginas de largo (y la sintaxis que puede ser positivamente desconcertante). Tan exitoso y popular como VB es, no hay un buen ejemplo de lenguaje de diseño. Sería hermoso tener la facilidad y poder de VB sin el código resultante inmanejable. Y hay donde pienso que Java deslumbrará: como el "próximo VB" **[8]**. Usted puede o no estremecerse para oír esto, pero piense sobre ello: como mucho de Java es orientado para hacer fácil a los programadores resolver problemas a nivel de aplicación como redes y UI para plataforma cruzada, y aún tiene un lenguaje de diseño que permita la creación de cuerpos muy grandes y flexibles de código. Agregue a esto el hecho de que el chequeo de tipo y manejo de error de Java es una impresionante mejora sobre la mayoría de los lenguajes y tiene la creación de un salto significativo hacia adelante en la productividad de programación.

[8] Microsoft está efectivamente diciendo "no es tan rápido" como C# y .NET. Numerosas personas han planteado la cuestión de si los programadores de VB desean cambiar *cualquier* opción, ya sea Java, C# o aún VB.NET.

Si está desarrollando todo su código desde el principio, entonces la simplicidad de Java sobre C++ significará acortar su tiempo de desarrollo - la evidencia anecdótica (cuentan los grupos de C++ con que yo he hablado quiénes han cambiados a Java) sugiere un doble de velocidad de desarrollo sobre C++. Si el rendimiento de Java no es problema o puede de alguna manera compensarlo, la cuestión de tiempo de mercado hace difícil elegir C++ por sobre Java.

La cuestión más grande es el rendimiento. La interpretación de Java ha sido lenta, cerca de 20 a 50 veces más lento que C en los intérpretes de Java original. Esto ha mejorado grandemente el sobre tiempo (especialmente con las más recientes versiones de Java), pero aún el sobrante es un número importante. Las computadoras están sobre la velocidad; si no es significativamente rápida para hacer algo entonces lo hago con mis manos. (Siempre oí sugerir que inicie con Java, engañe con el tiempo corto de desarrollo, entonces usa una herramienta y librerías de soporte para traducir su código a C++, si necesita velocidad de ejecución rápida).

La clave para hacer Java viable para muchos proyectos de desarrollo es la apariencia de mejoras de velocidad como los compiladores llamados "just-in-time (JIT)", la tecnología propietaria de Sun "hotspot" y aún compiladores de código nativo. De hecho, los compiladores de código nativo eliminarán la

ejecución de plataforma cruzada revendida de los programas compilados, pero también acercarán la velocidad de los ejecutables muy cerca de C y de C++. (En teoría, es recompilar, pero esta promesa ha sido hecha antes para otros lenguajes).

Resumen

Este capítulo intenta darle un sentido a los usos de la programación orientada al objeto y Java, incluyendo por qué POO es diferente, y porque Java en particular es diferente.

La POO y Java puede no ser para todo. es importante evaluar sus propias necesidades y decidir si Java óptimamente satisfará estas necesidades, o si puede ser mejor pasar a otro sistema de programación (incluyendo el que actualmente está usando). Si conoce que sus necesidades estarán muy especializadas para el para el futuro previsible y si tiene restricciones específicas que no pueden ser satisfechas con Java, entonces su deber para sí mismo es investigar alternativas (en particular, recomiendo mirar Python; vea www.Python.org). Incluso si eventualmente elige Java como su lenguaje, al menos entenderá que las opciones están y tendrá una visión clara de porque tomar esa dirección.

Usted conoce a qué se parece un programa procedural: definiciones de datos y llamadas a funciones. Para encontrar significado a tales programas, tiene que trabajar un poco, observar a través de las llamadas a funciones y conceptos de bajo nivel para crear un modelo en su mente. Esta es la razón porque necesitamos representaciones intermedias cuando diseñamos programas procedurales - por sí mismos, estos programas tienden a ser confusos porque los términos de expresión están orientados más hacia la computadora que al problema que está resolviendo.

Porque Java agrega algunos conceptos nuevos arriba de los que encuentra en un lenguaje procedural, su interpretación natural puede ser que el **main( )** en un programa Java será un poco más complicado que para un programa C equivalente. Aquí, estará placenteramente sorprendido: un programa bien escrito en Java es generalmente un tanto más simple y mucho más fácil de entender que el programa equivalente en C. Lo que verá son las definiciones de objetos que representan conceptos en el espacio del problema (en vez de las representaciones de las cuestiones de la computadora) y enviará mensajes a estos objetos para representar las actividades en ese espacio. Uno de los puntales de la programación orientada al objeto es que, con un programa bien diseñado, es fácil entender el código par leerlo. Usualmente, hay un poco menos de código asimismo, porque muchos de sus problemas serán resueltos reutilizando la librería de código existente.

2: Todo es un objeto

Aunque se basa en C + +, Java es más que un lenguaje orientado a objetos.

C + + y Java son idiomas híbridos, pero en Java los diseñadores consideraron que la hibridación no era tan importante como lo era en C + +. Un idioma híbrido consiente estilos múltiples de programación; La razón por la que C + + es híbrido es por la compatibilidad hacia atrás con el lenguaje C. Porque C + + es un actualización del lenguaje C, incluye muchas de características indeseables de ese idioma, lo cual puede hacer algunos aspectos de C + + excesivamente complicados.

El idioma Java da por supuesto que usted quiere sólo programar con programación orientada a objetos. Esto quiere decir que antes de que usted pueda comenzar debe desviar su disposición mental a un mundo orientado a objetos (a menos que esté ya allí). El beneficio de este esfuerzo inicial es la habilidad para programar en un idioma que es más simple para aprender y usar que muchos otros idiomas OOP. En este capítulo veremos que los componentes básicos de un programa Java y aprenderemos que todo en Java es un objeto, dentro de un programa Java.

Usted manipula objetos mediante referencias

Cada lenguaje de programación tiene su propia manera de manipular datos. Algunas veces el programador debe estar constantemente atento a que tipo de manipulación está ocurriendo. ¿Está usted manipulando el objeto directamente, o está usted ocupándose de alguna clase de representación indirecta (un puntero en C o C + +) que debe ser tratada con una sintaxis especial?

Todo esto es simplificado en Java. Usted trata todo como un objeto, usando una sintaxis coherente. Aunque usted *trata* todo como un objeto, el identificador que usted manipula es de hecho una "referencia" a un **object [1]**. Usted podría imaginar esta escena como una televisión (el objeto) con su control remoto (la referencia). Mientras usted sostiene esta referencia, usted tiene una conexión con la televisión, pero cuándo alguien dice "cambia el canal" o "baja el volumen", lo que usted manipula es la referencia, lo cual a su vez modifica el objeto. Si usted quiere ir de arriba abajo por el cuarto y todavía controlar la televisión, usted coge el mando / referencia, pero no la televisión.

[1] Este puede ser un punto de conflicto. Hay quien dice "claramente, es un puntero, "pero esto presupone una implementación subyacente. También, las referencias en Java son mucho más semejantes a las referencias en C + + que a los punteros en su sintaxis. En la primera edición de este libro, preferí

inventar un término nuevo, “handle”, porque las referencias C + + y las referencias Java tienen algunas diferencias importantes. Deje fuera a C + + y no quise confundir a los programadores de C + + que creí que sería la audiencia más importante para Java. En la edición 2, me decidí que “la referencia “era el término más comúnmente usado, y que cualquiera que viniese de C + + entendería mejor la terminología de referencias, y podrían entrar de un salto con ambos pies. Sin embargo, hay personas que disienten aun con el término “referencia.” Leí en un libro donde ponía que “totalmente incorrecto que Java soporte paso por referencia” porque los identificadores del objeto Java (según ese autor) son r*ealmente* “las referencias del objeto.” Y (él sigue) *de hecho* todo es pasado por el valor. Así es que no se pasa por referencia, si no que, “se pasa la referencia a un objeto por valor”.
Uno podría argumentar a favor de la precisión de tales explicaciones complejas, pero pienso que mi acercamiento simplifica la comprensión del concepto sin lastimar a ninguna (pues bien, los puristas del lenguaje pueden afirmar que le miento, pero diré que planteo una abstracción apropiada.)

También, el control remoto puede existir por el mismo, sin televisión. Es decir, una referencia no significa necesariamente que haya un objeto asociado a él. Así que si usted quiere tener una palabra o una sentencia, usted crea una referencia **String:**

```
String s;
```

Pero aquí usted ha creado *sólo* la referencia, no un objeto. Si usted envía un mensaje a **s** en este momento, usted obtendrá un error (durante la ejecución) porque **s** no tiene asociado ningún objeto (no hay televisión). Una práctica más segura, por tanto, es siempre inicializar una referencia cuando usted la crea:

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

Sin embargo, esto usa un rasgo especial de Java: los Strings pueden ser inicializados con un texto entrecomillado. Normalmente, usted debe usar un tipo de inicialización para los objetos más general.

Usted debe crear todos los objetos

Cuando usted crea una referencia, usted quiere asociarla a un objeto nuevo.
Usted hace eso, en general, con la palabra clave **new. New** dice “créame un nuevo objeto de este tipo.” Así en el ejemplo citado anteriormente, usted puede decir:

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

No sólo significa “Créame un nuevo **String**” pero también da información de *cómo* crear el **String** suministrando una cadena de caracteres inicial.

Por supuesto, **String** no es el único tipo que existe. Java viene con una colección de tipos prefabricados. Pero lo más importante es que usted puede crear sus propios tipos. De hecho, esa es la actividad fundamental de la programación en Java, y eso lo que usted aprenderá en el resto de este libro.

Donde el almacenamiento vive

Es útil visualizar algunos aspectos de cómo son las cosas ocurren mientras el programa se está ejecutando, en particular como es aprovechada la memoria. Hay seis lugares diferentes para almacenar datos:

- **Los registros.** Éste es el almacenamiento más rápido porque existe en un lugar distinto al resto de los datos almacenados: Dentro del procesador. Sin embargo el número de registros es bastante limitado, por eso los registros son usados por el compilador según sus necesidades. Usted no tiene control de estos ni en sus programas hay evidencias de que estos registros existan.

- **La pila.** Reside en el área de la memoria RAM (acceso aleatorio a memoria), pero tiene acceso directo desde el procesador mediante el *puntero de pila.* El puntero de pila se decrementa para crear El puntero de pila se decrementa para crear memoria nueva y se incrementa para liberar esa memoria. Ésta es una forma sumamente rápida y eficiente para ubicar datos, sólo inferior a los registros. El compilador de Java debe saber, mientras crea el programa, el tamaño exacto y la duración de vida de todos los datos que son almacenados en la pila, porque debe generar el código para mover de arriba abajo el puntero de pila. Esta restricción limita la flexibilidad de sus programas, así es que mientras haya datos en la pila —en particular, referencias a objetos— los objetos Java no pueden colocarse ellos mismos en la pila.

- **El montículo (Heap).** Éste es área de memoria multiusos (también en el área de RAM) donde todos los objetos Java viven. Lo interesante de esta área de memoria es que, al contrario que la pila, el compilador no necesita saber que cantidad de memoria es necesaria reservar o cuanto tiempo va a permanecer en esta área. Así, hay una gran flexibilidad en el almacenamiento utilizado esta área de memoria (heap). Cada vez que usted necesita crear un objeto, usted simplemente escribe el código para crearlo usando **new**, y es ubicado en el 'heap' cuando ese código es ejecutado. Por supuesto hay un precio que usted paga para esta flexibilidad: es más costoso en cuanto a tiempo el ubicar en este área que hacerlo en la pila (si *puede crear* objetos en la pila en Java, como lo puede hacer en C + +).

- **Almacenamiento estático.** “estático” es usado aquí en el sentido de “una posición fija” (aunque está también en RAM). La memoria estática contiene datos que están disponible durante todo tiempo entero que dure programa en ejecución. Usted puede usar la palabra clave **static** para

especificar que un elemento particular de un objeto es estático, porque en Java por defecto los objetos no se almacenan en memoria estática.

- **Almacenamiento constante.** Los constantes son a menudo creadas directamente en el código de programa, asegurando que nunca puedan cambiar. Algunas veces las constantes son declaradas de forma aislada a fin de que pueden ser opcionalmente ubicadas en memoria de sólo lectura (ROM), en sistemas empotrados.

- **El almacenamiento secundario (fuera de memoria RAM).** Son los datos que se mantienen fuera de un programa en ejecución que deben de ser almacenados fuera de la memoria RAM. Los dos ejemplos principales de esto son *los objetos corrientes (streamed objects),* donde los objetos son convertidos a una *corriente* o *stream* para ser enviados a otra máquina, y *objetos persistentes*, donde los objetos se guardan en disco, de forma que mantienen su estado una vez que el programa ha terminado. Estos objetos pueden ser de nuevo convertidos a objetos ordinarios en memoria. Java provee soporte para la *persistencia ligera* o *lightweight persistence*. Versiones futuras de Java darán soluciones más completas a la persistencia.

Un caso especial: los tipos primitivos

Un grupo de tipos, que usted usará muy a menudo en su programación, tiene un tratamiento especial. Usted puede pensar en estos como tipos “primitivos”. La razón del tratamiento especial es que crear un objeto con **new** — especialmente una variable pequeña, simple — es poco eficiente porque **new** ubica los objetos en el montículo. Para estos tipos Java recurre a la forma de hacerlo en C y C + +. Es decir, en lugar de crear la variable usando **new**, una variable “automática” es creada sin ser *una referencia.* La variable posee el valor, y es colocada en la pila siendo así mucho más eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no cambian de una arquitectura de máquina a otra como ocurre en la mayoría de los lenguajes. Que este tamaño sea fijo es una razón por la que los programas de Java son portables entre distintas arquitecturas.

Tipo Primitivo	Tamaño	Mínimo	Máximo	Tipo Envoltorio
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16}-1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}$—1	**Short**
int	32-bit	-2^{31}	$+2^{31}$—1	**Integer**
long	64-bit	-2^{63}	$+2^{63}$—1	**Long**
float	32-bit	IEEE754	IEEE754	**Float**

double	64-bit	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Todos los tipos numéricos tienen signo, así es que no busque tipos sin signo.

El tamaño del tipo **del boolean** no está explícitamente especificado; está sólo definido para poder tomar los valores literales **true** o **false.**

Las clases "envoltura" (wrappers) de los tipos primitivos de datos permiten que usted ubique un objeto no primitivo en el montículo para representar ese tipo primitivo. Por ejemplo:

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

O usted también podría usar:

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

Las razones para hacer esto serán mostradas en un capítulo posterior.

Los números de alta precisión

Java incluye dos clases para realizar aritmética de alta precisión: **BigInteger** y **BigDecimal.** Aunque estos aproximadamente están dentro de la misma categoría como las clases "envoltura", ninguno tiene similitudes primitivas.

Ambas clases tienen métodos que tienen similitudes con las operaciones que usted realiza con tipos primitivos. Es decir, usted puede hacer las mismas cosas con un **BigInteger** o **BigDecimal** de las que pueden hacer con un **int** o un **float**, pero debe usar llamadas a métodos en vez de operadores.

BigInteger soporta enteros de precisión arbitraria. Esto quiere decir que usted puede representar valores numéricos de cualquier tamaño sin perder precisión durante las operaciones.

BigDecimal es para números de coma flotante de precisión arbitraria; Usted puede usar este tipo, por ejemplo, para cálculos monetarios precisos.

Consulte la documentación JDK para los detalles acerca de los constructores y los métodos que puede usar en estas dos clases.

Arrays en Java

Supuestamente todos los lenguajes de programación soportan arrays. Usar arrays en C y C + + es peligroso porque esos arrays son sólo bloques de memoria. Si un programa accede a un array fuera de su bloque de memoria o

usa la memoria antes de la inicialización (errores comunes de programación) los resultados serán imprevisibles.

Una de las metas principales de Java es la seguridad, para que los quebraderos de cabeza de los programadores en C y C + + no los tengan repetidos en Java. Se garantiza que un array en Java es inicializado y no puede ser accedido fuera de su rango.

El precio a pagar es el usar un poquito de memoria de más en cada array así como también verificar el índice en tiempo de ejecución, pero se asume que el aumento de seguridad y la productividad lo vale.

Cuando usted crea un array de objetos, usted realmente crea un array de referencias, y cada uno de esas referencias son automáticamente inicializadas a un valor especial con su propia palabra clave: **null.** Cuando Java ve **null**, reconoce que la referencia no es un puntero a un objeto.

Usted debe asignar un objeto a cada referencia antes de que usted lo use, y si usted trata de usar una referencia que todavía esta a **null,** el problema será reportado en tiempo de ejecución.

Así, los errores típicos con arrays son prevenidos en Java. Usted también puede crear a un array de tipos primitivos. De nuevo, el compilador garantiza la inicialización porque pone en el cero la memoria para ese array. Los arrays serán estudiados en detalle en capítulos posteriores.

Usted nunca necesita destruir un objeto

En la mayoría de lenguajes de programación, el concepto de la duración de una vida de una variable ocupa una porción significativa del esfuerzo programador. ¿Cuánto tiempo dura la variable? Si se supone que usted debe destruirla, ¿cuándo debería hacerlo? La confusión sobre la vida de las variables puede conducir para un montón de problemas, y esta sección enseña cómo Java simplifica el asunto haciendo todo el trabajo de limpieza por usted.

Ámbito

La mayoría de idiomas procedurales tienen el concepto de *alcance.* Esto determina la visibilidad y la duración de vida de los nombres definidos dentro de ese alcance. En C, C + +, y Java, el alcance es determinado por la colocación de corchetes **{ }.** Por ejemplo:

```
{
  int x = 12;
  // Only x available
  {
```

```
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q “out of scope”
}
```

Una variable definida dentro de un bloque está disponible sólo dentro de ese bloque. Cualquier texto después de '//' hasta el fin de la línea es un comentario.

La sangría simplifica leer código Java. Java es un lenguaje donde los espacios adicionales, etiquetas, y retornos de carro no afectan el programa resultante. Nótese que usted *no puede hacer* lo siguiente, aunque si es correcto en C y C + +:

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

El compilador reportará un error diciendo que la variable **x** ya estaba definida. Así la habilidad de C y C + + para “silenciar” una variable de un bloque superior no es admitida porque los diseñadores de Java pensaron que esto da lugar a programas confusos.

El ámbito de objetos

Los objetos en Java no tienen las mismas duraciones de vida como los primitivos. Cuando usted crea un objeto Java usando **new**, sigue existiendo incluso fuera de su bloque. Así si usted usa:

```
{
  String s = new String("a string");
} // End of scope
```

La referencia **s** deja de existir al final del bloque. Sin embargo, el objeto **String** hacia el que **s** apuntaba, todavía ocupa memoria. En esta porción de código, no hay forma de acceder al objeto porque la única referencia para él no tiene alcance. En capítulos posteriores usted verá cómo la referencia al objeto puede ser reutilizada y duplicada durante el curso de un programa.

El resultado es que los objetos creados con **new** permanecen durante el tiempo que usted quiera, un montón de problemas de la programación en C + + simplemente desaparecen en Java. Los problemas más duros que ocurren en C + + es porque no se obtiene ninguna ayuda del idioma en lo referente a que los objetos estén disponibles cuando son necesarios. Y más importante, en C + + es que usted debe asegurarse de que destruye los objetos cuando ha terminado con ellos.

Eso trae a colación una pregunta interesante. ¿Si Java deja los objetos ocupando memoria, quién limpia esta memoria y detiene su programa? Éste es exactamente el tipo de problema que ocurriría en C + +.

Aquí es donde ocurre un poquito de 'magia'. Java tiene a un *colector de basura (garbage collector)*, el cual busca todos los objetos que se creó con **new** y no serán referenciadas de nuevo. Luego libera la memoria de esos objetos, así esta la memoria puede servir para nuevos objetos. Esto quiere decir que usted nunca necesita preocuparse por limpiar la memoria. Usted simplemente crea objetos, y cuando usted ya no los necesita se eliminarán por ellos mismos. Esto elimina algunos problemas para el programador: "falta de memoria" cuando un programador olvida liberar la memoria que ya no es usada.

Creando tipos nuevos de datos: clases

¿Si todo es un objeto, qué determina a una clase particular de objeto y su comportamiento? De otra forma, ¿qué establece el *tipo* de un objeto? Usted podría esperar que hubiera una palabra reservada "**type**" y eso ciertamente tendría sentido. Históricamente, sin embargo, la mayoría de lenguajes orientados a objetos han usado la **palabra reservada class** para querer decir "estoy a punto de decirle de que tipo de objeto soy" La palabra reservada **class** esta seguida por el nombre del tipo nuevo. Por ejemplo:

```
class ATypeName { /* Class body goes here */ }
```

Esto introduce un tipo nuevo, aunque el cuerpo de clase conste sólo de un comentario (las estrellas y barras serán explicadas más adelante en este capítulo), por tanto no se puede hacer mucho más con él. Sin embargo, usted puede crear un objeto de este tipo usando **new:**

```
ATypeName a = new ATypeName();
```

Pero usted no le puede decir que haga nada más (es decir, usted no le puede enviar ningún mensaje interesante) hasta que usted defina algunos métodos para esta clase.

Atributos y métodos

Cuando usted define una clase (y todo lo que usted hace en Java es definir clasifica, crear objetos de esas clases, y enviar mensajes a esos objetos), usted puede poner dos tipos de elementos en su clase: *Atributos* (algunas veces llamados datos miembro), y *métodos* (algunas veces llamados funciones miembro).

Un atributo es un objeto de cualquier tipo con el que usted puede comunicarse mediante su referencia. También puede ser uno de los tipos primitivos (el cuál no es una referencia). Si es una referencia a un objeto, usted debe inicializarla asociándola a un objeto real (usando **new**, como se vio anteriormente)

mediante un método especial llamado *constructor* (descrito completamente en el capítulo 4).

Si es un tipo primitivo usted lo puede inicializar directamente en el momento de la definición en la clase. (Como verá más tarde, las referencias también pueden ser inicializadas en el momento de la definición.)

Tipo Primitivo	Valor por Defecto
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Estos son los valores predeterminados que Java garantiza cuando la variable es usada *como un miembro de una clase.* Esto asegura que las variables miembro de tipos primitivos siempre serán inicializadas (algo que C + + no hace), reduciendo una fuente de problemas. Sin embargo, este valor inicial puede no ser correcto o el necesario para su programa. Es aconsejable inicializar explícitamente sus variables.

Esto no se garantiza en las variables "locales" — las que no son atributos de una clase. Así, si dentro de una definición de un método usted tiene:

```
int x;
```

X obtendrá algún valor arbitrario (como en C y C + +); y no será automáticamente inicializado a cero. Usted es responsable de asignar un valor apropiado antes de usar **x.** Si se olvida, Java definitivamente mejora a C + +: se genera un error en fase de compilación diciéndole que la variable no ha sido inicializada. (Muchos compiladores C + + le advertirán sobre la falta de inicialización de variables, pero en Java éstos son errores.)

Métodos, argumentos y valores de retorno

En muchos lenguajes (como C y C + +), el término *función* se usa para describir una subrutina. El término que es más comúnmente usado en Java es *método,* como "una forma para hacer algo." Si lo desea, usted puede continuar pensando en términos de funciones. Realmente es sólo una diferencia sintáctica, pero este libro sigue el uso común de Java con el término "método."

Los métodos en Java determinan los mensajes que un objeto puede recibir. En esta sección usted aprenderá lo fácil que es definir un método.

Las partes fundamentales de un método son el nombre, los argumentos, el tipo de retorno, y el cuerpo. Aquí está la forma básica:

```
tipoDeRetorno nombreDelMetodo( /* Lista de argumentos */ ) {
/* Cuerpo del método */
}
```

El tipo de retorno es el tipo del valor que se devuelve por el método después de que usted lo llame. La lista de argumentos da los tipos y los nombres de la información que usted quiere pasar al método. El nombre de método y lista de argumentos conjuntamente identifican inequívocamente al método.

Los métodos en Java pueden ser creados sólo como parte de una clase. Un método puede ser invocado sólo desde un **object [2]**, y ese objeto debe poder realizar la llamada a ese método. Si usted trata de llamar a un método equivocado para un objeto, usted obtendrá un mensaje de error en la fase de compilación. Para llamar a un método desde un objeto se nombra el objeto seguido por un punto, seguido por el nombre del método y su lista de argumentos, de la siguiente forma:

[2] métodos **estáticos**, que se explican más adelante, se pueden invocar desde la *clase*, sin un objeto.

```
NombreDelObjeto.nombreDelMetodo (arg1, arg2, arg3);
```

Por ejemplo, suponga que usted tiene un método **f()** que no tiene argumentos y devuelve un valor de tipo **int** . Entonces, si usted tiene un objeto que se llama **a** que puede invocar el método **f()**, usted puede decir esto:

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

El tipo del valor de regreso debe ser compatible con el tipo de **x.**

Este acto de invocar un método se denomina comúnmente como *enviar un mensaje a un objeto.* En el ejemplo citado anteriormente, el mensaje es **f ()** y el objeto lo es **a.** La programación orientada a objetos está a menudo resumida como simplemente “enviar mensajes *a* objetos.”

La lista de argumentos

La lista de argumentos de un método especifica qué información usted pasa en el método. Como usted podría adivinar, esta información — como todo lo demás en Java — se plasma en objetos. Entonces, lo que usted debe especificar en la lista de argumentos son los tipos de los objetos y su nombre que tiene cada uno. Como en cualquier situación en Java dónde a usted le parece que lo que usa son objetos, usted realmente usa referentes **[3]**. El tipo de la referencia debe ser correcta, sin embargo. Si el argumente debe de ser

supuestamente un **String**, usted debe pasar un **String** o el compilador dará un error.

[3] Con la excepción usual de tipos datos "especiales" **boolean, char**, **byte**, **short**, **int**, **long**, **float,** y **double .** En general, sin embargo, usted pasa objetos, lo que realmente usted pasa son las referencias a los objetos.

Considere un método que tomo a un **String** como argumento. Aquí está la definición, que debe ser colocada dentro de una definición de clase para para que sea compilada:

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

Este método devuelve cuántos bytes son necesarios para almacenar un **String** particular. (Cada **char** en un **String** usa 16 bits, o dos bytes, para representar los caracteres Unicode.) El argumento es de tipo **String** y su nombre es **s**. Una vez **s** es pasada al método, usted lo puede usarla como cualquier otro objeto. (Usted puede enviarle mensajes.) Aquí, el método **length( )**, que es uno de los métodos de **Strings;** devuelve el número de caracteres en un **string**.

Usted también puede ver el uso de la palabra reservada **return**, la cual hace dos cosas.

Primero, que quiere decir "dejar el método, terminar." En segundo lugar, si el método devuelve un valor, ese valor es colocado inmediatamente después de la palabra **return.** En este caso, el valor de retorno se produce evaluando la expresión:

s.length () * 2.

Usted puede devolver el tipo que quiera, sino si usted no quiere devolver nada, usted lo hace eso señalando que el método devuelve **void.** Aquí hay algunos ejemplos:

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

Cuando el tipo de retorno es **null**, la palabra **return** es usada sólo para salir del método, y está por consiguiente innecesaria cuando usted alcanza el fin del método. Usted puede abandonar un método en cualquier momento, pero si usted devuelve un tipo no nulo el compilador le obligará (con mensajes de error) a devolver el tipo apropiado de valor que se espera.

En este punto, puede parecer que un programa es simplemente un montón de objetos con métodos que otros objetos toman como argumentos y envían mensajes a esos otros objetos. Eso es ciertamente lo que pasa, pero en el

siguiente capítulo aprenderá a hacer el trabajo de bajo nivel tomando decisiones dentro de un método. Para este capítulo, los envíos de mensajes son suficientes.

Construyendo un programa Java

Hay varios otros asuntos que usted debe entender antes de escribir su primer programa en Java.

Nombre visibilidad

Un problema en cualquier lenguaje de programación es el control de nombres. Si usted usa un nombre en un módulo del programa, y otros programadores usan el mismo nombre en otro módulo, ¿cómo hace para distinguir un nombre de otro e impedir que los dos nombres “colisionen”?
En C este es uno de los problemas particulares porque un programa sea a menudo un mar inmanejable de nombres. Las clases en C + + (en las que se basa Java) anidan funciones dentro de clases entonces no se pueden llamar con nombres de funciones que anidó dentro otras clases. Sin embargo, C + + todavía consiente datos globales y funciones globales. Para solucionar este problema, C + + introdujo loss *namespaces* usando palabras reservadas adicionales.

Java fue capaz de evitar todo esto haciendo un nuevo refinamiento. Para producir un nombre inequívoco para una biblioteca, el identificador usado no es diferente a un nombre de dominio de Internet. De hecho, los creadores Java quieren que usted use su nombre de dominio de Internet al revés para garantizar la unicidad.

Desde que mi nombre de dominio es **BruceEckel.com**, mi biblioteca de utilidades de fechas pasó a llamarse **com.bruceeckel.utility.fechas**. Después de su nombre de dominio puesto al revés, los puntos son usados para representar subdirectorios.

En Java 1.0 y Java 1.1 las extensiones de dominio **com**, **edu**, **org**, **produzcan**, etc., fueron capitalizado por convención, así es que la biblioteca aparecería:

COM.bruceeckel.utility.foibles.

Tras el desarrollo de Java 2, sin embargo, fue descubierto que esto causó problemas, y por eso ahora el nombre entero del paquete se escribe con letras minúscula.

Este mecanismo hace que todos sus archivos automáticamente pertenezcan a sus **namespaces**, y cada clase dentro de un archivo tiene un identificador único. Así es que usted no necesita aprender el lenguaje especial para solucionar este problema — el lenguaje se encarga de esto por usted.

Usando otros componentes

Cada vez que usted quiere usar una clase predefinida en su programa, el compilador debe saber cómo hallarla. Por supuesto, la clase ya podría existir en el mismo archivo de código fuente desde el que está siendo invocada. En ese caso, usted simplemente usa la clase — aun si la clase está definida posteriormente en el archivo (Java elimine el problema "referencias posteriores" por lo que no se debe preocupar de él).

¿Qué pasa con las clases definidas en otros ficheros? Usted podría pensar que el compilador debería ser lo suficientemente listo para simplemente ir y encontrarlo, pero hay un problema. Imagine que usted quiere usar una clase con un nombre particular, pero hay más de clase definida con ese nombre (probablemente definiciones diferentes). O peor, se imagina que usted escribe un programa, y cuando usted lo construye agrega una nueva clase a su biblioteca que está en conflicto con el nombre de una clase existente.

Para solucionar este problema, usted debe eliminar todas las ambigüedades potenciales.

Esto se resuelve diciendo al compilador Java exactamente qué clases quiere usar con la palabra reservada **import. Import** dice al compilador que importe un paquete, que es una biblioteca de clases. (En otros idiomas, una biblioteca podría constar de funciones y datos así como también clases, pero recuerde que todo código en Java debe estar escrito dentro de una clase.)

La mayoría de las veces usted estará usando componentes de las bibliotecas estándar Java que vienen con su compilador. Con estos, no necesita preocuparse de nombres de dominio puestos al revés; Usted solamente escribe, por ejemplo:

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

Para decir al compilador que usted quiere usar la clase **ArrayList** de Java. Sin embargo, **util** contiene un gran número de clases y usted podría querer usar varias de ellas sin pronunciarlos a todos ellos explícitamente. Esto se facilita usando **‘ * ’** para indicar un comodín:

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

Es más común importar una colección de clases de esta manera que importar clases individualmente.

La palabra reservada **static**

Normalmente, cuando usted crea una clase usted describe cómo serán los objetos de esa clase y cómo se comportarán. Usted realmente no obtiene nada

hasta que crea un objeto de esa clase con **new**, y en ese momento se reserva la memoria para el objeto y los métodos se hacen disponibles.

Pero hay dos situaciones en las cuales este requisito no es suficiente. Uno es si usted quiere tener una reserva de memoria para un dato particular, a pesar de que varios objetos están creados, o si no hay ningún objeto creado. El otro es si usted necesita un método que no esté asociado a ningún objeto particular de esta clase. Es decir, usted necesita un método que usted puede llamar incluso si no hay objetos creados. Usted puede lograr ambos propósitos con la palabra reservada **static.** Cuando usted dice que algo es **static**, quiere decir esa información o el método no está asociado con ninguna instancia particular objeto de esa clase. Por lo tanto si usted nunca ha creado un objeto de esa clase usted puede llamar a un método **static** o puede acceder a datos **static.**

Con datos y métodos no estáticos usted debe crear un objeto y debe usar el objeto para acceder a los datos o el método, los datos y los métodos no estáticos deben conocer el objeto particular al que están asociados. Por supuesto, como los métodos **static** no necesitan que ningún objeto sea creado antes de que sean usados, no pueden acceder *directamente* a los métodos o miembros no estáticos simplemente llamándolos sin referenciar aun objeto creado (los miembros y los métodos **static** deben estar asociados a un objeto particular).

Algunos lenguajes orientados a objetos usan los *datos de clase* de términos y *métodos de clase*, queriendo decir que los datos y métodos existen sólo para la clase como un todo, y no para objetos particulares de la clase. Algunas veces la literatura Java usa estos términos también.

Para hacer un campo o método **static**, usted simplemente pone la palabra clave antes de la definición. Por ejemplo, las siguientes líneas declaran un campo **static** y lo inicializa:

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

Ahora si usted crea dos objetos **StaticTest**, habrá sólo un área de memoria reservada para **StaticTest.i.** Ambos objetos compartirán lo mismo. Considere:

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

En este punto, ambos **st1.i** y **st2.i** tienen el mismo valor 47 y apuntan a la isma zona de memoria. Hay dos formas para referirse a una variable **estática.** Como se indicó anteriormente, la puede referenciar por un objeto, diciendo, por ejemplo, **st2.i.** También puede referirse a eso directamente a través de su nombre de clase, algo que usted no puede hacer con un miembro no estático. (Ésta es la forma preferida para referirse a una variable **estática** ya que enfatiza la naturaleza **estática** de esa variable.)

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

El operador **+ +** incrementa la variable. En este punto, ambos **st1.i** y **st2.i** tendrán el valor 48.Una lógica similar se aplica a los métodos estáticos. Usted puede referirse a un método estático también a través de un objeto como con cualquier otro método, o con la sintaxis especial **ClassName.method ().** Usted define un método estático similarmente:

```
class StaticFun {
static void incr() { StaticTest.i++; }
}
```

Usted puede ver que el método **StaticFun incr ()** incrementa los datos **estáticos** usando el operador **+**. Usted puede llamar a **incr ()** en la forma típica, a través de un objeto:

```
StaticFun sf = new StaticFun();
sf.incr();
```

O, como **incr ()** es un método estático, usted lo puede llamar directamente a través su clase:

```
StaticFun.incr();
```

Mientras **static**, estando aplicada a un campo, definitivamente cambia la forma de que los datos son creados (uno para cada clase vs. un para cada objeto), estando aplicada a un método esto no es tan dramático. Un uso importante de **static** para los métodos es permitir que usted llame ese método sin crear un objeto.

Esto es esencial, como veremos, en definir el método **principal ()** que es el punto de inicio para arrancar una aplicación.

Como cualquier método, un método **estático** puede crear o usar objetos de su tipo, así es que un método **estático** es a menudo usado como un “pastor” para una bandada de instancias de su propio tipo.

Su primer programa Java

Finalmente, aquí está su primer programa completo. Comienza por imprimir un String, y luego la fecha, usando la clase **Date** de la biblioteca estándar 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());
  }
```

```
}
```

En el comienzo de cada programa, debe colocar la declaración **import** para importar las clases adicionales que usted necesitará para el código de ese archivo. El decir extra es porque hay alguna biblioteca de clases que es automáticamente traída en cada archivo Java: **Java.lang.** Abra su navegador de Internet y mire la documentación de Sun. (Si usted no se la ha descargado de *java.sun.com* la documentación Java, hágalo ahora **[4]**). Si usted mira la lista de los paquetes, usted verá todas las bibliotecas de clases que vienen incluidas con Java. Seleccione **java.lang.** Esto mostrará una lista de todas las clases que tiene esa biblioteca. Como **java.lang** está implícitamente incluido en cada archivo de código Java, estas clases están automáticamente disponibles. La clase **Date** no existe en **java.lang**, lo cual quiere decir que debe importar otra biblioteca para usar esa funcionalidad. Si usted no conoce la biblioteca donde se encuentra una clase particular, o si usted quiere ver todas las clases, usted puede seleccionar "Tree" en la documentación Java. Ahora usted puede encontrar cualquier clase que viene con Java. Luego usted puede usar la función "buscar" del navegador para encontrar **Date.** Cuándo la encuentre verá como pertenece a **java.util.Date**, por lo que para usar **Date** se debe importar la librería pertinente con **import java.util.***.

[4] El compilador Java y la documentación de Sun no fué incluida en el CD de este libro porque tiende a cambiar regularmente. Si lo descarga usted mismo, puede obtener la versión mas reciente.

Retroceda al principio, seleccione **java.lang** y luego **System**, verá que la clase **System** tiene varios campos, y si usted indaga usted descubrirá que es un objeto **static PrintStream.** Como es **static** no necesita crear nada. El objeto **out** existe siempre y usted lo puede usar cuando quiera. Lo que usted puede hacer con el objeto **out** es determinado por el tipo que es: **PrintStream.** Convenientemente, **PrintStream** es mostrado en la descripción como un hiperenlace, así que si usted hace clic verá una lista de todos los métodos que usted puede llamar para **PrintStream.** Un buen número de estos estarán descritos más tarde en este libro. Por ahora todo en lo que nos interesa es **println (),** que lo que hace es "imprimir en la consola y acabe con una línea nueva." Así, en cualquier programa Java usted puede escribir **System.out.println ("las cosas");** Cada vez que usted quiere imprimir algo en pantalla.

El nombre de la clase es idéntico al nombre del archivo. Cuando usted crea un programa autónomo como éste, una de las clases en el archivo debe tener el mismo nombre que el archivo. (El compilador se queja si usted no hace esto.) Esta clase debe contener un método que se llame **main ()** con esta apraiencia:

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

La palabra reservada **public** quiere decir que el método está disponible para el mundo exterior (descrito en detalle en el capítulo 5). Los argumentos para **main()** es un array de objetos **String.** Los **args** no serán usados en este

programa, pero el compilador Javanecesita que esten allí porque almacena los parámetros desde la línea de comando.

La línea que escribe la fecha es muy interesante:

```
System.out.println (el nuevo Date ());
```

El argumento es un objeto **Date** que es creado para enviar su valor a **println ().** Tan pronto como esta declaración se ejecute, el objeto **Date** es innecesario, y el colector de basuras puede limpiarlo. No necesitamos preocuparnos por limpiarlo.

Compilando y ejecutando

Para compilar y ejecutar el programa, y todos los de este libro, usted primero debe tener un entorno de programación Java. Hay un gran número de entornos de desarrollo de terceros, pero en este libro que daremos por supuesto que usted usa al JDK de Sun, el cual es gratis. Si usted usa otro sistema de desarrollo **[4]**, necesitará mirar en la documentación de ese sistema cómo compilar y dirigir programas.

[5] El compilador IBM's "**jikes**" es una alternativa común, este es significativamente más rápido que el **javac** de Sun.

Vaya a *java.sun.com.* Allí encontrará información y enlaces que le ayudarán a descargar e instalar el JDK en su ordenador.

Una vez que el JDK es instalado, y usted ha establecido el path de su ordenador para que encuentre **javac** y **java**, descargue y descomprima el código fuente para este libro (lo puede encontrar en el CD-ROM incluido con este libro, o en *www.BruceEckel.com*). Esto creará un subdirectorio por cada capítulo de este libro. Vaya al subdirectorio **c02** y escriba:

```
javac HelloDate.java
```

Esta orden no debería producir respuesta. Si usted obtiene cualquier clase de mensaje de error quiere decir que usted no ha instalado el JDK correctamente y necesita investigar esos problemas. Por otra parte, si usted vuelve a recibir el prompt, escriba:

```
Java HelloDate
```

Y usted obtendrá el mensaje y la fecha como salida.

Este es el proceso que seguirá cada vez que compile y ejecute un programa de este libro. Sin embargo, usted verá que además del código fuente de este libro también tiene un archivo llamado **build.xml** en cada capítulo, que contiene "ant " órdenes para compilar automáticamente los archivos para ese capítulo. Ficheros compilados y **ant** (incluyendo el enlace para descargarlo) están

descritos que más completamente en el capítulo 15, una vez usted tiene instalado **'ant'** (http://jakarta.apache.org/ant) ya puede escribir **'ant'** en la ventana de comandos para compilar y ejecutar los programas en cada capítulo. Si no ha instalado **ant** aún, puede escribir **javac** y **java** las órdenes manuales.

Los comentarios y la documentación embebida

Hay dos tipos de comentarios en Java. La primera es el comentario tradicional estilo C que fue heredado de C++. Estos comentarios comienzan con un **/ *** y continúan, posiblemente a través de muchas líneas, hasta un ***/.** Muchos programadores empiezan cada línea de un comentario continuado con un *****, así es más fácil reconocerlos:

```
/* éste es un comentario
 * que continúa
 * a través de líneas
 */
```

Recuerde, sin embargo, que todo dentro de **/ *** y *****/ será ignorado, por lo que hay diferencia en decir:

```
* éste es un comentario que
Continúa a través de líneas */
```

La segunda forma de comentarios proviene de C + +. Es el comentario de una línea sola, se empieza por // y continúa hasta el fin de la línea. Este carácter de imprenta de comentario es conveniente y comúnmente usado por su simplicidad. Usted no necesita buscar en el teclado /y luego ***** (en lugar de eso, usted presiona la misma tecla dos veces), y no hay necesidad de cerrar el comentario. Así es que usted a menudo verá:

```
// éste es un comentario de un línea
```

La documentación del comentario

Una de las mejores ideas en Java es que escribir código no es la unica actividad importante, documentar el código al menos es igual de importante. Posiblemente el problema más grande con documentar código ha sido mantener esa documentación. Si la documentación y el código están separados, se convierte en una molestia cambiar la documentación cada vez que usted cambia el código. La solución parece simple: Asocie el código a la documentación. La forma más fácil para hacer esto es poner todo en el mismo archivo. Para completar la descripción, sin embargo, usted necesita que una sintaxis especial de comentario para marcar la documentación, y una

herramienta para extraer esos comentarios y almacenarlos de una forma útil. Esto es lo que ha hecho Java.

La herramienta para extraer los comentarios es *javadoc*, y es una parte de la instalación JDK. Usa una parte de la tecnología del compilador Java para buscar etiquetas especiales del comentario que usted pone en sus programas. No sólo extrae la información marcada por estas etiquetas, si no que también extrae el nombre de clase y de los métodos que se anexa al comentario. Así por un incremento mínimo de trabajo puede conseguir una adecuada documentación de sus programas.

La salida de **javadoc** es un archivo de HTML que usted puede ver con su navegador de Internet. Por lo tanto, **javadoc** le permite crear y mantener un único archivo fuente y generar documentación útil. Gracias a **javadoc** tenemos un estándar para crear documentación.

Además, puede escribir sus propios manipuladores del **javadoc**, llamados a *doclets*, si usted quiere realizar operaciones especiales en la información tramitada por **javadoc** (devuelva en un formato diferente, por ejemplo). Doclets son explicados en el capítulo 15.

Lo que sigue es sólo una introducción y visión general de los fundamentos de **javadoc**. Una amplia descripción se puede encontrar en la documentación JDK descargable desde *java.sun.com* (Se trata de una descarga independiente a JDK). Cuando usted descomprima la documentación, mire en los subdirectorios "tooldocs" (o sigue a los enlaces " tooldocs ").

La sintaxis

Todas las órdenes del **javadoc** ocurren sólo dentro de comentarios **/ * *.** Los comentarios acaban con ***** / como siempre. Hay dos formas primarias para usar **javadoc**: Incruste HTML, o uso "doc tags." *Las etiquetas doc* son órdenes que empiezan con '**@**' y son colocadas al principio de la línea del comentario. Las etiquetas doc puede aparecer en cualquier parte de un comentario **javadoc**, siempre empezando con '**@**' pero dentro de llaves.

Hay tres " tipos " de comentarios de documentación, que corresponde al elemento que se va a comentar: clase, variable, o método. Es decir, un comentario de clase aparece antes de la definición de la clase; un comentario de variable aparece justo delante de la definición de una variable, y un comentario de método aparece delante de la definición de un método. Como un ejemplo simple:

```
/** A class comment */
public class docTest {
/** A variable comment */
public int i;
/** A method comment */
public void f() {}
```

```
}
```

Hay que destacar que javadoc sólo procesará documentación de comentario para miembros **públicos** y **protegidos.** Los comentarios para **private** y los miembros de acceso de paquete (vea a capítulo 5) son ignorados y usted obtendrá su salida. (Sin embargo, usted puede usar la opción **-private** para incluir a miembros **private** también.) Esto tien sentido, ya que sólo los miembros **públicos** y **protegidos** están disponible fuera del archivo. Sin embargo, todos los comentarios **de clase** son incluidos en la salida.

La salida para el código citado anteriormente es un archivo de HTML que tiene el mismo formato del estándar como el resto de la documentación Java, así es que los usuarios estarán acostumbrados al formato y fácilmente podrán navegar sus clases. Es digno de entrar en el código citado anteriormente, enviarlo a través de javadoc y mirar el archivo resultante de HTML para ver los resultados.

El HTML embebido

Javadoc pasa órdenes de HTML al documento HTML generado. Esto permite que pueda usar toda la funcionalidad de HTML; Sin embargo, el motivo primario es dejarle formatear código:

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

Usted también puede usar HTML tal como usted lo haría en cualquier otro documento de Web para formatear el texto en sus descripciones:

```
/**
* You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

Notese que dentro de la documentación, los asteriscos a principio de una línea son ignorados, junto con espacios principales. Javadoc *110 Pensar en Java Www.BruceEckel.com* reformatea todo a fin de que se conforme a la apariencia estándar de la documentación. No use encabezamientos HTML como **< h1** > o **< hr** > porque javadoc inserta sus propios encabezamientos y el interferiría con ellos.

Todos los tipos de documentación comentario — clase, variable, y método — pueden soportar HTML embebido.

Algunas etiquetas de ejemplo

Aquí hay alguna de las etiquetas del javadoc disponibles para documentación de código. Antes de tratar de hacer algo serio usando javadoc, usted debería consultar la referencia de javadoc en la documentación de JDK para tener un mayor conocimiento para usar javadoc.

@**see:** referenciando otras clases

Las etiquetas **@see** permiten referenciar la documentación en otras clases. Javadoc generará HTML con las etiquetas **@see** enlacadas a la otra documentación. Las formas son:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Cada uno agrega un hiper-enlace “See Also” a la documentación generada. Javadoc no comprobará que los enlaces que usted lo da hacer seguro son válidos.

{@link *package.class#member label*}

Muy similar para **@see** , excepto que puede ser usada como el texto del hiper-enlace en vez de “ See Also.”

{ @ docRoot }

Produce el camino relativo al directorio raíz de la documentación. Útil para crear hiper-enlaces a páginas del árbol de la documentación.

{ @ inheritDoc }

Hereda la documentación de la clase más próxima de este comentario de doctor de la corriente del intothe de clase.

@version

Así es su uso:

```
@version información de la versión
```

@versión en la cual **la información de versión** es la información más significativa usted escoge incluir. Cuando lo la opción **-version** es colocada en la línea de comando del javadoc, la información de versión será generada para el fichero HTML resultante.

@author

Así es su uso:

```
@author información del autor
```

Lógicamente es el nombre del autor, pero también podría incluir su dirección de correo electrónico o cualquier otra información apropiada. Con la opción **-author** en la línea de comando del javadoc, la información del autor será generada en la salida de la documentación generada de HTML.

Usted puede tener etiquetas múltiples de autor para una lista de autores, pero deben estar consecutivas. Toda la información del autor será tratada de forma conjunta en un párrafo solo en el HTML generado.

@since

Esta etiqueta permite que usted indique la versión de este código que tiene un rasgo particular. Usted lo verá aparecer en la documentación de HTML Java para indicar qué versión del JDK es usada.

@param

Esto sirve para la documentación de los métodos, y su uso es el siguiente:

```
@param parameter-name description
```

Por cada **nombre de parámetro** es el identificador en la lista de parámetros del método, y **la descripción** es del texto que puede continuar en subsiguientes líneas. La descripción es considerada acabada cuando se encuentra una etiqueta nueva de la documentación. Usted puede tener cualquier número de estos, probablemente un para cada parámetro.

@return

Esto sirve para la documentación de los métodos, y su uso es el siguiente:

```
@return description
```

La descripción le da el significado del valor de regreso. Puede continuar en subsiguientes líneas.

@throws

Las excepciones serán explicadas en el capítulo 9, pero brevemente son objetos que pueden ser lanzados desde un método si ese método falla. Aunque sólo un objeto de excepción puede ser lanzado cuando usted llama un método, un método particular podría producir varios de tipos diferentes de excepciones. El uso de la etiqueta de excepción es:

```
@throws fully-qualified-class-name description
```

Da un nombre inequívoco de clase, y **la descripción** (que pueda continuar en subsiguientes líneas) le dice por qué este tipo particular de excepción puede emerger de la llamada de método.

@deprecated

Esto se usa para indicar características que se han mejorado en versiones posteriores. La etiqueta deprecated es una sugerencia para no usar este rasgo particular, y que en el futuro tiene probabilidad de estar borrado. Un método que es marcado con **@deprecated** hace que el compilador de una advertencia si es usado.

Ejemplo de documentación

Aquí está el primer programa Java otra vez, esta vez con comentarios añadidos:

```
//: c02:HelloDate.java
import java.util.*;
/** The first Thinking in Java example program.
* Displays a string and today's date.
* @author Bruce Eckel
* @author www.BruceEckel.com
* @version 2.0
*/

public class HelloDate {
     /** Sole entry point to class & application
     * @param args array of string arguments
     * @return No return value
     * @exception exceptions No exceptions thrown
     */

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

La primera línea del archivo usa la técnica de escribir ‘ **//:**’ Como un indicador especial para la línea del comentario conteniendo el nombre del archivo fuente. Esa la línea contiene la información de la ruta del archivo (en este caso, **c02** indica a capítulo 2) seguido del nombre del archivo **[6]**. La última línea también termina con un comentario, y éste (‘ **///:**~’) indica el fin del código fuente, lo cual permite que automáticamente se actualice en el texto de este libro después de que ser ejecutado con un compilador y ser ejecutado.

[6] Originalmente, creé que una herramienta usando Python (vea a www.Python.org) usa esta información para extraer los archivos de código, ponerlos en subdirectorios apropiados, y crear makefiles. En esta edición, todos los archivos se guardan en CVS y automáticamente incorporados en este libro usando un macro VBA (Visual Basic for Applications). Esta nueva mejora parece ser mucho mejor en términos del mantenimiento de código, en su mayor parte por CVS.

Codificando con estilo

El estilo descrito en el *'Code Conventions for the Java Programming Language* **[7]** *'* es capitalizar la primera letra de un nombre de clase. Si el nombre de clase consta de varias palabras, se escriben de forma concatenada (es decir, no se usan guiones para separar los nombres), y la primera letra de cada palabra es capitalizada, de la siguiente forma:

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

Esto se denomina "escritura de camello." Para casi todo lo demás: métodos, campos (variables del miembro), y nombres de referencias a objetos, el estilo aceptado es como para las clases *excepto que* la primera letra del identificador es letra minúscula. Por ejemplo:

```
class AllTheColorsOfTheRainbow {
     int anIntegerRepresentingColors;

     void changeTheHueOfTheColor(int newHue) {
          // ...
     }

     // ...
}
```

Por supuesto, usted debería recordar que el usuario también debe escribir todo estos nombres largos, y por tanto hay que ser compasivos. El código Java que usted verá en las bibliotecas de Sun también sigue la colocación de llaves como se hace en este libro.

[7] Http://java.sun.com/docs/codeconv/index.html. Por el espacio físico en este libro como en diapositivas de cursos es imposible seguir todas las directivas que se exponen.

Resumen

La meta de este capítulo es aprender los mínimos conceptos básicos de Java cómo escribir un programa simple. Usted también ha recibido una visión general del lenguaje y algunas de sus ideas básicas. Sin embargo, todos los ejemplos hasta ahora han sido de la clase "haga esto, luego haga eso, luego si

no haga algo." Qué ocurre si usted quiere que el programa haga elecciones, tales como "si el resultado de hacer esto es rojo, haga eso; En caso de que no, entonces hace otra cosa". Los conocimientos de Java para esta actividad fundamental de programación se estudia en el siguiente capítulo.

Ejercicios

1. Siguiendo el ejemplo **HelloDate.java** de este capítulo, crear un programa "hola, mundo" que simplemente imprima esa frase. Solo necesitamos un método en nuestra clase ("**main**", que se ejecuta cuando empieza el programa). Recordemos hacerlo **static** e incluir la lista de argumentos, aunque no la utilicemos. Compilar el programa con **javac** y ejecutarlo con java. Si **alguién** está utilizando un entorno de desarrollo distinto del JDK, deberá aprender a compilar y ejecutar en él.

2. Encontrar el fragmento de código donde aparece **UnNombredeTipo** y convertirlo en un programa Java que se compile y ejecute.

3. Convertir el fragmento de código de **SoloDatos** en un programa que se compile y ejecute.

4. Modificar el ejercicio 3 de modo que los valores de las variables en **SoloDatos** se asignen e impriman en el **main**()

5. Escribir un programa que incluya y utilice el método **storage**() que definimos en este capítulo

6. Convertir el código de **StaticFun** en un programa funcional.

7. Escribir un programa que imprima tres argumentos tomados de la línea de comandos. Para esto, necesitaremos indicar en la línea de comandos un array de **strings**.

8. Convertir el ejemplo de **TodosLosColoresDelArcoIris** en un programa que se compile y ejecute.

9. Buscar el código de la segunda versión de **HelloDate.java** , que es un ejemplo sencillo de los comentarios de documentación. Ejecutar **javadoc** sobre ese fichero y ver los resultados con el navegador Web.

10. Convertir **docTest** en un fichero que se compile y pasarlo por **javadoc** . Verificar la documentación resultante con el navegador Web.

11. Añadir una lista de items HTML a la documentación en el ejercicio 10.

12. Coger el programa del ejercicio 1 y añadirle comentarios de documentación. Extraer esos comentarios a un fichero HTML utilizando **javadoc** y verlos en el navegador web.

3: Controlando el flujo del programa

Como una criatura consciente, un programa debe manipular su mundo y hacer cambios en el transcurso de su ejecución.

En Java manipulamos objetos y datos mediante el uso de operadores, y elegimos alternativas con las sentencias de control de ejecución. Java es heredado de C++, por lo que muchas de sus sentencias y operadores serán familiares a programadores de C y C++. Java ha añadido además algunas mejoras y simplificaciones.

Utilizando los operadores de Java

Un operador recibe uno o más argumentos y produce un nuevo valor. Los argumentos tienen un formato diferente que las habituales llamadas a métodos, pero el efecto es el mismo. Deberías estar razonablemente cómodo con el concepto general de operador gracias a la experiencia previa programando. Suma (+), resta y signo negativo (-),multiplicación (*), división (/) y asignación (=), todos funciona más o menos igual en cualquier lenguaje de programación.

Todos los operadores producen un valor de sus operandos. Adicionalmente un operador puede cambiar el valor de un operado. Esto se conoce como efecto lateral. El uso más común de los operadores que modifican sus operandos es generar el efecto lateral, pero debemos recordar que el valor generado estará disponible para su uso de la misma forma que en el caso de los operadores sin efecto lateral.

La mayoría de operadores trabajan solamente con tipos primitivos. Las excepciones son ' **=** ', ' **==** ' y ' **!=** ', que trabajan con todos los tipos de objetos (y son una fuente de confusiones en los objetos). Adicionalmente la clase **String** soporta ' **+** ' y ' **+=** '.

Precedencia

La precedencia de operadores define como se evalúa una expresión cuando contiene más de un operador.

Java tiene reglas especificas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división se evalúan antes que la suma y la resta. A menudo los programadores olvidan el resto de reglas de precedencia, por lo es recomendable usar paréntesis para especificar el orden de evaluación. Por

ejemplo:

```
A = X + Y - 2/2 + Z;
```

tiene un significado muy diferente que la misma sentencia agrupada con paréntesis:

```
A = X + (Y - 2)/(2 + Z);
```

Asignación

La asignación se representa con el operador =. Significa "coger el valor de la parte derecha (a menudo llamado *rvalue*) y copiarlo en la parte izquierda (a menudo llamado *lvalue*). Un rvalue es cualquier constante, variable o expresión que produce un valor, pero el lvalue debe ser una variable. (Esto es, tiene que haber un espacio físico donde almacenar el valor.) Por ejemplo, podemos asignar un valor constante a una variable (**A = 4;**), pero no podemos asignar nada a un valor constante - las constantes no pueden ser lvalue. (No podemos decir **4 = A;.**)

La asignación de tipos primitivos es bastante integra. Como los tipos primitivos contienen el valor actual y no una referencia a un objeto, cuando asignamos tipos primitivos copiamos el contenido de un sitio a otro. Por ejemplo, si ponemos **A = B** cuyos tipos son primitivos, entonces el contenido de **B** se copia en **A** . Si ahora modificamos **A** , obviamente **B** no se verá afectado por esta modificación.

Esto es lo que tenemos que esperar como programadores en la mayoría de situaciones. En cambio cuando asignamos objetos, las cosas cambian. Siempre que manipulamos un objeto lo que estamos manipulando es la referencia al objeto, por lo que cuando asignamos "de un objeto a otro" lo que estamos haciendo es copiar una referencia de un sitio a otro. Esto significa que si decimos **C = D** siendo ambos objetos, acabamos teniendo a **C** y **D** referenciando al objeto que originalmente solo referenciaba **D** . El siguiente ejemplo lo demuestra.

Lo primero que vemos como a parte es una sentencia **package** en **package c03** , que indica el Capítulo 3 de este libro. El primer listado de código de cada capítulo incluirá una sentencia de paquete como esta para establecer el número de capítulo del listado de código en ese capítulo. Como resultado veremos, en el Capítulo 17, que todos los listados del capítulo 3 (excepto aquellos que tengan un nombre de paquete diferente) se situarán automáticamente en un subdirectorio llamado **c03** , los listados del Capítulo 4 estarán en **c04** y así sucesivamente. Todo esto se hará a través del programa **CodePackager.java** visto en el Capítulo 17, y en el

Capítulo 5 se explicará de forma completa el concepto de paquetes. Lo que tenemos que saber en este punto es que, para este libro, las líneas de código del tipo **package c03** se utilizan para establecer el subdirectorio del capítulo para los listados que haya en el mismo.

Para ejecutar el programa debemos asegurarnos que el classpath contiene el directorio raíz donde se instaló el código de este libro. (A partir de este directorio verás los subdirectorios **c02** , **c03** , **c04** , etc.)

Para versiones futuras de Java (1.1.4 y en adelante), cuando el **main( )** está dentro de un fichero con una sentencia **package** , para ejecutar un programa deberemos especificar el nombre completo del paquete antes del nombre del programa. En este caso la línea de comandos es:

```
java c03.Assignment
```

Deberemos recordar esto siempre que ejecutemos una programa que está en un **package** .

Aquí está el ejemplo:

```
//: Assignment.java
// La asignación con objetos es un poco falsa.
package c03;
class Number {
	int i;
}
public class Assignment {
	public static void main(String[] args) {
		Number n1 = new Number();
		Number n2 = new Number();
		n1.i = 9;
		n2.i = 47;
		System.out.println("1: n1.i: " + n1.i +
		", n2.i: " + n2.i);
		n1 = n2;
		System.out.println("2: n1.i: " + n1.i +
		", n2.i: " + n2.i);
		n1.i = 27;
		System.out.println("3: n1.i: " + n1.i +
		", n2.i: " + n2.i);
	}
} ///:~
```

La clase **Number** es sencilla, en el **main( )** creamos dos instancias de esta clase (**n1** y **n2**).Al valor i de cada **Number** se le da un valor diferente, entonces asignamos **n2** a **n1** , y cambiamos **n1** . En muchos lenguajes de programación creeríamos que **n1** y **n2** son independientes todo el tiempo, pero como lo que hemos asignado es una referencia esta es la salida que veremos:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

¡Cambiando el objeto **n1** parece que el objeto **n2** también cambia! Esto es porque tanto **n1** como **n2** contienen la misma referencia, que está apuntando al mismo objeto. (La referencia que originalmente contenía **n1** y que apuntaba al objeto que contenía el valor 9 ha sido sobrescrita durante la asignación y se ha perdido; este objeto será eliminado por el recolector de basura.)

Frecuentemente se denomina a este fenómeno *alias* , y es esta es fundamentalmente la forma de trabajar de Java con objetos. ¿ Pero y si en este caso no queremos que aparezcan alias?. Podemos adelantarnos a la asignación y decir:

```
n1.i = n2.i;
```

Esto conserva los dos objetos por separado en vez de tirar uno y apuntar **n1** y **n2** al mismo objeto, pero pronto comprenderemos que manipulando los campos de los objetos es desordenado y va en contra de los principios del buen diseño orientado a objetos. Esto no es un tópico trivial, por lo que lo dejaremos para el Capítulo 12, que está dedicado a los alias. En lo sucesivo tendremos en cuenta que la asignación de objetos puede traer sorpresas.

Alias en las llamadas a métodos

Los alias también pueden aparecer cuando pasamos un objeto en un método:

```
//: PassObject.java
// Pasar objetos a métodos puede ser un poco falsa.
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';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
    }
} ///:~
```

En muchos lenguajes de programación puede parecer que el método **f( )** está haciendo una copia del argumento **Letter** y dentro del alcance del método. Pero una vez más se ha pasado una referencia por lo que la línea

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

está actualmente cambiando el objeto fuera de **f( )** . La salida muestra lo siguiente:

```
1: x.c: a
2: x.c: z
```

Los alias y su solución es un tema complejo y, aunque deberemos esperar hasta el Capítulo 12 para todas las respuestas, en este punto debemos tener cuidado con ello para así poder vigilar el peligro.

Operadores matemáticos

Los operadores matemáticos básicos son los mismos que están disponibles en la mayoría de lenguajes de programación: suma (+), resta (-), división (/), multiplicación (*) y modulo (%, produce el resto de una división de enteros). La división de enteros trunca más que redondea el resultado.

Java también utiliza una notación abreviada para permitir una operación y una asignación al mismo tiempo. Esto se indica con un operador seguido del signo de igual, y esto es consistente con todos los operadores del lenguaje (siempre que tenga sentido). Por ejemplo, para añadir 4 a la variable **x** y asignar el resultado a **x** , utilizaremos **x += 4;**

Este ejemplo muestra como utilizar los operadores matemáticos:

```
//: MathOps.java
// Muestra los operadores matemáticos
import java.util.*;
public class MathOps {
	// Crea una abreviatura para evitar el teclear:
	static void prt(String s) {
		System.out.println(s);
	}
	// abreviatura para escribir un string y un entero:
	static void pInt(String s, int i) {
		prt(s + " = " + i);
	}
	// abreviatura para escribir un string y un float:
	static void pFlt(String s, float f) {
		prt(s + " = " + f);
	}
	public static void main(String[] args) {
		// Crea un generador de números aleatorios,
		// por defecto a partir de la hora actual:
		Random rand = new Random();
		int i, j, k;
		// '%' limita el valor máximo a 99:
		j = rand.nextInt() % 100;
		k = rand.nextInt() % 100;
		pInt("j",j); pInt("k",k);
		i = j + k; pInt("j + k", i);
		i = j - k; pInt("j - k", i);
		i = k / j; pInt("k / j", i);
		i = k * j; pInt("k * j", i);
		i = k % j; pInt("k % j", i);
		j %= k; pInt("j %= k", j);
		// Comprueba números de coma flotante:
		float u,v,w; // se aplica también a dobles
		v = rand.nextFloat();
		w = rand.nextFloat();
		pFlt("v", v); pFlt("w", w);
		u = v + w; pFlt("v + w", u);
		u = v - w; pFlt("v - w", u);
		u = v * w; pFlt("v * w", u);
		u = v / w; pFlt("v / w", u);
		// lo siguiente también funciona con
		// char, byte, short, int, long,
		// y double:
		u += v; pFlt("u += v", u);
		u -= v; pFlt("u -= v", u);
		u *= v; pFlt("u *= v", u);
		u /= v; pFlt("u /= v", u);
	}
} ///:~
```

Lo primero que vemos son una serie de métodos abreviados para la escritura: el método **prt( )** escribe un **String** , el **pInt( )** escribe un **String** seguido de un **entero** y el **pFlt( )** escribe un **String** seguido de un **float** . Por supuesto, todos ellos finalmente acaban utilizando **System.out.println( )** .

Para genera números, el programa primero crea un objeto **Random** . Como no se pasan argumentos en la creación, Java utiliza la hora actual como base para el generador de números aleatorios. El programa genera una serie de números aleatorios de diferente tipos con el objeto **Random** simplemente llamando a diferentes métodos: **nextInt( )** , **nextLong( )** , **nextFloat( )** o **nextDouble( )** .

El operador módulo, cuando se utiliza con el resultado de un generador de números aleatorios, limita el resultado a un limite superior del operando menos uno (99 en este caso).

Operadores de signo positivo y negativo

El signo negativo (-) y el signo positivo (+) son los mismos operadores que para la suma y la resta.

El compilador se imagina que uso se le quiere dar por la forma en que escribimos la expresión. Por ejemplo, la sentencia

```
x = -a;
```

tiene un significado obvio. El compilador es capaz de imaginar:

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

pero el lector puede confundirse, por lo que es más claro decir:

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

El signo negativo produce el negativo del valor. El signo positivo proporciona el simétrico del signo negativo, aunque no es que no hace mucho.

Auto incremento y decremento

Java, como C, está lleno de abreviaturas. Las abreviaturas pueden facilitar mucho

la escritura de código y también facilitar o dificultar su lectura.

Dos de las mejores abreviaturas son los operadores de incremento y decremento (a menudo se refiere a ellos como operadores de auto-incremento y auto-decremento). El operador de decremento es -- y significa "decrementar una unidad." El operador de incremento es ++ y significa "incrementar una unidad." Si **A** es un **entero** , por ejemplo, la expresión **++A** es equivalente a **(A = A + 1)** . Los operadores de incremento y decremento producen como resultado el valor de la variable.

Hay dos versiones de cada uno de los operadores, habitualmente llamadas versiones prefijo y sufijo. El pre-incremento significa que el operador **++** aparece antes de la variable o expresión, y el post-incremento significa que el operador **++** aparece después de la variable o expresión. De forma similar, el pre-decremento significa que el operador **--** aparece antes de la variable o expresión, y el post-decremento significa que el operador **--** aparece después de la variable o expresión. En el pre-incremento o pre- decremento, (pe., **++A** o **--A**), se realiza la operación y luego se produce el valor. En el post-incremento o post-decremento (pe., **A++** o **A--**), el valor se produce y luego se realiza la operación. Como ejemplo:

```
//: AutoInc.java
// Muestra los operadores ++ y --
public class AutoInc {
	public static void main(String[] args) {
		int i = 1;
		prt("i : " + i);
		prt("++i : " + ++i); // Pre-incremento
		prt("i++ : " + i++); // Post-incremento
		prt("i : " + i);
		prt("--i : " + --i); // Pre-decremento
		prt("i-- : " + i--); // Post-decremento
		prt("i : " + i);
	}
	static void prt(String s) {
		System.out.println(s);
	}
} ///:~
```

La salida de este programa es:

```
i : 1
++i : 2
i++ : 2
i : 3
```

```
--i : 2
i-- : 2
i : 1
```

Podemos ver que para el prefijo obtenemos el valor después de que se ha realizado la operación, pero con la forma de sufijo obtenemos el valor antes de que se haya realizado la operación. Estos son los únicos operadores que tienen efectos laterales (distintos que los envueltos en la asignación). (Esto es, no solamente utilizan el valor del operando, sino que lo cambian.)

El operador de incremento es una explicación del nombre C++, implicando "un paso más allá de C." En un discurso inicial de Java, Bill Joy (uno de los creadores), dijo que "Java=C++--" (C más más menos menos), sugiriendo que Java es C++ sin las partes difíciles innecesarias quitadas y por tanto un lenguaje mucho más simple. Según progresemos en este libro veremos que varias partes son más simples, y aún así Java no es ese mucho más fácil que C++.

Operadores relacionales

Los operadores relacionales generan un resultado **boolean** . Evalúan la relación entre los valores y los operandos. Una expresión relacional produce **true** si la relación es verdadera y **false** si la relación es falsa. Los operadores relacionales son menor que (<), menor o igual que (<=), mayor que (>), mayor o igual que (>=), igualdad (==) y desigualdad (!=). La igualdad y la desigualdad funcionan con todos los tipos de datos predefinidos, pero las otras comparaciones no funcionarán con el tipo **boolean** .

Comprobando la equivalencia de un objeto

Los operadores relacionales **==** y **!=** también funcionan con todos los objeto, pero a menudo su significado puede confundir a los programadores principiantes de Java. Aquí hay un ejemplo:

```
//: 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);
	}
} ///:~
```

La expresión **System.out.println(n1 == n2)** mostrará el resultado **lógico** de la comparación. Seguramente la salida será **cierto** y luego **falso** , ya que los dos

objetos **Integer** son el mismo. Pero mientras el contenido es el mismo, las referencias no lo son y los operadores **==** y != comparan referencias a objetos. Luego la salida es actualmente **falso** y luego **cierto** . Naturalmente, a lo primero esto sorprende a la gente.

¿Y si lo que queremos comparar es la igualdad de los contenidos de un objeto? Deberemos usar el método especial **equals( )** que existe para todos los objetos (no para los tipos primitivos que trabajan bien con **==** y **!=**). Aquí está como se utiliza:

```
//: 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));
	}
} ///:~
```

El resultado será **cierto** , tal y como esperábamos. Ah, pero no es tan simple como esto. Si creamos nuestra propia clase, como esta:

```
//: EqualsMethod2.java
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));
	}
} ///:~
```

volvemos a la primera situación: el resultado es **falso** . Esto es porque el comportamiento por defecto de **equals()** es comparar las referencias. Por lo que a menos que *sobre escribamos* **equals()** en nuestra nueva clase no tendremos el comportamiento deseado. Desafortunadamente, no aprenderemos sobre como sobre escribir hasta el Capítulo 7, pero en este momento ser advertidos del comportamiento de **equals()** nos puede evitar algunos fracasos.

La mayoría de las clases de las librerías de Java implementan **equals()** de forma

que compare el contenido de los objetos en vez de sus referencias.

Operadores lógicos

Los operadores lógicos AND (**&&**), OR (**||**) y NOT (**!**) producen el valor **lógico** **true** o **false** basándose en la relación lógica entre sus argumentos. Este ejemplo utiliza operadores relacionales y lógicos:

```
//: Bool.java
// Operadores relacionales y lógicos
import java.util.*;
public class Bool {
	public static void main(String[] args) {
		Random rand = new Random();
		int i = rand.nextInt() % 100;
		int j = rand.nextInt() % 100;
		prt("i = " + i);
		prt("j = " + j);
		prt("i > j is " + (i > j));
		prt("i < j is " + (i < j));
		prt("i >= j is " + (i >= j));
		prt("i <= j is " + (i <= j));
		prt("i == j is " + (i == j));
		prt("i != j is " + (i != j));
		// Tratar un entero como si fuera un valor lógico
		// no está permitido en Java
		//! prt("i && j is " + (i && j));
		//! prt("i || j is " + (i || j));
		//! prt("!i is " + !i);
		prt("(i < 10) && (j < 10) is "
		+ ((i < 10) && (j < 10)) );
		prt("(i < 10) || (j < 10) is "
		+ ((i < 10) || (j < 10)) );
	}
	static void prt(String s) {
		System.out.println(s);
	}
} ///:~
```

Podemos aplicar AND, OR, o NOT solamente a valores **lógicos** . En una expresión no podemos utilizar un valor **no lógico** como si fuera **lógico** , como hacíamos en C y C++. Podemos ver los intentos fallidos quitando los comentarios marcados con el marcador //!. Las siguientes expresiones, en cambio, producen valores **lógicos** utilizando comparaciones relacionales, y luego utilizando operaciones lógicas con los resultados.

Un listado de la salida sería:

```
i = 85
j = 4
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 true
```

Fijémonos en que un valor **lógico** se convierte automáticamente al formato texto apropiado si se utiliza donde se espera un **String** .

Podemos cambiar la definición de **entero** en el programa a continuación con cualquier otro tipo de dato primitivo excepto **boolean** . De cualquier manera hay que tener cuidado con los números de coma flotante ya que su comparación es muy estricta. Un número que es la mínima fracción diferente de otro número es aún "no igual." Un número que es el bit más pequeño después del cero sigue sin ser un cero.

Cortocircuitados

Tratar con operadores lógicos con puede llevar a un fenómeno denominado "cortocircuitado." Esto significa que la expresión se evaluará solamente hasta que la falsedad o veracidad de la misma se pueda determinar de forma no ambigua. Como resultado de esto, pueden no evaluarse todas las partes de una expresión lógica. Aquí hay un ejemplo que muestra el cortocircuito:

```
 ShortCircuit.java
// Demuestra el comportamiento del cortocircuito
// con operadores lógicos.
public class ShortCircuit {
	static boolean test1(int val) {
		System.out.println("test1(" + val + ")");
		System.out.println("result: " + (val < 1));
		return val < 1;
	}
	static boolean test2(int val) {
		System.out.println("test2(" + val + ")");
		System.out.println("result: " + (val < 2));
		return val < 2;
	}
```

```
	static boolean test3(int val) {
		System.out.println("test3(" + val + ")");
		System.out.println("result: " + (val < 3));
		return val < 3;
	}
	public static void main(String[] args) {
		if(test1(0) && test2(2) && test3(2))
			System.out.println("expression is true");
		else
			System.out.println("expression is false");
	}
} ///:~
```

Cada comprobación realiza una comparación entre los argumentos y devuelve cierto o falso. También escribe la información para que podamos ver que se le está llamando. Las comprobaciones se realizan en la expresión:

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

Naturalmente podemos pensar que se ejecutan las tres comprobaciones, pero en cambio la salida nos muestra:

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

La primera comprobación produce un resultado **cierto** , luego la evaluación de la expresión continua. En cambio la segunda comprobación produce un resultado **falso** . Como esto significa que toda la expresión tiene que ser **falso** , ¿por qué continuar evaluando el resto de la expresión? Esto podría ser caro. La razón del cortocircuito, es precisamente esta; podemos tener un incremento potencial del procesamiento si todas las partes de una expresión lógica no necesitan ser evaluadas.

Operadores de bits

Los operadores de bits permiten la manipulación de bits individualmente dentro de un tipo de datos primitivo. Los operadores de bits ejecutan algebra booleana obre los bits correspondientes a sus dos argumentos para producir un resultado.

Los operadores de bits proviene de la orientación de bajo nivel de C, cuando se

manipulaba directamente el *hardware* y se tenían que modificar los bits en los registros *hardware* . Java se diseñó originalmente para programar aparatos de TV por lo que esta orientacó a bajo nivel sigue teniendo sentido. Sin embargo, lo más probable es que no utilice los operadores de bits muy a menudo.

El operador de bits AND (&) produce un uno en el bit de salida si los dos bits de entrada son uno; si no, produce un cero. El operador de bits OR (|) produce un uno en el bit de salida si uno de los bits de entrada es un uno y produce un cero solamente si ambos bits de entrada son cero. El operador OR EXCLUSIVO o XOR (^) produce un uno si uno de los dos bits de entrada es un uno pero no ambos. El operador de bits NOT (~, también llamado operador complemento) es un operador unario, que tiene solamente un argumento. (Todos los demás operadores son operadores binarios). El operador NOT produce el opuesto del bit de entrada - un uno si el bit de entrada es cero, un cero si el bit de entrada es uno.

Los operadores de bits y los operadores lógicos utilizan los mismos caracteres, por lo que resulta útil tener un macanismo mnemónico para ayudar a recordar los significados: como los bits son "pequeños", los operadores de bits solamente tienen un caracter.

Los operadores de bits pueden combinarse con el signo = para unir la operación y la asignación: &=, |= y ^= son expresiones legítimas. (Debido a que ~ es un operador unario, no puede combinarse con el signo =).

El tipo **booleano** se trata como un valor de un bit por lo que es algo diferente. Se pueden realizar operaciones AND, OR y XOR, pero no se puede usar el operador de bits NOT (presumiblemente para evitar la confusión con el NOT lógico). Para los **booleanos** , las operaciones de bits tienen el mismo efecto que los operadores lógicos excepto que en estos no hay "corto-circuito". Además, las operaciones de bits sobre **booleanos** incluye un operador lógico XOR que no está incluido en la lista de operadores "lógicos". Estas avisado del uso de **booleanos** en expresiones de desplazamiento, las cuales se describen a continuación.

Operadores de desplazamiento

Los operadores de desplazamiento tambié manipulan bits. Solamente pueden utilizarse con el tipo primitivo entero. El operador de desplazamiento a la izquierda (<<) da como resultado el desplazamiento a la izquierda del operando de la izquierda del operador el número de bits especificado por el número escrito tras el operador (Se insertan ceros en los bits de órden más bajo). El operador desplazamiento a la derecha con signo (>>) da como resultado el desplazamiento a la derecha del operando a la izquierda del operador desplazamiento el número de bits especificado por el número situado tras el operador. El operador de desplazamiento a la derecha con signo >> utiliza *extensión de signo* : Si el valor es positivo, se insertan ceros en los bits de órden más alto. Si el valor es negativo, se insertan unos en los bits de órden más alto. Java también dispone el desplazamiento a la derecha sin signo >>>, el cual usa *extensión de ceros* : Sea cual sea el signo, se insertan ceros en los bits de los bits de órden más alto. Este

operador no existe en C o en C++.

Si se desplazan **char** , **byte** o **short** se prmueven a **int** antes de que el desplazamiento tenga lugar y el resultado será un **int** . Solamente se utilizarán los cinco bits de órden bajo del lado derecho. Esto impide desplazar más del número de bits que hay en un **int** . Si se opera sobre un **long** , se obtendrá un **long** . Solamente se usarán los seis bits de órden bajo del lado derecho por lo que no se podrán desplazar más que el número de bits en un **long**

Los desplazamientos pueden combinarse con el signo igual (<<= o >>= o >>>=). El "ivalor" (valor de la izquierda) se sustituye por el "ivalor" desplazado por el "rvalor" (valor de la derecha). Existe un problema, sin embargo, con el desplazamiento a la derecha sin signo combinado con la asignación. Si se utiliza con **byte** o **short** no se obtienen resultados correctos. Por el contrario, estos tipos se promocionan a **int** y se desplaza a la derecha, pero se trunca cuando vuelve a asignarse a la variable, por lo que siempre se obtiene **-1** en estos casos. El siguiente ejemplo demuestra esto:

```
//: c03:URShift.java
// Prueba del desplazamiento de bits a la derecha sin signo
public class URShift {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
    short s = -1;
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
    b = -1;
    System.out.println(b>>>10);
  }
} ///:~
```

En la última línea, el valore resultante no vuelve a asignarse a **b** , sino que se imprime directamente y, por lo tanto, tenemos el comportamiento correcto.

Aquí tenemos un ejemplo que demuestra el uso de todos los operadores que implican bits

```
//: c03:BitManipulation.java
// Uso de los operadores de bits
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);

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

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

```
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
    .System.out.print(" ");
    for(int j = 31; j >= 0; j--)
      if(((1 << j) & i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print(" ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

Los dos métodos del final, **pBinInt()** y **pBinlong()** toman un **int** o un **long** , respectivamente, y lo imprimen en formato binario junto con una descripción. Puede ignorar la implementación de estas funciones por ahora.

Se habrá dado cuenta del uso de **System.out.print()** en vez de **System.out.println()** . El método **print()** no envía una nueva línea por lo que permite sacar una línea a trozos.

Además de demostrar el efecto de todos los operadores de bits para **int** y **long** , este ejemplo también muestra los valores mínimo, máximo, 1 y -1 para **int** y **long** por lo que se puede ver el aspecto que presentan. Puede notarse lo que representa el bit de órden superior es el signo: 0 significa positivo y 1 significa negativo. La salida de la parte relativa a **int** es:

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 59081716, binary:
```

```
   00000011100001011000001111110100
~i, int: -59081717, binary:
   11111100011110100111110000001011
-i, int: -59081716, binary:
   11111100011110100111110000001100
j, int: 198850956, binary:
   00001011110110100011100110001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011110111111011101111111100
i ^ j, int: 140491384, binary:
   00001000010111111011101001111000
i << 5, int: 1890614912, binary:
   01110000101100000111111010000000
i >> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
   11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
   00000111111000111101001111100000
```

La representación binaria de los números se refiere a la representación en *Complemento a dos* .

Operador ternario if-else

Este operador es inusual porque tiene tres operandos. Es un verdadero operador porque produce un valor, a diferencia de la sentencia if-else que se verá en la siguiente sección de este capítulo. La expresión es de la forma:

```
boolean-exp ? value0 : value1
```

Si evaluar *boolean-exp* resulta **true** , se calcula *value0* y su resultado es el resultado del operador. Si *boolean-exp* es **false** , se calcula *value1* y su resultado es el resultado producido por el operador.

Naturalmente, se puede usar un **if-else** típico (descrito más adelante), pero el operador ternario es mucho más breve. Aunque en C (de donde este operador es originario) presume de ser un lenguaje breve y el operador ternario podría haber sido introducido parcialmente por razones de eficiencia, debería ser cauteloso a la hora en usarlo habitualmente - es fácil escribir codigo ilegible.

El operador condicional puede utilizarse por sus efectos secundarios o por el valor

que genera pero, en general, lo que se desea es un valor puesto que es esto lo que diferencia al operador de la sentencia **if-else** . Aquí tenemos un ejemplo:

```
static int ternary(int i) {
   return i < 10 ? i * 100 : i * 10;
}
```

Puede comprobarse que este código es más compacto que

```
static int alternative(int i) {
if (i < 10)
   return i * 100;
else
   return i * 10;
}
```

La segunda forma es más fácil de entender y no requiere teclear mucho más. Asegurese de considerar bien sus razones cuando elija utilizar el operador ternario

El operador punto y coma (;)

El punto y coma usado en C y C++ no solamente es un separador de listas de argumentos, sino también se usa como operador de evaluación secuencial. El único sitio donde se usa el *operador* en Java es en lo bucles **for** que se describen más tarde en este capítulo

El operador + de String

Tenemos en Java un uso especial de un operador: El operador + puede utilizarse para concatenar cadenas de caracteres, como ya ha visto. Parece ser un uso natural de + aunque no se adecúe a la forma tradicionalen la que se usa +. Esta capacidad pareció una buena idea en C++, por lo que se añadió la *sobrecarga de operadores* en C++ para permitir a los programadores de C++ añadir significado a casi cualquier operador. Desafortunadamente, la sobrecarga de operadores combinada con algunas otras restricciones de C++ hace que esta característica sea bastante complicada para los programadores cuando intentan incorporarla a sus clases. Aunque la sobrecarga de operadores podría haber sido mucho más simple de implementar en Java que en C++, esta característica sigue siendo considerada muy compleja por lo que los programadores de Java no pueden implementar sus propios operadores sobrecargados como pueden hacerlo los programadores de C++.

El uso de **String +** tiene un comportamiento interesante. Si una expresión

comienza con un **String** , entonces todos los operadores que siguen deben ser **String** (Recuerde que el compilador convertirá una secuencia de caracteres entre comillas simples en un **String**):

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Aquí, el compilador Java convertirá **x** , **y** y **z** en sus representaciones **String** en lugar de sumarlas antes. Si dice:

```
System.out.println(x + sString);
```

Java convertirá **x** en un **String** .

Trampas comunes en el uso de operadores

Una de los problemas que se presentan cuando se usan operadores es intentar evaluar sin paréntesis una expresión en la que se tiene una pequeña duda en como será evaluada. Esto sigue siendo cierto en Java.

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

El programador intentaba comprobar la equivalencia (==) en lugar de hacer una asignación. En C y C++, el resultado de esta asignación siempre será **true** si **y** no es cero por lo que, probablemente, entrará en un bucle infinito. En Java, el resultado de esta expresión no es un **booleano** , que es lo que espera el compilador que no lo convertirá en **int** , por lo que, convenientemente, le dará un error en tiempo de compilación antes incluso de ejecutar el programa. Por lo tanto, no se encontrará con esta trampa en Java. (El único caso en el que no se obtendrá un error de compilación es cuando **x** e **y** sean **boolean** , en cuyo caso **x = y** es una expresión legal y, en el caso anterior, probablemente un error).

Un problema similar en C y C++ es usar los operadores de bits AND y OR en lugar de sus verisones lógicas. Los operadores de bits usan un caracter (**&** o **|**)) mientras que los operadores lógicos AND y OR usan dos (**&&** y **||**). Lo mismo que con = y ==, es fácil teclear sólo un caracter en lugar de dos. En Java, el compilador, de nuevo, previene esto porque no le deja usar despreocupadamente

usar un tipo donde no corresponde,

Moldeado de operadores

La palabra *moldeado* (del inglés *cast*)se usa en el sentido de "ajustar a un molde". Java cambiará automáticamenteun tipo de dato a otro cuando estime apropiado. Por ejemplo, si asigna un valor entero a una variable en punto-flotante, el compilador convertirá automáticamente el **int** en **float** . El moldeado le permite realizar esta conversión explícitamente o forzarla cuando no vaya a suceder.

Para realizar un moldeado hay que poner el tipo de dato deseado (incluidos todos los modificadores) entre paréntesis a la izquierda de cualquier valor. Aquí tenemos un ejemplo:

```
void casts() {
   int i = 200;
   long l = (long)i;
   long l2 = (long)200;
}
```

Como puede ver, es posible ralizar un moldeado sobre valores numéricos tanto como sobre variables. En los casos aquí mostrados, sin embargo, el moldeado es completamente superfluo puesto que el compilador promocionará automáticamente un valor **int** a **long** cuando sea necesario. Pero puede hacer este cast superfluo para asegurarse o hacer el código más claro. En otras situaciones, un cast puede ser esencial para compilar el código.

En C y C++, el moldeado puede provocar dolores de cabeza. En Java, el moldeado seguro con la excepción hecha de la *conversión a menor* (esto es, cuando va desde un tipo de dato que puede contener más información a otro que no puede con tanta) en la que se corre el rieso de perder información. Aquí, el compilador le fuerza a realizar un moldeado. En efecto, diciendo "esto puede ser algo peligroso de hacer - si usted quiere que lo haga debe hacer un moldeado explícito". Con una *conversión a mayor* un moldeado explícito no es necesario porque el nuevo tipo podrá contener mayor información que el tipo anterior por lo que no habrá pérdida de información.

Java permite moldear cualquier tipo primitivo en otro tipo primitivo, escepto el **boolean** , al cual no le está permitido ningún moldeado. Los tipos de clase no permiten moldeado. Para convertir de uno a otro debe haber métodos especiales. (**String** es un caso especial y encontrará más tarde en este libro que los objetos pueden moldearse dentro de una *familia* de tipos; un **Roble** puede moldearse a un **Árbol** pero no hay un tipo extraño como una **Roca**)

Literales

Normalmente, cuando inserta un valor literal en un programa, el cokpilador sabe exactamente de que tipo hacerlo. A veces, por el contrario, el tipo es ambiguo. Cuando esto ocurre, debe guiar al compilador añadiendo información extra en forma de caracteres asociados con el valor literal. El siguiente código muestra esos caracteres:

```
//: c03:Literals.java
class Literals {
   char c = 0xffff; // valor hex char máximo
   byte b = 0x7f; // valor hex byte máximo
   short s = 0x7fff; // valor hex short máximo
   int i1 = 0x2f; // Hexadecimal (minúsculas)
   int i2 = 0X2F; // Hexadecimal (mayúsculas)
   int i3 = 0177; // Octal (cero inicial)
   // Hex and Oct también funcionan con long.
   long n1 = 200L; // sufijo long
   long n2 = 200l; // sufijo suffix
   long n3 = 200;
   //! long l6(200); // no permitido
   float f1 = 1;
   float f2 = 1F; // sufijo float
   float f3 = 1f; // sufijo float
   float f4 = 1e-45f; // potencia de 10
   float f5 = 1e+9f; // sufijo float
   double d1 = 1d; // sufijo double
   double d2 = 1D; // sufijo double
   double d3 = 47e47d; // potencia de 10
} ///:~
```

El hexadecimal (base 16), que funciona con todos los tipos de datos enteros, se denota por un **0x** o **0X** inicial seguido por =-9 y a-f ya se en minúsculas o en mayúsculas. Si trata de inicializar una variable con un valor mayor del que puede contener (sin importar la forma numérica del valor), el compilador mostrará un mensaje de error. Notar en el código anterior el valor hexadecimal máximo posible para **char** , **byte** y **short** . Si se exceden estos valores, el compilador automáticamente convertirá este valor en **int** y le indicará que tendrá que realizar un moldeado a menor en la asignación. Usted lo sabrá porque se le indicará la línea.

El octal (base 8) se denota por un cero inicial en el número y los dígitos0-7. No hay representación literal para números binarios en C, C++ o Java

Un caracter final después de un valor literal establece su tipo. Una **L** mayúsucula o minúscula significa **long** , una **F** mayúscula o minúscula signigfica **float** y una **D**

mayúscula o minúscula significa **double** .

Los exponentes usan una notación que siempre he encontrado consternadora: **1.39e-47f** . En ciencia e ingeniería, 'e' se refiere a la base de los logaritmos naturales, aproximadamente 2.718 (Un valor **double** más preciso está disponible en Java como **Math.E**). Este se usa en expresiones de exponencicion tales como 1.39 x e ^-47, que significa 1.39 x 2.718 ^-47. Sin embargo, cuando se inventó FORTRAN se decidió que **e** seignificaba naturalmente "potencia de diez", lo que es una curiosa decisión puesto que FORTRAN se diseño para la ciencia y la ingeniería y uno tendría que pensar que los diseñadores deberían haber sido sensible a tal ambigüedad[1]. En cualquier caso, esta decisión se siguió en C, C++ y en Java. Por tanto, si está acostumbrado a pensar en **e** como en la base de los logaritmos naturales, debe hacer una traducción mental cuando use una expresión como **1.39e-47f** en Java donde significa 1.39 x 10^-47.

Note que no es necesario usar el carácter final cuando el compilador puede saber el tipo apropiado. Con

```
long n3 = 200;
```

no hay ambigüedad por lo que la **L** después del 200 es superflua. Sin embargo, con

```
float f4 = 1e-47f; //potencia de 10
```

el compilador normalmente hace toma los números exponenciales como dobles, por lo que la **f** final dará un error diciendo que debe usar un moldeado para convertir el **double** en **float** .

Promoción

Descubrirá que si realiza cualquier operación matemática o de bits sobre tipos de datos primitivos que son menores que un **int** (Esto es, **char** , **byte** o **short**), esos valores se promocionaran a **int** antes de realizar las operaciones y el valor resultante será de tipo **int** . Por lo tanto, si se desea volver al tipo menor, debe utlizar un moldeado (Y, puesto que está asignando a un tipo menor, puede perder información). En general, el tipo de dato más grande en una expresión es el que determina el tamaño del resultado de la expresión. Si multiplpica un **float** y un **double** , el resultado será un **double** , si añade un **int** y un **long** , el resultado será un **long** .

Java no tiene "sizeof"

En C y C++, el operator **sizeof()** satisface una necesidad específica: le dice en número de bytes ocupados por cada elemento de datos. La necesidad más respetable de **sizeof()** en C y C++ es la portabilidada. Diferentes tipos de datos pueden ser de diferentes tamaños en máquinas diferentes por lo que el programador debe saber como son de grandes esos tipos cuando realiza operaciones que son sensibles al tamaño. Por ejemplo, un ordenador puede almacenar enteros en 32 bits mientras que otro puede almacenar enteros como 16 bits. Los programas pueden utilizar valores enteros más grandes en la primera máquina. Como puede imaginar, la portabilidad es un enorme dolor de cabeza para los programadores C y C++.

Java no necesita el operador **sizeof()** para este propósito porque todos los tipos de datos son del mismo tamaño en todas las máquinas. No necesita pensar en la portabilidad en este nivel - está diseñado dentro del lenguaje.

Revisión de la precedencia

Al oir quejarme sobre la complejidad de recordar la precedencia de operadores durante uno de mis seminarios, un estudiante sugirió una regla mnemónica que es, a la vez, un comentario: "Ulcer Addicts Really Like C A Lot".

Mnemónico	Tipo de operador	Operadores
Ulcer	Unario	+ - ++ --
Addicts	Aritméticos (y desplazamientos)	* / % + - << >>
Really	Relacional	> < >= <= == !=
Like	Lógicas (y de bits)	&& || & | ^
C	Condicionales (ternarios)	A > B ? X : Y
A Lot	Asignación	= (y las compuestas como *=)

Naturalmente, con los operadores de desplazamiento y de bits distribuidos por la tabla, no se tiene un mnemónico perfecto, pero para operaciones que no sean con bits funciona.

Un compendio de operadores

El siguiente ejemplo muestra como pueden usarse los tipos de datos primitivos con operadores particulares. Básicamente, es el mismo ejemplo repetido una y otra vez pero usando tipos primitivos diferentes. El fichero se compilará sin error porque las líneas erroneas están comentadas con un //!

```
//: c03:AllOps.java
// Prueba todos los operadores sobre todos los
```

```
// tipos de datos primitivos para mostrar
// cuales son los aceptados por el compilador Java
class AllOps {
   // 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 de bits:
      //! 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;
      // Moldeado:
      //! 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 de bits:
   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 <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (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);
   // Bitwise operators:
   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 <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (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 de bits:
   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 <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (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 de bits:
   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;
   // Moldeado:
   //! boolean b = (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 de bits:
   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;
   // Moldeado:
   //! boolean b = (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 de bits:
   //! 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;
   // Moldeado:
   //! boolean b = (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 de bits:
   //! 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;
   // Moldeado:
   //! boolean b = (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;
   }
} ///:~
```

Es de notar que **boolean** es bastante limitado. Se le pueden asignar valores **true** y **false** y se puede comprobar su verdad o su falsedad pero no se pueden sumar booleabos o realizar ningún otro tipo de operación con él.

En **char** , **byte** y **short** puede efectuarse la promoción con los operadores aritméticos. Cada operación aritmética sobre cualquiera de esos tipos da un resultado **int** , que debe ser moldeado explícitamente al tipo original. (Una conversión a un tipo menor que podría producri una pérdida de información) para asignárselo a ese tipo. Con los valores **int** , sin embargo, no necesita moldeado porque ya es todo **int** . Por el contrario, no hay que dormirse pensando que todo es seguro. Si multiplica dos **int** s que sean lo suficientemente grandes, se tendrá un desbordamiento como resultado. El siguiente ejemplo muestra esto:

```
//: c03:Overflow.java
// Surprise! Java lets you overflow.
public class Overflow {
      public static void main(String[] args) {
      int big = 0x7fffffff; // valor entero máximo
      prt("big = " + big);
      int bigger = big * 4;
      prt("bigger = " + bigger);
   }
   static void prt(String s) {
      System.out.println(s);
   }
} ///:~
```

El resultado de salida es

```
big = 2147483647
bigger = -4
```

y no se obtienen errores o mensajes de advertencia del compilador ni excepciones en ejecución. Java es bueno. pero no es *así* de bueno.

Las asignaciones compuestas *no* requieren moldeado para **char** , **byte** o **short**

aunque ellos realicen promociones que tienen el mismo resultado que las operaciones aritméticas directas. Por otro lado, la falta de moldeado simplifica el código.

Como puede ver, con la excepción de **boolean** , cualquier tipo primitivo puede amoldarse a cualquier otro tipo primitivo. De nuevo, debe darse cuenta del efecto de la conversión a un tipo menor cuando se moldea a un tipo más pequeño, de otra forma perderá información sin percatarse durante el moldeado.

Control de ejecución

Java utiliza todas las sentencias de control de ejecución de C. De tal modo que si has programado en C o C++ la mayor parte de lo que vas a ver te será familiar. La mayoría de los lenguajes de programación de procedimiento poseen alguna clase de sentencia de control. Muchas veces éstas sentencias se repiten en los distintos lenguajes. En Java el conjunto de palabras clave incluye al **if-else, while, do-while, for** y una sentencia de selección llamada **switch.** Java en cambio no posee el nunca bien ponderado **goto** (el cual, sin embargo, puede ser la forma más rápida de resolver ciertos tipos de problemas). Aún es posible utilizar saltos al estilo goto, pero de una manera mucho más restringida.

true y false

Todas las sentencias condicionales utilizan el valor de verdad o falsedad devuelto por una expresión condicional. Un ejemplo de expresión condicional es **A == B.** Esta expresión usa el operador condicional **==** para determinar si el valor de **A** es equivalente al valor de **B.** La expresión devuelve **true** o **false.** Cualquiera de los operadores relacionales que has visto previamente en este capítulo puede usarse para producir una sentencia condicional. Ten en cuenta que, aunque permitido en C y C++ (donde un valor distinto de cero es falso y un valor igual a cero es verdadero), en Java no es legal utilizar un número como **boolean.** Si quieres usar un valor no booleano en una comprobación booleana, como por ejemplo **if(a),** debes primero convertirla a un valor booleano usando una expresión condicional como **if(a != 0).**

if-else

La sentencia **if-else** es, probablemente, la manera más básica de controlar el flujo de un programa. El **else** es opcional, así que es posible escribir el **if** de dos formas:

```
if(Expresión booleana)
   sentencia
```

o

```
if(Expresión booleana)
   sentencia
else
   sentencia
```

El condicional debe producir un valor booleano. *Sentencia* significa bien una sentencia simple terminada por un punto y coma, o una serie de sentencias encerradas entre llaves. De aquí en más siempre que el término *"sentencia"* sea usado, implicará que la sentencia puede ser simple o compuesta.

Como ejemplo de **if-else,** el método **test()** te mostrará si una estimación está por encima, debajo o es igual que el número buscado:

```
//: c03:IfElse.java
 public class IfElse {
   static int test(int testval, int target) {
      int result = 0;
         if(testval > target)
           result = +1;
         else if(testval < target)
           result = -1;
         else
           result = 0; // Match
         return result;
   }
   public static void main(String[] args) {
      System.out.println(test(10, 5));
      System.out.println(test(5, 10));
      System.out.println(test(5, 5));
   }
} ///:~
```

Por convención, el cuerpo de las sentencias de control de flujo se indenta de manera tal que quien lee el programa pueda determinar fácilmente cuál es el comienzo y el fin de cada una.

return

La palabra reservada **return** tiene dos fines: indicar cuál es el valor que devolverá un método (siempre y cuando el método no sea **void**) y provocar que ese valor sea devuelto en forma inmediata. El método **test()** presentado más arriba puede volver a escribirse para aprovechar esta característica:

```
//: c03:IfElse2.java
```

```
public class IfElse2 {
   static int test(int testval, int target) {
      int result = 0;
      if(testval > target)
         return +1;
      else if(testval < target)
        return -1;
      else
        return 0; // Match
   }
   public static void main(String[] args) {
      System.out.println(test(10, 5));
      System.out.println(test(5, 10));
      System.out.println(test(5, 5));
   }
} ///:~
```

No hay necesidad de un **else** porque el método no continúa luego de ejecutar un **return.**

Iteración

while, do-while y **for** controlan los ciclos y a menudo son clasificadas como *sentencias de iteración.* Una *sentencia* se repite hasta que la *expresión booleana* que la controla se evalúa como falsa. La forma de un ciclo **while** es

```
while(Expresión booleana)
   sentencia
```

La *expresión booleana* se evalúa antes de cada ejecución del ciclo (incluyendo a la primera).

Este es un simple ejemplo que genera números aleatorios hasta que cierta condición se cumpla:

```
//: c03:WhileTest.java
public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

Aquí se usa el método estático **random()** de la biblioteca **Math,** el cual genera un valor de tipo **double** comprendido entre 0 y 1 (incluyendo al 0 pero no al 1). La expresión condicional para el **while** indica lo siguiente: "siga ejecutando este ciclo hasta que el número sea 0.99 o mayor". Cada vez que corras este programa obtendrás una lista de números de diferente longitud.

do-while

La forma para un **do-while** es

```
do
   sentencia
while(Expresión booleana);
```

La única diferencia entre un **while** y un **do-while** es que la sentencia del **do-while** siempre se ejecuta al menos una vez, aún cuando la expresión se evalúe como falsa la primera vez. En un **while,** el hecho de que el condicional sea falso la primera vez que se evalúa provoca que la sentencia nunca llegue a ejecutarse. En la práctica un **do-while** es menos frecuente que un **while.**

for

Un ciclo **for** realiza una inicialización antes de la primera iteración. Luego realiza una comprobación condicional y, al final de cada iterración, cierta forma de "dar un paso" hacia la próxima. La forma para un cliclo **for** es:

```
for(inicializacion;expresión booleana;paso)
   sentencia
```

Cualquiera de las expresiones de *inicialización, expresión booleana* o *paso* puede estar vacía. La expresión booleana se comprueba antes de cada iteración y ni bien ésta evalúe como falso la ejecución continúa en la línea siguiente a la sentencia del **for.** Al final de cada ciclo se ejecuta el *paso.*

los ciclos **for** se utilizan normalmente para tareas en las que es necesario contar:

```
//: c03:ListCharacters.java
public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
```

```
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

Observa que la variable **c** se define en el punto en que la misma se utiliza, dentro de la expresión de control del ciclo **for,** y no al comienzo del bloque indicado por la llave de apertura. El ámbito de **c** es la expresión controlada por el **for.**

Los lenguajes de procedimiento como C requieren que todas las variables se definan al comienzo de un bloque, de tal manera que cuando el compilador cree un bloque pueda reservar espacio para estas variables. En Java y C++ se puede distribuir las declaraciones de variables en todas las partes del bloque, definiéndolas en el lugar en que se las precisa. Esto facilita un estilo de codificación más natural y hace que el código sea más fácil de entender.

Es posible definir varias variables dentro de una sentencia **for** , pero éstas deben ser del mismo tipo:

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
     i++, j++)
 /* cuerpo del ciclo for */
```

La definición de **int** dentro de la sentencia **for** abarca tanto a **i** como a **j.** La posibilidad de definir variables en la expresión de control estás limitada al ciclo **for.** No es posible usar esta estrategia en ningún otro tipo de sentencia de selección o iteración.

El operador coma

Previamente en este capítulo indiqué que el *operador coma* (no el *separador coma,* que se utiliza para separar definiciones o argumentos de funciones) posee un único uso en Java: en la expresión de control de un ciclo **for.** Es posible tener varias sentencias separadas por comas tanto en la porción de inicialización como en la porción de paso de la expresión de control, y esas sentencias serán evaluadas secuencialmente. El ejemplo de código anterior utiliza esta capacidad. Aquí hay otro ejemplo:

```
//: c03: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);
    }
  }
} ///:~
```

La salida sería

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

Puede verse que tanto en la porción de inicialización como en la porción de paso las sentencias se evalúan en orden secuencial. Además, la porción de inicialización puede tener cualquier cantidad de definiciones *de un tipo.*

break y continue

Dentro del cuerpo de cualquiera de las sentencias de iteración también es posible controlar el flujo del loop usando **break** y **continue. break** sale del ciclo sin ejecutar el resto de las sentencias del mismo. **continue** detiene la ejecución de la iteración actual y vuelve al comienzo del ciclo para comenzar la siguiente iteración.

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

```
//: c03:BreakAndContinue.java
public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
```

```
} ///:~
```

En el ciclo **for** el valor de **i** nunca llega a 100 ya que la sentencia **break** corta la ejecución del ciclo cuando **i** vale 74. Normalmente se usaría un **break** como este sólo si no se conoce el momento en que una condición de fin va a ocurrir. La sentencia **continue** provoca que la ejecución vuelva al principio del ciclo de iteración (incrementando **i**) siempre y cuando **i** no sea divisible entre 9. Cuando lo es, su valor se imprime.

La segunda parte muestra un "ciclo infinito", el cual, en teoría, continuaría para siempre. Sin embargo, dentro del ciclo existe una sentencia **break** que finalizará el ciclo. Además puede verse que el **continue** vuelve al comienzo del ciclo sin completar el resto (así es que la impresión en el segundo ciclo ocurre solamente cuando el valor de **i** es divisible entre 10). La salida es:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

Se imprime el valor cero porque 0 % 9 produce 0.

Una segunda forma de ciclo infinito es **for(;;).** El compilador trata tanto a **while (true)** como a **for(;;)** de la misma manera, así es que el uso de uno u otro es simplemente una cuestión de gustos.

El infame "goto"

La palabra reservada **goto** ha estado presente desde el principio en los lenguajes de programación. Es más, **goto** fue el génesis del control de programas en lenguaje ensamblador: "Si se da la condición A, entonces saltar hacia aquí, de otra forma saltar hacia allí". Si se lee el código de ensamblador que en última instancia es generado por virtualmente cualquier compilador se verá que el control del programa posee muchos saltos. Sin embargo, un **goto** es un salto al nivel de código fuente, y eso es lo que le valió su desprestigio. Si un programa tiene siempre que saltar de un punto a otro ¿no hay alguna manera de reorganizar el

código de tal forma que el flujo de control no sea tan saltarín? el **goto** tuvo al fin su desaprobación general con la publicación del famoso artículo de Edsger Dijkstra "Goto considerado dañino", y desde ese entonces el "pegarle" al goto ha sido un deporte popular, con los partidarios de la palabra clave maldecida corriendo a buscar refugio.

Como es típico en casos como éste, lo mejor resulta un punto intermedio. El problema no es el uso del **goto,** sino más bien el abuso del mismo. En contadas ocasiones el **goto** es, en realidad, la mejor manera de estructurar el control de flujo.

Aún cuando **goto** es una palabra reservada en Java, la misma no se utiliza en el lenguaje: Java no posee **goto.** Sin embargo sí tiene algo que se parece bastante a un salto y que se usa conjuntamente con las palabras clave **break** y **continue.** No es un salto, sino más bien una manera de salirse de una sentencia de iteración. La razón por la cual aparece normalmente con el tratamiento del **goto** es que usa el mismo mecanismo que éste: una etiqueta.

Una etiqueta es un identificador seguido de dos puntos, como este:

```
etiqueta1:
```

El *único* lugar en donde una etiqueta es útil en Java es inmediatamente antes de una sentencia de iteración.Y esto significa *justo* antes, no es bueno agregar ninguna otra sentencia entre la etiqueta y la iteración. Y la única razón de poner una etiqueta delante de una iteración es que se vaya a anidar otra iteración o un **switch** dentro de ésta. Esto es porque las sentencias de **break** y **continue** normalmente interrumpen sólo el ciclo actual, pero cuando son usadas con una etiqueta las mismas interrumpen todos los ciclos hasta el nivel en que existe la etiqueta:

```
label1:
outer-iteration { i
   nner-iteration {
     // ...
     break; // 1
     //...
     continue; // 2
     //...
     continue label1; // 3
     //...
     break label1; // 4
 }
}
```

En el caso 1, el **break** finaliza la iteración interna, y el programa continúa en la iteración externa. En el caso 2, el **continue** vuelve al comienzo de la iteración interna. En el caso 3, sin embargo, el **continue label1** interrumpe *tanto* la iteración interna *como* la interna externa y vuelve directamente a **label1.** A partir de ese momento continúa con el ciclo, pero partiendo desde la iteración externa. En el caso 4, el **break label1** también interrumpe hasta **label1,** pero no vuelve a entrar en la iteración. En realidad interrumpe por completo ambas iteraciones.

Aquí hay un ejemplo utilizando ciclos **for:**

```
//: c03:LabeledFor.java
public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue
```

```
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Aquí se usa el método **prt()** definido en otros ejemplos

Observa que **break** interrumpe el ciclo **for,** y que la expresión de incremento no ocurre hasta el final del pasaje por el ciclo **for.** Ya que el **break** se saltea la expresión de incremento, el incremento se realiza directamente en el caso de **i == 3.** La sentencia **continue outer** en el caso de **i == 7** también vuelve al comienzo del ciclo y también se saltea el incremento, así es que también se incrementa directamente.

Esta es la salida

```
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
```

Si no fuera por la sentencia **break outer** no habría manera de salir del loop externo desde dentro del loop interno, ya que **break** por sí mismo sólo puede terminar sólo el ciclo más interno (lo mismo se aplica a **continue**)

Por supuesto que en los casos donde el terminar un ciclo también finalizaría la ejecución del método es posible utilizar un simple **return.**

Esta es una demostración del uso de **break** y **continue** con etiquetas usados en conjunción con ciclos **while:**

```
//: c03:LabeledWhile.java
// From 'Thinking in Java, 2nd ed.' by Bruce Eckel
// www.BruceEckel.com. See copyright notice in CopyRight.txt.
// Java's "labeled while" loop.

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
        }
      }
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Las mismas reglas se aplican al **while:**

1. Un **continue** simple va al comienzo del ciclo más interno y continúa.
2. Un **continue** etiquetado vuelve a entrar al ciclo inmediatamente después de la etiqueta.
3. Un **break** continúa al final del ciclo.
4. Un **break** etiquetado continúa al final del ciclo marcado por la etiqueta.

La salida del método deja todo esto en claro:

```
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
```

Es importante recordar que la *única razón* para usar etiquetas en Java es el tener ciclos anidados para los cuales se desea ejecutar un **break** o **continue** a través de más de un nivel de anidación.

En el artículo de Dijkstra "Goto considerado dañino", el autor objetó específicamente las etiquetas, no el goto. Puntualizó que la cantidad de errores de programación parecía incremenentarse en relación con el número de etiquetas en un programa. Las etiquetas y los gotos hacen que un programa sea difícil de analizar en forma estática, ya que introducen ciclos en el grafo de ejecución del programa. Observa que las etiquetas de Java no sufren de ese problema, ya que están restringidas en su ubicación y no pueden ser utilizadas para transferir el ontrol de una manera arbitraria. También es interesante considerar que este es un caso en el que un lenguaje de programación se hace más útil por medio de restringir el poder de una sentencia.

switch

El **switch** normalmente se clasifica como una *sentencia de selección.* La sentencia **switch** realiza una selección de entre varias secciones de código basada en el valor de una *expresión entera* (en este contexto, *expresión entera* no significa necesariamente de tipo **int,** sino cualquier tipo que represente un número que no sea de punto flotante como por ejemplo **short** y **byte**). Su forma es:

```
switch(selector entero) {
 case valor entero 1 : sentencia; break;
 case valor entero 2 : sentencia; break;
 case valor entero 3 : sentencia; break;
```

```
 case valor entero 4 : sentencia; break;
 case valor entero 5 : sentencia; break;
 // ...
 default: sentencia;
}
```

El *selector entero* es una expresión que produce un valor entero. El **switch** compara el resultado del *selector entero* con cada uno de los *valores enteros.* Si encuentra una coincidencia entonces la correspondiente *sentencia* (simple o compuesta) se ejecuta. De no ocurrir ninguna coincidencia se ejecuta la *sentencia* marcada como **default.**

En la definición anterior puede apreciarse que cada **case** finaliza con un **break,** lo cual causa que la ejecución salte al final del cuerpo del **switch.** Esta es la forma convencional de contruir la sentencia **switch,** no obstante, el **break** es opcional. Si el **break** falta, se ejecuta el código de los siguientes **case,** hasta que se encuentre un **break.** Si bien normalmente este no es el comportamiento que se busca, puede llegar a ser útil para un programador experimentado. Hay que tener en cuenta que la última sentencia, la que sigue al **default** no lleva un **break** ya que la ejecución del programa de todas formas seguirá exactamente en donde hubiese saltado el **break.** Puedes usar un **break** de todas formas sin causar ningún daño, si es que lo consideras importante debido a una cuestión de estilo.

La sentencia **switch** es una manera limpia de implementar una selección de entre varios cursos de ejecución diferentes, pero requiere un selector que se evalúe dando un valor entero como por ejemplo **int** o **char.** Si quieres usar, por ejemplo, una cadena de caracteres o un valor de punto flotante como selector no puedes hacerlo con una sentencia **switch.** Para tipos no enteros se debe usar una serie de sentencias **if.**

Este es un ejemplo en el que se generan letras al azar y se determina si éstas son vocales o consonantes:

```
//: c03:VowelsAndConsonants.java
public class VowelsAndConsonants {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vowel");
```

```
                  break;
      case 'y':
      case 'w':
                  System.out.println(
                    "Sometimes a vowel");
                  break;
      default:
                  System.out.println("consonant");
      }
    }
  }
} ///:~
```

Como **Math.random()** genera un valor comprendido entre 0 y 1, sólo se necesita multiplicarlo por el límite superior del rango de números que se desea producir (26 para las letras del alfabeto) y sumar el valor del límite inferior.

Aunque pareciera que aquí se está usando un **switch** con un caracter, en realidad el **switch** está usando el valor entero del carácter. Los caracteres encerrados entre apóstrofos en las sentencias **case** también producen valores enteros que son utilizados para realizar las comparaciones.

Observa cómo los **cases** pueden apilarse uno sobre el otro para brindar múltiples coincidencias para un solo segmento de código. También deberías tener en cuenta que es esencial agregar una sentencia **break** al final de un **case** particular, en caso contrario el flujo del programa simplemente continuará con el proceso del siguiente **case.**

Detalles de cálculo

La sentencia

```
char c = (char)(Math.random() * 26 + 'a');
```

merece una mirada más atenta. **Math.random()** produce un **double,** así que el valor de 26 se convierte a **double** para realizar la multiplicación, la cual también produce un **double.** Esto significa que **'a'** debe convertirse a **double** a fin de realizar la suma. El resultado **double** se vuelve a convertir en **char** utilizando un moldeado (cast).

¿Qué es lo que produce un moldeado a **char** ? O sea, si se tiene un valor de 29.7 y se lo moldea a **char** ¿el resultado es 30 o 29? La respuesta a esta pregunta puede averiguarse utilizando este ejemplo:

```
//: c03:CastingNumbers.java
public class CastingNumbers {
  public static void main(String[] args) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
      (char)('a' + below));
  }
} ///:~
```

La salida es:

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

Así que la respuesta es que el moldeado de

```
float
```

o

```
double
```

a un valor entero siempre trunca.

Una segunda pregunta concerniente a **Math.random().** ¿El mismo produce un valor comprendodo entre 0 a 1, incluyendo o excluyendo el valor 1? En lenguaje

matemático, ¿es (0,1), [0,1], (0,1] o [0,1)? (el corchete significa "inclusive", mientras que el peréntesis significa "excluyendo"). Otra vez un programa de prueba puede darnos la respuesta

```
//: c03:RandomBounds.java
public class RandomBounds {
  static void usage() {
    System.err.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds 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)
        ; // Keep trying
      System.out.println("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Keep trying
      System.out.println("Produced 1.0!");
    }
    else
      usage();
  }
} ///:~
```

Para ejecutar el programa simplemente se ingresa la línea de comandos

```
java RandomBounds lower
```

o

```
java RandomBounds upper
```

En ambos casos te verás forzado a interrumpir el programa en forma manual, así que *parecería* que **Math.random()** nunca produce ni 0.0 ni 1.0. Sin embargo, aquí es cuando este experimento puede resultar engañoso. Si consideras que hay 2^62 valores de doble precisión entre 0 y 1, la posibilidad de conseguir cualquiera

de ellos experimentalmente excedería el tiempo de vida de una computadora, y también el del experimentador. Sucede que 0.0 *está* incluído en la salida de **Math.random().** O sea que en lenguaje matemático, es [0,1).

Chuck Allison escribe: La cantidad total de números en un sistema de números de punto flotante es **2(M-m+1)^(p-1)+1,** donde **b** es la base (normalmente 2), **p** es la precisión (dígitos en la mantisa), **M** es el exponente más grande y **m** es el exponente más pequeño. IEEE 754 utiliza: **M=1023, m=-1022, p=53, b=2,** así que el número total es **2(1023 + 1022 + 1)2^52 = 2((2^10 - 1) + (2^10 - 1))2^52 = (2^10 -1)2^54 = 2^64 - 2^54.** La mitad de estos números (correspondientes a exponentes en el rango [-1022,0]) son menores que 1 (aproximadamente 2^62 valores están en el rango [0,1). Ver mi ensayo en http://www.freshsources.com/1995006a.htm

Resumen

Este capítulo concluye el estudio de las características fundamentales que aparecen en la mayoría de los lenguajes de programación: cálculo, precedencia de operadores, moldeado de tipos, selección e iteración. Ahora estás listo para comenzar a dar pasos que te lleven a estar más cerca del mundo de la programación orientada a objetos. El siguiente capítulo cubre aspectos importantes de la inicialización y purgado de objetos, seguido por el concepto esencial del ocultamiento de la implementación.

Ejercicios

Las soluciones a ejercicios seleccionados puede hallarse en el documento electrónico *The Thinking in Java Annotated Solution Guide,* disponible por un precio accesible en *www.BruceEckel.com*

1. Hay dos expresiones que se encuentran en la sección "precedencia" de este capítulo. Pon esas expresiones en un programa y demuestra que ellas producen diferentes resultados
2. Pon los métodos ternary() y alternative() en un programa.
3. Pon los métodos test() y test2() de las secciones tituladas "if-else" y "return" en un programa.
4. Escribe un programa que imprima los valores que van del 1 al 100.
5. Modifica el ejercicio 4 de tal manera que el programa finalize mediente una sentencia break al llegar al n'umero 47. Luego cámbialo para utilizar return.
6. Escribe una función que reciba dos argumentos de tipo String y que utilice todas las comparaciones booleanas para comparar esas dos cadenas e imprima los resultados. Para las comparaciones == y != también utilizar la comprobación equals(). En el método main () llama a tu función con diferentes objetos de tipo String
7. Escribe un programa que genere 25 valores int aleatorios. Para cada valor, usa una sentencia if-then-else para clasificarlo como mayor, menor o igual que un segundo valor generado aleatoriamente.

8. Modifica el ejercicio 7 de tal manera que tu código quede rodeado por un ciclo while "infinito". El programa se ejecutará entonces hasta que lo interrumpas desde el teclado (normalmente presionando la combinación control-C).
9. Escribe un programa que utilice dos ciclos anidados y el operador módulo (%) para detectar e imprimir números primos (números enteros que no son divisibles por ningún otro número, con excepción de sí mismos y el 1).
10. Crea una sentencia switch que imprima un mensaje por cada case, y pon el switch dentro de un ciclo que recorra cada case. Pon un break luego de cada case, ejecuta el programa, luego quita los breaks y observa qué sucede.

4: Inicialización y Limpieza

Con el avance de la revolución informática, la "inestabilidad" de la programación se ha convertido en uno de los factores principales en el incremento del coste del software.

Dos de estos elementos de estabilidad son la *inicialización* y el *purgado* , o liberación de recursos del sistema cuando estos dejan de ser necesarios. Muchos de los errores en C se producen cuando el programador olvida inicializar una variable.Esto es cierto sobre todo cuando se trabaja con librerías ya que los usuarios a menudo desconocen como inicializar sus componentes o incluso si deben ser inicializados o no. El purgado presenta un problema especial ya que es fácil olvidarse de un elemento cuando ha dejado de usarse y ha dejado de ser interesante. Es así como los recursos usados por ese elemento no son liberados con lo que se puede acabar por agotar los recursos del sistema (normalmente la memoria).

C++ introdujo el concepto de *constructor* , un método especial que es invocado automáticamente cuando se crea un objeto. Java ha adaptado el constructor y además tiene un recogedor de basura que automáticamente libera las zonas de memoria que ya no están en uso. Este capítulo examina los conceptos de inicialización y purgado y su implementación en Java.

El constructor garantiza la inicialización

Se puede imaginar que hubiese que crear un método llamado **inicializar()** cada una de las clases que se escribiese. El nombre del es una indicación de que el método debebería ser invocado antes de usar el objeto. Pero esto significaría que el usuario debe acordarse siempre de llamar a este método. En Java el diseñador de una clase puede garantizar que todos los objetos son inicializados definiendo un método especial llamado *constructor* . Si una clase tiene un constructor Java invocará automáticamente ese constructor cuando el objeto es creado, antes de que el usuario pueda ponerle las manos encima.

El siguiente problema a resolver es que nombre debería darse a este método. Hay que considerar dos cuestiones. La primera es que, cualquier nombre que se elija podría coincidir con el que se desease usar un miembro de la clase. La segunda cuestión es que el compilador debe generar la invocación al constructor por lo que deberá saber en todo momento que método llamar. La solución que da C++ parece la más sencilla y la más lógica por lo que ha sido adoptada por Java: el nombre del constructor es el mismo que el de la clase. Es lógico que así se

invocado durante la inicialización.

Veamos el constructor de una clase sencilla:

```
//: c04:ConstructorSimple.java
// Ejemplo de un constructor simple.

class Roca {
   Roca () { // Este es el constructor
      System.out.println("Creando Roca)");
   }
}

public class ConstructorSimple {
   public static void main(String[] args) {
      for (int i = 0; i < 10; i++)
         new Roca();
   }

} ///:-
```

Entonces, cuando se crea un objeto:

```
new Roca();
```

se reserva un área de memoria y se invoca el constructor. De este modo queda garantizado que el objeto es inicializado adecuadamente antes de que nadie pueda tocarlo

Se observa que, la regla habitual de poner en minúscula la primera letra del nombre de los métodos no se aplica a los constructores, ya que estos deben tener *exactamente* con el nombre de su clase.

Como cualquier otro método los constructores pueden tener argumentos que permiten especificar como se ha de crear el objeto.El ejemplo anterior puede ser fácilmente modificado que el constructor tenga un argumento.

```
//: c04:ConstructorSimple2.java
// Los constructores pueden tener argumentos.

class Roca2 {
   Roca2(int i) {
      System.out.println(
```

```
         "Creando Roca numero " + i);
   }
}

public class ConstructorSimple2 {
   public static void main(String[] args) {
      for (int i = 0; i < 10; i++)
         new Roca2(i);
   }

} ///:-
```

Los argumentos del constructor proporcionan al programador un medio parametrizar la creación de objetos. Por ejemplo, si la clase **Arbol** tiene un constructor con un único argumento entero representando la altura, se puede crear un objeto **Arbol** del siguiente modo:

```
Arbol a = new Arbol (12);  // árbol de 12 pies.
```

Si **Arbol(int)** es el único constructor, el compilador no permitirá que se creen objetos de ninguna otra manera.

Los constructores eliminan una amplia categoría de errores y hacen que el código sea más fácil de leer. Por ejemplo, en el fragmento de código precedente no aparece ninguna llamada explícita a ningún método **inicializar()** separado conceptualmente de la definición. En Java la definición y la inicialización son dos conceptos que van de la mano: no se puede tener uno sin el otro.

El constructor es un método atípico ya que no devuelve ningún valor. Esto es completamente diferente de un valor **void** en que un método no devuelve nada pero en un método ordinario siempre se tiene la posibilidad de elegir entre devolver un valor o no. Los constructores no devuelven nada y, además, no es posible modificar este comportamiento. Si se pudiese devolver un valor y se pudiese elegir cual, el compilador debería saber que hacer con ese valor.

Sobrecarga de métodos

Un aspecto importante en cualquier lenguaje de aplicación es el uso de los nombres. Cuando se crea un objeto se le asigna un nombre a una zona de almacenamiento. El uso de nombres describir un sistema permite crear programas que son más fáciles de entender y modificar. Es parecido a escribir en prosa, el objetivo es comunicarse con los lectores.

Se usan nombres referirse a los objetos y a los métodos. La elección de nombres

adecuados facilita la comprensión del código uno mismo y los demás.

Cuando se intenta trasladar el concepto de matiz en el lenguaje humano a un lenguaje de programación surgen problemas. A menudo una simple palabra tiene más de un posible significado, es decir, esta *sobrecargada* . Esto resulta especialmente útil cuando las diferencias entre los posibles significados son solo de detalle. Decimos "lavar la camisa", "lavar el coche" o "lavar al perro". Sería absurdo tener que decir "camisa-lavar la camisa", "coche-lavar el coche" o "perro-lavar al perro" porque el que escucha no necesita que se haga ninguna distinción entre las diferentes acciones. La mayoría de los lenguajes humanos son redundantes por lo que es posible reconstruir el significado de una frase incluso cuando se han perdido unas pocas palabras; podemos deducir el significado por el contexto.

La mayoría de los lenguajes de programación (C en particular) obligan a que cada función tenga un nombre único. Por lo tanto no se puede tener una función **print ()** imprimir enteros y otra función **print()** números en coma flotante. Cada función debe tener un nombre único.

En Java (y en C++) hay otra razón que obliga a la sobrecarga en los nombres de los métodos. Como el nombre del constructor debe se el mismo que el de la clase, solo existe un nombre posible él. Pero qué sucede si se desea poder construir un objeto en más de un modo diferente? Por ejemplo, si se diseña una clase que pueda inicializarse de un modo estándar o leyendo la información necesaria de un fichero, se necesitará dos constructores: uno sin argumentos (el constructor *predeterminado* , también llamado constructor *no-args*) y otro que tenga un argumento de tipo **String** que contenga el nombre del fichero. Ambos son constructores por lo que deben tener el mismo nombre: el nombre de la clase. Por lo tanto la *sobrecarga de métodos* es esencial que mismo nombre pueda se usado por dos procedimientos con argumentos diferentes. La sobrecarga de métodos es necesaria los constructores y en general es un mecanismo conveniente que puede usarse con cualquier método.

En el ejemplo siguiente muestra tanto constructores como métodos ordinarios sobrecargados:

```
//: c04:Sobrecargando.java
// Ejemplo de sobrecarga en constructores
// y metodos ordinarios.
import java.util.*;

class Arbol {

   int altura;
   Arbol() {
      prt("Plantando una semilla");
      altura=0;
```

```
   }
   Arbol(int i) {
      prt("Creando un arbol que mide "
          + i + " pies de alto");
      altura = i;
   }
   void info() {
      prt("El arbol mide " + altura +
          " pies de alto");
   }
   void info() {
      prt(s + ": El arbol mide "
          + height + " pies de alto");
   }
   static void prt(String s) {
      system.out.println(s);
   }
}

public class Sobrecargando {
   public static void main (String[] args) {
      for (int i = 0; i < 5; i++) {
         Arbol a = new Arbol(i);
         a.info();
         a.info("metodo sobrecargado");
      }
      // Constructor sobrecargado:
      new Arbol();
   }
} ///:~
```

Se puede crear un **Arbol** a partir de una semilla o a partir de un árbol joven criado en un invernadero. que ésto sea posible hay dos constructores; uno sin argumentos (a los constructores sin argumentos se les denomina *constructores predeterminados*) y otro que tiene por argumento la altura inicial.

También se puede querer llamar al método **info()** de más de una manera.Por ejemplo con un argumento **String** , si de desea presentar un mensaje extra, o sin argumentos ,si no se tiene nada que añadir. Resultaría extraño tener que usar nombres diferentes dos cosas que representan el mismo concepto. La sobrecarga de métodos nos permite usar el mismo nombre ambos.

Decidiendo entre métodos sobrecargados

Si es posible tener varios métodos con el mismo nombre Ücómo puede Java determinar que método ha de ser invocado? La regla es sencilla: cada método debe tener una lista de argumentos diferente.

Si se reflexiona sobre ello, tiene sentido: de qué otra manera podría el programador indicar la diferencia entre los dos métodos si no es mediante la lista de argumentos?

Se puede distinguir entre dos métodos incluso por el orden de los argumentos (Sin embargo, no es conveniente usar esta opción ya que conduce a un código difícil de mantener).

```
//: c04:OrdenSobrecarga.java
// Sobrecarga de metodos basada en el
// orden de los argumentos.

public class OrdenSobrecarga {
   static void print(String s, int i) {
      System.out.println(
         "String: " + s +
         " , int: " + i);
   }
   static void print(int i, String s) {
      System.out.println(
         "int: " + i +
         ", String: " + s);
   }
   public static void main (String[] args) {
      print("Primero String", 11);
      print(99, "Primero int");
   }
} ///:~
```

Los dos métodos **print()** tienen los mismos argumentos, pero en diferente orden lo que les hace diferentes.

Sobrecarga con primitivas

Una primitiva puede ser elevada de modo automático a un tipo más general lo que puede dar lugar a una cierta confusión cuando se combina con la sobrecarga de métodos. El ejemplo siguiente muestra que es lo que ocurre cuando una primitiva es usada como argumento en un método sobrecargado.

```
//: c04:SobrecargaConPrimitivas.java
// Promocion de primitivas y sobrecarga.

public class SobrecargaConPrimitivas {
   // La conversion automatica no se aplica
   // a booleanos
```

```
static void prt(String s) {
   System.out.println(s);
}

void f1(char x) { prt("f1(char)"); }
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }

void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }

void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }

void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }

void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }

void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }

void f7(double x) { prt("f7(double)"); }

void TestValConst() {
   prt("Probando con 5");
   f1(5);f2(5);f3(5);f4(5);f6(5);f7(5);
}
void testChar() {
   char x = 'x';
   prt("Argumento char:");
   f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
}
void testByte() {
   byte x = 0;
```

```
      prt{"Argumento byte:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testShort() {
      short x = 0;
      prt{"Argumento short:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testInt() {
      int x = 0;
      prt{"Argumento int:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testLong() {
      long x = 0;
      prt{"Argumento long:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testFloat() {
      float x = 0;
      prt{"Argumento float:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   void testDouble() {
      double x = 0;
      prt{"Argumento double:");
      f1(x);f2(x);f3(x);f4(x);f6(x);f7(x);
   }
   public static void main(String[] args) {
      SobrecargaConPrimitivas p =
        new SobrecargaConPrimitivas();
      p.testValConst();
      p.testChar();
      p.testByte();
      p.testShort();
      p.testInt();
      p.testLong();
      p.testFloat();
      p.testDouble();
   }
} ///:~
```

Si se examina la salida que produce este programa se puede ver que el valor constante 5 es tratado como un **int** , por lo tanto se usa el método sobrecargado con argumento **int** siempre que este disponible. En el resto de los casos, si el tipo que es inferior que el del argumento del método, es promovido al tipo superior. El tipo **char** se comporta de un modo ligeramente diferente ya que, cuando no se puede encontrar un método que tenga argumento **char** , es promovido a **int** .

Un variación del ejemplo anterior muestra que ocurre cuando el argumento es de un tipo superior al esperado por un método sobrecargado:

```
//: c04:Degradado.java
// Degradado de primitivas y sobrecarga.

public class Degradado {
   static void prt(String s) {
      System.out.println(s);
   }

   void f1(char x) { prt("f1(char)"); }
   void f1(byte x) { prt("f1(byte)"); }
   void f1(short x) { prt("f1(short)"); }
   void f1(int x) { prt("f1(int)"); }
   void f1(long x) { prt("f1(long)"); }
   void f1(float x) { prt("f1(float)"); }
   void f1(double x) { prt("f1(double)"); }

   void f2(char x) { prt("f2(char)"); }
   void f2(byte x) { prt("f2(byte)"); }
   void f2(short x) { prt("f2(short)"); }
   void f2(int x) { prt("f2(int)"); }
   void f2(long x) { prt("f2(long)"); }
   void f2(float x) { prt("f2(float)"); }

   void f3(char x) { prt("f3(char)"); }
   void f3(byte x) { prt("f3(byte)"); }
   void f3(short x) { prt("f3(short)"); }
   void f3(int x) { prt("f3(int)"); }
   void f3(long x) { prt("f3(long)"); }

   void f4(char x) { prt("f4(char)"); }
   void f4(byte x) { prt("f4(byte)"); }
   void f4(short x) { prt("f4(short)"); }
   void f4(int x) { prt("f4(int)"); }

   void f5(char x) { prt("f5(char)"); }
   void f5(byte x) { prt("f5(byte)"); }
   void f5(short x) { prt("f5(short)"); }

   void f6(char x) { prt("f6(char)"); }
   void f6(byte x) { prt("f6(byte)"); }

   void f7(char x) { prt("f7(char)"); }

   void testDouble() {
      double x = 0;
```

```
      prt("Argumento double:");
      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) {
      Degradado p = new Degradado();
      p.testDouble();
   }
} ///:~
```

En este caso hay métodos que tienen argumentos con tipos inferiores a los de los argumentos reales. Cuando se pasa a un método un valor de un tipo superior su argumento formal es necesario reducirlo al tipo aceptado por el método. ello se ha de poner entre paréntesis el nombre del tipo del argumento del método. Si no se hace así el compilador generará un mensaje de error.

Se debe tener muy en cuenta que se trata de una *conversión a un tipo inferior* por lo que se puede perder informacíon.Por ese motivo el compilador fuerza a que la transformación se haga de modo explícito.

Sobrecarga por valores de retorno

Es común preguntarse "?Por qué solo los nombres de las clases y la lista de argumentos? ?Por qué no distinguir entre métodos también usando el tipo de valor que devuelven? Por ejemplo los dos métodos siguientes tienen el mismo nombre y la misma lista de argumentos y pueden ser distinguidos fácilmente:

```
void f() {}
int f() {}
```

Esto funciona cuando el compilador puede determinar unívocamente el significado a partir del contexto, como en **int x=f()** . Sin embargo, es posible invocar un método e ignorar el valor que devuelve, lo que frecuentemente se llama *llamar al método por su efecto lateral* , ya que no lo importante no es el valor que el método devuelve sino otros efectos que produce la invocación. Por lo tanto si se invoca el método de la siguiente forma:

```
f();
```

Java no podría determinar cual de los dos métodos **f()** debe ser llamado. Nadie que leyese el código podría diferenciar entre uno y otro. Es por este tipo de problemas que no se puede usar el tipo de los valores devueltos por los métodos

como un criterio distinguir entre métodos sobrecargados.

Constructores predeterminados

Como ya se dijo más arriba el constructor predeterminado (también conocido como constructor "no-args") no tiene argumentos y se utiliza construir "objetos vainilla". Si se crea una clase que no tenga constructores, el compilador generará automáticamente un constructor predeterminado. Por ejemplo:

```
//: c04:CosntructorPredeterminado.java

class Pajaro {
   int i;
}

public class ConstructorPredeterminado {
   public static void main(String[] args) {
      Pajaro nc = new Pajaro(); // predeterminado!
   }
} ///:~
```

La línea:

```
new Pajaro();
```

crea un objeto nuevo y llama al constructor predeterminado, aunque no se haya definido explícitamente ninguno. Sin él no habría ningún método que invocar crear el objeto. Sin embargo, si se define algún constructor (con o sin argumentos) el compilador *no* genera ninguno por si mismo:

```
class Arbusto {
   Arbusto(int i) {}
   Arbusto(double d) {}
}
```

Si se intenta:

```
new Arbusto();
```

el compilador se quejará diciendo que no puede encontrar un constructor adecuado. Es como si, cuando no se incluye constructores, el compilador dijese: "Vas a necesitar *algún* constructor, así que déjame que prepare uno ti". Pero si se define algún constructor el compilador dirá: "Tienes un constructor, así que sabes lo que estás haciendo. Si no has definido un constructor predeterminado es por que no quieres que haya uno".

La palabra clave this

Si se tiene dos objetos, **a** y **b** , del mismo tipo cabe preguntarse como es posible invocar el mismo método **f()** cada uno de ellos:

```
class Banana { void f(int i) { /*...*/ } }
Banana a = new Banana(); b = new Banana();
a.f(1);
b.f(2);
```

Dado que solamente existe un método **f()** , éste debe ser capaz de determina cual de los dos objetos **a** o **b** ha sido invocado.

que sea posible escribir el código usando una sintaxis "orientada a objetos" en la que "se envía un mensaje a un objeto", el compilador debe hacer un poco de trabajo entre bastidores. Existe un argumento secreto que es pasado al método **f ()** en primer lugar. Este argumento es una referencia al objeto que se esta manipulando. Por lo tanto, las llamadas anteriores se traducen en algo así como:

```
Banana.f(a,1);
Banana.f(b,2);
```

Esta transformación se realiza internamente.No se puede escribir y compilar las dos líneas anteriores pero dan una buena idea de lo que ocurre.

Supongamos que dentro de un método se quiere obtener una referencia al objeto que se esta manipulando. Ya que el compilador ha pasado la referencia en *secreto* no hay ningún identificador que la contenga. Sin embargo, existe una palabra clave ello: **this** . La palabra clave **this** - que puede ser usada únicamente dentro de un método - devuelve la referencia al objeto el que el método ha sido invocado. Se puede usar esta referencia como cualquier otra referencia a un objeto. Conviene observar que, si se invoca un método de una clase desde otro método de la misma clase, no es necesario usar **this** ya que se puede invocar el método directamente. La referencia **this** es usada de modo automático. Por lo tanto es posible escribir:

```
class Melocoton {
   void coger() { /*...*/ }
   void pelar() { coger(); /*...*/ }
}
```

Dentro de **pelar()** se *puede* poner **this.coger()** pero no es necesario. El compilador lo hace automáticamente. La palabra clave **this** se usa únicamente en aquellos casos especiales en los que es necesita usar una referencia explícita al objeto que se está manipulando. Por ejemplo, es usando frecuentemente en la cláusula **return** cuando se quiere devolver una referencia al objeto en uso:

```
//: c04:Hoja.java
// Ejemplo simple de uso de la palabra clave "this"

public class Hoja {
   int i;
   Hoja incremento() {
      i++;
      return this;
   }
   void print () {
      System.out.println("i = " + i);
   }
   public static void main(String[] args) {
      Hoja x = new Hoja();
      x.incremento().incremento().incremento().print();
   }
} ///:~
```

Como **incremento()** devuelve la referencia al objeto en uso usando la palabra clave **this** se pueden realizar varias operaciones sobre el mismo objeto de un modo sencillo.

Invocación de constructores dentro de constructores

En algunas ocasiones, cuando una clase tiene varios constructores, resulta conveniente llamar a un constructor desde dentro de otro no tener que duplicar el código. La palabra clave **this** permite realizarlo.

Normalmente, **this** se usa con el significado de "este objeto" o "el objeto actual" y produce una referencia al objeto con el que se está trabajando. Dentro de un constructor **this** toma otra significado cuando se le añade una lista de argumentos. En este caso representa una llamada explícita al constructor que tiene los mismos argumentos que los incluidos en la lista. Proporciona por lo tanto

un medio directo de invocar otros constructores:

```
//: c04:Flor.java
// Invocacion a constructores usando "this."

public class Flor {
   int contPetalos = 0;
   String s = new String("null");
   Flor (int petalos) {
      contPetalos = petalos;
      System.out.println(
        "Constructor con arg int solo, contPetalos= "
        + contPetalos);
   }
   Flor(String ss) {
      System.out.println(
        "Constructor con arg String solo, s=" + ss);
      s=ss;
   }
   Flor(String s, int petalos) {
     this(petalos);
//!     this(s);  // No se puede hacer dos veces.
      this.s = s; // Otro uso de "this"
      System.out.println("args String y int");
   }
   Flor() {
      this("hi",47);
      System.out.println(
        "constructor predeterminado (sin argumentos)");
   }
   void print() {
//!      this(11);  // Fuera de un constructor!
      System.out.println(
      "contPetalos = " + contPetalos + " s = " + s);
   }
   public static void main(String[] args) {
      Flor x = new Flor();
      x.print();
   }
} ///:~
```

El constructor **Flor(String s, int petalos)** muestra como se puede invocar un constructor usando **this** y que no se puede hacer dos veces. Además la llamada al constructor debe ser la primera sentencia o el compilador generará un error.

El ejemplo muestra también otra situación en la que se usa **this** . Como el nombre del argumento **s** y el nombre del dato miembro **s** es el mismo, se produce una

ambigüedad. Esta ambigüedad puede ser resuelta usando **this.s** referirse al dato miembro. Es frecuente el uso de esta forma en Java y aparece en numerosos lugares dentro de este libro.

En **print()** se ve que el compilador no permitirá que se llame a un constructor desde un método que no sea él mismo un constructor.

El significado de static

Con la palabra clave **this** en mente, se puede comprender mejor que significa que un método sea **static** . los métodos **static** no existe **this** . No es posible invocar un método no- **static** desde un método **static** (aunque lo contrario sí es posible) y se puede invocar un método **static** usando la clase, sin objeto alguno. De hecho, los métodos **static** son existen principalmente eso. Es el equivalente a un función global (de C), pero las funciones globales no están permitidas en Java. Definiendo un método **static** dentro de una clase se puede acceder a otros métodos **static** y a campos **static**

Hay quien opina que los métodos **static** no son orientados a objeto ya que significan lo mismo que una función global; con un método **static** no es posible enviar un mensaje a un objeto ya que no existe una referencia **this** asociada. Es un argumento razonable y si se empieza a usar *con frecuencia* es recomendable replantearse la estrategia que se está usando. Sin embargo, los métodos y campos **static** son prácticos y en ocasiones son realmente necesarios. Luego, si son o no programación orientada a objetos en estado puro, es una cuestión que se puede dejar a los teóricos. En realidad incluso Smalltalk tiene algo equivalente con sus "métodos de clase".

Purgado: finalización y recogida de basura

Los programadores son conscientes de la importancia de la inicialización, pero a menudo se olvidan de la importancia del purgado. Después de todo, ?quién necesita deshacerse de un **int** ?. Pero cuando se trabaja con librerías, la política de "déjalo estar" cuando se ha terminado con un objeto no es siempre segura. Por supuesto, Java tiene un recogedor de basura que se encarga de reclamar la memoria que los objetos ya no usan. Consideremos ahora un caso muy extraño. Supongamos que nuestros objetos reservan memoria "especial" sin usar **new** . El recogedor de basura solo sabe como liberar la memoria que ha sido reservada usando **new** , por lo que no sabrá como liberar la memoria "especial" de los objetos. estos casos Java proporciona un método llamado **finalize()** que puede ser definido dentro de una clase. Veamos como *se supone* que este método trabaja. Cuando el recogedor de basura está listo liberar la zona de memoria de memoria usada por un objeto invoca en primer lugar su **finalize()** y solo en el siguiente ciclo de recogida de basura reclamará la memoria del objeto. Por lo tanto si se usa **finalize()** es posible realizar alguna operación de limpieza importante durante *la recogida de basura* .

Esto puede ser un problema por que algunos programadores, especialmente para aquellos que vienen de C++, tienden a confundir **finalize()** con el destructor de C++: una función que es invocada automáticamente cuando un objeto es destruido. Es importante entender desde ahora mismo las diferencias entre C++ y Java en este punto: en C++ *los objetos son siempre destruidos* (en un programa sin errores) mientras que en Java los objetos no siempre terminan por ser arrojados a la basura. En otras palabras:

Recogida de basura no es lo mismo que destrucción

Si se recuerda esto es posible mantenerse al margen de los problemas. El enunciado anterior implica que, si es necesario hacer alguna cosa antes de que deje de ser necesario un objeto, ese "algo" debe ser hecho por uno mismo. Java no dispone de destructores o de cualquier otra construcción similar, por lo que es necesario crear métodos ordinarios que se encargue de esta tarea de purgado. Pongamos el ejemplo de un objeto se pinta a si mismo en la pantalla del ordenador al ser creado. Si la imagen no es borrada explícitamente de la pantalla, puede que el objeto nunca llegue a ser purgado del sistema. Solo si se introduce dentro de **finalize()** las instrucciones de borrado pertinentes, la imagen será borrada cuando el objeto sea enviado a la basura, de otro modo la imagen permanecerá pintada en la pantalla. Por lo tanto, el segundo punto a recordar es:

Es posible que los objetos no sean procesados por el recogedor de basura

Si un programa nunca llega a un punto en el que los recursos disponibles empiecen a escasear, el espacio ocupado por los objetos no es liberado. Si el programa termina y el recogedor de basura no ha liberado espacio para ninguno de los objetos del mismo, todo el espacio reservado por los objetos durante la ejecución del programa el liberado y devuelto al sistema operativo de una sola vez a la finalización del programa. Dado que la recogida de basura supone un trabajo extra que es necesario realizar, resulta adecuado evitarla cuando sea posible.

¿Para qué sirve finalize() ?

En este punto de la discusión, es posible pensar que **finalize()** no debería ser usado como un método de borrado de propósito general. ¿Qué tal es?

El tercer punto a recordar es:

La recogida de basura afecta solo a la memoria.

La única razón de exitir del recogedor de basura es recuperar la memoria que el programa ha dejado de utilizar. Por lo tanto cualquier actividad relacionada con la recogida de basura, en particular el método **finalize()** , debe ocuparse únicamente de la memoria y de su liberación.

Esto no significa que, si un objeto contiene otros objetos, **finalize()** debe eliminar explícitamente todos estos objetos: el recogedor de basura se encarga de liberar

la memoria asignada a cada objeto independientemente de como hayan sido creados. El método **finalize()** se necesita únicamente en aquellos casos especiales en los que un objeto reserve espacio para si de algún modo que no sea la creación de otros objetos. Pero si en Java todo es un objeto, ¿cómo puede ser esto posible?

Parece como si **finalize()** estuviese ahí para el caso en que, en el estilo de C, se reservase memoria usando un mecanismo diferente del normal en Java. Esto se puede hacer, principalmente, usando *métodos nativos* que permiten invocar desde Java código escrito en otro lenguaje de programación. (En el Apéndice B se puede encontrar un amplia descripción de los métodos nativos). De momento, C y C++ son los únicos lenguajes que pueden utilizarse con los métodos nativos pero, dado que desde C y C++ es realizar llamadas a subprogramas escritos en otros lenguajes, en realidad se pueden hacer llamadas a cualquier cosa. En un fragmento de código que no sea Java, se puede usar la familia de funciones **malloc()** de C para reservar espacio. A menos que se use luego la función **free()** el espacio nunca será liberado lo que producirá una pérdida de memoria (memory leak). Como **free()** es una función de C y C++, se necesita un método nativo para poder llamarla desde dentro de **finalize()** .

De la lectura de los párrafos anteriores se desprende que **finalize()** no se usa con mucha frecuencia. No es le lugar adecuando para purgar el sistema de los objetos que ya no están en uso. Entonces ¿donde debe hacerse este purgado?

El purgado es necesario

Para eliminar un objeto, el usuario de dicho objeto debe invocar un método de purgado cuando se desee hacerlo. Esto parece bastante claro pero se contradice con el concepto de destructor en C++. En C++ todos los objetos son destruidos. O mejor dicho: todos los objetos *deberian ser* destruidos. Si, en C++, un objeto es creado localmente (es decir, en la pila, lo que no es posible en Java), su destrucción tiene lugar en el punto en el que se cierra el ámbito de definición (scope) en que fue credo el objeto. Si, en cambio, el objeto fue creado usando **new** (como en Java) el destructor es invocado cuando el programador llama al operador **delete** de C++ (que no tiene equivalente en Java). Si el programador de C++ olvida incluir la llamada a **delete** el destructor nunca será invocado y se producirá una fuga de memoria; además los otros elementos del objetos no podrá ser eliminados nunca, Los errores de este tipo son muy difíciles de detectar.

Al revés que C++, Java no permite la creación de objetos locales: es necesario usar siempre **new** . Pero en Java no existen un "delete" que deba ser llamado para eliminar el objeto ya que cuenta con el recogedor de basura que se encarga de liberar el espacio previamente reservado. Por lo tanto de puede decir simplificando que Java no necesita destructores por que tiene un recogedor de basura. El lector comprobará, según avance en la lectura de este libro, que la presencia de un recogedor de basura no elimina la necesidad o utilidad de los destructores. (además nunca debería llamarse directamente **finalize()** , por que

ese no es el camino adecuado hacia una solución) Si se quiere realizar algún tipo de purgado, además de la liberación de espacio, se debe hacer de modo explícito llamando al método apropiado desde Java, lo que equivale a un destructor de C++ sin comodidad que éste proporciona.

Una de las cosas para las que puede ser útil **finalize()** , es para ver como funciona la recogida de basura. El ejemplo siguiente muestra que es lo que ocurre y resume las descripciones de la recogida se basura dadas hasta ahora:

```
//: c04:Basura.java
// Demostracion del funcionamiento del
// recogedor de basura y la finalizacion

class Silla {
   static boolean gcrun = false;
   static boolean f = false;
   static int creadas = 0;
   static int finalizadas = 0;
   int i;
   Silla() {
      i = ++creadas;
      if (creadas == 47)
         System.out.println("Creada 47");
   }

   public void finalize() {
      if(!gcrun) {
         // La primera que finalize es llamado:
         gcrun = true;
         System.out.println(
           "Empezando a finalizar despues de que " +
           creadas + " Sillas hayan sido creadas");
      }
      if(i == 47) {
         System.out.println(
           "Finalizando Silla #47, " +
           "Fijando indicador para detener la creacion de Sillas");
         f = true;
      }
      finalizadas++;
      if (finalizadas >= creadas)
         System.out.println(
           "Todas " + finalizadas + " finalizadas");
   }
}

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

```
      // Mientras no se fije el indicador,
      // hacer Sillas y Strings:
      while(!Silla.f) {
         new Silla();
         new String("Para gastar un poco de espacio");
      }
      System.out.println(
         "Despues de que todas las sillas han sido creadas:\n" +
         "Total creadas = " + Silla.creadas +
         ", total finalizadas = " + Silla.finalizadas);
       // Argumentos opcionales para forzar
       // recogida de basura y finalizacion
       if(args.length > 0) {
          if(args[0].equals("gc") ||
             args[0].equals("todo")) {
             System.out.println("gc():");
             System.gc();
          }
          if(args[0].equals("finalizar") ||
             args[0].equals("todo")) {
             System.out.println("runFinalization():");
             System.runFinalization();
          }
       }
       System.out.println("Adios!");
   }
} ///:~
```

El programa anterior crea muchos objetos **Silla** , y en algún momento el recogedor de basura comienza a ejecutarse, el programa detiene al creación de **Silla** s. Como el recogedor de basura puede ejecutarse en cualquier momento, no se puede saber con exactitud cuando arrancará. Por este motivo se usa el indicador **gcrun** para señalar si el recogedor de basura se ha comenzado a ejecutar ya. El segundo indicador **f** sirve para que **Silla** le diga a bucle en **main()** que debe dejar de crear objetos. Ambos indicadores son activados dentro de **finalize()** que es invocado durante la recogida de basura.

Otras dos variables **static** , **creadas** y **finalizadas** llevan la cuenta del número de **Silla** s creadas frente al de finalizadas por el recogedor de basura. Finalmente, cada **Silla** tiene su propia (no- **static**) **int i** , de modo que puede llevar la cuenta de que número hace dentro de la secuencia. Cuando la **Silla** número 47 es finalizada, el indicador **f** se le asigna el valor **true** para detener el proceso de creación de **Silla** .

Todo esto sucede en **main()** , dentro del bucle:

```
while(!Silla.f) {
```

```
        new Silla();
        new String("Para gastar un poco de espacio");
    }
```

El lector puede preguntarse como puede detenerse este bucle si no hay nada dentro de él que cambie el valor de **Silla.f** . Sin embargo, será **finalize()** quien lo haga con el tiempo, tras finalizar la **Silla** número 47.

El objeto **String** que se crea en cada iteración sirve tan solo para gastar un poco de espacio y acelerar la entrada del recogedor de basura, lo que ocurrirá en cuanto empiece a ponerse nervioso sobre la cantidad de memoria disponible.

Al lanzar el programa desde la línea de comandos es posible incluir un parámetro: "gc", "finalizar", o "todo". Con el argumento "gc" se llama al método **System.gc()** (para forzar la ejecución del recogedor de basura). Con "finalizar" se llama a **Systen.runFinalization()** que - en teoría - hará que sean finalizados todos los objetos que no lo estuviesen ya. Con el argumento "all" se llama a ambos métodos.

El comportamiento de este programa y la versión en la primera edición de este libro, ponen de manifiesto que todo el asunto de la recogida de basura y la finalizacion ha evolucionado y que, la mayor parte de esta evolución se ha producido en secreto. De echo, puede ser que el comportamiento del programa haya cambiado de nuevo cuando el lector este leyendo esta líneas.

Si se llama a **System.gc()** , todos los objetos son finalizados. Éste no era necesariamente el caso con la implementaciones anteriores del JDK, aunque la documentación insista en lo contrario. Además, se puede comprobar que no hay ninguna diferencia aparente si la llamada se hace a **System.runFinalization()**

Sin enbargo, se puede observar que, solamente si **System.gc()** es llamado después de que todos los objetos estén creados y descartados, serán invocados todos los finalizadores. Si no se llama a **System.gc()** solo seran finalizados algunos de los objetos creados. En Java 1.1 se introdujo el método **System.runFinalizersOnExit()** que forzaba a los programas a invocar a todos los finalizadores antes de termina. Sin embargo, el diseño resulto incorrecto y el método fue retirado (deprecated). Este es otro indicio más de que los diseñadores de Java seguían trabajado para intentar resolver el problema de la recogida de basura y la finlización. Solo cabe esperar que las cosas funcionen en Java2.

El ejemplo precedente muestra que el compromiso de que los finalizadores son ejecutados siempre, continúa siendo cierta, pero únicamente si se fuerza explícitamente. Si no se fuerza la llamada a **System.gc()** , se obtiene una salida como la siguiente:

```
Creada 47
```

```
Empezando a finalizar despues de que 3486 Sillas hayan sido creadas
Finalizando Silla #47, Fijando indicador para detener la creacion de S
Despues de que todas las sillas hayan sido creadas:
Total creadas = 3881, total finalizadas = 2684
Adios!
```

Por lo tanto, no todos los finalizadores son llamados cuando el programa termina. Si **System.gc()** es llamado, finalizará y destruirá todos los objetos que ya no estén en uso en ese momento.

Hay que recordar que ni la recogida de basura ni la finalización están garantizadas. Si la Máquina Virtual Java (JVM - Java Virtual Machine) no corre el riesgo de quedarse sin menoria, no malgastará el tiempo recuperando memoria a través de la recogida de basura.

Death Condition

En general, no se puede confiar en que **finalize()** sea llamado, y se deben crear funciones independientes de "purgado" que han de ser llamadas explícitamente. Luego, parece que **finalize()** es útil únicamente en extraños purgados de memoria que la mayoría de los programadores nunca usan. Existe, sin embargo, un uso muy interesante de **finalize()** que no depende en que sea llamada todas las veces. Este uso es la comprobación de la *death condition* de un objeto.

En el momento en que un objeto deja de ser de interés - cuando el está listo para ser eliminado - debe encontrarse en un estado tal que la memoria a él asignada pueda ser liberada de un modo seguro. Por ejemplo, si el objeto representa un fichero abierto, ese fichero debería ser cerrado por el programador antes de que el objeto sea enviado a la basura. Si alguno de los componentes del objeto no es eliminado adecuadamente se produce un error en el programa que puede ser muy difícil de encontrar. El valor de **finalize()** puede ser usado para descubrir este tipo de situaciones, aún cuando no sea llamado en todas la ocasiones. Si una de la finalizaciones revela el error, el problema queda al descubierto, que es precisamente lo que se pretende.

El siguiente ejemplo muestra como usar **finalize()** con esta finalidad:

```
//: c04:DeathCondition.java
// Uso de finalize() para detectar un objeto
// que no ha sido eliminado correctamente.

class Libro {
   boolean retirado = false;
   Libro(boolean retirar) {
      retirado = retirar;
   }
```

```
   void ingresar() {
      retirado =  false;
   }
   public void finalize() {
      if (retirado)
         System.out.println("Error: libro retirado");
   }
}

public class DeathCondition {
   public static void main(String[] args) {
      Libro novela = new Libro(true);
      // Purgado correcto
      novela.ingresar();
      // Desprecia la referencia, olvida limpiar
      new Libro(true);
      // Fuerza la recogida de basura y la finalizacion
      System.gc();
   }
} ///:~
```

En este caso se supone que todo **Libro** tiene que ser ingresado antes que pueda ser enviado a la basura. Pero en **main()** el programador no registra un de los libros por error. Si **finalize()** no verificase la death condition, este error sería muy difícil de descubrir.

System.gc() se usa para forzar la finalización (lo que resulta conveniente durante el desarrollo para facilitar la depuración de errores). Pero, incluso aunque no estuviese, es más que probable que el **Libro** errante termine por aparecer durante alguna de las ejecuciones del programa (supuesto que se llegue a reservar memoria suficiente para que el recogedor de basura entre en acción).

El funcionamiento del recogedor de basura

Para aquellos que vengan de lenguajes de programación en los que la creación de objetos en el heap es una operación costosa, pueden pensar que el esquema de crear todo (salvo las primitivas) en heap adoptado por Java, es ineficiente.

Resulta, sin embargo, que la existencia del recogedor de basura puede tener un impacto significativo en la rapidez a la que se crean los objetos. Puede sonar un poco raro al principio que la liberación de memoria afecte a reserva de espacio, pero es como trabajan algunas JVMs y significa, al final, que la creación de objetos en el heap puede ser tan rápida en Java como la creación de objetos en la pila en otros lenguajes de programación.

Por ejemplo, en C++ el heap puede ser visto como un jardín en el que cada objeto se ocupa de cada su propio trozo de césped. Esta propiedad puede que ser abandonada y, con el tiempo, reutilizada por otro objeto. En varias JVMs, el heap

es bastante diferente: se asemeja más a una cinta transportadora que se mueve hacia adelante cada vez que se crea un objeto nuevo. Esto implica que la creación de objetos es notablemente rápida. El "puntero del heap" simplemente se mueve hacia adelante, adentrándose en territorio virgen, por lo que, a final de cuentas, es lo mismo que la reserva de espacio en la pila con C++ (Naturalmente, hay un pequeño trabajo extra que realizar para mantener la información acerca de la ocupación en el heap, pero es nada en comparación com andar buscando espacio libre por el heap).

El heap, sin embargo, no es una cinta transportadora y, si se manejase como tal tarde o temprano el programa empezaría a paginar (lo que influye muy negativamente en el rendimiento) y acabaría por agotar la memoria disponible. El truco está en que el recogedor de basura, al mismo tiempo que elimina la basura, compacta todos los objetos en el heap con lo que el "puntero del heap" es desplazado hacia el comienzo de la cinta transportadora y evitando el fallo de página. El recogedor de basura reordena las cosas y hace posible que el modelo del heap infinito de alta velocidad pueda ser utilizado para la asignación de espacio.

Para entender como funciona el recogedor de basura (GC del inglés, "garbage collector"), es necesario conocer mejor como trabajan los diferentes algoritmos de recogida de basura. Una técnica simple pero lenta de recogida de basura es el conteo de referencias (reference counting). En este modelo cada objeto debe contener un contador de referencias que se incrementa cada vez que se asigna una nueva referencia a dicho objeto. Cada vez que el programa abandona el ámbito de validez de una referencia o se le asigna el valor **null** , el contador de referencias del objeto asociado se decrementa en una unidad. La gestión de los contadores de referencias supone por lo tanto una pequeña, pero constante, carga adicional que se entiende durante toda la vida del programa. El recogedor de basura recorre toda la lista de objetos y, cuando encuentra uno cuyo contador de referencias es cero, libera el espacio reservado para dicho objeto. Esta técnica, sin embargo, no resulta adecuada para identificar conjuntos de objetos que contenga referencias de unos a otros de una manera circular de modo que, aun siendo solo basura, sus contadores de referencias sean distintos de cero. La identificación de estos grupos autoreferenciados requiere del recogedor de basura un trabajo extra considerable. El conteo de referencias se usa con frecuencia para describir un tipo concreto de recogida de basura pero probablemente no se use en ninguna implementación de la JVM.

Otros esquemas de recogida de basura más eficientes no usan la técnica de conteo de referencias. Es su lugar usan la idea de que cualquier objeto que permanezca vivo debe, en última instancia, ser rastreado hacia atrás hasta encontrar una referencia en la pila o en la zona de almacenamiento estático. La cadena puede extenderse a través de varias capas de objetos. Por lo tanto, es posible identificar que objetos están vivos sin más que seguir las referencias que se encuentran en la pila y en la zona de almacenamiento estático. Por cada referencia que se encuentre, hay que entrar dentro del objeto apuntado por ella, identificar todas las

referencias contenidas en *ese* en ese objeto, y entrar en los objetos apuntados por estas referencias, y así sucesivamente hasta recorrer por completo red de objetos generada a partir de las referencias en la pila y en la zona de almacenamiento estático. Cada objeto que se atraviese en este proceso debe ser un objeto vivo. En este caso, los conjuntos autoreferenciados no presentan ningún problema; simplemente no aparecen por ningún lado, y son, por lo tanto, enviados automáticamente a la basura.

En el enfoque descrito aquí, la JVM utiliza un esquema *adaptativo* de recogida de basura: lo que haga con los objetos vivos que encuentre dependerá de la variante que se encuentre activa en ese momento. Una de estas variantes es *para-y-copiar (stop-and-copy)* . Esto significa que - por razones que se harán evidentes más adelante - el programa es parado en primer lugar (por lo tanto no se trata de un esquena de recogida de basura en segundo plano). A continuación, cada objeto vivo que se encuentre en el heap es copiado en otro heap, con lo que la basura se queda en el primero. Además, los objetos son copiados en el nuevo heap uno a continuación del otro con lo que se optimiza de paso la utilización del espacio (y permitiendo así que los objetos que se creen puedan ser simplemente añadidos al final del heap como ya se dijo más arriba)

Naturalmente, cuando se mueve un objeto de un lugar a otro, todas las referencias que apuntan al (es decir esa *refencia*) objeto deben ser actualizadas. Las referencias que van del heap????? o la zona de almacenamiento estático al objeto pueden ser actualizadas inmediatamente; pero puede que existan otras referencias, apuntando a este objeto, que serán encontradas más tarde durante el paseo. Estas referencias son corregidas según se van encontrando (el lector puede imaginar una tabla conteniendo la correspondencia entre las direcciones nuevas y antiguas)

Hay dos razones por las que estos "recogedores por copia" (copy collectors") resultan ineficientes. La primera es que se necesitan dos heaps y copiar la información de uno a otro continuamente, por lo que hay mantener dos veces mas memoria de la que es realmente necesaria. Algunas JVMs tratan de aliviar el problema realizando las asignaciones de espacio en el heap por segmentos, según va siendo necesario, y copiando de un segmento a otro.

La segunda razón tiene que ver con la copia de un heap al otro. Una vez que el programa se ha estabilizado puede ser que genere muy poca basura o incluso que no genere basura en absoluto. Aún así, el recogedor por copia continuará todo el contenido de la memoria de un lugar a otro. Para evitar esta situación, algunas JVMs son capaces de detectar que no se está generando basura nueva y cambiar su modo de funcionamiento (de aquí lo de *adaptativa*). El segundo modo de trabajo recibe el nombre de *marcar y barrer (mark and sweep)* y fue el método usado por Sun en las primeras versiones de su JVM. Este método, usado en condiciones generales, es un poco lento pero, si se sabe que la cantidad de basura que se está generando es pequeña, es bastante rápido.

Marcar y barrer sigue la misma lógica de comenzar en la pila y en la zona de

almacenamiento estático y seguir todas la referencias para encontrar todos los objetos vivos. Cada objeto vivo encontrado se marca activando un indicador dentro del objeto. Solo cuando el proceso de marcado ha finalizado, tiene lugar el barrido, en el que los objetos muertos son eliminados. En cualquier caso, no se realiza copia de objetos, como con el método anterior, y cuando el recogedor de basura decide compactar el heap, lo hace desplazando los objetos dentro del heap.

El nombre "parar-y-copiar" hace referencia a hecho de que este tipo de recogida de basura no se realiza en segundo plano. El programa ha de ser detenido mientras el GC cumple con su misión. En la documentación de SUN se puede encontrar multitud de referencias en la que se habla de la recogida de basura como un proceso de prioridad baja que se ejecuta en segundo plano. Sin embargo, el GC nunca ha sido implementado de ese modo, al menos en las primeras versiones de la JVM de Sun. El recogedor de basura de Sun se ejecutaba siempre que la cantidad de memoria disponible alcanzaba un mínimo. Además el esquema marcar-y-barrer requiere que el programa sea detenido.

Como ya se dijo mas arriba, en la JVM que se está describiendo aquí, la memoria se asigna en bloques grandes. Cuando se crea un objeto grande, recibe su propio bloque. En un sistema en el que se aplique la política parar-y-copiar de un modo estricto, hay que copiar cada objeto vivo del heap original al nuevo antes de que se puedan liberar la memoria asignada el primero; esto implica grandes cantidades de memoria. Los bloques permiten al CG copiar los objetos vivos que va recogiendo en bloques muertos. Cada bloque tiene un *numero de generación* para llevar la cuenta de si está vivo o no. Normalmente, solo los bloques creados desde le ultima ejecución del GC son compactados; con el resto de los bloques se aumenta su numero de generación si se ha hecho alguna referencia a ellos. Esta política resulta adecuada para el caso en que se tenga muchos objetos de vida breve. Periódicamente, se realiza un barrido completo; los objetos grandes no son copiados (tan solo sus números de generación son actualizados) y los bloques con objetos pequeños son copiados y compactados. La JVM monitoriza el rendimiento del GC; si detecta que este caído bajo un cierto limite, debido a que la mayoría de los objetos tienen un tiempo de vida largo, cambia al modo marcar-y-barrer; si el heap comienza a estar fragmentado, cambia de nuevo al modo parar-y-copiar. Por este motivo se dice que el esquema es adaptativo, por lo que al final se tiene el bonito nombre de "adaptativo generacional parar-y-copiar marcar-y-barrer".

Hay varias posibilidades de optimizar el proceso de la recogida de basura dentro de la JVM. Una de las más importantes está relacionada con el funcionamiento del cargador y el compilador Just-In-Time (JIT). Cuando una clase tiene que ser cargada en el sistema (normalmente cuando se va a crear el primer objeto de la clase), ha de localizarse el fichero **.class** y código de bytes tiene que ser cargado en memoria. En ese momento, se puede compilar, usando el JIT, todo el código pero hay dos inconvenientes. El primero es que se introducen pequeños retardos que pueden combinarse unos con otros relentizando, al final, la ejecución del programa. El segundo inconveniente es que aumenta el tamaño de los ejecutables (el código en bytes en mucho más compacto que el código que genera el JIT) lo

que puede provocar que el programa empiece a paginar reduciendo drásticamente la velocidad de ejecución. Una estrategia alternativa es la *evaluacion débil (lazy evaluation)* que consiste en no compilado por el JIT hasta que no sea necesario. De este modo, el código que no llegue nunca a ser ejecutado, no será compilado por el JIT.

Inicialización de miembros

Java hace todo lo necesario para garantizar que las variables son inicializadas correctamente antes de que sean usadas. En el caso de que la variable esté definida localmente dentro de un método, el compilador generará un mensaje de error si no ha sido inicializada explícitamente. Por ejemplo, al intentar compilar:

```
void f() {
   int i;
   i++;
}
```

se obtiene un mensaje de error diciendo que **i** no pudo ser inicializada. El compilador podría asignar a **i** un valor por defecto, pero lo más probable es que se trate de un error del programador que quedaría oculto si se asignase un valor por defecto. Es más fácil localizar un error obligando al programador a inicializar las variables.

Las cosas son un poco diferentes si la primitiva es un dato miembro. Como cualquier método puede iniciar o use el dato, no seria practico obligar al usuario a inicializarlo con un valor adecuado antes de utilizarlo. Pero, dejar que el dato contuviese basura es peligroso, así que Java se encarga de que cada dato miembro de la clase reciba un valor inicial. El siguiente ejemplo muestra cuales son estos valores:

```
//: c04:ValoresIniciales.java
// Muestra los valores iniciales por defecto.

class Medida {
   boolean t;
   char c;
   byte b;
   short s;
   int i;
   long l;
   float f;
   double d;
   void print() {
```

```
      System.out.println(
         "Tipo          Valor Inicial\n" +
         "boolean       " + t + "\n" +
         "char          [" + c + "] " + (int)c + "\n" +
         "byte          " + b + "\n" +
         "short         " + s + "\n" +
         "int           " + i + "\n" +
         "long          " + l + "\n" +
         "float         " + f + "\n" +
         "double        " + d);
   }
}

public class ValoresIniciales {
   public static void main(String[] args) {
      Medida d = new Medida();
      d.print();
      /* En este caso se puede poner tambien:
      new Medidas().print();
      */
   }
} ///:~
```

El programa produce la siguiente salida:

```
Tipo          Valor Inicial\n" +
boolean       false
char          [ ] 0
byte          0
short         0
int           0
long          0
float         0.0
double        0.0
```

El valor de **char** es cero, que aparece como un espacio.

Más adelante se verá que cuando se define una referencia a un objeto dentro de una clase sin inicializarla con un objeto, la referencia toma el valor especial **null** (que es una palabra clave de Java).

El lector puede comprobar que, aunque no se especifique ningún valor, todas las variables son inicializadas. Por lo tanto, no hay peligro de trabajar con variables no inicializadas.

Especificando la inicialización

¿Qué ocurre si se desea dar un valor inicial a una variable? Una forma de hacerlo es simplemente asignar el en donde la variable es definida dentro de la clase (No es posible hacerlo en C++ aunque los programadores novatos lo intentan siempre). En el siguiente fragmento de código se han modificado la definición de los campos de la clase **Medida** para asignarles valores iniciales:

```
class Medida {
   boolean b = true;
   char c = 'x';
   byte b = 47;
   short s = 0xff;
   int i = 999;
   long l = 1;
   float f = 3.14f;
   double d = 3.14159;
   //. . .
```

No solo las primitivas, cualquier objeto puede ser inicializado del mismo modo. Si **Profundidad** es una clase, se puede insertar una variable e inicializarla como sigue:

```
class Medida {
   Profundidad o = new Profundidad();
   boolean b = true;
   // . . .
```

Si se intenta usar **o** sin haberle inicializado, se obtiene en tiempo de ejecución un error que recibe el nombre de *excepción(exception)* . Las excepciones son tratadas en captítulo 10)

También se puede incluir la llamada a un método en la inicialización:

```
class CInit {
   int i = f();
   //...
}
```

El método puede tener argumentos, pero los argumentos no pueden ser miembros de la clase que todavía no hayan sido inicializados. Luego el siguiente segmento

de código es correcto:

```
class CInit {
   int j = g(i);
   int i = f();
   /...
}
```

En este caso el compilador se quejará, apropiadamente, de que ha encontrado una forward reference, ya que el problema está en el orden de la inicialización y en como es compilado el programa.

Esta modo de inicialización es simple y directo. Tiene, por contra, la limitación de que *todos* los objetos del tipo **Medida** serán inicializados con los mismos valores.Unas veces es esto justamente lo que se quiere; en otras ocasiones se necesita más flexibilidad.

Inicialización en el constructor

El constructor se puede usar realizar la inicialización. Esta posibilidad, da una gran flexibilidad a la hora de programar, ya que se puede hacer llamadas a métodos y realizar otras acciones en tiempo de ejecución determinar los valores iniciales. De cualquier modo, hay que tener siempre en mente que de este modo no se evita la inicialización automática que tiene lugar inclusa antes de que se haya invocado el constructor. Así, en el ejemplo siguiente,

```
class Contador {
   int i;
   Contador() { i = 7; }
   // . . .
```

la variable **i** será inicializada primero con 0 y luego con 7. Este principio se aplica a todos los tipos primitivos y a las referencias a objetos, incluso en los casos en que se hace una inicialización explícita en el momento de la definición. Por esta razón, el compilador no obliga a inicializar elementos en el constructor o antes de que sean usados: la inicialización está, por lo tanto, garantizada.

Orden de inicialización

Dentro de una clase, el orden de inicialización queda determinado por el orden en que las variables son definidas dentro de la clase. Las definiciones de las variables pueden estar esparcidas por o entre la definición de los métodos pero las variables son inicializadas antes de que cualquier método pueda ser invocado; incluido el

constructor. Por ejemplo:

```
//: c04:OrdenDeInicializacion.java
// Muestra cual es el orden de inicialización.

// Cuando el constructor es llamado crea
// un objeto Etiqueta, se vera el mensaje:
class Etiqueta {
   Etiqueta(int marcador) {
      System.out.println("Etiqueta(" + marcador + ")");
   }
}

class Tarjeta {
   Etiqueta t1 = new Etiqueta(1); // Antes del constructor
   Tarjeta() {
      // Indica que estamos dentro del constructor:
      System.out.println("Tarjeta()");
      t3 = new Etiqueta(33);      // Reinicializacion de t3
   }
   Etiqueta t2 = new Etiqueta(2); // Despues de constructor.
   void f() {
      System.out.println("f()");
   }
   Etiqueta t3 = new Etiqueta(3); // Al final.
}

public class OrdenDeInicializacion {
   public static void main(String[] args) {
      Tarjeta t = new Tarjeta();
      t.f();   // Demuestra que se ha completado el constructor
   }
} ///:~
```

En **Tarjeta** las definiciones de los objetos **Etiqueta** se han desperdigado demostrar que todos ellos son inicializados antes de que se ejecute el constructor y antes de que pueda ocurrir cualquier otra cosa. Además, **t3** es reinicializado dentro del constructor. La salida es:

```
Etiqueta(1)
Etiqueta(2)
Etiqueta(3)
Tarjeta()
Etiqueta(33)
f()
```

De este modo, la referencia **t3** es inicializado dos veces; la primera antes de la llamada al constructor y la segunda durante ella. (El primer objeto es abandonado , por lo que podrá ser eliminado por el recogedor de basura más adelante). Esta estrategia puede parecer poco eficiente, que garantiza que la inicialización se realiza adecuadamente; por que ¿qué ocurriría si se definiese un segundo constructor que *no* inicializase **t3** y no hubiese hubiese una inicialización "por defecto" en su definición?

Inicialización de datos estáticos

Lo espuesto en el párrafo anterior se aplica también a los datos estáticos. Si una primitiva no es inicializada explícitamente, recibe el valor inicial estándar que le corresponda. Si es una referencia a un objeto, se le asigna el valor **null** a menos que se cree un objeto y se le asocie dicha referencia.

Si se quiere inicializar un datos en el mismo sitio en que se define, se hace como el caso de datos no estáticos. Hay solo una zona de memoria un dato estático, independientemente de cuantos objetos sean creados, lo que plantea la cuestión de cuando es inicializada la zona de memoria que corresponde al datos estático. El ejemplo siguiente aclara esta cuestión:

```
//: c04:InicializacionEstatica.java
// Especificacion de valores iniciales
// en la definicion de una clase.

class Bol {
   Bol(int marcador) {
      System.out.println("Bol(" + marcador + ")");
   }
   void  f(int marcador) {
      System.out.println("f(" + marcador + ")");
   }
}

class Mesa {
   static Bol b1 = new Bol(1);
   Mesa() {
      System.out.println("Mesa()");
      b2.f(1);
   }
   void f2(int marcador) {
      System.out.println("f2(" + marcador + ")");
   }
   static Bol b2 = new Bol(2);
}
```

```
class Armario {
   Bol b3 = new Bol(3);
   static Bol b4 = new Bol(4);
   Armario() {
      System.out.println("Armario()");
      b4.f(2);
   }
   void f3(int marcador) {
      System.out.println("f3(" + marcador + ")");
   }
   static Bol b5 = new Bol(5);
}

public class InicializacionEstatica {
   public static void main(String[] args) {
      System.out.println(
         "Creando nuevo Armario() en main");
      new Armario();
      System.out.println(
         "Creando nuevo Armario() en main");
      new Armario();
      t2.f2(1);
      t3.f3(1);
   }
   static Mesa t2 = new Mesa();
   static Armario t3 = new Armario();
} ///:~
```

Bol permite ver la creación de la clase y **Mesa** y **Armario** crean miembros del tipo **Bol** en diferentes posiciones dentro de la definición de sus clases respectivas. Se observa que **Armario** crea el objeto no estático **Bol b3** antes que las definiciones **static** . El programa produce la siguiente salida:

```
Bol(1)
Bol(2)
Mesa()
f(1)
Bol(4)
Bol(5)
Bol(3)
Armario()
f(2)
Creando nuevo Armario() en main
Bol(3)
Armario()
f(2)
```

```
Creando nuevo Armario() en main
Bol(3)
Armario()
f(2)
f2(1)
f3(1)
```

Los datos estáticos son inicializados únicamente si es necesario. Si no se crea ninguna **Mesa** o si nunca se accede a **Mesa.b1** o **Mesa.b2** miembros estáticos **b1** y **b2** del tipo **Bol** nunca serán creados. Además, serán inicializados únicamente una vez: la *primera* que un objeto *Mesa* sea creado (o cuando se produzca el primer acceso estático) Después los objetos estáticos no vuelven a ser inicializados.

Los miembros estáticos son, pues, los primeros en ser inicializados, si no lo han sido ya como consecuencia de la creación un objeto anterior, y a continuación se inicializan los datos no estáticos, como se puede verificar examinando la salida del ejemplo anterior.

Resulta útil resumir aquí el proceso de creación de un objeto. Considérese la clase **Perro** :

1. La primera vez que un objeto del tipo **Perro** es creado, *o* la primera vez que se accede a un método estático o a un campo estático, el interprete de Java debe localizar el fichero **Perro.class** buscando por entre las diferentes localizaciones indicadas por *classpath*
2. A medida que **Dog.class** es cargado (creando un objeto **Class** como se verá más adelante) se ejecutan todos sus inicializadores estáticos. De este modo, la inicialización estática se realiza solo una vez: mientras el objeto **Class** es cargado por primera vez.
3. Cuando se crea un nuevo objeto **Perro** , mediante **new Perro()** , el proceso de construcción del objeto **Perro** reserva, en primer lugar, espacio suficiente en el heap contener el objeto.
4. El espacio reservado en el heap es automáticamente con ceros, lo que automáticamente asigna a las primitivas definidas dentro de **Perro** sus valores por defecto (cero los números y su equivalente los campos **boolean** y **char** y **null** a las referencias.
5. Se ejecutan todas las inicializaciones especificadas junto a la definición de los campos.
6. Se ejecutan los constructores. Como se verá en el capítulo 6 esto puede significar una gran cantidad de trabajo, especialmente cuando hay involucrada herencia.

Inicialización estática explícita

Java cuenta con una "construcción estática" que permite agrupar varias inicializaciones de datos **static** en un solo lugar dentro de una clase. Esta construcción recibe a veces el nombre de *bloque estático (static block)* . El siguiente ejemplo muestra un ejemplo de esta construcción.

```
class Cuchara {
   static int i;
   static {
      i = 47;
   }
   // . . .
```

Aunque parece la definición de un método, se trata solamente de la palabra clave **static** seguida por el cuerpo de método. Este fragmento de código, como cualquier otra inicialización estática, se ejecuta tan solo una vez: la primera vez que se cree un objeto de esa clase o la primera vez que se acceda a un miembro estático de la clase (incluso cuando no se llegue a crear un solo objeto de esa clase). Por ejemplo:

```
//: c04:ExplicitaEstatica.java
// Inicializacion explicita de datos
// estaticos con una clausula "static"

class Taza {
   Taza(int marcador) {
      System.out.println("Taza(" + marcador + ")");
   }
   void f(int marcador) {
      System.out.println("f(" + marcador + ")");
   }
}

class Tazas {
   static Taza t1;
   static Taza t2;
   static {
      t1 = new Taza(1);
      t2 = new Taza(2);
   }
   Tazas() {
      System.out.println("Tazas()");
   }
}

public class ExplicitaEstatica {
   public static void main(String[] args) {
      System.out.println("Dentro de main()");
      Tazas.t1.f(99);   // (1)
   }
   // static Tazas x = new Tazas();  // (2)
   // static Tazas y = new Tazas();  // (2)
```

```
} ///:~
```

Los inicializadores estáticos de **Copas** se ejecutan cuando se accede al objeto estático **t1** en la línea (1) o si la línea (1) se comenta la línea (1) y se descomentan las líneas marcadas con (2). Si tanto la línea marcada con (1) como las marcadas con (2) son comentadas, la inicialización estática en **Tazas** no llega nunca a tener lugar. Por otro lado, no cambia nada si se quitan las marcas de comentario de ambas o solo una de las líneas (2): la inicialización estática solo ocurre una vez.

Inicialización de instancias no estáticas

La sintaxis en Java para inicializar las variables de cada objeto que no sean estáticas, es similar a la del caso estático. Por ejemplo:

```
//: c04:Tazones.java
// Inicializacion de instancias en Java.

class Tazon {
   Tazon(int marcador) {
      System.out.println("Tazon(" + marcador + ")");
   }
   void f(int marcador) {
      System.out.println("f(" + marcador + ")");
   }
}

public class Tazones {
   Tazon c1;
   Tazon c2;
   {
      c1 = new Tazon(1);
      c2 = new Tazon(2);
      System.out.println("c1 y c2 inicializados");
   }
   Tazones() {
      System.out.println("Tazones()");
   }
    public static void main(String[] args) {
      System.out.println("Dentro de main()");
      Tazones x = new Tazones();
   }
} ///:~
```

Puede comprobarse que la clausula de inicializacion de la instancia:

```
{
      c1 = new Tazon(1);
      c2 = new Tazon(2);
      System.out.println("c1 y c2 inicializados");
   }
```

parece exactamente igual que la clausula de inicialización estática salvo que no contiene la palabra clave **static** . Esta sintaxis es necesaria para permitir la inicialización de *clases internas anónimas (anonymous inner class)* como se verá en el capítulo 8.

Inicialización de arrays

La inicialización de arrays en C es una tarea tediosa en la que es fácil cometer errores. En su lugar, C++ utiliza inicialización de agregados (aggregate initialization) lo que hace la tarea más segura. Java no tiene "agregados" como C++ ya que en Java todo es siempre un objeto. Java tiene arrays y además proporciona los medios necesarios para su inicialización.

Un array no es más que una secuencia de primitivas u objetos, todos del mismo tipo y agrupados bajo un único identificador. Los arrays se definen y se utilizan haciendo uso del *operador indexación (indexing operator)* **[]** . Un array se puede definir, simplemente, añadiendo un par de paréntesis cuadrados detrás del nombre del tipo:

```
int[] a1;
```

También se pueden poner los paréntesis detrás del identificador: el efecto es el mismo:

```
int a1[];
```

Esta forma de la notación satisfacera las expectativas de los programadores de C y C++. La primera en cambio, es quizás más adecuada ya que se puede leer como "el tipo es un array de **int** . En este libro se utilizará por lo tanto este estilo cuando se definan arrays.

El compilador no permite indicar cual será el tamaño del array, lo que nos lleva de vuelta a asunto de las "referencias". Después de la definición del array lo único que se tiene es una referencia a un array, y además, no se reservará espacio alguno para contener el array. Para reserva espacio para el array se necesita

escribir una expresión de inicialización. La inicialización de un array puede aparecer en cualquier lugar dentro del código, pero también se puede usar una tipo especial de expresión de inicialización que tiene que hacerse en el mismo lugar donde se crea el array. Este tipo especial de inicialización consiste en un conjunto de valores encerrados entre llaves. En este caso es el compilador el que se encarga de reservar el espacio necesario para el array:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Según esto, ¿Por qué motivo va a querer nadie crear una referencia sin un array?

```
int [] a2;
```

Una razón posible, es que se quiera asignar un array a otro, lo que es una operación permitida en Java. Por ejemplo:

```
a2 = a1;
```

Lo que se está copiando realmente en este caso es una referencia como revela este ejemplo:

```
//: c04:Arrays.java
// Arrays de primitivas.

public class Arrays {
   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]++;
      for (int i = 0; i < a1.length; i++)
         System.out.println(
            "a1[" + i + "] = " + a1[i]);
   }
} ///:~
```

En este ejemplo, a **a1** se le asigna un valor de inicialización pero no a **a2** , a la

que se le asigna posteriormente otro array.

En el ejemplo anterior aparece también algo nuevo: todos los arrays tiene un miembro intrínseco (tanto si son arrays de objetos como si son arrays de primitivas), que puede examinar - pero no modificar -, que contiene el número de elementos del array. Este miembro es **length** . Como, al igual que en C y C++, en Java los índices de los arrays comienzan a contar desde cero el índice del último elemento del array es **length - 1** . C y C++ aceptarán calladamente que un programa sobrepase los límites del array y acceda de este modo al resto de la zona de memoria reservada para él, lo que es una fuente de continuos problemas. Por contra, Java generará un error en tiempo de ejecución (es decir una *excepción* , tema del capítulo 10) si el programa intenta sobrepasar los límites del array. Naturalmente, el control de todos los accesos a arrays tiene un coste, tanto en tiempo como en código, que no se puede evitar, por lo que el acceso a arrays puede perjudicar el rendimiento del programa, especialmente si estos accesos ocurren en una juncture crítica. Los diseñadores de Java pensaron que, la seguridad en Internet y el aumento de productividad de los programadores, justificaban el sobrecarga que supone el control de los accesos a los arrays.

Pero ¿qué ocurre si se desconoce, cuando se esta escribiendo el programa, cuantos elementos tendrá que contener? Para crear nuevos elementos del array se usa simplemente **new** . En este ejemplo, **new** funciona también creando creando un array de primitivas (**new** no creará ??????????????????):

```
//: c04:ArrayNuevo.java
// Creacion de arrays con new.
import java.util.*;

public class ArrayNuevo {
   static Random rand = new Random();
   static int pRand(int mod) {
          return Math.abs(rand.nextInt()) % mod + 1;
   }
   public static void main(String[] args) {
            int[] a;
            a = new int[pRand(20)];
            System.out.println(
                 "num. elementos de a = " + a.length);
              for(int i = 0; i < a.length; i ++)
                System.out.println(
                  "a[" + i + "] = " + a[i]);
   }
} ///:~
```

Como el tamaño del array es elegido de modo aleatorio (usando el método **pRand ()**), queda claro que la creación del array se está haciendo en tiempo de

ejecución. Además, se puede comprobar examinando la salida del programa anterior, los elementos del array que son de un tipo primitivo son inicializadas automáticamente con sus valores "vacíos"(para números y **char** este valor es cero y para **boolean** es **false**)

El array podría haber sido definido e inicializado en la misma línea:

```
int[] a = new int[pRand(20)];
```

Si se está trabajando con arrays de objetos que no sean primitivas, el necesario usar siempre el operador **new** . Vuelve a aparecer aquí el tema de la referencia ya que lo que realmente se crea es un array de referencias. Así, por ejemplo, con el tipo wrapper **Integer** , que es una clase y no una primitiva:

```
//: c04:ArrayClassObj.java
// Creacion de un array de objetos
// de tipo no primitivo.
import java.util.*;

public class ArrayClassObj {
   static Random rand = new Random();
   static int pRand(int mod) {
      return Math.abs(rand.nextInt()) % mod +1;
   }
   public static void main(String[] args) {
      Integer[] a = new Integer[pRand(20)];
      System.out.println(
         "num. elementos de a = " + a.length);
      for (int i = 0; i < a.length; i++) {
         a[i] = new Integer(pRand(500));
         System.out.println(
            "a[" + i + "] = " + a[i]);
      }
   }
} ///:~
```

En este ejemplo, incluso después de haber llamado a **new** crear el array:

```
Integer[] a = new Integer[pRand(20)];
```

lo único que se tiene es un array de referencias, y la inicialización, por lo tanto, no

está completa hasta que no se inicializa la referencia misma creando un nuevo objeto **Integer** .

```
a[i] = new Integer(pRand(500));
```

Si el objeto no es creado, se obtendrá una excepción en tiempo de ejecución al intentar leer la posición vacía del array.

Si se examina como se construye el objeto **String** dentro de las sentencias de impresión, se puede ver que la referencia al objeto **Integer** es transformada en una referencia a un objeto **String** que contiene el valor del objeto **Integer**

También se pueden inicializar arrays usando una lista de elementos encerrada entre llaves. Existen dos formas diferentes:

```
//: c04:ArrayInit.java
// Inicializacion de arrays.

public class ArrayInit {
   public static void main(String[] args) {
      Integer[] a = {
         new Integer(1),
         new Integer(2),
         new Integer(3),
      };

      Integer[] b = new Integer[] {
         new Integer(1),
         new Integer(2),
         new Integer(3),
      };
   }
} ///:~
```

Esta manera de inicializar arrays puede ser útil en ocasiones, pero tiene la limitación de que el tamaño del array queda fijado durante la compilación. La coma al final de la lista de los inicializadores es opcional; su única función es hacer más sencillo el mantenimiento de listas largas.

La segunda forma de inicialización de arrays proporciona una sintaxis adecuada crear e invocar métodos que pueden producir el mismo efecto que las *listas variables de argumentos* de C (en C reciben el nombre de "varargs"). Estas listas pueden contener un número indeterminado de argumentos que, además pueden ser de cualquier tipo. Ya que todas las clases derivan de la misma clase raíz

Object (el lector podrá aprender más sobre este tema según prosiga en la lectura de este libro), es posible crear un método que tenga como argumento un array de **Object** e invocarlo del siguiente modo:

```
//: c04:VarArgs.java
// Ejemplo de uso de un array para crear
// una lista variable de argumentos.

class A { int i; }

public class VarArgs {
   static void f(Object[] x) {
      for (int i = 0; i < x.length; i++)
         System.out.println(x[i]);
   }
   public static void main(String[] args) {
      f(new Object[] {
         new Integer(47), new VarArgs(),
         new Float(3.14), new Double(111.11) });

      f(new Object[] {"one", "two", "three" });
      f(new Object[] {new A(), new A(), new A()});
   }
} ///:~
```

De momento no hay mucho más que se pueda hacer con estos objetos de tipo desconocido, y el programa solo utiliza la conversión automática a **String** poder hacer lago útil con cada **Object** . En el capítulo 12, que trata de la identificación de tipos durante la ejecución del programa (*run-time type identification* - RTTI), se verá como se puede averiguar el tipo preciso de cada objeto de modo que se posible hacer con ellos algo más interesante.

Arrays multidimensionales

Java permite crear arrays multidimensionales fácilmente:

```
//: c04:ArrayMultiDim.java
// Creacion de arrays multidimensionales
import java.util.*;

public class ArrayMultiDim {
   static Random rand = new Random();
   static int pRand(int mod) {
      return Math.abs(rand.nextInt()) % mod +1 ;
   }
```

```
static void prt(String s) {
   System.out.println(s);
}
public static void main(String[] args) {
   int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
   };
   for (int i = 0; i < a1.length; i++)
      for (int j = 0; j < a1[i].length; j++)
         prt("a1[" + i + "][" + j +
             "] = " + a1[i][j]);
   // Array 3-D con dimesiones fijas.
   int[][][] a2 = new int[2][2][4];
   for (int i = 0; i < a2.length; i++)
      for (int j = 0; j < a2[i].length; j++)
         for (int k = 0; k < a2[i][j].length;
              k++)
            prt("a2[" + i + "][" +
                j + "][" + k +
                "] = " + a2[i][j][k]);
   // Array 3-D con vectores de longitud variable
   int [][][] a3 = new int[pRand(7)][][];
   for (int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for (int j = 0; j < a3[i].length; j++)
         a3[i][j] = new int[pRand(5)];
   }
   for (int i = 0; i < a3.length; i++)
      for (int j = 0; j < a3[i].length; j++)
         for (int k = 0; k < a3[i][j].length;
              k++)
            prt("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
   // Arrays de objetos no primitivos
   Integer [][] a4 = {
      { new Integer(1), new Integer(2) },
      { new Integer(3), new Integer(4) },
      { new Integer(5), new Integer(6) },
   };
   for (int i = 0; i < a4.length; i++)
      for (int j = 0; j < a4[i].length; j++)
         prt("a4[" + i + "][" + j +
             "] = " + a4[i][j]);
   Integer[][] a5;
   a5 = new Integer[3][];
   for (int i = 0; i < a5.length; i++) {
      a5[i] = new Integer[3];
      for (int j = 0; j < a5[i].length; j++)
```

```
            a5[i][j] = new Integer(i*j);
      }
      for (int i = 0; i < a5.length; i++)
         for (int j = 0; j < a5[i].length; j++)
            prt("a5[" + i + "][" + j +
                "] = " + a5[i][j]);

   }
} ///:~
```

En el programa anterior se utiliza **length** en los bucles de escritura, lo que muestra que **length** se puede usar también con arrays que no tengan un tamaño fijo.

El primer ejemplo muestra un array multidimensional de primitivas. Cada vector dentro del array es delimitado mediante el uso de llaves:

```
int[][] a1 = {
         { 1, 2, 3, },
         { 4, 5, 6, },
      }
```

Cada pareja de corchetes define una nueva dimensión dentro del array.

El segundo ejemplo muestra un array tridimensional que se construye usando el operador **new** . En este caso, el espacio para el array es reservado de una sola vez:

```
int[][][] a2 = new int[2][2][4];
```

El tercer ejemplo muestra como cada uno de los vectores que forman el array puede tener su propia longitud:

```
int [][][] a3 = new int[pRand(7)][][];
      for (int i = 0; i <  a3.length; i++) {
         a3[i] = new int[pRand(5)][];
         for (int j = 0; j <  a3[i].length; j++)
            a3[i][j] = new int[pRand(5)];
      }
```

El primer operador **new** crea un array en el que el primer elemento tiene una

longitud aleatoria y deja el resto indeterminado. El segundo operador **new** , dentro del bucle **for** rellena los elementos pero deja el tercer índice sin definir hasta que se alcanza el tercer operador **new** .

Se pude verificar, examinando la salida que produce el programa anterior, que los valores de los arrays son inicializados con valor cero si no se les asigna explícitamente un valor de inicialización.

El cuarto ejemplo muestra que es posible trabajar con objetos no primitivos de una manera similar y como se puede agrupar varios operadores **new** entre llaves:

```
Integer [][] a4 = {
        { new Integer(1), new Integer(2) },
        { new Integer(3), new Integer(4) },
        { new Integer(5), new Integer(6) },
      };
```

El quinto ejemplo, por último, muestra como se puede construir un array de objetos no primitivos elemento a elemento:

```
Integer[][] a5;
      a5 = new Integer[3][];
      for (int i = 0; i <  a5.length; i++) {
         a5[i] = new Integer[3];
         for (int j = 0; j <  a5[i].length; j++)
            a5[i][j] = new Integer(i*j);
      }
```

donde **i*j** se utiliza simplemente para poner un valor interesante en la inicialización de **Integer** .

Resumen

Lo .....elaborado mecanismo de inicialización, el constructor, da una idea de la prioridad que se ha puesto en lo relativo a la inicialización en el lenguaje. Cuando Stroustrup estaba diseñando C++, una de las primeras observaciones que hizo acerca de la productividad en C, fue que la inicialización inadecuada de variables está en el origen de muchos de los problemas que se presentan en la programación. Estos errores son, además difíciles de descubrir. Lo mismo se aplica a un purgado inadecuado. Dado que los constructores garantizan la inicialización y el purgado adecuados (el compilador no permite que un objeto sea creado sin que se invoque el constructor adecuado),y de este modo, el programador retiene todo

el control en la creación y destrucción de objetos.

En C++ la destrucción de objetos es muy importante ya que los objetos creados mediante el operador **new** tienen que ser destruidos explícitamente. En Java, el recolector de basura es el encargado de liberar la memoria reservada por los objetos. Por lo tanto, no es necesario, en muchos casos, definir un método de purgado equivalente al destructor de C++, lo que simplifica en gran medida la programación en Java, y añade la tan necesitada seguridad en la gestión de memoria. Algunos recolectores de basura pueden incluso pueden incluso gestionar otros recursos como gráficos y ficheros de .....(handles). Pero la utilización de un recolector de basura conlleva un sobrecarga en la ejecución de los programas. La importancia que esta sobrecarga tenga es difícil de valorar debido a la lentitud de los interpretes de Java en el momento en que este libro esta siendo escrito. Solo cuando esta situación mejore, será posible determinar si el uso de recolector de basura permitirá, o no, el uso de Java en algún tipo de aplicaciones (uno de los asuntos a considerar es el carácter, impredecible a priori, del recolector de basura).

Dado que la construcción de objetos en Java está garantizada, hay muchos mas asuntos relativos a la construcción de objetos de los que se han tratado en este capítulo. En particular, cuando se crea una nueva clase usando *composición (composition)* o *herencia (inheritance)* , la garantía sobre la inicialización se sigue cumpliendo, por lo que es necesario ampliar la sintaxis de modo que estos casos queden cubiertos. La composición, la herencia y su efecto en como un objeto es construido, son tratados en otros capítulos, más adelante, en este libro.

Ejercicios

La soluciones a los ejercicios seleccionados pueden ser encontradas en el documento electrónico *The Thinking in Java Annotated Solution Guide* , disponible en *www.BruceEckel.com* a un precio muy asequible.

1. Crear una clase con un constructor por defecto (es decir, un constructor sin argumentos) que imprima un mensaje. Crear un objeto de esta clase.
2. Añadir a la clase del ejercicio 1 un constructor sobrecargado que tenga un argumento de tipo **String** y que imprima la cadena junto con el mensaje anterior.
3. Crear un array de referencias a objetos de la clase creada en el ejercicio anterior, pero si llegar a crear los objetos. Al ejecutar el programa observar si se imprimen los mensajes de inicialización.
4. Completar el ejercicio 3 creando objetos para asignarlos a las referencias contenidas en el array.
5. Crear un array de objetos **String** y asignar cadenas a cada elemento. Imprimir los resultados usando un bucle **for** .
6. Crear una clase **Perro** con un método **ladrar()** sobrecargado. El método debe ser sobrecargado usando diferentes tipos de primitivas e imprimir diferentes tipos de

ladridos, auyidos, etc., dependiendo de que versión del método es invocada. Escribir un método **main()** que llame a todas las versiones.

7. Modificar el Ejercicio 6 para que dos de los métodos sobrecargados tengan dos argumentos (cada uno de un tipo diferente) pero en distinto orden. Verificar que la sobrecarga de métodos funciona también en este caso.
8. Crear una clase sin constructores, crear un objeto de esta clase dentro del método **main ()** y verificar que el constructor por defecto es sintetizado automáticamente.
9. Crear una clase con dos métodos. Desde el primer método llamar al segundo dos veces: la primera sin usar **this** y la segunda con **this**
10. Crear una clase con dos constructores sobrecargados. Usar **this** para llamar al segundo constructor desde dentro del primero.
11. Crear una clase llamada **Tanque** que puede ser llenada y vaciada, y que además tiene como *"death condition"* que ha de estar vacía cuando el objeto es eliminado. Escribir un método **finalize()** que compruebe que se cumple la "death condition". Comprobar en el método **main()** todas las posibles situaciones que pueden darse.
12. Crear una clase que tenga un **int** y un **char** que no sean inicializados e imprimir su valores comprobar que Java realiza la inicialización por defecto.
13. Crear una clase que tenga un referencia a **String** sin inicializar. Demostrar que la referencia es inicializada con **null** .
14. Crear una clase con un campo **String** que sea inicializado en la definición y otro que se inicializado por el constructor. ¿Cual es la diferencia entre estos dos planteamientos?
15. Crear una clase con dos campos **static Sring** de modo que uno sea inicializado en la definición y el otro dentro de bloque **static** . Añadir un método **static** que imprima ambos campos y demuestre que ambos son inicializados antes de ser usados.
16. Crear una clase con un **String** que es inicializado usando "inicialización de instancia". Describir una utilidad que puede darse a esta construcción (diferente de la apuntada en este libro)
17. Escribir un método que cree e inicializa un array bidimensional de **double** . El tamaño del array debe venir definido los argumentos del método, y los valores de inicialización deben estar dentro de un rango cuyos valores máximo y mínimo sean también parámetros del método. Crear un segundo método que imprima el array generado por el primer método. En **main()** verificar el correcto funcionamiento de los dos métodos creando e imprimiendo arrays de diferentes tamaños.
18. Repetir el ejercicio 19 con un array tridimensional
19. Comentar la línea marcada con (1) en **ExplicitaEstatica** y comprobar que la cláusula de inicialización estática no es invocada. Quitar la marca de comentario de una de las líneas marcadas con (2) y comprobar que *sí* es invocada. Hacer lo mismo con la otra línea marcada con (2) y comprobar que la inicialización estática solo tiene lugar una vez.
20. Experimentar con **Basura.java** ejecutando el programa pasándole los argumentos "gc", "finalizar" o "todo". Repetir el proceso si se observa que se repite algún modelo en la salida. Modificar el código que **System.runFinalization()** sea llamado *antes* que **System.gc()** y observar el resultado.

5:Ocultando la Implementación

Una consideración fundamental en el diseño orientado a objetos es "separar las cosas que cambian de las cosas que permanecen igual".

Esto es especialmente importante para las librerías. El usuario *(programador cliente)* de esa librería tiene que confiar en la parte que usa, y saber que no necesitará reescribir código si una nueva versión de la librería aparece. En la cara opuesta, el creador de la librería debe tener libertad para realizar modificaciones y mejoras con la seguridad que el código de los programadores no será afectado por esos cambios.

Esto se puede conseguir gracias a la convención. Por ejemplo, el programador de la librería debe estar de acuerdo en no eliminar métodos existentes cuando modifique una clase de la librería, ya que eso estropearía el código del programador cliente. Sin embargo, la situación contraria es más espinosa. En el caso de un dato miembro, ¿cómo puede saber el creador de la librería que datos miembros han sido accedidos por los programadores cliente? Esto también ocurre con métodos que son sólo parte de la implementación de una clase, y no significa que sean usados directamente por el programador cliente. Pero, y si el creador de la librería quiere deshacerse de una vieja implementación y poner una nueva? Cambiar cualquiera de esos miembros podría estropear el código del programador cliente. Por tanto, el creador de la librería lleva una camisa de fuerza y no puede cambiar ninguna cosa.

Para solucionar este problema, Java proporciona *especificadores de acceso* que permiten al creador de la librería decir que está disponible al cliente programador, y que no está. Los niveles de control de acceso son, en orden creciente de permisividad, **public** (público), **protected** (protegido), "friendly" (amiga, no tiene palabra reservada) y **private** (privado). Del párrafo anterior se podría pensar que el diseñador de librerías mantendrá todo cuanto sea posible como privado (private) y dejar accesibles los métodos que quiere que el cliente programador use. Esto es totalmente correcto, incluso aunque frecuentemente no sea intuitivo para la gente que programaba en otros lenguajes (especialmente C) y accedía a todo sin restricciones. Al final del capítulo, usted deberá ser convencido de la importacncia del control de acceso en Java.

No obstante, el concepto de librería de componentes y el control sobre quien puede acceder a los componentes de esa librería no está completo. Todavía está la pregunta de cómo los integrantes son envueltos en una unidad de librería (library unit). Esto se controla con la palabra reservada **package** en Java, y los especificadores de acceso influyen en si una clase está en el mismo paquete o en

paquetes separados. Asi que para empezar este capítulo, aprenderá cómo los integrantes de la librería son alojados en paquetes. Luego, será capaz de comprender el significado completo de los especificadores de acceso.

paquetes: la librería unidad

Un paquete (package) es lo que se obtiene cuando usa la palabra reservada **import** para importar una librería entera, tal como la instrucción

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

Esto importa la librería de utilidades que es parte de la distribución estándar de Java. Ya que, por ejemplo, la clase **ArrayList** está en **java.util** , usted ahora puede especificiar el nombre completo **java.util.ArrayList** (lo cual puede hacer sin la sentencia import), o simplemente decir **ArrayList** (debido al **import**).

Si quiere importar sólo una clase, puede nombrar esa clase en la sentencia **import**

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

Ahora puede usare ArrayList sin limitación. Sin embargo, ninguna de las otras clases en java.util puede usarse.

La razón de hacer esto es proporcionar un mecanismo que gestione los "espacios de nombres". Los nombres de todos los miembros de clase están aislados unos de otros. Un método **f( )** dentro de una clase A no colisionará con **f( )** que tenga la misma signatura (lista de argumentos) de una clase B. Pero, Qué ocurre con el nombre de las clases? Suponga que crea una clase **stack** que está instalada en una máquina que ya tiene otra clase **stack** escrita por otra persona. Con Java en Internet, esto puede ocurrir sin que el usuario lo sepa, ya que las clases pueden descargarse automáticamente en el proceso de arranque de un programa en Java.

Esta posible colisión de nombres es la causa de la importancia de tener control completo sobre los nombres de espacio en Java, y ser capaz de crear un nombre completo único a pesar de todas las restricciones de Internet.

Además, la mayor parte de los ejemplos de este libro han estado en un solo fichero y han sido diseñados para uso local, y no se preocupan de los nombre de paquetes. (En este caso el nombre de la clase es situado en el "package default"). Esto es con certeza una opción, y por simplicidad esta aproximación será usada cuando sea posible durante el resto de l libro. Sin embargo, si está planeando crear librerias o programas que son "amigos" con otros programas Java en la

misma máquina, debe preocuparse de evitar conflictos de nombres de clases.

Cuando crea un fichero de código fuente para Java, se le conoce comúnmente como *unidad de compilación* (a veces unidad de translación). Cada unidad de compilación tiene que tener un nombre terminado **.java** , y dentro de la unidad de compilación puede haber una clase **public** que debe tener el mismo nombre que el fichero (incluso si está en mayúsculas, aunque no hay que añadir la extensión **.java** al nombre de la clase). Puede haber sólo una clase **public** en cada unidad de compilación, de otro modo el compilador dará error. El resto de las clases en esa unidad de compilación, si hay otras, estarán ocultas del mundo exterior a ese paquete, porque *no* son **public** , y comprenden clases de "apoyo" para la clase principal **public** (pública).

Cuando compila un fichero **.java** , obtiene otro fichero de salida con el mismo nombre pero de extensión **.class** *para cada clase en el fichero* **.java** . Así, puede terminar con bastantes ficheros **.class** de un pequeño número de ficheros **.java** files. Si ha programado con un lenguaje compilado, tal vez el compilador generaba una forma intermedia (normalmente un fichero "obj") que es luego empaquetado junto con otros de su mismo tipo usando un enlazador (para crear un fichero ejecutable) o un generador de librerías (para crear una librería). Así no es como funciona Java. Un programa funcionando es un grupo de ficheros **.class** , que pueden ser empaquetados y comprimidos dentro de un fichero JAR (usando el archivador **jar** de Java).

El intérprete de Java es responsable de encontrar, cargar e interpretar estos ficheros [1].

Una librería es también un grupo de ficheros class. Cada fichero tiene una clase que es public (no está obligado de tener una clase public class, pero es lo típico), por lo que hay un componente para cada fichero. Si quiere decir que todos esos componentes (que están en su correspondiente fichero .java y .class) belong together, that's where the package keyword comes in.

Cuando dice:

```
package mypackage;
```

al comienzo del fichero (si usa una sentencia **package** , debe aparecer al principio sin comentarios en el fichero), está indicando que esta unidad de compilación es parte de una librería denominada **mypackage** . O, dicho de otra forma, está diciendo que la clase **public** dentro de esta unidad de compilación está bajo la sombrilla del nombre **mypackage** , y si alguien quiere usarla tienen que especificar completamente el nombre de la clase o usar la palabra reservada **import** en combinación con **mypackage** (usando las instrucciones dadas anteriormente). Note que la convención para los nombres de paquete en java es

usar minúsculas, incluso para palabras intermedias.

Por ejemplo, supongamos que el nombre del fichero es **MyClass.java** . Esto quiere decir que puede haber una y solo una clase **public** en ese fichero, y el nombre de esa clase debe ser **MyClass** (incluyendo las que sean mayúsculas):

```
package mypackage;
public class MyClass {
// . . .
```

Ahora, si alguien quiere usar **MyClass** o, del mismo modo para cualquiera otra clase **public** en **mypackage** , tiene que usar la palabra reservada **import** para tener acceso al nombre o nombres en **mypackage** . La alternativa es dar el nombre completo:

```
mypackage.MyClass m = new mypackage.MyClass();
```

---

1 No hay nada en Java que obligue el uso de un intérprete. Existen compiladores de código nativo en Java que generan ficheros ejecutable.

La palabra import puede hacer esto más claramente:

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

Es bueno tener en mente que las palabras reservadas **package** e **import** lo que le permiten hacer, como diseñador de librerías, es dividir todo el espacio de nombres para que no se produzcan colisiones, sin importar cuanta gente tenga e Internet y comience a escribir clases en Java.

Creación de nombres de paquete único

Puede observar que, ya que un package nunca "empaqueta" todo realmente en un único fichero, puede estar compuesto de muchos ficheros .class y eso podría dar lugar a un poco de desorden. Para evitar esto, algo lógico de hacer es situar todos los ficheros **.class** de un paquete en particular en un solo directorio; esto es, usar la estructura jerárquica de ficheros del sistema operativo para su aprovechamiento. Esto es un modo en que Java da solución al problema del desorden; verá otra modo más tarde cuando se comente la utilidad **jar** .

Reunir los ficheros de los paquetes en un solo directorio resuelve otros dos problemas: creación de nombres de paquetes únicos, y encontrar aquellas clases que pudieran estar enterradas en algún lugar de la estrucutura del directorio. Esto está realizado, ya que fue presentado en el Capítulo 2, codificando la ruta de localización del fichero **.class** dentro del nombre del **package** . El compilador obliga a esto, pero por convención, la primera parte del nombre del **package** es el nombre de dominio de Internet del creador de la clase, revertido. Ya que está garantizado que los nombres de dominio de Internet sean únicos, si sigue esta convención está garantizando que su nombre de **package** será único y así nunca tendrá un conflicto de nombres. (Esto es, hasta que pierda el nombre de dominio y éste vaya a parar a alguien que además empiece a escribir código Java con la misma ruta que usó usted.) Por supuesto, si no tiene su propio nombre de dominio entonces debe fabricarse una combinación que difícilmente coincida con otra (como su nombre y apellidos) para crear nombres de paquete únicos. Si ha decidido empezar a publicar código Java merece la pena el relativamente pequeño esfuerzo de conseguir un nombre de dominio.

La segunda parte de este truco es tomar como nombre de **package** un directorio de su máquina, así cuando el programa en Java se ejecute y necesite cargar el fichero **.class** (lo cual se hace dinámicamente, en un punto del programa donde se necesita crear un objeto de esa clase particular, o la primera vez que accede a un miembro estático de la clase), puede localizar el directorio donde el fichero **.class** reside. El intérprete de Java procede como sigue. Primero, encuentra la variable de ambiente CLASSPATH (colocada por el sistema operativo, a veces por el programa de instalación que instala Java o una herramienta basada en Java de su máquina). CLASSPATH contiene uno o más directorios que son usados como raíz para la búsqueda de ficheros **.class** . Comenzando en esa raíz, el intérprete tomará el nombre del paquete y reemplazará cada punto con un paréntesis para generar la ruta desde la raíz del CLASSPATH (asi **package foo.bar.baz** se convierte en foo\bar\baz o foo/bar/baz o en alguna otra cosa, dependiendo de su sistema operativo) Esto es luego concatenado para varias entradas en el CLASSPATH. Ahí es donde busca el fichero .class con el nombre correspondiente a la clase que está intentando crear. (También busca algún directorio estándar relativo a donde el intérprete de Java reside).

Para entender esto, considere mi nombre de dominio, que es **bruceeckel.com** . Inviertiendo esto, **com.bruceeckel** establece mi nombre global único para mis clases. (Las extensiones com, edu, org, etc. fueron antiguamente puestas en mayúsculas en los paquetes Java, pero esto fue cambiando en Java 2 asi que el nombre del paquete entero es en minúsculas) Puedo además subdividir esto, decidiendo que quiero crear un librería llamada **simple** , así terminaré con un nombre de paquete:

```
package com.bruceeckel.simple;
```

Ahora este nombre de paquete puede ser usado como como una sombrilla de espacio de nombres para los siguientes dos ficheros:

```
//: com:bruceeckel:simple:Vector.java
// Creando un paquete.
package com.bruceeckel.simple;
public class Vector {
   public Vector() {
      System.out.println(
         "com.bruceeckel.util.Vector");
   }
} ///:~
```

Cuando cree sus propios paquetes, descubrirá que la sentencia **package** tiene que ser la primera línea de código sin comentar en el fichero. El segundo fichero se parece mucho:

```
//: com:bruceeckel:simple:List.java
// Creando un paquete.
package com.bruceeckel.simple;
public class List {
   public List() {
      System.out.println(
         "com.bruceeckel.util.List");
   }
} ///:~
```

Ambos ficheros son situados en el subdirectorio de mi sistema:

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Si retrocede, puede ver el nombre de paquete **com.bruceeckel.simple** , pero Qué ocurre con la primera parte de la ruta? Eso es tomado en cuenta en la variable de ambiente CLASSPATH , que es, en mi máquina:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Observe que el CLASSPATH puede contener un número alternativo de rutas de

búsqueda.

Sin embargo, hay una diferencia cuando se usan ficheros JAR. Debe poner el nombre del ficher JAR en el classpath, no solo la ruta donde se encuentra. Así, para un JAR llamado **grape.jar** su classpath incluiría:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Una vez que el classpath está instalado correctamente, el siguiente fichero puede ser situado en cualquier directorio:

```
//: c05:LibTest.java
// Usa la libreria.
import com.bruceeckel.simple.*;
public class LibTest {
   public static void main(String[] args) {
      Vector v = new Vector();
      List l = new List();
   }
} ///:~
```

Cuando el compilador encuentra la sentencia **import** , comienza investigando en los directorios especificados por CLASSPATH, buscando el subdirectorio com\bruceeckel\simple, y hallando los ficheros compilados con el nombre adecuado (**Vector.class** para **Vector** y **List.class** para **List**).

Observe que tanto las clases como los métodos utilizados in **Vector** y **List** deben ser **public** .

Ajustar el CLASSPATH ha sido tal aventura para los usuarios novatos en Java (para mí lo fue, cuando empecé) que Sun hizo el JDK en Java 2 un poco más listo. Encontrará que cuando lo instale, incluso si no establece un CLASSPATH será capaz de compilar y ejecutar programas básicos en Java. Sin embargo, para compilar y ejecutar el paquete del código fuente para este libro (disponible en el CD ROM incluido con este, o en www.BruceEckel.com), necesitará hacer algunas modificaciones a su CLASSPATH (estas son explicadas en el paquete del código fuente).

Conflictos

Qué ocurre si 2 librerias son importadas via * e incluyen los mismos nombres? Por ejemplo, suponga un programa que hace esto:

```
import com.bruceeckel.simple.*;
import java.util.*;
```

Ya que **java.util.*** también contiene una clase **Vector** , esto causa un conflicto en potencia. No obstante, mientras no escriba código que no cause conflictos es correcto- de esta forma no terminará escribiendo mucho para evitar colisiones que tal vez no se produciesen.

El conflicto ocurre si ahora intenta crear un **Vector** :

```
Vector v = new Vector();
```

A qué clase **Vector** se hace referencia? El compilador no lo sabe, y el lector no puede saberlo tampoco. Por tanto el compilador da error y le obliga a ser explícito. Si quiero el **Vector** Java, por ejemplo, debo decir:

```
java.util.Vector v = new java.util.Vector();
```

Ya que esto (junto con el CLASSPATH) especifica completamente la localización de ese **Vector** , no hay necesidad para la sentencia **import java.util.*** a menos que yo esté usando otra cosa de **java.util** .

Una librería de herramientas personalizada

Con lo que ya sabe, puede crear sus propias librerías de herramientas para reducir o eliminar código duplicado. Considere, por ejemplo, la creación de un alias para **System.out.println( )** y así no tener que escribir algo tan largo. Esto puede ser parte de una paquete llamado **tools:**

```
//: com:bruceeckel:tools:P.java
// Las abreviaturas P.rint y P.rintln.
package com.bruceeckel.tools;
public class P {
   public static void rint(String s) {
      System.out.print(s);
   }
   public static void rintln(String s) {
      System.out.println(s);
   }
} ///:~
```

Puede utilizar esta contracción para imprimir un **String** con salto de línea (**P.rintln( )**) or sin salto (**P.rint( )**).

Puede apostar que la ubicación de ese fichero debe ser en un subdirectorio que cominece en una de las ubicaciones del CLASSPATH y que luego continua con com/bruceeckel/tools. Después de compilar, el fichero P.class puede ser utilizado en cualquier lugar de su sistema con la sentencia import:

```
//: c05:ToolTest.java
// Usa la libreía tools.
import com.bruceeckel.tools.*;
public class ToolTest {
   public static void main(String[] args) {
      P.rintln("Disponible de ahora en adelante!");
      P.rintln("" + 100); // Obligada a ser un String
      P.rintln("" + 100L);
      P.rintln("" + 3.14159);
   }
} ///:~
```

Observe que todos los objetos pueden fácilmente ser convertidos al tipo **String** poniéndolos en una expresión de tipo **String** ; en el caso de arriba, comenzar la expresión con una cadena vacía sirve de truco. Pero esto nos trae una interesante observación.Si llama a **System.out.println(100)** , esto funciona sin convertir a cadena(**String**). Con un poco de sobrecarga, puede hacer que la clase **P** haga esto también (esto es un ejercicio al final de este capítulo).

Por tanto, de ahora en adelante, siempre que aparezca algo que sea de utilidad, puede añadirlo al directorio tools. (O a su directorio personal util o tools.)

Usando importaciones para cambiar el comportamiento

Una carácterística que está ausente de Java es la *compilación condicional* de C, que le permite cambiar un parámetro (switch) y conseguir diferentes comportamientos sin cambiar ningún otro código. La razón de que tal característica haya sido dejada en Java es probablemente el que la mayor parte del uso que se le da en C es para resolver cuestiones de cambio de plataforma: diferentes partes de código son compiladas dependiendo de la plataforma donde se compile. Ya que se pretende que Java sea multiplataforma, tal característica no debería ser necesaria.

Si embargo, hay otros usos importantes para la compilación condicional. Uno muy común es para depurar código. Las características de depurado son activadas durante el desarrollo y desactivadas en el producto final. Allen Holub

(www.holub.com) propuso la idea de utilizar paquetes para simular la compilación condicional. EL usó esto para crear una versión Java del muy útil mecanismo de afirmación de C, donde usted usted decir "esto debería ser verdadero" o "esto debería ser falso" y si la sentencia no coincide con su afirmación, lo sabrá. Tal herramienta es bastante útil durante la depuración.

Aquí está la clase que usará para depurar:

```
//: com:bruceeckel:tools:debug:Assert.java
// Herramienta de afirmación para depurar.
package com.bruceeckel.tools.debug;
public class Assert {
   private static void perr(String msg) {
      System.err.println(msg);
   }
   public final static void is_true(boolean exp) {
      if(!exp) perr("Assertion failed");
   }
   public final static void is_false(boolean exp){
      if(exp) perr("Assertion failed");
   }
   public final static void
   is_true(boolean exp, String msg) {
      if(!exp) perr("Assertion failed: " + msg);
   }
   public final static void
   is_false(boolean exp, String msg) {
      if(exp) perr("Assertion failed: " + msg);
   }
} ///:~
```

Esta clase simplemente encapsula un test para los Booleanos, que imprime un mensaje de error si falla. En el capítulo 10, aprenderá una herramienta más sofisticada para tratar con errores llamada *manejo de excepciones* , pero el método **perr()** funcionará bien mientras tanto.

La salida es impresa al dispositivo estándar de error (*standard error stream*) escribiendo a **System.err** .

Cuando quiera usar este clase, añada una linea en su programa:

```
import com.bruceeckel.tools.debug.*;
```

Para eliminar las las afirmaciones y poder despachar el código , se crea una

segunda clase **Assert** , pero en un paquete diferente:

```
//: com:bruceeckel:tools:Assert.java
// Desactivando la salida de la afirmación
// para poder despachar el programa.
package com.bruceeckel.tools;
public class Assert {
   public final static void is_true(boolean exp){}
   public final static void is_false(boolean exp){}
   public final static void
   is_true(boolean exp, String msg) {}
   public final static void
   is_false(boolean exp, String msg) {}
} ///:~
```

Ahora si cambia la anterior sentecia **import** :

```
import com.bruceeckel.tools.*;
```

El programa ya no imprimirá las afirmaciones. Aquí tiene un ejemplo:

```
//: c05:TestAssert.java
// Demostración de la herramienta de afirmación.
// Comente y descomente las líneas siguientes
// para cambiar el comportamiento de afirmación:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;
public class TestAssert {
   public static void main(String[] args) {
      Assert.is_true((2 + 2) == 5);
      Assert.is_false((1 + 1) == 2);
      Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
      Assert.is_false((1 + 1) == 2, "1 +1 != 2");
   }
} ///:~
```

Al cambiar el **package** que es importado, puede cambiar su código de la versión en pruebas a la version final. Esta técnica puede ser usada para todo tipo de código condicional.

Package caveat

Merece la pena recordar que siempre que cree un paquete, especifica implícitamente una estructura de directororio al dar nombre al paquete. El paquete debe residir en el directorio indicado por su nombre, el cual debe ser un directorio que se pueda buscar comenzando desde el CLASSPATH.

Experimentar con la palabra reservada **package** puede ser un poco frustrante al principio, porque a menos que ajuste el nombre del paquete a las reglas anteriores obtendrá muchos mensajes en tiempo de ejecución que hablan de no ser capaz de encontrar una clase, incluso si esa clase esta ahí en el mismo directorio. Si obtiene un mensaje asi, intente poner entre comentarios la sentencia **package** y si así funciona, sabrá donde está el problema.

Especificadores de acceso en Java

Cuando son usados, los especificadores de acceso en Java **public** , **protected** , y **private** se colocan delante de cada declaración para cada miembro de su clase, si se trata de un campo o un método. Cada especificador controla el acceso pero solo para una definición en concreto. Esto es una diferencia con C++, en el que los especificadores de acceso afectan todas las definiciones que aparecen tras ellos hasta que otro especificador apararezca.

De una u otra forma, todo los elementos tienen especificado algun tipo de acceso. En las siguientes secciones, aprenderá varios tipos de acceso, empezando con el acceso por defecto.

"Friendly"

Qué ocurre si no pone especificador de acceso tal y como ocurre en los ejemplos anteriores a este capítulo? El acceso por defecto no tiene palabra resarvada, pero es normalmente conocido como "friendly" (amiga). Significa que todo el resto de clases del paquete actual tienen acceso a un miembro friendly, pero para el resto de clases fuera del paquete, el miembro aparece como **private** (privado). Ya que una unidad de compilación -un fichero- puede pertenecer a un solo paquete, todas las clases dentro de una unidad de compilación son automáticamente amigas (friendly) unas de otras. Así, los elementos friendly también se dice que tienen *acceso de paquete* .

El acceso Friendly le permite agrupar clases relacionadas en un paquete para que puedan interactuar fácilmente unas con otras. Cuando coloca clases juntas en un paquete (garantizando así el acceso mutuo a sus miembros friendly; haciéndolos "amigos") usted "posee" el código en ese paquete. Tiene sentido que sólo el código que usted posee pueda tener acceso a oltro código de su propiedad. Usted podría decir que el acceso amistoso le d aun signifcicado o una razón al agrupamiento de clases un un paquete. En muchos lenguajes la forma en que

usted organiza sus definiciones en archivos puede ser willy-nilly, pero en Java está obligado a hacerlo en una forma sensible. En suma, probablemente quiera excluir clases que no deberían tener acceso a las clases definidas en el paquete actual.

La clase contorla qué código tiene acceso a sus miembros. No hay froma mágica de "entrar a la fuerza". El código de otro paquete no puede aparecer y decir "Hola, soy amigo de **Bob** !" y esperar ver los miembros **protected** , friendly, y **private** de **Bob** . El único modo de conceder acceso a un miembro es:

1. Hacer el miembro **public** . Entonces todo el mundo, en cualquier lado, puede acceder a él.
2. Hacer el miembro friendly al no indicar ningún especificador de acceso, y poniendo las otras clases en el mismo paquete. Entonces las otras clases pueden acceder al miembro.
3. Como verá en el Capítulo 6, cuando comentemos la herencia, una clase derivada puede acceder a un miembro **protected** igual que a un miembro **public** (pero no a miembros **private**). Puede acceder a los miembros friendly sólo si las dos clases están en el mismo paquete. Pero no se preocupe por eso ahora.
4. Proporcionar métodos para "acceso/mutación" (también conocidos como métodos "get/set") que leen y cambian el valor. Esta es la aproximación más civilizada en terminos de POO, y es fundamental con JavaBeans, como verá en el Capítulo13.

public : interfaz de acceso

Cuando usa la palabra reservada **public** , significa que la declaración del identificador que va detrás de public está disponible para todo el mundo, en particular para el programador cliente que usa la librería. Suponga que define un paquete **dessert** que contiene la siguiente unidad de compilación:

```
//: c05:dessert:Cookie.java
// Crea una librería.
package c05.dessert;
public class Cookie {
   public Cookie() {
      System.out.println("Constructor de Cookie");
      void bite() { System.out.println("bite"); }
   } ///:~
```

Recuerde, **Cookie.java** debe estar en un subdirectorio llamado **dessert** , en un directorio bajo **c05** (indicando Capítulo 5 de este libro) que debe estar bajo uno de los directorios del CLASSPATH. No cometa el error de pensar que Java siempre mirará en el directorio actual como uno de los puntos de comienzo a la hora de buscar. Si no tiene un punto '.'como ruta en su CLASSPATH, Java no mirará ahí. Ahora si crea un programa que utilice **Cookie** :

```
//: c05:Dinner.java
// Usa la librería
import c05.dessert.*;
public class Dinner {
   public Dinner() {
      System.out.println("Constructor de Dinner");
   }
   public static void main(String[] args) {
      Cookie x = new Cookie();
      //! x.bite(); // No puede acceder
   }
} ///:~
```

Puede crear un objeto **Cookie** , ya que su constructor es **public** y la clase es **public** . (Profundizaremos en el concepto de clase **public** más tarde.) Pero, el miembro **bite( )** no es accesible dentro de **Dinner.java** ya que **bite( )** es amigo solo dentro del paquete **dessert** .

El paquete por defecto

Puede que le sorprenda descubrir que el siguiente código compila, aunque a primera vista parece que no cumple las reglas:

```
//: c05: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();
   }
} ///:~
```

En un segundo fichero, en el mismo directorio:

```
//: c05:Pie.java
// La otra clase.
class Pie {
   void f() { System.out.println("Pie.f()"); }
} ///:~
```

Podría inicialmente ver esto como ficheros totalmente ajenos, y sin embargo la

clase **Cake** es capaz de crear un objeto **Pie** y llamar a su método **f( )** !! (Observe que debe tener '.' en su CLASSPATH para que los ficheros compilen.) Normalmente pensaría que **Pie** y **f( )** son friendly y por lo tanto no disponible para **Cake** . Son friendly-eso es correcto. La razón de que esté disponible en **Cake.java** es porque están en el mismo directorio y tienen nombre de paquete no explícito. Java trata ficheros así como si fueran parte del "paquete por defecto" para ese directorio, y por lo tanto friendly a otros ficheros de ese directorio.

private: no puedes tocar eso!

La palabra reservada **private** significa que nadie puede acceder a ese miembro excepto la clase en sí, dentro de los métodos de la misma. Las otras clases en el mismo paquete no pueden acceder a miebros **private** , de manera que es como si la estuviera aislando de usted mismo. Por otro lado, es probable que un paquete pueda ser creado por varias personas colaborando juntas, asi que **private** le permite cambiar ese miembro sin preocuparse de que eso afectará a otra clase del mismo paquete. El acceso de paquete por defecto "friendly" a menudo proporciona una adecuado grado de ocultación; recuerde, un miembro "friendly" es inaccesible para el usuario del paquete. Esto está bien, ya que el acceso por defecto es el que normalmete usa (y el que obtendrá si olvida añadir cualquier control de acceso). Así, normalmente pensará en el acceso para los miembros que quiera hacer explícitamente **public** para el programador cliente y, como resultado, podría pensar incialmente que no usará la palabra clave **private** frecuentemente, ya que es tolerable estar sin ella. (Este es un contraste distinto con C++). Sin embargo, ocurre que el uso consistente de **private** es muy importante, especialmente cuando nos concierne el multithreading. (Como verá en el Capítulo 14.)

Aquí tiene un ejemplo del uso de **private** :

```
//: c05:IceCream.java
// Demuestra 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();
   }
} ///:~
```

Esto muestra un ejemplo en el cual **private** viene bien: podría querer controlar

como un objeto es creado y evitar que alguien acceda a un constructor en particular (o a todos). En el ejemplo de arriba, no puede crear un objeto **Sundae** a través de su constructor; en cambio debe llamar al método **makeASundae( )** para que lo haga por usted[2].

Cualquier método que que está seguro que es sólo un método de ayuda para esa clase puede ser **private** , para asegurar que no lo use accidentalmente en otro lugar del paquete y prohibiéndose así de cambiar o remover el método. Hacer un método **private** garantiza que retiene esta opción.

---

2 Hay otro efecto en este caso:ya que el constructor por defecto es el único definido y es **private** , se evitará la herencia de esta clase. (A tema que será presentado en el Capítulo 6.)

Lo mismo es cierto para un campo **private** dentro de una clase. A menos que tenga que exponer la implementación subyacente (lo cual es una situación más rara de lo que usted podría pensar), debe hacer todos los campos private. Si enmbargo, solo porque una referencia a un objeto dentro de una clase es private no significa que otro objeto no puede tener una referencia public al mismo objeto. (Ver Apéndice A para temas acerca de aliasing.)

protected : "algo parecido a friendly"

Necesita leer más adelante para entender el especificador de acceso **protected** . En primer lugar, debe ser consciente de que no necesita entender esta sección para continuar con el libro pasando a la herencia (Capítulo 6). Pero para no dejarse nada atrás aquí tiene una breve descripción y un ejemplo usando **protected** . La palabra reservada **protected** se relaciona con un concepto llamado *herencia* , que toma una clase creada y añade nuevos miembros a esa clase sin modificarla, y a la que nos refereriremos como clase base. Puede también cambiar el comportamiento de los métodos creados de la clase. Para heredar de una clase creada, se dice que nuestra nueva clase extiende (extends) una clase existente, de este modo:

```
class Foo extends Bar {
```

El resto de la definición de la clase no varía. Si crea un nuevo paquete y hereda de una clase en otro paquete, a los únicos miembros que tiene acceso son los miembros public del paquete original. (Por supuesto, si realiza la herencia en el *mismo* paquete, tiene el acceso normal a todos los miembros "friendly".) A veces el creador de la clase base, le gustaría tomar un miembro en particular y garantizar acceso a las clases derivadas pero no al resto. Eso es lo que **protected** hace. Si nos fijamos en el anterior fichero **Cookie.java** , la siguiente clase no puede acceder al miembro "friendly":

```
//: c05:ChocolateChip.java
// No puede acceder a un miembro friendly
// de otra clase.
import c05.dessert.*;
public class ChocolateChip extends Cookie {
   public ChocolateChip() {
      System.out.println(
         "Constructor de ChocolateChip");
   }
   public static void main(String[] args) {
      ChocolateChip x = new ChocolateChip();
      //! x.bite(); // No puede acceder a bite
   }
} ///:~
```

Una de las cosas interesantes de la herencia es que sin un método **bite( )** existe en la clase **Cookie** , también existe luego en cualquier clase derivada de **Cookie** . Pero dado que **bite()** es "friendly" en un paquete distinto, no está disponible para nosotros en este. Por supuesto, podría hacerlo **public** , pero luego todos tendrían acceso y quizás no quiera eso. Si cambiamos la clase **Cookie** como sigue:

```
public class Cookie {
   public Cookie() {
      system.out.println("Constructor de Cookie");
   }
   protected void bite() {
      System.out.println("bite");
   }
}
```

entonces **bite()** todavía tiene acceso "friendly" dentro del paquete **dessert** , pero es también accesible para todos los que hereden de **Cookie** . Sin embargo, no es **public** .

Interfaz e implementación

El control de acceso está con frecuencia se refiere al *ocultamiento dela implementación* . Envolver datos y métodos dentro de clases combinado con la ocultación de la implementación es con frecuencia llamado *encapsulamiento* [3]. El resultado es un tipo de dato con características y comportamientos.

---

3 No obstante, la gente frecuentemente se refiere al ocultamiento de la implementación sólo como encapsulamiento.

El control de acceso crea fronteras dentro de un tipo de datos por dos importantes razones. La primera es establecer lo que el programador cliente puede y no puede usar. Puede construir mecanismos internos a la estructura sin preocuparse de que los programadores cliente accidentalmente traten las partes internas como parte de la interfaz que deberían usar.

Esto nos condunce directamente a la segunda razon, que es separar la interface de la implementación. Si se usa la estructura en un conjunto de programas, pero los programadores cliente no pueden hacer nada aparte de enviar mensajes a la interfaz pública, entonces usted puede cambiar cualquier cosa que no sea pública ("friendly", protected, o private) sin requerir modificaciones al código del cliente.

Estamos ahora en el mundo de la programación orientada a objetos, donde una **clase** en realidad se describe como "una clase de objetos", tal y como describiría una clase de pescados o pajaros. Cualquier objeto perteneciente a esta clase compartirá estas características y comportamientos. La clase es una descripción del estado de los objetos de ese tipo y de su forma de actuar.

En el primer lenguaje OO, Simula-67, la palabra reservada **class** era usada para describir un nuevo tipo de datos. La misma palabra ha sido usada por la mayor parte de los lenguajes orientado a objetos. Este el punto clave de todos los lenguajes: la creación de nuevos tipos de datos que son más que cajas conteniendo datos y métodos.

La clase es el concepto OO fundamental en Java. Es una de las palabras reservadas (class) que *no* será puesta en negrita en este libro-llega a ser molesta una palabra tantas veces repetida como "class".

Por claridad, podría preferir un estilo de creación de clases que ponga los miembros **public** al principio, seguido por los miembros **protected** , friendly, y **private** . La ventaja es que el usuario de la clase puede leer de abajo a arriba y ver al principio que es lo importante (los miembros **public** , porque ellos pueden ser accedidos fuera del fichero), y dejar de leer cuando encuentran los miembros no **public** , que son parte de la implementación interna:

```
public class X {
   public void pub1( ) { /* . . . */ }
   public void pub2( ) { /* . . . */ }

   public void pub3( ) { /* . . . */ }
   private void priv1( ) { /* . . . */ }
   private void priv2( ) { /* . . . */ }
   private void priv3( ) { /* . . . */ }
   private int i;
   // . . .
}
```

Esto solo lo hará fácil de leer en parte porque la interfaz y la implementación están mezcladas aún. De ese modo, verá el código fuente -la implementación- porque está justo ahí en la clase. Además, la documentación comentada soportada por javadoc (descrito en el Capítulo 2) reduce la importancia de la habilidad de leer código por el cliente programador. Enseñar la interfaz al consumidor de la clase es en realidad trabajo del *class browser* , una herramienta cuyo trabajo es mirar todas las clases disponibles y mostrarle que puede hacer con ellas(por ejemplo, los miebros que están disponibles) de un modo útil. En el momento en que lee esto, los browsers deben ser una parte a incluir en cualquier buen entorno de desarrollo Java.

Acceso a las clases

En Java, los especificadores de acceso pueden también ser usados para determinar las clases de una librería que estarán disponibles para los usuarios de esa librería. Si quiere que una clase esté disponible para un programador cliente, coloque la palabra **public** en algún lugar antes del la llave de apertura del cuerpo de la clase. Esto controla incluso si el programador cliente puede crear un objeto de esa clase.

Para controlar el acceso de una clase, el especifiacador debe aparecer antes de class. Así, puede poner:

```
public class Widget {
```

ahora si el nombre de su librería es **mylib** cualquier programador cliente puede acceder a **Widget** diciendo

```
import mylib.Widget;
```

o

```
import mylib.*;
```

No obstante, hay un conjunto de restricciones extra:

1. Puede haber solo un una clase **public** por unidad de compilación (fichero). La idea es que cada unidad de compilación tiene una única interfaz pública, representada por esa clase **public** . Puede tener muchas clases "friendly" de apoyo comoa quiera. Si tiene más

de una clase **public** dentro de una unidad de compilación, el compilador le dará un mensaje de error.

2. El nombre de la clase **public** debe ser exactamente el mismo que el del fichero que contiene la unidad de compilación, distinguiendo mayúsculas de minúsculas. Así para **Widget** , el nombre del fichero debe ser **Widget.java** , no **widget.java** o **WIDGET.java** . Una vez más, obtendría un error en tiempo de compilación si son diferentes.
3. Es posible, aunque no habitual, tener una unidad de compilación sin clases **public** . En ese caso, puede llamar al fichero como más le guste.

Que pasa si usted tiene una clase dentro de **mylib** que está usando sólo para llevar a cabo las tareas realizadas por **Widget** o alguna otra clase pública en **mylib** ? Usted no quiere tomarse la molestia de escribir documentación para el programador cliente y piensa que en algún momento más tarde podría querer cambiar completamente las cosas y desmenuzar completamente su clase, sustituyéndola por una diferente. Para conseguir esta flexibilidad, necesita asegurase de que ningún programador cliente se vuelva dependiente de sus detalles particulares de implementación ocultos dentro de **mylib** , Para llevar a cabo esto, sólo quite la palabra clave **public** de la clase, de manera que se vuelva "friendly" (Esta clase puede ser usada sólo dentro de este paquete).

Note que la clase no puede ser **private** (que la haría inaccesible a todos, excepto a ella misma), o **protected** 4. De manera que sólo tiene dos opciones para el acceso a la clase: "friendly" o **public** . Si no quiere que otro tenga acceso a esa clase, puede hacer que todos los constructores sean **private** , previniendo consecuentemente a todos de crear un objeto de esa clase, excepto a usted desde dentro de un miembro **static** de la clase5. Aquí hay un ejemplo:

```
//: c05:Lunch.java
// Demuestra los especificadores de acceso a clase.
// Hace una clase efectivamente privada
// con constructores privados:
class Soup {
   private Soup() {}
   // (1) Permite la creación por medio de un método estático:
   public static Soup makeSoup() {
      return new Soup();
   }
   // (2) Crea un objeto estático y
   // retorna una referencia al requerimiento.
   // (El patrón "Singleton"):
   private static Soup ps1 = new Soup();
   public static Soup access() {
      return ps1;
   }
   public void f() {}
```

```
}
class Sandwich { // Usa Lunch
   void f() { new Lunch(); }
}
// Sólo se permite una clase pública por archivo:
public class Lunch {
   void test() {
      // No permitido! Constructor privado:
      //! Soup priv1 = new Soup();
      Soup priv2 = Soup.makeSoup();
      Sandwich f1 = new Sandwich();
      Soup.access().f();
   }
} ///:~
```

---

4 Realmente, una clase interna puede ser privada o protegida, pero es un caso especial. Estas serán presentadas en el capítulo 7.

---

5 También lo puede hacer heredando (Capítulo 6) de esa clase.

Hasta ahora, la mayoría de los métodos han estado retornando **void** o un tipo primitivo, de manera que la definición:

```
public static Soup access() {
   return ps1;
}
```

puede parecer un poco confusa al principio. La palabra previa al nombre del método (**access**) dice qué retorna este método. Hasta ahora, esto ha sido en su mayor parte **void** , lo que significa que no retorna nada. Pero además se puede retornar una referencia a un objeto, que es lo que ocurre aquí. Este método retorna una referencia a un objeto de clase **Soup** .

La clase **Soup** muestra como prevenir la creación directa de una clase haciendo todos los constructores **private** . Recuerde que si no crea al menos un constructor explícitamente, el constructor por defecto (un constructor sin argumentos) será creado automáticamente. Escribiendo el constructor por defecto, no será creado automáticamente. Haciéndolo **private** , nadie puede crear un objeto de esa clase. Pero ahora Cómo usa alguien esta clase? El ejemplo anterior muestra dos opciones. Primero, se escribe un método **static** que crea un **Soup** nuevo y retorna una referencia al mismo. Esto puede ser útil si quiere hacer alguna operación extra sobre el **Soup** antes de retornarlo o si quiere mantener la cuenta de cuantos objetos **Soup** crea (quizás para restringir su población).

La segunda opción usa lo que se llama un *patrón de diseño* , el cual es cubierto en *Thinking in Patterns with Java* , descargable de *www.BruceEckel.com* . Este patrón

particular se llama "singleton" porque permite que se cree sólo un único (single) objeto. El objeto de clase **Soup** es creado como un miembro **static private** de **Soup** , de manera que hay sólo uno, y no puede obtenerlo a no ser por el método **access()** público.

Como se mencionó previamente, si no coloca un especificador de acceso para la clase, por defecto es "friendly". Esto significa que un objeto de esa clase puede ser creado por cualquier otra clase en el paquete, pero no por una externa al mismo. (Recuerde que todos los archivos del mismo directorio que no tienen una declaración de **package** explícita son implícitamente parte del paquete por defecto para ese directorio). Por supuesto, si un miembro **static** de esa clase es **public** , el programador cliente puede todavía acceder a ese miembro aunque no puedan crear un objeto de esa clase.

Resumen

En toda relación es importante tener límites que sean respetados por todas las partes involucradas. Cuando usted crea una librería, establece una relación con el usuario de la misma -el programador cliente- quien es otro programador, pero que está reuniendo una aplicación o usando su librería para construir una mayor.

Sin reglas, los programadores cliente pueden hacer lo que quieran con los miembros de una clase, incluso si usted prefiere que no manipulen directamente algunos miembros. Todo está desnudo al mundo.

Este capítulo dio un vistazo a cómo construir clases para formar librerías; primero, la forma en que un grupo de clases es empaquetado dentro de una librería y, segundo, la forma en que la clase controla el acceso a sus miembros.

Se estima que un proyecto de programación en C comienza a averiarse al llegar a algún lugar entre las 50.000 y 100.000 líneas de código porque C tiene un "espacio de nombres" único, de manera que los nombres comienzan a conflictuar, causando una sobrecarga extra en la gestión. En Java, la palabra clave **package** , el esquema de nombres de paquetes y la palabra clave **import** le dan un control completo sobre los nombres, de manera que la cuestión del conflicto de nombres es fácilmente evitada.

Hay dos razones para controlar el acceso a los nombres. La primera es para mantener las manos de los usuarios lejos de las herramientas que no deberían tocar; las herramientas son necesarias para las maquinaciones internas del tipo de dato, pero no una parte de la interfaz que los usuarios necesitan para resolver sus problemas particulares. De manera que hacer los **private** los métodos y campos es un servicio al usuario porque pueden ver fácilmente qué es importante para ellos y qué pueden ignorar. Simplifica su entendimiento de la clase.

La segunda razón más importante para el control de acceso es permitirle al diseñador de la librería cambiar el trabajo interno de la clase sin preocuparse acerca de cómo afectará al programador cliente. Podría construir una clase de una

forma primero y luego descubrir que reestructurando su código se obtendrá una velocidad mayor. Si la interfaz y la implementación son separadas y protegidas claramente, usted puede realizar esto sin forzar al usuario a reescribir su código.

Los especificadores de acceso en Java le dan al creador de la clase un control valioso. El usuario de la clase puede ver claramente qué puede usar y qué puede ignorar. Más importante aún, es la capacidad de asegurar que ningún usuario se vuelva dependiente de cualquier parte de la implementación subyacente de una clase. Si usted, como creador de la clase, sabe esto puede cambiar la implementación subyacente con el conocimiento de que ningún programador cliente será afectado por los cambios porque ellos no pueden acceder a esa parte de la clase.

Cuando tiene la capacidad de cambiar la implementación subyacente, usted puede no sólo mejorar su diseño más tarde, sino que además tiene la libertad de cometer errores. No importa que tan cuidadosamente planee o diseñe, cometerá errores. Saber que es relativamente fácil cometer esos errores significa que será más experimental, aprenderá más rápido y terminará su proyecto más pronto.

La interfaz pública de una clase es lo que el usuario *ve* , de manera que la parte más importante de la clase a hacer bien durante el análisis y diseño. Incluso esto brinda algo de libertad para el cambio. Si usted no hace la interfaz bien la primera vez, puede *agregar* métodos, en cuanto no remueva ninguno de los que los programadores cliente han usado ya en su código

Ejercicios

Las soluciones a los ejercicios se pueden encontrar en el documento electrónico The Thinking in Java Annotated Solution Guide, disponible por un pequeño monto en www.BruceEckel.com.

1. Escriba un programa que cree un objeto **ArrayList** sin importar explícitamente **java.util.*** .
2. En la sección titulada "paquetes: la librería unidad" convierta los fragmentos de código concernientes a **mypackage** en un conjunto de archivos Java que compilen y corran.
3. En la sección titulada "Conflictos" tome los fragmentos de código y conviértalos en un programa y verifique los conflictos que de hecho ocurren.
4. Haga más general la clase **P** definida en este capítulo agregando todas las versiones sobrecargadas de **rint( )** y **rintln( )** necesarias para manejar todos los tipos básicos de Java.
5. Cambie la sentencia import en **TestAssert.java** para habilitar y deshabilitar el mecanismo de afirmación.
6. Construya una clase con datos y métodos miembro que sean **public** , **private** , **protected** y "friendly". Cree un objeto de esta clase y vea que tipo de errores de compilador obtiene cuando intenta acceder a todos los miembros de la clase. Sea consiente de que las clases del mismo directorio son parte del mismo paquete "por

defecto".

7. Construya una clase con datos **protected** . Construya una segunda clase en el mismo archivo que manipule los datos **protected** de la primera clase.
8. Cambie la clase **Cookie** como se especifica en la sección "protected: algo parecido a friendly". Verifique que **bite( )** no es **public** .
9. En la sección titulada "Acceso a las clases" encontrará fragmentos de código que describen a **mylib** y a **Widget** . Construya esta librería, luego cree un **Widget** en una clase que no sea parte del paquete **mylib** .
10. Cree un nuevo directorio y edite su CLASSPATH para que lo incluya. Copie el archivo **P.class** (producido compilando **com.bruceeckel.tools.P.java**) a su nuevo directorio y luego cambie los nombres del archivo, la clase **P** adentro y los nombres de los métodos. (También podría agregar salida adicional para ver cómo funciona). Construya otro programa en un directorio diferente que use su nueva clase.
11. Siguiendo la forma del ejemplo **Lunch.java** , construya una nueva clase llamada **ConnectionManager** que gestione un arreglo fijo de objetos **Connection** . El programador cliente debe ser ahora capaz de crear objetos **Connection** explícitamente, pero sólo puede obtenerlos vía un método **static** en **ConnectionManager** . Cuando **ConnectionManager** se queda sin objetos, retorna una referencia nula (**null**). Pruebe las clases en **main( )** .
12. Construya el siguiente archivo en el directorio c05/local (presumiblemente en su CLASSPATH):

```
package c05.local;
class PackagedClass {
   public PackagedClass() {
      System.out.println(
         "Creación de una clase empaquetada");
   }
} ///:~
```

Luego construya el siguiente archivo en un directorio distinto de c05:

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```

6: Reusando Código y clases

Una de las características más complejas de Java es el reuso de código. Pero, para ser revolucionario, se debe poder hacer más que copias de código y cambios.

Esto es el enfoque usado en lenguajes procedimentales como C, que no ha funcionado demasiado bien. Como todo en Java, el problema se resuelve con las clases. Para reusar código creamos nuevas clases pero, en lugar de partir de cero, partimos de clases, relacionadas con nuestra clase, que han sido ya creadas y depuradas.

El truco está en usar las clases sin ensuciar el código existente. En este capítulo veremos dos formas de llevar a cabo esto. La primera es bastante sencilla: simplemente consiste en creas objetos de nuestra clase existente dentro de la nueva clase. Esto se conoce como *composición* porque la nueva clase está compuesta de objetos de clases existentes. Estamos reusando la funcionalidad del código, y no la forma.

La segunda manera de aprovechar el código es más ingeniosa. Se crea una nueva clase como un tipo de una clase ya existente. Tomamos la forma de la clase existente y añadimos código a la nueva, sin modificar la clase existente. Esta forma de crear nuevos objetos se llamada *herencia* , y el compilador hace la mayoría del trabajo. La herencia es una de las piedras angulares de la programación orientada a objetos, y tiene implicaciones adicionales que exploraremos en el próximo capítulo.

Tanto la composición como la herencia tienen sintaxis y comportamiento similares (lo que tiene sentido porque ambos son formas de crear nuevos tipos a partir de tipos existentes). En este capítulo, aprenderemos esos mecanismos de reuso del código.

Sintaxis de la Composición

Hasta ahora, hemos utilizado la composición con cierta frecuencia. Simplemente, se trata de colocar manejadores de objetos dentro de nuevas clases. Por ejemplo, supongamos que queremos un objeto que contenga varios objetos **String** , un par de primitivas (variables simples) y un objeto de otra clase. Para los objetos que no son primitivas, colocamos los manejadores (manejadores de los objetos de una clase) dentro de la nueva clase, y las primitivas las definimos dentro de la clase: (Vea la pagina XX si tiene problemas al ejecutar este programa.)

```
//: SprinklerSystem.java
// Composition for code reuse
package c06;

class WaterSource {
private String s;

  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }

  public String toString() { return s; }
  }

public class SprinklerSystem {
private String valve1, valve2, valve3, valve4;
WaterSource source;
int i;
float f;

  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }

  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

Uno de los métodos definidos en **WaterSource** es especial: **toString( )** . Más tarde mostraremos que todos los objetos presentan un método **toString( )** , que es invocado en situaciones especiales cuando el compilador quiere una cadena pero está en uno de esos objetos. La expresión es de esta manera:

```
System.out.println("source = " + source);
```

el compilador ve que estamos intentando añadir un objeto **String** ("source = ") a un **WaterSource** . Esto no tiene sentido, porque a un objeto **String** sólo podemos añadirle un objeto **String** , así que dice "transformaré **source** en un **String** mediante una llamada a **toString( )** ". Después de hacer esto puede combinar las dos **Strings** y pasar la cadena resultante a **System.out.println( )** . Si deseamos que una de nuestras clases exhiba este comportamiento, sólo necesitamos escribir un método **toString( )** .

A primera vista, puede suponer (siendo Java tan seguro y fiable como lo es) que el compilador construiría automáticamente objetos para cada uno de los manejadores de objetos (variables) de la parte de arriba del código, por ejemplo llamando al constructor por defecto de **WaterSource** para inicializar el objeto. La salida del método **print** es, en realidad:

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

Los objetos primitivos que son campos en una clase son automáticamente inicializados a cero, como se observa en el capitulo 2. Pero las variables objeto (tipos de objetos) son inicializadas a **null** y, si intentamos llamar a métodos de algunos de ellos se obtendrá excepción. Es realmente bastante bueno (y útil) que puedas sacarlos sin excepción:

Tiene sentido que el compilador no cree un objeto por defecto para cada tipo de objeto (clases) porque eso provocaría gastos innecesarios en muchos casos. Si deseamos inicializar las variables objeto (objetos de las distintas clases), podemos hacerlo:

1. En el punto en que definamos los objetos. Esto significa que ellos siempre serán inicializados antes de que el constructor sea llamado.
2. El constructor para esa clase.
3. Antes de que realmente necesitamos usarlos. Esto puede reducir gastos, si hay situaciones donde el objeto no necesitar ser creado.

Los tres enfoques se muestran aquí:

```
//: Bath.java
// Constructor initialization with composition

class Soap {
private String s;

  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }

  public String toString() { return s; }
}

public class Bath {
private String
// Initializing at point of definition:
  s1 = new String("Happy"),
  s2 = "Happy",
  s3, s4;
Soap castille;
int i;
float toy;

  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }

  void print() {
  // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }

  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

Observe que en el constructor **Bath** se ejecuta un método antes que tenga lugar ninguna de las inicializaciones. Si, al principio de la definición, no inicializamos, no hay garantías de que realicemos ninguna inicialización antes de que enviemos un mensaje a un manejador de objeto (excepto por una inevitable excepción de ejecución.

Aquí está la salida del programa:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

Cuando **print( )** es llamado se rellena **s4** así que todos los campos son iniciados apropiadamente en el tiempo en que ellos son usados (o sea cuando se usan son iniciados).

Sintaxis de la herencia

La herencia es una parte integral de Java (y se usa en lenguajes OOP en general) que fué introducida en el capítulo 1 y que ha sido usada en otros capítulos anteriores cuando la situación lo requería. Además, cada vez que creamos un objeto estamos haciendo herencia porque, aunque no indiquemos otra cosa, estamos heredando de Java el objeto **Object** , que es la raíz de la jerarquía de clases.

La sintaxis para la composición es obvia pero, para realizar la herencia, hay una forma claramente distinta. Cuando heredamos, estamos diciendo "Esta nueva clase es como esa clase antigua". Afirmamos esto en el código dando el nombre de la clase como siempre pero, antes de la apertura del Ímite cuerpo de clase, pondremos la palabra clave " **extends** " seguida por el nombre de la clase base. Cuando hagamos esto, obtendremos automáticamente todos los datos miembros y métodos de la clase base. Aquí hay un ejemplo:

```
//: Detergent.java
// Inheritance syntax & properties
```

```
class Cleanser {
  private String s = new String("Cleanser");

  public void append(String a) { s += a; }

  public void dilute() { append(" dilute()"); }

  public void apply() { append(" apply()"); }

  public void scrub() { append(" scrub()"); }

  public void print() { System.out.println(s); }

  public static void main(String[] args) {
  Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}

public class Detergent extends Cleanser {
// Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

Esto demuestra una serie de cosas. Primero, en el método **append( )** de **Cleaner** , las cadenas son concatenadas a s usando el operador +=, que es uno de los operadores (junto con "+") que los diseñadores de Java han sobrecargado para trabajar con cadenas.

Segundo, tanto **Cleaner** como **Detergent** contienen un método **main( )** . Podemos crear un método **main( )** para cada una de tus clases, y es una buena costumbre codificar de esta forma; así nuestro código de prueba (no definitivo)

estará dentro de la clase. Incluso aunque tengamos muchas clases en un programa, sólo podremos invocar al método **main( )** de la clase pública que fue invocada (y, como sólo podemos tener una clase pública por fichero, todas las posibles ambigüedades están eliminadas). Así, en este caso, cuando decimos **java Detergent** , estamos invocando a **Detergent.main( )** . Pero también podríamos decir **java Cleanser** para invocar a **Cleanser.main( )** , incluso aunque **Cleanser** no fuera una clase pública. Esta técnica de introducción el main( ) en cada clase permite una unidad fácil de prueba para cada clase. Además, no es necesario que eliminemos el método **main( )** cuando hayamos terminado la prueba; podemos dejarlo para pruebas posteriores.

Aquí, podemos ver que **Detergent.main( )** llama a **Cleanser.main( )** explícitamente.

Es importante que todos los métodos de **Cleanser** sean públicos. Recuerde que si no se indica algún acceso específico a los miembros, éstos son "amigables" por defecto, lo que sólo permite acceso a una agrupación de miembros. Entonces, dentro de este agrupamiento, alguien podía usar esos métodos sin tener acceso específico. **Detergent** por ejemplo, no tendría problemas. Sin embargo, si una clase de algunas otras agrupaciones era heredada de **Cleanser** solo podría acceder a los miembros públicos. Por tanto, para planificar por herencia, haremos por lo general privados a todos los campos y públicos a todos los métodos (posteriormente veremos que los miembros protegidos también permiten acceso por clases derivada). Claro que, en casos particulares habrá que hacer algunos ajustes, pero esto es una directiva útil.

Observe que Cleanser tiene un conjunto de métodos en su interfaz: **append( ), dilute( ), apply( ), scrub( )** y **print( )** . Como **Detergent** es una clase derivada de **Cleanser** (por la palabra reservada **extends**) automáticamente se obtiene todos esos métodos en su interface, incluso aunque no estén todos definidos explícitamente en **Detergent** . Puede entonces pensar que la herencia es, entonces, la forma de reusar el interface. (La implementación viene libremente, pero esta parte no es un punto primordial).

Cuando vimos el método **scrub( )** , es posible tomar un método que ha sido definido en la clase base y modificarlo. En este caso, podemos llamar al método de la clase base dentro de la nueva versión. Pero dentro de **scrub( )** no podemos llamar a **scrub( )** , ya que produciría una llamada recursiva, que no es lo que deseamos. Para resolver este problema Java tiene la palabra reservada **super** , que se refiere a la "superclase" de la que ha sido heredada de la clase actual. Así, la expresión **super.scrub( )** llama a la versión de la clase base del método **scrub ( )** .

Cuando se hereda, no estamos restringidos a usar los métodos de la clase base, sino que también podemos añadir nuevos métodos a la clase derivada, exactamente del mismo modo a como lo hacemos cuando ponemos algún método en una clase. La palabra reservada **extends** sugiere que vamos a añadir un nuevo

método al interfaz de la clase base, y el método **foam( )** es un ejemplo de esto.

En **Detergent.main( )** podemos ver que, para un objeto **Detergent** , podemos llamar tanto a los métodos que están disponibles en **Cleanser** como a los disponibles en **Detergent** (por ejemplo, **foam()**).

Iniciando la clase base

Como ahora hay dos clases involucradas (la clase base y la clase derivada) en lugar de una, puede ser un poco confuso intentar imaginar el objeto final producido por una clase derivada. Desde fuera, parece como si la clase nueva tuviera el mismo interfaz que la clase base y, quizás, algunos métodos y campos adicionales. Pero la herencia no es copiar el interfaz de la clase base, Cuando creamos un objeto de una clase derivada, este contiene dentro un subobjeto de la clase base. Este subobjeto es el mismo que si hubiésemos creado un objeto de la clase base. Lo que pasa es que es, desde fuera, el subobjeto de la clase base está envuelto dentro del objeto de la clase derivada.

Por supuesto, es esencial que el subobjeto de la clase base sea inicializado correctamente y solo hay una forma para garantizar eso: realizar la inicialización en el constructor, llamando al constructor de la clase base, que tiene todos los privilegios y conocimiento apropiados para realizar la inicialización de la clase base. Java inserta automáticamente llamadas al constructor a la clase base en el constructor de la clase derivada. El siguiente ejemplo muestra esto trabajando con tres niveles de herencia:

```
//: Cartoon.java
// Constructor calls during inheritance

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
```

```
  }
} ///:~
```

La salida de este programa muestra las llamadas automáticas:

```
Art constructor
Drawing constructor
Cartoon constructor
```

Podemos comprobar que la construcción ocurre desde la base más exterior. Así, la clase base es inicializada antes de poder acceder al constructor de la clase derivada.

Incluso aunque no creemos un constructor para **Cartoon( )** , el compilador sintetizará un constructor por defecto para que llamemos al constructor de la clase base.

Constructores con argumentos

El ejemplo anterior tiene constructores por defecto; es decir, ellos no tienen ningún argumento. Esto facilita al compilador llamar a estos (constructores) porque no hay pregunta sobre que argumentos pasar. Si tu clase no tiene argumentos por defecto o si deseamos invocar a un constructor de la clase base que tiene un argumento, debemos escribir explícitamente las llamadas al constructor de la clase base usando la palabra reservada **super** y la lista apropiada de argumentos:

```
//: Chess.java
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}
```

```
public class Chess extends BoardGame {
  Chess() {
   super(11);
   System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

Si no llamamos al constructor de la clase base en **BoardGame( )** , el compilador se quejará de que no se puede encontrar un constructor de la forma **Game( )** . Además, la llamada al constructor de la clase base debe ser la primera cosa que hagamos en el constructor de la clase derivada. (El compilador dará un mensaje de error si no se hace así).

Capturando excepciones del constructor base

Como habrá observado, el compilador nos fuerza a colocar la primera llamada al constructor de la clase base en el cuerpo del constructor de la clase derivada. Esto significa que no se puede colocar nada antes. Como veremos en el Capítulo 9, esto también impide a un constructor de la clase derivada coger alguna excepción que viene de una clase base, lo cual puede resultar un inconveniente a veces.

Combinando composición y herencia

Es muy común usar composición y herencia juntos. El siguiente ejemplo muestra la creación de una clase más compleja, usando tanto herencia como composición, junto con la necesaria inicialización del constructor:

```
//: PlaceSetting.java
// Combining composition & inheritance

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println(
                    "DinnerPlate constructor");
  }
}
```

```
class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
 }

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  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);
    System.out.println(
                     "PlaceSetting constructor");
  }
  public static void main(String[] args) {
```

```
    PlaceSetting x = new PlaceSetting(9);
  }
} ///:~
```

Aunque el compilador nos obliga fuerza a inicializar las clases base, y exige que lo hagamos correctamente al principio del constructor, él no comprueba si hemos iniciado los objetos miembros. Por tanto, hemos de poner atención en este punto.

Garantizando limpieza correcta

Java no tiene lo que en C++ es el concepto de un *destructor* , un método que es automáticamente llamado cuando un objeto es destruido. La razón es probablemente que la practica en Java es simplemente olvidar objetos antes que destruirlos, permitiendo al recolector de basura recuperar la memoria como sea necesario.

A menudo esto es magnífico, pero hay ocasiones en las que nuestra clase realiza algunas actividades que exigen labores de limpieza. Como mencionamos en el Capítulo 4, podemos saber cuándo va a se invocado el recolector de basura, o si va a ser invocado. De esta manera, si queremos limpiar algo de una clase, debemos escribir un método especial para hacerlo explícitamente, y asegurar que al programador cliente sabe que ellos deben llamar a este método. Además de esto, como se describirá Capítulo 9 (manejando excepciones), debemos evitar una excepción poniendo tal limpieza (la orden o llamada para limpiar) en la cláusula **finally** .

Consideramos un ejemplo de un sistema de diseño asistido por computadora que pinta dibujos en la pantalla:

```
//: CADSystem.java
// Ensuring proper cleanup
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void cleanup() {
    System.out.println("Shape cleanup");
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing a Circle");
```

```
  }
  void cleanup() {
    System.out.println("Erasing a Circle");
    super.cleanup();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing a Triangle");
  }
  void cleanup() {
    System.out.println("Erasing a Triangle");
    super.cleanup();
  }
}

class Line extends Shape {
private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;

    System.out.println("Drawing a Line: " +
                       start + ", " + end);
  }
  void cleanup() {
    System.out.println("Erasing a Line: " +
                       start + ", " + end);
    super.cleanup();
  }
}

public class CADSystem extends Shape {
private Circle c;
private Triangle t;
private Line[] lines = new Line[10];
  CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
       lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  void cleanup() {
    System.out.println("CADSystem.cleanup()");
    t.cleanup();
```

```
    c.cleanup();
    for(int i = 0; i < lines.length; i++)
      lines[i].cleanup();
    super.cleanup();
  }
  public static void main(String[] args) {
  CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.cleanup();
    }
  }
} ///:~
```

Todo en este sistema es algún tipo de objeto **Shape** (que es a su vez hijo de **Objet** , porque es heredado implícitamente de la clase principal). Cada clase redefine un método **cleanup( )** de **Shape** ; además, para llamar a la versión de la clase base de ese método se usa **super** . Las clases específicas de **Shape** como **Circle** , **Triangle** y **Line** tienen todas constructores que "dibujan", aunque algún método llamado durante el tiempo de vida del objeto podía ser responsable de hacer algo que necesita limpieza. Cada clase tiene su propio método **cleanup( )** para restaurar cosas olvidadas de vuelta a la forma que ellas antes eran el objeto que existía.

En **main( )** , podemos ver dos palabras reservadas que son nuevas, y no serán oficialmente introducidas hasta el Capítulo 9: se trata de **try** y **finally** . La palabra reservada **try** indica que el bloque que sigue (delimitado por llaves) es una *región protegida* , que quiere decir que recibe un tratamiento especial. Uno de esos tratamientos especiales es que el código en la cláusula **finally** que sigue a esta región protegida es siempre ejecutado, no importa cómo se salga del bloque **try** . (Con el manejo de excepciones, es posible abandonar un bloque **try** en un número extraordinario de formas). Aquí, la cláusula **finally** está diciendo "llama siempre a **cleanup( )** de x, no importa que ocurra". Estas palabras reservadas serán explicadas a fondo en el Capítulo 9.

Observe que en el método **cleanup** debemos también poner atención al orden en que invoquemos a los métodos **cleanup** de la clase base y de los objetos miembros en el caso de que un subobjeto dependa de otro. En general, deberíamos seguir la misma forma que viene impuesta por un compilador de C++ a sus destructores: Primero realizar todo el trabajo específico para nuestra clase (la cual puede requerir que los elementos de la clase base también sean viables) entonces llamar al método **cleanup** de la clase base como se demostró aquí.

Puede haber muchos casos en los que el asunto de la limpieza no sea un problema; en estos casos dejamos que el recolector de basura sea el que haga el trabajo. Pero, cuando debamos hacerlo explícitamente, diligencia y atención son

requeridas.

Orden de recolección de basura

No hay mucho de lo que tú puedas fiarte cuando se acumula la colección de objetos olvidados. El recolector de objetos olvidados puede que nunca sea llamado. Si eso ocurre, puede recuperar objetos en el orden que quiera (el recolector). Además, las implementaciones del recolector de objetos olvidados en Java 1.0 a menudo no llaman a los métodos **finalize( )** . Es mejor no depender de la colección de objetos olvidados para nada excepto de recuperación de memoria. Si tú quieres realizar limpieza, haz tus propios métodos **cleanup** y no dependas de **finalize( )** . (Como mencionamos anteriormente, Java 1.1 puede ser forzado para llamar a todos los finalizadores).

Ocultando nombres

Los programadores de C++ pueden estar sorprendido por la ocultación de nombres, ya que se trabaja diferente en ese lenguaje. Si una clase base de Java tiene un nombre de método que es sobrecargado varias veces, redefinir ese nombre de método en la clase derivada no ocultará ninguna de las versiones de la clase base. Por eso, la sobrecarga funciona sin importar si el método fue definido a este nivel o en una clase base:

```
//: Hide.java
// Overloading a base-class method name
// in a derived class does not hide the
// base-class versions

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {}
}

class Hide {
```

```
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1); // doh(float) used
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
} ///:~
```

Como veremos en el próximo capítulo, es muy común sobreescribir métodos del mismo nombre usando exactamente el mismo prototipo y tipo de retorno de la clase base. Eso puede ser confuso (de ahí que C++ lo rechace, para prevenir cometer un error).

Eligiendo composición frente a herencia

Tanto la composición como la herencia permiten poner subobjetos dentro de tu nueva clase. Podríamos preguntarnos cuál es la diferencia entre los dos, y cuándo elegir uno en lugar del otro.

La composición es generalmente usada cuando deseamos las características de una clase existente dentro de una nueva clase, pero no su interfaz. Es decir, ponemos un para poder usarlo para implementar características de nuestra nueva clase, pero el usuario de esa nueva clase verá el interfaz que hemos definido en lugar del interfaz del objeto insertado. Por este efecto, insertamos objetos privados de clases existentes dentro de la nueva clase.

A veces tiene sentido permitir al usuario de la clase acceder directamente a la composición de nuestra nueva clase, es decir, hacer públicos los objetos miembros. Los objetos miembros usan la implementación ocultándose a sí mismos, por lo que esto es una cosa segura a hacer y, cuando el usuario sabe que estamos uniendo un conjunto de partes, hace que el interfaz sea más fácil de entender. Un objeto **car** es un buen ejemplo:

```
//: Car.java
// Composition with public objects

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
  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);
  }
} ///:~
```

Como la composición de un coche es parte del análisis del problema (y no simplemente parte de la base del diseño), hacer que los miembros sean públicos asiste a la compresión del cliente programador de cómo usar la clase y necesita menos complejidad en el código para el creador de la clase.

Cuando heredamos, estamos cogiendo una clase existente y creando una versión especial de esa clase. En general, esto significa que estamos tomando una clase de propósito general, especializándola para un caso o necesidad particular. Pensando un poco, podrá entender que no tendría sentido construir un coche usando un objeto **vehiculo** (un coche no contiene un vehículo, ¡es un vehículo!). La relación *es- un* viene expresada por la herencia, y la relación *tiene un* viene expresada por la composición.

protected

Ahora que hemos hecho una introducción al concepto de la herencia, la palabra reservada **protected** tendrá significado. En un mundo ideal, los miembros privados serían siempre irrevocablemente privados pero, en realidad, hay ocasiones en la que podemos desear crear algo oculto del mundo en general permitiendo acceso a los miembros de las clases derivadas. La palabra reservada

protected es un toque de pragmatismo. Decimos "esto es privado en lo que al usuario de la clase se refiere, pero está disponible para cualquiera que herede esta clase o cualquier otro en el mismo grupo de miembros". O sea, **protected** en Java es (o quiere decir) automáticamente "amigable".

La mejor dirección o sentido a tomar es dejar los datos miembros privados (deberíamos proteger siempre nuestro derecho a poder cambiar la implementación base), y permitir un acceso controlado a los herederos de la clase a través de métodos **protected** :

```
//: Orc.java
// The protected keyword
import java.util.*;

class Villain {
  private int i;
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}

public class Orc extends Villain {
  private int j;
  public Orc(int jj) { super(jj); j = jj; }
  public void change(int x) { set(x); }
} ///:~
```

Como puede comprobar, **change( )** tiene acceso a **set( )** y, por esto se ha definido **protected** .

Desarrollo incremental

Uno de las ventajas de la herencia es que soporta el *desarrollo incremental* , permitiéndonos introducir nuevo código sin causar daños al código existente. Esto también aísla los errores en el nuevo código. Heredando de una clase funcional existente y añadiendo datos miembros y métodos (y redefiniendo métodos existentes), dejamos intacto el código existente (que otra persona aún puede estar usando) sin daños ni errores. Si ocurre un daño, sabremos que está en el código nuevo, que es mucho más breve y fácil de leer que si hubiésemos modificado el cuerpo de un código existente.

Es extraordinaria la forma tan limpia en que son separadas las clases. Para reusar el código ni siquiera necesitamos el código fuente de los métodos. A lo sumo, importaremos un paquete (esto es verdad tanto para herencia como para

composición).

Es importante darse cuenta de que el desarrollo o evolución del programa es un proceso incremental (evolutivo), exactamente como el conocimiento humano. Uno podrá realizar todos los análisis que quiera, pero no sabrá todas las respuestas cuando se plantee el proyecto. Tendrá muchos más éxitos (¡ y más resultados inmediatos!) si comienza a "desarrollar" el proyecto como un criatura viva y evolutiva, en lugar de construirla de una vez como un rascacielos de cristal.

Aunque la herencia por experimentación puede ser una técnica útil, en algún momento, cuando las cosas se estabilicen tendrá que examinar detenidamente su jerarquía de clases con vistas a convertirla en una estructura sensible. Recuerde que, debajo de todo, la herencia es una forma de expresar una relación que dice "esta nueva clase es un tipo de esa antigua clase". Su programa no debería tratar de aprovechar bits, sino crear y manipular objetos de varios tipos para expresar un modelo en términos que procedan del espacio del problema.

Upcasting

El aspecto más importante de la herencia no es proporcionar métodos a la nueva clase. Es la relación expresada entre la nueva clase y la clase base. Esta relación puede ser resumida diciendo "la nueva clase es un tipo de una clase existente".

Esta descripción no es una forma extravagante de explicar lo que es la herencia (está soportada directamente en el lenguaje). Por ejemplo, consideremos una clase base llamada **Instrument** que representa a los instrumentos musicales y una clase derivada **Wind** para los instrumentos de viento. Como herencia significa que todos los métodos de la clase base están también disponibles en la clase derivada, todos los mensajes que se pueden enviar a la clase base también se pueden enviar a la clase derivada. Si la clase **Instrument** tiene un método **play ( )** , los **Wind** también los presentarán. Esto quiere decir que nosotros podemos exactamente decir que un objeto **Wind** es también un tipo de **Instrument** . El siguiente ejemplo muestra como el compilador apoya esta idea:

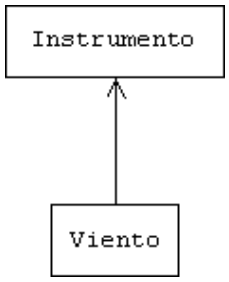
```
//: Wind.java
// Inheritance & upcasting
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}
```

```
// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

Lo que interesa en este ejemplo es el método **tune( )** , que acepta un manejador del objeto de **Instrument** . Sin embargo, en **Wind.main( )** el método **tune( )** es llamado pasándole un manejador de **Wind** . Dado que Java es de un tipo particular de control, parece extraño que un método que acepta un tipo acepte de buena gana a otro tipo, hasta que comprendemos que un objeto **Wind** es también un objeto **Instrument** , y no hay método que **tune( )** no pueda llamar para un **Instrument** que no esté también en **Wind** . Dentro de t **une( )** , el código funciona para **Instrument** y para cualquier objeto derivado de **Instrument** , y el hecho de convertir una variable o manejador **Wind** en una variable **Instrument** se denomina *upcast* (cast hacia arriba).

¿Por qué upcast?

La razón del término es histórica y está basada en los diagramas de herencia de clases que han sido tradicionalmente dibujados con la clase principal en la parte superior de la página, y creciendo hacía abajo (por supuesto, uno puede dibujar sus diagramas en la forma que desee). El diagrama de herencia para **Wind.java** es entonces:



La conversión de derivada a base es un movimiento *ascendente* en el diagrama de herencia, de ahí que hablemos de upcast, cast hacia arriba. Upcast es algo seguro porque se va desde un tipo más específico a un tipo más general. Es decir, la clase derivada es un superconjunto de la clase base. Ella puede contener más métodos que la clase base, pero debe contener por lo menos los métodos de la clase base. La única cosa que puede ocurrir al interfaz de la clase durante la conversión ascendente es que puede perder métodos, no ganarlos. Por esto el compilador

permite la conversión ascendente sin formas explícitas u otra notación especial.

Podemos también realizar la inversa de la conversión ascendente, downcast o conversión descendente, pero esto supone un dilema el cual es el tema del Capítulo 11.

Composición frente a herencia revisitada

En programación orientada a objetos, la forma más apropiada de crear y usar código es agrupar datos y métodos junto dentro de una clase, y usar objetos de esta clase. Ocasionalmente, podemos usar las clases existentes para construir nuevas clases mediante composición. Incluso menos frecuentemente podemos usar herencia. Por tanto, aunque la herencia gana una mayor importancia mientras más se conoce la OOP, eso significa que debamos usarla en todas partes. Por el contrario, deberíamos usarla escasamente, solo cuando esté claro que la herencia es útil. Una de las formas más claras de determinar si deberíamos usar composición o herencia es preguntar si siempre hemos de ascender desde nuestra nueva clase a la clase base. En caso afirmativo, la herencia es necesaria pero, en caso negativo, deberías buscar examinar nuestro problema más detenidamente y ver si necesitamos la herencia. El siguiente capítulo (polimorfismo) proporciona una de las razones más irresistibles para la conversión ascendente pero, si recordamos responder a la pregunta "¿Necesito conversión ascendente?, tendremos una buena herramienta para decidir entre composición y herencia.

La palabra reservada final

La palabra reservada **final** tiene significados diferentes dependiendo del contexto pero, en general, significa "esto no puede ser cambiado". Existen dos razones por las que queramos evitar cambios: diseño o eficiencia. Como estas dos razones son bastantes diferentes, es posible emplear mal la palabra reservada **final** .

En las siguientes secciones se habla de los tres lugares donde puede usarse **final** : para datos, métodos y para una clase.

Datos final

Muchos lenguajes de programación tienen alguna forma de decir al compilador que un trozo de datos es "constante". Una constante es útil por dos razones:

1. Puede ser una *constante en tiempo de compilación* que nunca cambiará.
2. Puede ser un valor inicializado en tiempo de ejecución que no queremos cambiar.

En el caso de una constante en tiempo de compilación, se permite al compilador que recoja el valor de la constante dentro los cálculos en los que la use; es decir, el cálculo puede ser realizado en tiempo de compilación, perdiendo algo de

sobrecarga en tiempo de ejecución. En Java, estas constantes deben ser de tipos primitivos y ser expresados usando la palabra reservada **final** . En el momento que se define una constante de este tipo hay que darle un valor.

Un campo que es **static** y **final** es una única porción de almacenamiento que no puede ser cambiada.

Cuando usamos **final** con manejadores de objetos en lugar de con tipos primitivos el significado parece un poco confuso. Con un dato primitiva, **final** convierte el valor en una constante pero, con un manejador de objeto, **final** convierte el manejador en una constante. El manejador debe ser inicializado a un objeto al principio de la declaración, y no puede nunca ser cambiado para apuntar a otro objeto. Sin embargo, el objeto si que puede modificarse; Java no proporciona una forma de convertir arbitrariamente un objeto en constante (podemos, sin embargo, escribir una clase de manera que los objetos tengan el efecto de ser constantes). Esta restricción incluye a arrays, que son también objetos.

Aquí hay un ejemplo que muestra los campos final:

```
//: FinalData.java
// The effect of final on fields

class Value {
  int i = 1;
}

public class FinalData {
// Can be compile-time constants
final int i1 = 9;
static final int I2 = 99;
// Typical public constant:
public static final int I3 = 39;
// Cannot be compile-time constants:
final int i4 = (int)(Math.random()*20);
static final int i5 = (int)(Math.random()*20);

Value v1 = new Value();
final Value v2 = new Value();
static final Value v3 = new Value();
//! final Value v4; // Pre-Java 1.1 Error:
                                    // no initializer
// Arrays:
final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
```

```
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
  FinalData fd1 = new FinalData();
  //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Error: Can't
    //! fd1.v3 = new Value(); // change handle
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
} ///:~
```

Como **i1** e **i2** son datos primitivos **final** con valores en tiempo de compilación, ambos pueden ser usados como constantes en tiempo de compilación y no son diferentes de forma importante. **i3** es la forma más típica de definir las constantes: **public** para que puedan ser utilizadas fuera del agrupamiento de miembros, **estáticas** para acentuar que sólo hay una, y **final** para decir que son constantes. Observe que, por convenio, nombramos con letras mayúsculas a las primitivas **final static** con valores iniciales constantes (es decir, a las constantes en tiempo de compilación). También, es interesante observar que i5 no puede ser conocida en tiempo de compilación, así que no está en mayúsculas.

El hecho de que un dato sea **final** no significa que su valor se conozca en tiempo de compilación. Esto se demostrado inicializando **i4** e **i5** con números generados aleatoriamente en tiempo de ejecución. Este trozo de ejemplo también muestra la diferencia entre crear un valor estático final (final static) o no estático. Esta diferencia se nota sólo cuando los valores son iniciados en tiempo de ejecución. Así, los valores en tiempo de compilación son tratados iguales por el compilador. (Y supuestamente optimizados fuera de existencia). La diferencia se muestra en la salida de una ejecución:

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```

Los valores de **i4** para **fd1** y **fd2** son únicos, pero el valor para i5 no se cambia al crear el segundo objeto **FinalData** . Esto es porque es estático y está inicializado una vez en la carga de la clase y no cada vez que se crea un nuevo objeto.

Las variables **v1** a **v4** demuestran el significado de un manejador de objeto **final** . Como podemos ver en **main( )** , solo porque **v2** sea final no significa que no podamos cambiar su valor. Sin embargo, no podemos reasignar **v2** a un nuevo objeto, precisamente porque es **final** . Esto es lo que **final** quiere decir para un manejador de objeto. Podemos también comprobar que este mismo significado se mantiene en el caso de los arrays, que son otro tipo de manejadores (hay forma conocida que yo sepa de crear un manejador de array final). Crear manejadores final parece menos útil que creando primitivas final.

finales vacíos

Java 1.1 permite la creación de *datos finales en blanco* , que son campos que se han declarado como **final** pero a los que no se ha dado un valor de inicialización. En todos los casos, el **final** vacío debe ser inicializado antes de que sea usado, y el compilador asegura esto. Sin embargo, los finales en blanco proporcionan mucho más flexibilidad en el uso de la palabra reservada **final** . Así, por ejemplo, un campo **final** dentro de una clase puede ahora ser diferente para cada objeto y aún retiene su calidad inmutable. He aquí un ejemplo:

```
//: BlankFinal.java
// "Blank" final data members

class Poppet { }

class BlankFinal {
final int i = 0; // Initialized final
final int j; // Blank final
final Poppet p; // Blank final handle
// Blank finals MUST be initialized
// in the constructor:
  BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet();
  }
  BlankFinal(int x) {

    j = x; // Initialize blank final
    p = new Poppet();
  }
  public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

Estamos obligados a realizar asignaciones para los campos finales con una expresión al principio de la definición del campo o en cada constructor. De esta forma se garantizada que el campo final siempre será iniciado antes de ser usado.

Argumentos final

Java 1.1 permite crear argumentos **final** declarándolos así en la lista de argumentos. Esto significa que dentro del método no podremos cambiar el manejador de argumento al que se apunta en la llamada al método:

```
//: FinalArguments.java
// Using "final" with method arguments

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
  //! g = new Gizmo(); // Illegal -- g is final
    g.spin();
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  // void f(final int i) { i++; } // Can't change
  // You can only read from a final primitive:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

Es interesante observar que podemos asignar un manejador vacío a un argumento que es **final** sin que el compilador lo coja, exactamente como se puede hacer con un argumento no final.

Los métodos **f( )** y **g( )** muestran qué ocurre cuando los argumentos primitivos son **final** : sólo se puede leer el argumento, pero no puede cambiarse.

Métodos final

Hay dos razones para los métodos **final** . La primera es poner un "cerrojo" en el método para evitar que alguna clase heredera cambie su significado. Esto se hace hecho por razones de diseño, cuando queremos asegurar que el comportamiento de un método no puede ser anulado mediante la herencia.

La segunda razón para los métodos **final** es la eficiencia. Si creamos un método **final** , estamos permitiendo al compilador convertir las llamadas al método en llamadas en línea. Cuando el compilador ve una llamada a un método final puede, en su juicio, saltar el normal camino de inserción de código para realizar mecanismo de llamada del método (mover argumentos en la pila, saltar al principio del código del método y ejecutarlo, saltar al final y limpiar los argumentos de la pila, y ocuparse del valor a devolver) y, en su lugar, sustituir la llamada al método con una copia del código actual del cuerpo del método. Esto elimina la sobrecarga de llamada de métodos. Claro que, si el método es grande, entonces el código comenzaría a agrandarse y la realización en línea no supondría ventajas en el rendimiento. Se supone que el compilador de Java está preparado para detectar estas situaciones y elegir adecuadamente si ejecutar en línea un método final. Sin embargo, es mejor no confiar en que el compilador está preparado para hacer esto y crear un solo métodos finales si son suficientemente pequeños o si deseamos evitar la sobreescritura.

Los métodos privados de una clase son implícitamente **final** . Como nadie puedes acceder a ellos, nadie puede anularlos (el compilador da un mensaje de error si lo intentamos). Podemos añadir el especificador final para un método privado pero no da a ese método un significado extra.

Clases final

Cuando decimos que una clase completa es **final** (precediendo a su definición con la palabra reservada final), afirmamos que no queremos heredar de esta clase ni permitir que cualquier otro lo haga. En otras palabras, por alguna razón, el diseño de la clase es tal que no hay nunca necesidad de hacer cambios, o por razones de seguridad no queremos crear subclases. Alternativamente, podemos estar preocupados por la eficiencia y queremos estar seguros de que cualquier actividad asociada con objetos de esta clase será lo más eficiente posible.

```
//: Jurassic.java
// Making an entire class final

class SmallBrain {}

final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
```

```
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Los datos miembros pueden ser o no **final** . Se aplican las mismas reglas a los datos miembros sin reparar en si la clase es definida como final. Definiendo la clase como final simplemente evitamos la herencia. Sin embargo, como evitamos la herencia, todos los métodos de la clase **final** son implícitamente **final** , de manera no hay forma de sobreescribirlos. Así, el compilador tiene las mismas opciones de optimización que tendría si declaráramos explícitamente al método como final. Podríamos añadir el especificador **final** a un método en una clase **final** , pero se lo añadiría ningún significado.

Aviso final

Puede parecer ser práctico crear un método **final** mientras estamos diseñando una clase. Podríamos sentir que la eficiencia es muy importante cuando usas tu clase y no deseamos que se sobreescriban nuestros tus métodos en ninguna circunstancia. A veces esto es cierto.

Pero hay que ser cuidadoso con estas suposiciones. En general, es difícil prever si una clase puede ser reusada, especialmente una clase de propósito general. Si definimos un método como **final** estamos evitando la posibilidad de que nuestra clase se reuse mediante herencia en algún otro proyecto de programación, simplemente porque no habíamos imaginarlo que se podría usar de esa forma.

La librería estándar de Java es un buen ejemplo de esto. En particular, la clase **Vector** es muy usada y podrían ser aún más útil si, en aras de la eficiencia, todos los métodos no hubiesen sido creados como **final** . Es fácil concebir que podríamos desear heredar y sobreescribir esta clase pero, los diseñadores de todas formas decidieron que no era lo apropiado. Esto es irónico por dos razones. Primero, **Stack** es heredada de **Vector** , lo que significa que una pila (**Stack**) es un **Vector** , lo cual no es realmente cierto. Segundo, muchos de los más importantes métodos de **Vector** , tal como **addElement( )** y **elementAt( )** son sincronizados, lo cual, como veremos en el Capítulo 14, provoca una sobrecarga de rendimiento que probablemente elimina algunas ventajas proporcionadas por **final** . Esto lleva a creer en la teoría a que los programadores están

constantemente fallando en adivinar donde deberían ocurrir las optimizaciones. Es demasiado malo que un diseño malo esté creado dentro de la librería estándar donde nosotros debemos poder con todo eso.

Es también interesante observar que **Hashtable** , otra importante clase de la librería estándar, no tiene métodos **final** , Como se menciona en otro lugar en este libro, es bastante obvio que algunas clases son diseñadas por gente completamente diferente que otras. (Observa la brevedad de los nombres de los métodos en **Hashtable** comparados con los de **Vector**). Esto es precisamente el tipo de cosa que no debería ser obvia para los clientes de una librería de clases. Cuando cosas son inconsistentes hace que haya más trabajo para el usuario. Aún otro himno al valor del diseño y código campo a través.

Iniciación y carga de clases

En muchos lenguajes más tradicionales, los programas son cargados todos de una vez como parte del proceso de comienzo. Esto va seguido de la iniciación y entonces el programa comienza. El proceso de iniciación en estos lenguajes debe ser cuidadosamente controlado de manera que el orden de iniciación de estáticos no cause problemas. C++, por ejemplo, tiene problemas si un estático cuenta con otro estático para ser valido antes que el segundo haya sido inicializado.

Java no tiene este problema porque toma un enfoque diferente para cargar. Como todo en Java es un objeto, muchas actividades se vuelven más fáciles, y ésta es una de ellas. Como mostraremos en el próximo capítulo, el código para cada objeto existe en un fichero distinto. Este fichero se carga hasta que se necesita el código. En general, podemos afirmar que, hasta que no se construye un objeto de la clase es construido, el código de la clase no será cargado. Como puede haber algunas sutilezas los métodos estáticos, también podemos decir, "El código de la clase es cargado al principio del primer uso".

El punto del primer uso es también donde la se lleva a cabo iniciación estática. Todos los objetos y bloque de código estáticos ser inicializado en orden textual (es decir, el orden en que se escriben en la definición de la clase) al principio de la carga. Los estáticos, por supuesto, son iniciados solo una vez.

Iniciación con herencia

Es útil para observa el proceso completo de inicialización, incluyendo herencia, realizar un esquema completo de lo que ocurre. Consideremos el siguiente código:

```
//: Beetle.java
// The full process of initialization.

class Insect {
int i = 9;
```

```
int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =
    prt("static Beetle.x2 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 63;
  }
  public static void main(String[] args) {
    prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

La salida de este programa es:

```
static Insect.x initialized
static Beetle.x initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 63
j = 39
```

La primera cosa que ocurre cuando ejecutamos la clase **Beetle** de Java es que el cargador sale y encuentra esa clase. En el proceso de cargar, el cargador observa que tiene una clase base (que es la que dice la palabra reservada **extends**), que entonces carga. Esto es lo que ocurre estemos o no creando un objeto de esa clase base. (Intente comentar la creación del objeto para comprobarlo por sí

mismo).

Si la clase base tiene una clase base, la segunda clase base entonces sería cargada, etc. Después se realiza la inicialización estática en la clase principal (en este caso, **Insect**), y después en la siguiente clase derivada, etc. Esto es importante porque la inicialización estática de la clase derivada puede depender de que algún miembro de la clase base sea inicializado apropiadamente.

En este punto, las clases necesarias han sido todas cargadas así que el objeto puede ser creado. Primero, todas las primitivas de este objeto son asignadas a sus valores por defecto y los manejadores de objetos son asignados a **null** . Entonces se invocará al constructor de la clase base. En este caso la llamada es automática, pero también se puede especificar la llamada al constructor (como la primera operación en el constructor **Beetle( )**) usando **super** . La construcción de la clase base pasa por el mismo proceso en el mismo orden como en el constructor de la clase derivada. Después de completar el constructor de la clase base, las instancias de las variables son iniciadas en orden textual. Finalmente, el resto del cuerpo del constructor es ejecutado.

Resumen

Tanto la herencia como la composición nos permiten crear un nuevo tipo de tipos existentes. Típicamente, sin embargo, usaremos la composición para reusar tipos existentes como parte de la base de la implementación del nuevo tipo y la herencia cuando queramos reusar el interfaz. Entonces la clase derivada tiene el interface de la clase base, esto puede ser convertirse ascendentemente hacia a la base, lo cual es crítico para el polimorfismo, como tú estudiaremos en el próximo capítulo.

A pesar del fuerte énfasis que programación orientada a objetos pone en la herencia, cuando empiece un diseño debe preferir, por lo general, la composición y usar la herencia sólo cuando sea claramente necesaria. Como veremos en el siguiente capítulo, la composición tiende a ser más flexible. Además, usando el artificio añadido de la herencia con miembro de un tipo, podemos cambiar el tipo exacto, y entonces el comportamiento, de esos objetos miembros en tiempo de ejecución. Por lo tanto, podemos cambiar el comportamiento del objeto compuesto en tiempo de ejecución.

Aunque reusar el código a través de la composición y la herencia es práctico para un rápido desarrollo del proyecto, generalmente querremos rediseñar nuestra jerarquía de clases permitiendo a otros programadores comenzar a depender de esto. Tu meta es una jerarquía en la que cada clase tiene un uso específico y sea ni demasiado grande (abarcando mucha funcionalidad que es difícil de manejar para reusar) ni molestamente pequeño (tú no puedes usarla por eso o sin añadir funcionalidad). Tus clases finalizadas deberían ser fácilmente reusadas.

Ejercicios

1. Crea dos clases, A y B, con constructores por defecto (con lista de argumentos vacía) que se llaman por si mismos. Heredar una nueva clase llamada C de A, y crear un miembro B dentro de C. No cree un constructor para C. Crea un objeto de la clase C y observa los resultados.
2. Modifica el ejercicio 1 para que A y B tengan los constructores con argumentos en lugar de constructores por defecto. Escribe un constructor para C y realiza todas la inicializaciones dentro del constructor de C.
3. Crea una clase sencilla. Dentro de ella, una segunda clase. Define un campo para un objeto de la primera clase. Usa inicialización difusa para instanciar este objeto.
4. Crea una clase "hija" de la case **Detergent** . Sobreescribe el método **scrub()** y añade un nuevo método llamado **sterilize()** .
5. Toma el fichero **Cartoon.java** y comenta fuera el constructor de la clase **Cartoon** . Explica que ocurre.
6. Coge el fichero **Chess.java** y comenta fuera el constructor de la clase **Chess** . Explica que ocurre.
7. Demuestra que el constructor por defecto ha sido creado para tí por el compilador.
8. Demuestra que los constructores de la clase base son (a) siempre llamados, y (b) llamados después de los constructores de las clases derivadas.
9. Crea una clase base con un sólo constructor que no sea el constructor por defecto, y una clase derivada con un constructor por defecto y otro que no lo sea. En los constructores de la clase derivada, llama al constructor de la clase base.
10. Crea una clase que se llame **Root** que contenga una llamada a cada clase (que también debes crear) llamadas **Component1** , **Component2** , y **Component3** . Deriva una clase **Stem** de **Root** que también contenga una llamada a cada "componente". Todas las clases deben tener constructores por defecto que impriman un mensage explicativo de la clase que la que se hallan.
11. Modifica el ejercicio 10 para que cada clase sólo tenga constructores que no sean por defecto.
12. Añade una herencia apropiada del método **cleanup()** a todas las clases del ejercicio 11.
13. Crea una clase con un método que sea sobreescrito tres veces. Crea una clase heredada de ella, añade una nueva sobreescritura al método y muestra que los cuatro métodos son accesibles en la clase derivada.
14. En **Car.java** añade un método **service()** a **Engine** y llama a este método desde el **main ()** .
15. Crea una clase dentro de un paquete. Tu clase debe contener un método **protected** . Fuera del paquete, intenta llamar al método **protected** y explica los resultados. Ahora, crea una clase derivada y llama al método **protected** desde dentro de un método de tu clase derivada.
16. Crea una clase llamada **Anfibio** . Desde esta, hereda a una clase llamada **Rana** . Pon los métodos apropiados en la case base. En **main()** crea una **Rana** y haz un "upcasting"

hasta **Anfibio** . Demuestra que todos los métodos todavía funcionan.

17. Modifica el ejercicio 16 para que **Rana** sobreescriba los métodos definidos en la clase base, es decir, que añada nuevas definiciones usando la misma signatura para cada método. Comprueba qué ocurre en el **main()** .
18. Crea una clase con un campo **static final** y un campo **final** y demuestra la la diferencia entre las dos.
19. Crea una clase con una referencia **final** vacía para un objeto. Optimiza la inicialización de **final** dentro de un método(no del constructor) ante de usarlo. Demuestra con garantías que **final** debe ser inicializado antes de su uso y que no puede ser cambiado una vez inicializado.
20. Crea una clase con un método **final** . Hereda de esa clase e intenta sobreescribir ese método.
21. Crea una clase **final** e intenta heredar de ella.
22. Prueba que una clase se carga sólo una vez. Prueba que dicha carga puede ser debida por la creación de una llamada a esa clase o del acceso a una miembro **static** .
23. En **Beetle.java** hereda un tipo específico de "beetle" desde la clase **Beete** , siguiendo el mismo formato que la clase existente. Tracea y explica la salida.

7: Polimorfismo

El polimorfismo es la tercera característica esencial de los lenguajes de programación orientados a objetos, después de la abstracción de datos y la herencia.

Esto aporta otra dimensión para separar interfaz de implementación, para separar el qué del cómo. El polimorfismo nos permite mejorar la organización y comprensión del código así como la creación de programas extensibles que pueden "crecer" no sólo durante la creación del proyecto, sino también cuando se diseñen nuevas características.

La encapsulación crea un nuevo tipo de datos combinando características y comportamientos. La implementación oculta la interfaz de la implementación haciendo los detalles privados. Esta clase de organización mecánica tiene sentido para alguien que proceda de un lenguaje procedimental. Pero el polimorfismo trata del desacoplamiento de en términos de *tipos* . En él ultimo capitulo se vio cómo la herencia permite el tratamiento de un objeto como su propio tipo o su tipo base. Esta habilidad es crítica ya que permite que muchos tipos (derivados del mismo tipo base) sean creados como si fueran uno solo, y una sola pieza de código funcionará con todos estos tipos diferentes de la misma forma. La llamada a un método polimórfico permite a un tipo expresar su diferencia con otro similar, ambos derivados de la misma clase. Esta distinción es expresada a través de diferencias en el comportamiento de los métodos a los que podemos invocar a través de la clase.

En este capitulo aprenderemos a cerca del polimorfismo (también llamado *enlazado dinámico, enlazado diferido o enlazado en tiempo de ejecución*) comenzando por lo básico, con simples ejemplos en los que lo fundamental es el comportamiento polimórfico del programa.

Conversión ascendente

En el capitulo 6 mostramos como puede usarse un objeto como su propio tipo o como un objeto de su tipo base. Tomar un manejador de objeto y tratarlo como el manejador de su tipo base se denomina upcasting (conversión ascendente), dado que el árbol de herencia se dibujado con la clase base en la cima.

En el siguiente ejemplo mostramos un problema importante que surge: (ver la página XX si tiene problemas ejecutando este programa)

```
//: Music.java
// Inheritance & upcasting
package c07;
```

```
class Note {
private int value;
private Note(int val) { value = val; }
public static final Note
  middleC = new Note(0),
  cSharp = new Note(1),
  cFlat = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
// Redefine interface method:
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

public class Music {
  public static void tune(Instrument i) {
    // ...
    i.play(Note.middleC);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
  }
} ///:~
```

El método **Music.tune()** acepta un manejador **instrumental** , pero también cualquier tipo derivado de un instrumento. En la función **main()** podemos ver como se pasa el manejador **Wind** al método **tune()** sin necesidad de cast. Esto es aceptable; en **Wind** debe existir el interface de **Instrument** , ya que **Wind** lo hereda de él. La conversión hacia arriba desde **Wind** hacia **Instrument** puede "estrechar" el interfaz, pero no puede hacer nada menos el interface completo de **Instrument** .

¿Por qué upcast?

Este programa puede parecer extraño. ¿Por qué debe alguien olvidar intencionadamente el tipo de un objeto? Esto es lo que sucede cuando se realiza el

upcast, y parece que seria mucho más simple si **tune()** simplemente el manejador **Wind** como su argumento. Esto trae un punto esencial: suponga que necesita escribir un nuevo **tune()** para cada tipo de **Instrument** en su sistema. Suponga que seguimos ese razonamiento y añadimos instrumentos de cuerdas y metálicos:

```
//: Music2.java
// Overloading instead of upcasting

class Note2 {
private int value;
private Note2(int val) { value = val; }
public static final Note2
  middleC = new Note2(0),
  cSharp = new Note2(1),
  cFlat = new Note2(2);
} // Etc.

class Instrument2 {
  public void play(Note2 n) {
    System.out.println("Instrument2.play()");
  }
}

class Wind2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Wind2.play()");
  }
}

class Stringed2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Stringed2.play()");
  }
}

class Brass2 extends Instrument2 {
  public void play(Note2 n) {
    System.out.println("Brass2.play()");
  }
}

public class Music2 {
  public static void tune(Wind2 i) {
    i.play(Note2.middleC);
  }
  public static void tune(Stringed2 i) {
    i.play(Note2.middleC);
  }
```

```
  public static void tune(Brass2 i) {
    i.play(Note2.middleC);
  }
  public static void main(String[] args) {
    Wind2 flute = new Wind2();
    Stringed2 violin = new Stringed2();
    Brass2 frenchHorn = new Brass2();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
  }
} ///:~
```

Esto funciona, aunque hay mejores formas: Debemos escribir el tipo especifico del método para cada clase del nuevo **Intrument2** que se añada.

Esto significa que programar más en el primer lugar, pero también significa que si se necesita añadir un nuevo método como **tune()** o un nuevo tipo de instrumento, tienes mucho trabajo que hacer. Añadir el echo que el compilador no dará mensajes de error si olvida sobrecargar uno de tus métodos y todo el proceso de trabajo con los tipos no se puedan manejar.

¿No será muy agradable si solo escribe un método simple que toma como clase base como su argumento, y ninguno de las derivadas clases especificas? ¿Esto es, no será agradable si se olvida que hay clases derivadas, y escribe el código para hablar solo con las clases base?

Esto es exactamente lo que el polimorfismo te permite hacer. De todos modos, la mayoría de los programadores (que proceden de programación procedural) tienen un pequeño problema con la forma que trabaja el polimorfismo.

El enriedo

La dificultad con Music.java pueden verse ejecutando el programa. La salida es Wind.play() Este es una salida "limpia", pero no parece que funcione de esa forma. Mire el método tune():

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.middleC);
}
```

Este recibe un manejador Instument. ¿De esta forma como puede saber el compilador que este manejador Instrument apunta a viento en este caso y no a metálicos o de cuerda? El compilador no puede saberlo. Para tomar un profundo

conocimiento de la fuente, es útil examinar el tema de *binding* .

Enlazado en las llamadas a métodos

Conectar una llamada de un método con el cuerpo de un método se denomina enlazado. Cuando se *binding* se permite antes de ejecutar el programa (por el compilador y linkador, si hay alguno), es llamado primero *binding* . Puede que no haya oído el termino antes por que nunca ha sido una opción con lenguajes procedurales. Los compiladores de "C" solo tienen un tipo de llamada a métodos, y esta es antes de *binding* .

La parte confusa de arriba del programa resuelve a cerca del *early binding* por que el compilador no puede saber el método correcto para llamar cuando este solo tiene un manejador Instument.

La solución se llama *late binding* , que significa que el *binding* ocurre en tiempo de ejecución basado en el tipo de objeto. *Late binding* también es llamado *binding* dinámico o tiempo de ejecución *binding* . Cuando un lenguaje implementa *late binding* , debe haber algún mecanismo para determinar el tipo de un objeto en tiempo de ejecución y llamar al apropiado método. Esto es, el compilador todavía no saber el tipo del objeto, pero el mecanismo de llamada al método busca y llama el cuerpo del método correcto. El mecanismo *late-binding* varia de lenguaje a lenguaje, pero puede imaginar que algunos tipos de información deben ser instalados en los objetos.

Todos los métodos *binding* en Java usa *late binding* menos un método que ha sido declarado final. Esto significa que normalmente no necesita tomar ninguna decisión a cerca del cuando ocurrirá el *late binding* . (Sucederá automáticamente)

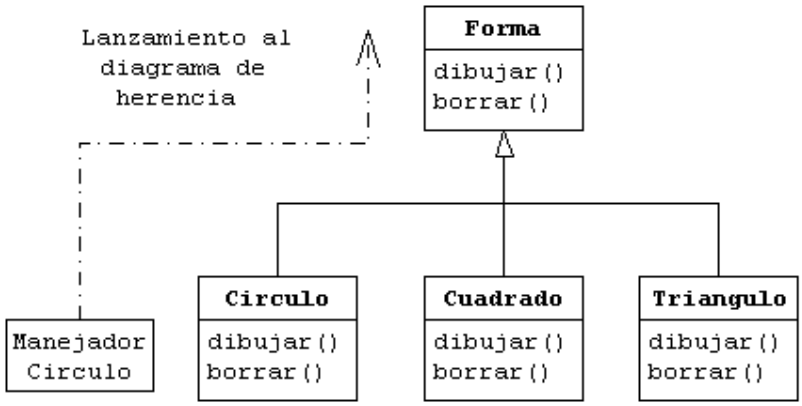
¿Por que se declara un método final? Como habrá notado en el ultimo capitulo, este previene de el sobreescribir métodos. Quizás mas importante, es efectivamente "turns off" *dynamic binding* , o mejor este dice al compilador que *dynamic binding* no es necesario. Esto permite al compilador generar un código mas eficiente para llamadas a métodos finales.

Produciendo el comportamiento correcto

Una vez que se sabe que todos los *binding* en Java suceden polimorficamente vía *late binding* , puedes escribir tu código para hablar a clases base y saber que todos las casos de clases derivadas funcionaran correctamente usando el mismo código. O para ponerlo de otra forma, tu mandas un mensaje a un objeto y dejas que el objeto escoge la correcta cosa a hacer.

El clásico ejemplo en OOP es el ejemplo de las formas. Esto es normalmente usado por que es fácil visualizarlo, pero desafortunadamente puede confundir a los programadores noveles pensando que OOP es programación solamente para gráficos, que por supuesto no es el caso.

El ejemplo de las formas tiene una clase base llamada Shape y varios tipos derivados: Circle, Square, Triangle, etc. La razón del ejemplo funciona por lo fácil que es decir "Circle es un tipo de Shape" y ser entendido. El diagrama de herencia nos enseña las relaciones:



El *upcast* puede ocurrir en una sentencia tan simple como:

```
Shape s = new Circle();
```

Aquí, un objeto Circle es creado y el manejador resultante es inmediatamente asignado a Shape que puede parecer ser un error (asignando un tipo a otro) y esta bien por que un es un "Shape" por la herencia. Por eso el compilador añade con las sentencias no da ningún mensaje de error.

Cuando se llama a un método de clase base (que ha sido sobrescrito en las clases derivadas):

```
s.draw();
```

De nuevo, se puede esperar que draw() de Shape es llamado por que esto es un manejador Shape, por eso, ¿como puede el compilador saber hacer algo mas? Y todavía el propio Circle.draw() es llamado por el *late binding* (polimorfismo). El ejemplo siguiente lo hace de una forma diferente:

```
//: Shapes.java
// Polymorphism in Java
```

```
class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()");
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}

class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }
}

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default: // To quiet the compiler
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
  public static void main(String[] args) {
  Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
```

```
} ///:~
```

La clase base Shape establece el interface común a todos los afectados por la herencia de Shape, esto es, todos los "shapes" pueden ser dibujados y borrados. Las clases derivados sobrescritas estas definiciones para provee un único comportamiento para cada especifico tipo de Shape.

La clase principal Shape continúe un método estático randShape() que produce un manejador a un objeto shape elegido aleatoriamente cada vez que se le llama. Notar que el *upcasting* sucede en las sentencias de retorno, que se toma un manejador a Circle, Squeare, o Triangle y lo envía fuera del método como un tipo devuelto, Shape. Por eso siempre que se llame a este método nunca se obtendrá un cambio para ver que tipo especifico es, desde que se retorna un manejador plain Shape. Main() contiene un array de manejadores Shape llenos de llamadas a randShape(). En este punto sabemos que tenemos Shapes, pero no sabemos nada mas especifico. De cualquier forma, cuando se avanza hacia este array y se llama Draw() para cada uno, el correcto tipo especifico comportamiento mágicamente ocurre, como se puede ver de la salida del siguiente ejemplo:

```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

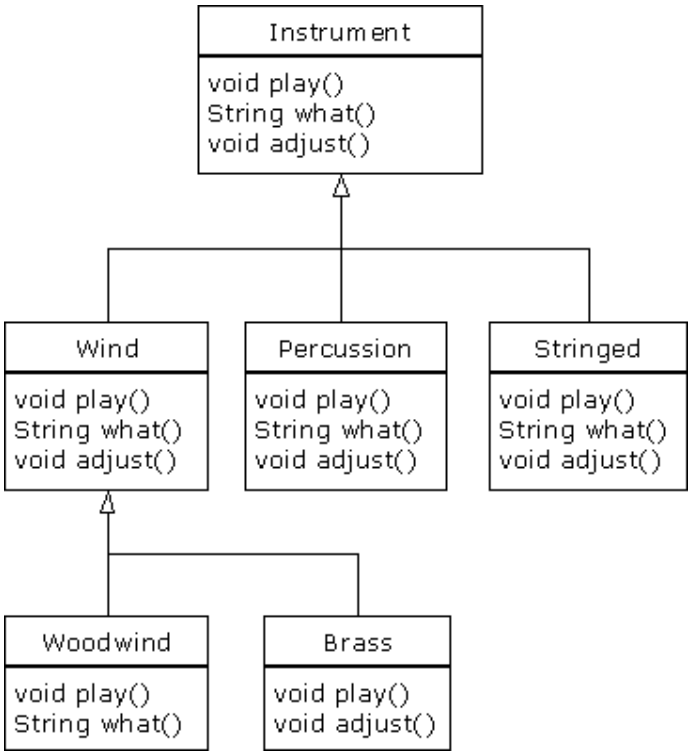
Por supuesto, desde que todas las figuras son elegidas aleatoriamente cada vez, la ejecución tendrá diferentes resultados. El punto de elegir las figuras aleatoriamente es la unidad "home" el entendimiento que el compilador puede no tener conocimientos especiales que le permitan hacer las llamadas correctas en tiempo de compilación. Todas las llamadas a draw() se hacen a través de *dynamic binding* .

Extensibilidad

Ahora volvamos al ejemplo de los instrumentos musicales. Por el polimorfismo usted puede añadir tantos tipos nuevos como desee al sistema sin cambiar el método tune(). En un programa POO bien diseñado, la mayoría o todos nuestros métodos seguirán el modelo de tune() y se comunican solo con el interface de clases base. Semejante a un programa extensible, ya que se puede añadir nuevas funcionalidades por la heredación de nuevos tipos de datos procedentes de las

clases base comunes. Los métodos que manipulan el interface de las clases base no necesitaran ser cambiados en absoluto para acomodarlo a las nuevas clases.

Considerar que sucede si tomamos el ejemplo de los instrumentos y añadir mas métodos en las clases base y un numero de nuevas clases. Aquí esta el diagrama:



Todas estas nuevas clases funcionan correctamente con el método antiguo, sin cambiar tune(). Incluso si "tune()" esta en un fichero separado y nuevos métodos son añadidos al interface de Instrumentos tune() funciona correctamente sin recompilar. Aquí esta la implementación del diagrama siguiente:

```
//: Music3.java
// An extensible program
import java.util.*;

class Instrument3 {
  public void play() {
    System.out.println("Instrument3.play()");
  }
  public String what() {
    return "Instrument3";
  }
  public void adjust() {}
```

```
}

class Wind3 extends Instrument3 {
  public void play() {
    System.out.println("Wind3.play()");
  }
  public String what() { return "Wind3"; }
  public void adjust() {}
}

class Percussion3 extends Instrument3 {
  public void play() {
    System.out.println("Percussion3.play()");
  }
  public String what() { return "Percussion3"; }
  public void adjust() {}
}

class Stringed3 extends Instrument3 {
  public void play() {
    System.out.println("Stringed3.play()");
  }
  public String what() { return "Stringed3"; }
  public void adjust() {}
}

class Brass3 extends Wind3 {
  public void play() {
    System.out.println("Brass3.play()");
  }
  public void adjust() {
    System.out.println("Brass3.adjust()");
  }
}

class Woodwind3 extends Wind3 {
  public void play() {
    System.out.println("Woodwind3.play()");
  }
  public String what() { return "Woodwind3"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument3 i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument3[] e) {
```

```
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
  Instrument3[] orchestra = new Instrument3[5];
  int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind3();
    orchestra[i++] = new Percussion3();
    orchestra[i++] = new Stringed3();
    orchestra[i++] = new Brass3();
    orchestra[i++] = new Woodwind3();
    tuneAll(orchestra);
  }
} ///:~
```

Los nuevos métodos son what() que devuelve un manejador String con una descripción de la clase y adjust() que da unas formas de ajustar cada instrumento.

En Main(), cuando se pone algo en el array Intrument3 automáticamente *upcast* el inrumento3.

Puedes ver que el método tune() es dichosamente ignorante de todos los cambios de código que ocurren por el, y todavía funciona correctamente. Esto es exactamente lo que el polimorfismo es supuestamente provisto. Los cambios del código no causaran daños a partes del programa que debería no ser afectado. De otra forma, el polimorfismo es una de las mas importantes técnicas que permiten al programador "separar las cosas que cambian de las cosas que continúan igual".

Sobreescritura vs. Sobrecarga

Echemosle un vistazo diferente al primer ejemplo de este capítulo. En el siguiente programa el interfaz del método **play()** se cambia en el proceso de sobreescritura del mismo, lo que significa que no hemos *sobreescrito* el método, sino que lo hemos *sobrecargado* . Como el compilador permite la sobrecarga de métodos no da ningún error, pero el comportamiento probablemente no es lo que pretendiamos.

Aquí está el ejemplo:

```
//: WindError.java
// Accidentally changing the interface

class NoteX {
  public static final int
  MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
```

```
}

class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
// OOPS! Changes the method interface:
  public void play(NoteX n) {
    System.out.println("WindX.play(NoteX n)");
  }
}

public class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.MIDDLE_C);
  }
  public static void main(String[] args) {
    WindX flute = new WindX();
    tune(flute); // Not the desired behavior!
  }
} ///:~
```

Aquí esta otro aspecto confuso que nos muestra el programa. En **InstrumentX** el método **play()** recibe un **int** con el identificativo **NoteX** . Aunque **NoteX** es un nombre de clase, también se puede usar como identificativo sin ningún problema. Pero en **WindX** , **play( )** recibe una referencia a **NoteX** que tiene el identificador **n** . (No obstante también se podría haber puesto **play(NoteX NoteX)** sin que produzca ningún error.) Parece como si el programador intentara sobreeescribir **play( )** , pero equivocandose un poco a la hora de escribirlo. El compilador, no obstante, asume que lo que se pretendía era sobrecargar y no sobreescribir el método. Notar que si seguimos la convención de nombres estándar de Java, el identificador del argumento hubiera sido **noteX** ('n' minuscula), que lo distinguiría del nombre de la clase.

En tune, el InstrumentX "i" es enviado el mensaje play(), con uno de los miembros de NoteX (MIDDLE_C) como un argumento. Dado que NoteX contiene definiciones enteras, esto significa que la versión entera del método *ahora sobrecargado* play() es llamado, y como no ha sido *sobreescrito* la versión clase-base es usada.

La salida es:

```
InstrumentX.play()
```

Ciertamente esto no parece ser una llamada a un método polimórfico. Una vez se entiende que esta sucediendo, usted puede arreglar el problema fácilmente, pero imagine lo dificíl que puede ser encontrar el error si este está enterrado en un programa de tamaño significante.

Clases y métodos Abstractos

En todos los ejemplos de instrumentos, los métodos en la base de clase instrumento hay siempre métodos de pruebas. Si esos métodos suenan llamadas alguna vez, estarás haciendo algo erróneo. Esto es porque el intento de un Instrumento crear un interface común para todos las clases derivadas de él.

La única razón para establecer este interface común es que pueda ser expresado diferente para cada diferente subtipo. Esto establece una forma básica, por eso puedes decir que es común a todas las clases derivadas. Otra forma de ver esto es llamar a Intrument una clase base abstracta (o simplemente clase abstracta). usted crea una clase abstracta cuando se quiere manipular un juego de clases a través de un interface común. Todos los métodos derivados de clases que emparejados la firma de la declaración de clase base será usada por el mecanismo de *dynamic binding* . (Sin embargo como hemos visto en la ultima sección, si el nombre del método es el mismo que el de la clase base, pero los argumentos son diferentes tendrás *sobreescritura* , que probablemente no es lo que quiere.)

Si tiene una clase abstracta como intrument, los objetos de esta clase no siempre tendrá significado. Esto es, Intrument pretende expresar solo el interface y no una implementación particular, por eso creando un objeto intrument no tiene sentido y probablemente querrá prevenir el uso de hacer esto. Esto puede ser llevado a cabo haciendo todos los métodos en el mensaje de error de Instrument, pero este aplazar la información hasta el tiempo de ejecución y requiere un test confiable y exhaustivo en la parte de usuario. Esto es siempre mejor coger problemas en tiempo de compilación.

Java provee un mecanismo para hacer estos métodos llamados abstractos. Este es un método que esta incompleto; tiene solo la declaración y no tiene cuerpo del método. Aquí esta la sintaxis para la declaración de un método abstracto:

```
abstract void X();
```

Una clase que contiene métodos abstractos se llama clase abstracta. Si una clase contiene uno o mas métodos abstractos, la clase debe ser calificada como abstracta. (De otro modo el compìlador dará un mensaje de error.)

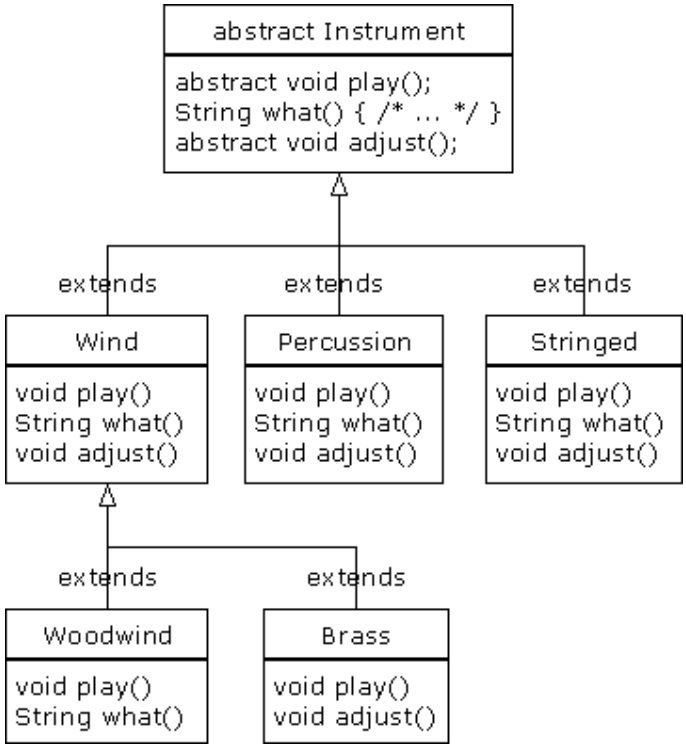
Si una clase abstracta esta incompleta, ¿que supondrá el compilador que debe hacer cuando alguien intente hacer un objeto de esa clase? Seguramente no podrá crear un objeto de una clase abstracta, por eso dará un error el compilador. De

este modo el compilador se asegura de la pureza de la clase abstracta y no necesitaras preocuparse por usarla mal.

Se hereda de una clase abstracta y quieres hacer objetos de un tipo nuevo, deberás proporcionar las definiciones de los métodos para todos los métodos abstractos en la clase base. Si no lo hace (puede no hacerlo) entonces las clases derivadas serán también abstractos y el compilador le obligará a identificar la clase como abstracta con la palabra "abstract".

Es posible declarar una clase como abstracto sin incluir ningún método abstracto. Esto es útil cuando se tiene una clase en la que no se hace buen uso para tener ningún método abstracto y todavía quieres prevenir cualquier ejemplo de esa clase.

La clase instrument puede ser cambiado fácilmente a una clase abstracto. Solo algunos de los métodos serán abstractos, desde que haciendo una clase abstracta no se fuerza a hacer todos los métodos abstractos. Aquí esta como aparece:



Aquí esta el ejemplo orquesta modificado para usar clases y métodos abstractos:

```
//: Music4.java
// Abstract classes and methods
import java.util.*;
```

```
abstract class Instrument4 {
int i; // storage allocated for each
  public abstract void play();
  public String what() {
    return "Instrument4";
  }
  public abstract void adjust();
}

class Wind4 extends Instrument4 {
  public void play() {
    System.out.println("Wind4.play()");
  }
  public String what() { return "Wind4"; }
  public void adjust() {}
}

class Percussion4 extends Instrument4 {
  public void play() {
    System.out.println("Percussion4.play()");
  }
  public String what() { return "Percussion4"; }
  public void adjust() {}
}

  class Stringed4 extends Instrument4 {
    public void play() {
    System.out.println("Stringed4.play()");
  }
  public String what() { return "Stringed4"; }
  public void adjust() {}
}

class Brass4 extends Wind4 {
  public void play() {
    System.out.println("Brass4.play()");
  }
  public void adjust() {
    System.out.println("Brass4.adjust()");
  }
}

class Woodwind4 extends Wind4 {
  public void play() {
    System.out.println("Woodwind4.play()");
  }
  public String what() { return "Woodwind4"; }
}
```

```
public class Music4 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument4 i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument4[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
  Instrument4[] orchestra = new Instrument4[5];
  int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind4();
    orchestra[i++] = new Percussion4();
    orchestra[i++] = new Stringed4();
    orchestra[i++] = new Brass4();
    orchestra[i++] = new Woodwind4();
    tuneAll(orchestra);
  }
} ///:~
```

Puede ver que no hay realmente cambio excepto en la clase base. Esto es útil para crear clases y métodos abstractos por que hacen la abstracción de una clase concreta y dicen tanto al usuario como al compilador como debe ser usado.

Interfaces

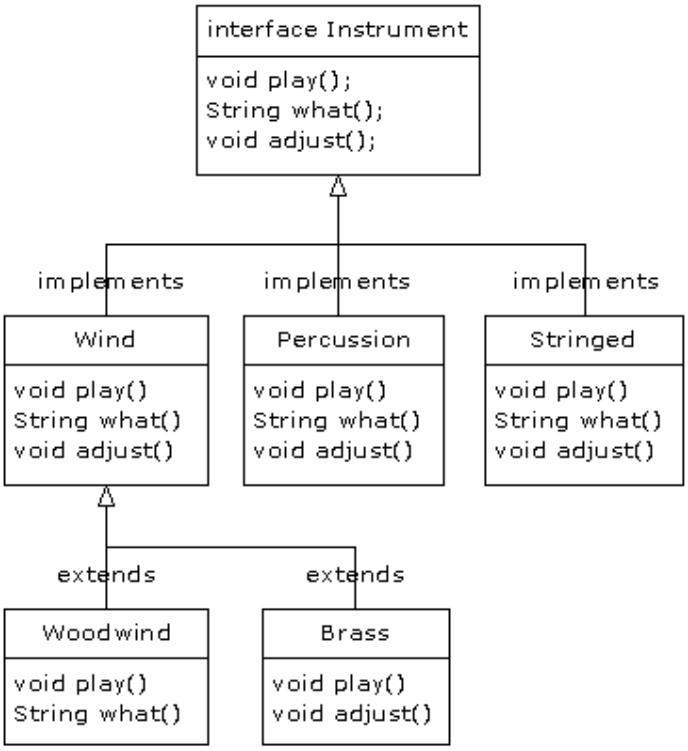
La palabra "clave" interface toma un concepto abstracto un paso más lejano. Podría pensarse que es como una clase abstracta "pura". Esto permite al creador establecer la forma para una clase: nombre de los métodos, lista de argumentos y tipos devueltos, pero no los cuerpos de los métodos. Un interface puede también contener datos de tipos primitivos, pero estos son implícitamente estáticos y final. Un interface provee solo una forma, pero no implementación.

Un interface dice: "esto es que todas las clases que implementan este particular interface se parecerán". Por eso ningún código que use un interface particular sabe que métodos deben ser llamados por ese interface, y esto es todo. Por lo que el interface es usado para establecer un "protocolo" entre clases. (Algunos lenguajes de programación orientados o objetos tienen una palabra reservada llamada "protocol" para hacer lo mismo.

Para crear un interface, usar la palabra interface en vez de class. Como una clase, puedes añadir la palabra public antes de interface (pero solo si el interface esta definido en un fichero del mismo nombre) o dejarlo para darle el estado

"amistoso".

Para hacer una clase que conforma un interface particular (o grupo de interfaces) usar la palabra implements. Con esto se esta diciendo: "El interface es lo que parece y aquí esta como funciona". otro como este, lleva un parecido fuerte para heredar. El diagrama para el ejemplo de instrumentos es este:



Una vez implementado un interface, esa implementación comienza una clase ordinaria que puede ser extendida a una forma regular.

Usted puede elegir declarar explícitamente las declaraciones del método en un interface como publico. Pero serán públicos incluso si no se le indica. Por eso cuando se implementa un interface, los métodos del interface deben ser definidos como públicos. De otro modo serán por defecto "amistosos" y podría ser restringido el acceso a un método durante la herencia, que no es permitida por el compilador de Java.

Puedes ver en esta versión modificada del ejemplo de instrumentos. Apuntar que todo método en el interface es estrictamente una declaración, que es lo único que el compilador permite. Además, ninguno de los métodos en Intrument5 son declarados como públicos, pero ellos son públicos automáticamente:

```
//: Music5.java
```

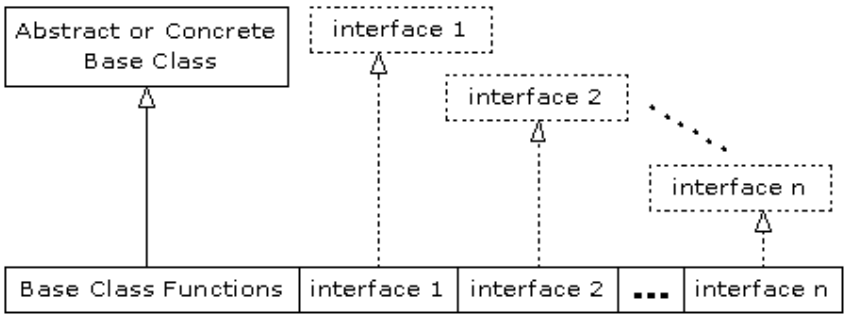
```
// Interfaces
import java.util.*;

interface Instrument5 {
  // Compile-time constant:
  int i = 5; // static & final
  // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind5 implements Instrument5 {
  public void play() {
    System.out.println("Wind5.play()");
  }
  public String what() { return "Wind5"; }
  public void adjust() {}
}

class Percussion5 implements Instrument5 {
  public void play() {
    System.out.println("Percussion5.play()");
  }
  public String what() { return "Percussion5"; }
  public void adjust() {}
}

class Stringed5 implements Instrument5 {
  public void play() {
    System.out.println("Stringed5.play()");
  }
  public String what() { return "Stringed5"; }
  public void adjust() {}
}

class Brass5 extends Wind5 {
  public void play() {
    System.out.println("Brass5.play()");
  }
  public void adjust() {
    System.out.println("Brass5.adjust()");
  }
}

class Woodwind5 extends Wind5 {
  public void play() {
    System.out.println("Woodwind5.play()");
  }
  public String what() { return "Woodwind5"; }
```

```
}

public class Music5 {
// Doesn't care about type, so new types
// added to the system still work right:
  static void tune(Instrument5 i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument5[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
  Instrument5[] orchestra = new Instrument5[5];
  int i = 0;
  // Upcasting during addition to the array:
    orchestra[i++] = new Wind5();
    orchestra[i++] = new Percussion5();
    orchestra[i++] = new Stringed5();
    orchestra[i++] = new Brass5();
    orchestra[i++] = new Woodwind5();
    tuneAll(orchestra);
  }
} ///:~
```

El resto del código funciona lo mismo. No importa si esta *upcasting* a una clase "regular" llamada Intrument5, una clase abstracta llamada Instrument5, o a un interface llamado Intrument5. La acción es la misma. De echo puede ver en el método tune() que no hay ninguna evidencia a cerca tanto si instrument5 es una clase "regular", una clase abstracta o un interface. Esto es el intento: cada entrada da al programador diferentes controles sobre la forma de que son creados y usados los objetos.

Herencia múltiple en Java

El interface no es simplemente una forma pura mas de una clase abstracta. Tiene un propósito mayor que esto. Por que un interface no tiene implementación, entonces, no hay *storage* asociado con el interface -no hay nada para impedir interfaces siendo combinados. Esto es evaluable por que hay veces cuando se necesita decir: "Una x es una a y a b y a c " En C++ esta acción se hace combinando múltiples interfaces de clases y se llama herencia múltiple, y este conlleva algunos raras consecuencias por que cada clase puede tener una implementación. En Java, puedes permitir el mismo acto, pero solo una de las clases puede tener una implementación, por eso los problemas vistos en C++ no ocurren con Java cuando se combinan múltiples interfaces:



En una clase derivada no estas obligado a tener una clase base que sea o un abstracto o "concreto" (que no tiene métodos abstractos). Si usted hereda desde un no-interface, puedes heredar de uno solo. El resto de los elementos básicos deben ser interfaces. Se colocan todos los nombres de interfaces después de la palabra implements y separados por comas. Puedes tener tantos interfaces como quieras y cada uno trae un tipo independiente que puedes *upcast* también. El ejemplo siguiente muestra una clase concreta combinada con varios interfaces para producir una clase nueva:

```
//: Adventure.java
// Multiple interfaces
import java.util.*;

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 {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero i = new Hero();
    t(i); // Treat it as a CanFight
    u(i); // Treat it as a CanSwim
    v(i); // Treat it as a CanFly
    w(i); // Treat it as an ActionCharacter
  }
} ///:~
```

Se puede ver que Hero combina la clase concreta ActionCharacter con los interfaces CanFight, CanSwin, y CanFly. Cuando se combina una clase concreta con interfaces esta forma, la clase concreta debe ir primero, después los interfaces. (El compilador dará error si no es así.)

Notar que la firma para fight() es el mismo en el interface CanFight y la clase ActionCharacter, y que fight() no es provista con una definición en Hero. La regla para un interface es que tu puedes heredar de él (como se verá pronto), pero entonces tienes otro interface. Si quiere crear un objeto de un nuevo tipo, este debe ser una clase con todas las definiciones provistas. Incluso si Hero no provee explícitamente una definición para fight(), la definición viene con Action Character por eso es automáticamente provista y es posible crear objetos de Hero.

En la clase Adventure, puedes ver que hay cuatro métodos que toman como argumento los diferentes interfaces y la clase concreta. Cuando un objeto Hero es creado, este puede ser pasado a cualquiera de esos métodos, que significa que se esta *upcast* a cada interface *in turn* . Por que la forma de los interfaces son diseñados en Java, estos funcionan sin tropiezos y sin ningún particular esfuerzo por parte del programador.

Conserva en mente que *the core reason* para interfaces se muestra en el ejemplo de abajo: para poder *upcast* a mas de un tipo base. Como sea, la segunda razón para usar interfaces es el mismo que usando una clase base abstracta: para prevenir al cliente programador de hacer un objeto de esta clase y establecer que solo es un interface. Esto nos trae una pregunta: ¿Usarías un interface o una clase abstracta? Un interface te da los beneficios de una clase abstracta y los beneficios de un interface, por eso es posible crear tu clase base sin definiciones de métodos o variables podrías preferir siempre interfaces que clases abstractas. De echo, si sabes algo será una clase base, tu primera elección podría ser hacer un interface, y solo si te ves forzado a tener definiciones de métodos o variables cambiarías a una clase abstracta.

Extendiendo un interface con herencia

Usted puede añadir fácilmente nuevas declaraciones de métodos a un interface usando la herencia y puedes también combinar varios interfaces en un nuevo interface con la herencia. En ambos casos obtendrás un nuevo interface, como se ve en este ejemplo:

```
//: HorrorShow.java
// Extending an interface with inheritance

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 HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  public static void main(String[] args) {
  DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
  }
} ///:~
```

Dangerous Monster es una extensión de Monster que produce un nuevo interface.

Esto es implementado en DragonZilla.

La sintaxis usada en Vampire funciona solamente cuando heredas interfaces. Normalmente, puedes usar extendidos con solo una clase simple, pero desde un interface puede ser echo desde otros múltiples interfaces, extendidos pueden referirse a interfaces múltiples base cuando se construye un nuevo interface. Como puedes ver los nombres del interface son simplemente separados con comas.

Agrupando constantes

Por que ningún campo que pongas en un interface será automáticamente estático y final, el interface es una utilidad conveniente para crear grupos de valores constantes, como una enumeración en C o C++, por ejemplo:

```
//: Months.java
// Using interfaces to create groups of constants
package c07;

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;
} ///:~
```

Notar que el estilo de Java para usar las letras mayúsculas (con underscores para separar múltiples palabras con un identificador simple) para las primitivas finales estáticas que tienen inicializadores constantes que es constantes en tiempo de compilación.

Los campos en un interface son públicos automáticamente, por eso no es necesario especificarlo.

Ahora puede usar las constantes importándolas del paquete c07.* o c07.Months o de cualquier otro paquete, y referenciando los valores con expresiones como Months.JANUARY. Pr supuesto, que tendrás un entero por eso no hay el tipo extra seguro que C++'s enum tenia, pero esta (usada comúnmente) técnica que es ciertamente un desarrollo del *hard-coding numbers* en tus programas. (Esto suele ser referido para usar "números mágicos" y estos producen un código muy difícil de mantener) Si quiere un tipo seguro extra, puede construir una clase como esta: (Esta aproximación fue inspirada por un e-mail de Rich Hoffarth.)

```
//: Month2.java
// A more robust enumeration system
package c07;

public final class Month2 {
private String name;

  private Month2(String nm) { name = nm; }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2("January"),
    FEB = new Month2("February"),
    MAR = new Month2("March"),
    APR = new Month2("April"),
    MAY = new Month2("May"),
    JUN = new Month2("June"),
    JUL = new Month2("July"),
    AUG = new Month2("August"),
    SEP = new Month2("September"),
    OCT = new Month2("October"),
    NOV = new Month2("November"),
    DEC = new Month2("December");
  public final static Month2[] month =  {
    JAN, JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public static void main(String[] args) {
  Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.month[12];
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
  }
} ///:~
```

La clase se llama Month2 desde que hay un mes en la librería standard de Java. Este es una clase final con un constructor privado por eso ninguno puede heredar de el o hacer ningún ejemplo de él. El único ejemplo es el final estático creado en la clase él mismo: JAN, FEB, MAR, etc. Estos objetos son también usados en el array Month, que le deja elegir meses por el número en vez del por el nombre. (Notar el extra JAN en el array para proveer un *offset* por uno, por eso que Diciembre es el mes 12) En main() puedes ver el tipo safety: m que es un objeto Month2 por lo que puede ser asignado solo a Month2. El anterior ejemplo Months.java provisto solo valores enteros, que no era demasiado seguro.

Este ejemplo también te permite usar == o "equals()" intercambiables, como se muestra al final de main().

Inicializando campos en interfaces

Los campos definidos en los interfaces son automáticamente estáticos y finales. Estos no pueden ser "blank finals", pero pueden ser inicializados con expresiones no constantes. Por Ejemplo:

```
//: RandVals.java
// Initializing interface fields with
// non-constant initializers
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

Desde que los campos son estáticos, son inicializados cuando la clase es cargada por primera vez, incluso el primer acceso a cualquier campo. Un simple test:

```
//: TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.rint);
    System.out.println(RandVals.rlong);
    System.out.println(RandVals.rfloat);
    System.out.println(RandVals.rdouble);
  }
} ///:~
```

Los campos, por supuesto, no son parte del interface pero en vez de esto son almacenados en el área de almacenamiento estático para ese interface.

Clases internas

En Java 1.1 es posible poner una definición de clase sin otra definición de clase. Este es llamado una clase internar. La clase interna es un rasgo hasta por que permite agrupar clases que lógicamente pertenecen juntos y controlar la visibilidad de uno sin otro. Como sea, es importante entender que las clases internas son distintas de la composición.

Con frecuencia, mientras se esta aprendiendo a cerca de ellas, la necesidad por las clases internas no es obvia inmediatamente. Al final de esta sección, después de todas las sintaxis y semánticas de las clases internas sean descritas, encontrarás un ejemplo que dará una idea clara de los beneficios de estas clases.

Al crear una clase interna podrías esperar: poniendo la definición de la clase dentro de la clase vecina (Ver página 94 si tiene problemas ejecutando este programa)

```
//: Parcel1.java
// Creating inner classes
package c07.parcel1;

public class Parcel1 {
  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; }
  }
  // Using inner classes looks just like
  // using any other class, within Parcel1:

  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tanzania");
  }
} ///:~
```

Las clases internas, cuando se usan dentro ship(), parece el uso de muchas clases. Aquí la única diferencia práctica es que los nombre están anidadas sin Parcel1. Verás que esta no es la única diferencia.

Mas típicamente, una clase de salida tendrá un método que devuelve un manejador a una clase interna, como esta:

```
//: Parcel2.java
// Returning a handle to an inner class
package c07.parcel2;

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 cont() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Defining handles to inner classes:
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
  }
} ///:~
```

Si quieres hacer un objeto de la clase interna en cualquier sitio excepto de uno con métodos no estáticos de una clase de salida, debes especificar el tipo de este objeto como OuterClassName.InnerClassName, como se ve en main().

Clases internas y *upcasting*

Pues, las clases internas no parecen tan dramaticas. Despues de todo, si esta escondido Java ya es un perfecto mecanismo de esconder- solo permite a las clases ser "amistosas" (visibles solo con un paquete) mejor que crearlas como una clase interna.

De cualquier forma, las clases internas realmente vienen de ellas mismas, cuando

empiezas a *upcasting* a una clase base, y en particular a un interface. (El efecto que produce un manejador de interface de un objeto que lo implementa esencialmente lo mismo que *upcasting* a una clase base.) Esto es por que la clase interna puede entonces estar completamente invisibles y no disponibles para cualquiera, que es conveniente para esconder la implementación. Todo lo que consigues es un manejador de la clase base o del interface, y es posible que incluso no encuentres el tipo exacto, como se enseña a continuación:

```
//: Parcel3.java
// Returning a handle to an inner class
package c07.parcel3;

abstract class Contents {
  abstract public int value();
}

interface Destination {
  String readLabel();
}

public class Parcel3 {
  private class PContents extends 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 dest(String s) {
    return new PDestination(s);
  }
  public Contents cont() {
    return new PContents();
  }
}

class Test {
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
    // Illegal -- can't access private class:
    //! Parcel3.PContents c = p.new PContents();
```

```
  }
} ///:~
```

Ahora Contents y Destinatination representan interfaces disponibles para el programador. (El interface, recuerda, automáticamente hace todos los miembros públicos.) Por conveniencia, estos se ponen en un solo fichero, pero ordinariamente Contents y Destination seria cada uno público en sus propios ficheros.

En Parcel3, algo nuevo ha sido añadido: la clase interna Pcontents es privada por eso ninguno excepto Parcel3 puede acceder a él. Pdestination esta protegido, por eso uno, pero Parcel3, clases en el paquete Parcel3 (desde que se protege dados los accesos a los paquetes; es, protegido es también "amistoso"), y los herederos de Parcel3 pueden acceder a Pdestination. Esto significa, que el programador ha restringido el conocimiento y el acceso de estos miembros. De echo, no puedes *downcast* a una clase interna (o a una clase interna protegida a menos que seas un heredero.), por que no puedes acceder al nombre, como puedes ver en la clase Test. Así, las clases internas privadas proveen una forma para el diseño de la clase para prevenir completamente ninguna dependencia *type-coding* y esconder los detalles de la implementación. En adicción, la extensión de un interface es menos útil para la perspectiva del programador desde que este no accede a ningún método adicional que no esta en la parte publica del interface de la clase. Esto también provee una oportunidad para el compilador de Java para generar código mas eficiente.

Las clases normales (no internas) no pueden ser privadas o protegidas - solo públicas o "amistosas".

Notar que contents no necesitan ser una clase abstracta. Podrías usar una clase ordinaria, pero los mas típicas empiezan como un diseño en un interface.

Clases internas en métodos y ámbitos

Lo que hemos visto en *encompasses* el uso típico de clases internas. En General, el código que escribirías y leer envolviendo las clases internas serán "simples" clases internas que son simples y fácil de entender. Sin embargo, el diseño de las clases internas esta completo y hay un número de otro, mas oscuro, formas que puedes usar si eliges: las clases internas pueden ser creadas con un método o incluso con un *scope* arbitrario. Hay dos razones para hacer esto:

1. Como se ha visto anteriormente, estas implementando un interface de un tipo por lo que puedes crear y devolver un manejador.
2. Estas resolviendo un complicado problema y quieres crear una clase para ayudar en tu solución, pero no quieres hacerlo disponible.

En los siguientes ejemplos, el código previo será modificado para usar:

1. Una clase definida con un método.
2. Una clase definida con un *scope* dentro de un método.
3. Una clase anónima implementando un interface.
4. Una clase anónima extendiendo una clase que tiene un constructor no por defecto.
5. Una clase anónima que permita la inicialización de campos.
6. Una clase anónima que permita construcción usando inicialización de ejemplos. (una clase interna anónima no puede tener constructores).

Esto te dará con el paquete innerscopes. Primero, los interfaces comunes del código previo que será definido en sus ficheros propios, por eso pueden ser usados en todos los ejemplos:

```
//: Destination.java
package c07.innerscopes;

interface Destination {
  String readLabel();
} ///:~
```

El punto que ha sido construido que *Contents* podría ser una clase abstracta, por eso aquí esta en una forma mas natural, como un interface:

```
//: Contents.java
package c07.innerscopes;

interface Contents {
  int value();
} ///:~
```

Aunque es una clase ordinaria con una implementación, Envoltura es algo que se viene usando como un interface común para que estas clases derivadas:

```
//: Wrapping.java
package c07.innerscopes;

public class Wrapping {
  private int i;
```

```
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

Notaras abajo que **Wrapping** tiene un constructor que requiere un argumento para hacer cosas un poco mas interesante.

El primer ejemplo enseña la creación de una clase entera con un método *scope* (en vez de el *scope* de otra clase).

```
//: Parcel4.java
// Nesting a class within a method
package c07.innerscopes;

public class Parcel4 {
  public Destination dest(String s) {
    class PDestination
        implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

La clase Pdestination es parte de dest() raro que sea parte de Parcel4. (Notar también que podrías usar el identificador de clase Pdestination para una clase interna dentro de cada clase en el mismo subdirectorio sin un nombre de clase). Entonces, Pdestination no puede ser accedido fuera de dest(). Notar que el *upcasting* que sucede, en la orden return -nada sale de dest() excepto un manejador de la clase base de destino. Por Supuesto, el echo de que el nombre de la clase Pdestination esta dentro de dest(), no significa que Pdestination no sea un objeto valido una vez que dest() acabe.

El siguiente ejemplo enseña como puede anidar una clase interna con un *scope* aleatorio:

```
//: Parcel5.java
// Nesting a class within a scope
package c07.innerscopes;

public class Parcel5 {
  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();
    }
    // Can't use it here! Out of scope:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
  }
} ///:~
```

La clase TrackingSlip esta anidada dentro de *scope* de una orden if. Esto no significa que la clase sea condicionalmente creada es compilado con todo. En cualquier caso, no esta disponible fuera de *scope* en el que esta definido. Otro, parece una clase ordinaria.

El siguiente ejemplo parece un poco extraño:

```
//: Parcel6.java
// A method that returns an anonymous inner class
package c07.innerscopes;
public class Parcel6 {
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
```

```
  }
} ///:~
```

El método cont() combina la creación de un valor devuelto con la definición de la clase que representa ese valor devuelto! . Además la clase es anónima, no tiene nombre. Para asegurarnos un pequeño mal, parece que estas creando un objeto Contents:

```
return new Contents()
```

Pero entonces, antes de conseguir el *semicolon* , dirás, "pero, espera, yo pienso yo huiré en una definición de clase:

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

Que significa esta extraña sintaxis "crear un objeto de una clase anónima que es heredada de Contents." El manejador devuelto por la nueva expresión es *upcast* automáticamente a un manejador Contents. La sintaxis de la clase anónima interna es una abreviatura para:

```
class MyContents extends Contents {
  private int i = 11;
  public int value() { return i; }
}return new MyContents();
```

En la clase anónima interna, contents es creado usando un constructor por defecto. El código siguiente enseña que hacer si tu clase base necesita un constructor con un argumento:

```
//: Parcel7.java
// An anonymous inner class that calls the
// base-class constructor
package c07.innerscopes;

public class Parcel7 {
```

```
  public Wrapping wrap(int x) {
    // Base constructor call:
    return new Wrapping(x) {
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
      Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

Esto es simplemente pasando el argumento apropiado al constructor de la clase base, mirando aquí como se pasa x en el nuevo Wrapping(x). Una clase anónima no puede tener un constructor donde normalmente llamaríamos a super().

Tanto en los ejemplos anteriores, los *semicolon* no marcan el final de la clase doby (como se hace en C++). En vez de esto, marca el final de la expresión que resulta de contiene la clase anónima. Así, es idéntico al uso de *semicolon* en cualquier otro sitio.

¿Qué sucede si necesitas realizar algún tipo de inicialización para un objeto de una clase anónima interna? Desde que es anónima, no hay nombre que darle al constructor, por eso no puedes tener un constructor. Podrías, como sea, realizar la inicialización en el punto de definición de los campos:

```
//: Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
// of Parcel5.java.
package c07.innerscopes;

public class Parcel8 {
// Argument must be final to use inside
// anonymous inner class:
  public Destination dest(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
```

```
} ///:~
```

Si estas definiendo una clase anónima interna y quieres usar un objeto que esta definido fuera de la clase anónima interna, el compilador requiere que el objeto de salida sea final. Esto es por que el objeto de salida sea final. Esto es por que el argumento para dest() es final. Si lo olvida, obtendrá un mensaje de error en tiempo de compilación.

Como estas asignando un campo, el ejemplo de abajo es excelente. Pero, ¿qué sucede si necesitas realizar algún constructor como actividad? Con Java 1.1 la inicialización de ejemplos, puedes en efecto, crear un constructor para una clase anónima interna.

```
//: Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class
package c07.innerscopes;

public class Parcel9 {
  public Destination
    dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
// Instance initialization for each object:
    {
       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) {
  Parcel9 p = new Parcel9();
  Destination d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

Dentro del inicializador de ejemplos puedes ver el código que no podrás ejecutar como parte de un inicializador de un campo (esto es, la orden if). Por eso en efecto, un inicializador de ejemplos es el constructor para una clase anónima interna. Por supuesto, esta limitada, no puedes *overload* inicializadores de ejemplos por eso solo puedes tener uno de los constructores.

El enlace a la clase exterior

Más adelante, aparece que las clases internas son justo un nombre escondido y un esquema de organización del codigo, que es útil, pero no totalmente completo. Aunque, hay otro problema. Cuando se crea una clase interna, los objetos de esa clase interna tienen un enlace al objeto "padre" que los creó, y por eso ellos pueden acceder a los miembros del objeto "padre" sin ninguna calificación especial. Además, las clases internas tienen derechos de acceso a todos los elementos en la clase "padre".[1] El siguiente ejemplo demuestra esto:

---

[1]Esto es muy diferente del diseño de clases anidadas en C++, que es simplemente un mecanismo de esconder el nombre. No hay enlace a un objeto "padre" y sin permisos implicitos en C++.

```
//: Sequence.java
// Holds a sequence of Objects

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] o;
  private int next = 0;
  public Sequence(int size) {
    o = new Object[size];
  }
  public void add(Object x) {
    if(next < o.length) {
      o[next] = x;
      next++;
    }
  }
  private class SSelector implements Selector {
  int i = 0;
    public boolean end() {
      return i == o.length;
    }
    public Object current() {
      return o[i];
    }
    public void next() {
      if(i < o.length) i++;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
  Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
       System.out.println((String)sl.current());
       sl.next();
    }
  }
} ///:~
```

La secuencia es simplemente un array de tamaño fijo de objetos con una clase envuelta en ella. Llama a add() para añadir un nuevo objeto al final de la secuencia (Lista) (si hay sitio). Para buscar cada objeto en una lista, hay un interface llamado Selector, que permite ver si estas en el final end(), mirar el objeto actual current(), y pasar al siguiente next(). Ya que el Selector es un interface, muchas otras clases pueden implementar el interface de su propia forma, y muchos métodos pueden tomar un interface como argumento para crear un código genérico.

Aquí, el Sselector es una clase privada que provee la función de Selector. En main () puedes ver la creación de una lista, seguida por añadir un numero de objetos string. Entonces, un selector se produce con una llamada a get Selector() y esta es usada para moverse a través de la lista y seleccionar cada elemento.

Primero, la creación de Sselector parece otra clase interna. Pero examínelo cuidadosamente. Note que cada uno de los métodos end() y next() se refieren a "o", que es un manejador que no es parte de Sselector, pero si es un campo privado en la clase "padre". Sin embargo, la clase interna puede acceder a métodos y a campos desde una clase "padre" como si fuera suya. Esto quita que sea muy conveniente, como puedes ver abajo en el ejemplo.

Por eso una clase, interna tuvo acceso a los miembros de las clases "padres". ¿Cómo puede suceder esto? La clase interna debe conservar una referencia a un objeto particular de la clase "padre" que fue responsable para crearlo. Entonces cuando te refieres a un miembro de la clase "padre", que (escondido) referencia es usada para seleccionar ese miembro. Afortunadamente el compilador tiene cuidado con todos estos detalles por ti, pero puedes también entender ahora que un objeto en una clase interna puede ser creado solo asociado con un objeto de una clase interna. El proceso de construcción requiere la inicialización del manejador del objeto de la clase "padre" y el compilador reclamará si no puede acceder al manejador. Muchas veces esto ocurre sin intervención por parte del programador.

Clases internas estáticas

Para comprender el significado de "estático" aplicando a las clases internas, debes recordar que el objeto de las clases internas conserva implícitamente un manejador del objeto de la clase "padre" que lo creó. Esto no es verdad, aunque, cuando se dice una clase interna es estática. Una clase interna estática significa:

1. No necesitas un objeto de una clase de salida para crear un objeto de una clase interna estática.
2. No puedes acceder a un objeto de una clase de salida desde un objeto de una clase de salida desde un objeto de una clase interna estática.

Hay algunas restricciones: los miembros estáticos, pueden estar en un nivel de

una clase por eso las clases internas no pueden tener datos estáticos o clases internas estáticas.

Si no necesita crear un objeto de una clase de salida para crear un objeto de la clase interna, puedes hacerlo todos estático. Para esto para trabajar, debes también hacer las clases internas estáticas:

```
//: Parcel10.java
// Static inner classes
package c07.parcel10;

abstract class Contents {
  abstract public int value();
}

interface Destination {
  String readLabel();
}

public class Parcel10 {
  private static class PContents
                   extends Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class PDestination
                       implements Destination {
  private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
  public String readLabel() { return label; }
  }
  public static Destination dest(String s) {
                     return new PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
  }
} ///:~
```

En main no es necesario objetos de Parcel10; en vez de eso use la sintaxis normal para seleccionar un miembro estático para llamar a los métodos que devuelve

manejadores a Contents y Destination.

Normalmente no puedes poner ningún código dentro de un interface, pero una clase interna estática puede ser parte de un interface. Desde que las clases son estáticas no violan las reglas para los interfaces - las clases internas estáticas es puesta solo en el espacio del nombre del interface:

```
//: IInterface.java
// Static inner classes inside interfaces

class IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
} ///:~
```

Anteriormente en el libro I sugería poner un main() en todas las clases *to act as a test bed* para esa clase. Un inconveniente de este es la cantidad de código extra debes arrastrar. Si este es un problema, puedes usar una clase interna estática para contener el código de test:

```
//: TestBed.java
// Putting test code in a static inner class

class TestBed {
  TestBed() {}
  void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
} ///:~
```

Esto genera una clase aparte llamada TestBed$Tester (para ejecutar el programa poner java TestBed$Tester) Puedes usar esta clase para testear, pero no necesitas incluir en tu *shipping product* .

Refiriéndose al objeto de la clase más exterior

Si necesita producir el manejador de salida de la clase objeto, ponga: nombre a la

clase de salida seguido por un punto. Por ejemplo, en la clase Sequence.Sselector ninguno de sus métodos puede producir el manejador almacenado a la clase de salida Sequence escribiendo Sequence.this. El manejador resultante es automáticamente el tipo correcto. (Esto se conoce y es chequeado en tiempo de compilación, por eso no hay *run-time overhead*). A veces quieres decir a algún otro objeto que cree un objeto de uno de estas clases internas. Para hacer esto debes proveer un manejador a la otra clase de salida en la nueva expresión, como esta:

```
//: Parcel11.java
// Creating inner classes
package c07.parcel11;

public class Parcel11 {
  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) {
    Parcel11 p = new Parcel11();
    // Must use instance of outer class
    // to create an instances of the inner class:
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d =
      p.new Destination("Tanzania");
  }
} ///:~
```

Para crear un objeto de la clase interna directamente, no sigue la misma forma y referidos al nombre de la clase d salida Parcel11 como puedes esperar, pero en vez de esto debes usar un objeto de la clase de salida para hacer un objeto de la clase interna:

```
Parcel11.Contents c = p.new Contents();
```

Así, no es posible crear un objeto de una clase interna a menos que ya tenga un objeto de la clase de salida. Esto es por que el objeto de la clase de salida es

conectado al objeto de la clase de salida de la que fue creado. Como sea, si creas una clase interna estática entonces no necesitas un manejador de la clase del objeto de salida.

Heredando de una clase interna

Por que el constructor de la clase interna debe "engancharse" al manejador del objeto de la clase "padre", las cosas se complican cuando heredas de una clase interna. El problema es que el manejador "secreto" del objeto de la clase "padre" debe ser inicializado, y todavía en la clase derivada no hay mas que un objeto por defecto a quien engancharse. La respuesta es usar la sintaxis provista para hacer la asociación explícita:

```
//: InheritInner.java
// Inheriting an inner class
class WithInner {
  class Inner {}
}

public class InheritInner
    extends WithInner.Inner {
  //! InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

Puedes ver que InheritInner es extendido solo en las clases interna, y no las de salida. Pero cuando va a crear un constructor, el de defecto no es bueno no puedes pasarle un manejador a un objeto "padre". Además, debes usar la sintaxis:

```
enclosingClassHandle.super();
```

dentro del constructor. Este provee el manejador necesario y el programa compilará entonces.

¿Pueden las clases internas ser sobreescritas?

¿Qué sucede cuando creas una clase interna, entonces heredas de una clase

"padre" y redefinir las clases interna? Es posible sobreescribir una clase interna? Esto parece que seria un concepto poderoso, pero sobreescribir una clase interna como si fuera otro método de una clase de salida no hace realmente nada:

```
//: BigEgg.java
// An inner class cannot be overriden
// like a method

class Egg {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  private Yolk y;
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() {
      System.out.println("BigEgg.Yolk()");
        }
    }
    public static void main(String[] args) {
      new BigEgg();
    }
  } ///:~
```

El constructor por defecto es sintetizado automáticamente por el compilador, y este llama al constructor por defecto de la clase base. Puede pensar que desde un BigEgg a sido creado, la versión *sobreecrita* de Yolk será usada, pero este no es el caso. La salida es:

```
New Egg()
Egg.Yolk()
```

Este ejemplo simplemente enseña que no hay ninguna clase mágica interna extra va cuando heredas de una clase mas exterior. Como sea, es posible todavía heredar explícitamente de la clase interna:

```
//: BigEgg2.java
// Proper inheritance of an inner class

class Egg2 {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  public Egg2() {
    System.out.println("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() {
      System.out.println("BigEgg2.Yolk()");
    }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
  }
} ///:~
```

Ahora BiggEgg2.Yok extiende explícitamente Egg2.Yolk y sobreescribe estos métodos. El método insertYolk() permite a BigEgg2 para *upcast* uno de sus propios objetos Yok dentro del manejador "y" en Egg2, por eso cuando g() llama a y.f() la versión *sobreescrita* de f() es usada. La salida es:

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

La segunda llamada a Egg2.Yok() es la llamada al constructor de la clase base llamada del constructor BigEgg2.Yolk. Puedes ver que la versión *overridden* de f() es usada cuando g() es llamada.

Identificadores de clases interna

Desde que todas las clases producen un fichero .class que contiene toda la información a cerca de cómo crear objetos de ese tipo (esta información produce meta-clases llamadas Clases objeto), puedes esperar que las clases interna debe también producir archivos .class para contener la información para su clases objeto. Los nombres de esos ficheros (clases tienen una formula estricta: el nombre de la clase "padre", seguida por un $, mas el nombre de la clase interna. Por ejemplo, los ficheros .class creados por InheritInner.java incluye:

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

Si las clases interna son anónimas el compilador simplemente empieza a generar números como identificadores de clases interna. Si la clase interna es anidada con clases interna, sus nombres serán simplemente añadidos después de un $ y el identificador/es de la clase de salida.

Aunque este esquema de generar nombres internos es simple y derecho, es también robusto y manejable en muchas situaciones. 3 Desde que este es el esquema de la forma de nombrar estándar en Java, los ficheros generados son automáticamente independientes de la plataforma. (Notar que el compilador de Java cambia tus clases interna en todos los tipos para hacerlos funcionar.)

¿Por qué las clases internas?. Esqueleto de control

En este punto has visto muchas sintaxis y descripciones semánticas la manera en que las clases interna funcionan, pero esto no contestan a la pregunta de por que existen. ¿Por qué trae tantos problemas para añadirlo como un rasgo fundamental del lenguaje en Java 1.1? La respuesta es algo que referiré como un esqueleto de control (framework)

Una aplicación framework es una clase o un conjunto de clases que son diseñadas para resolver un tipo particular de problema. Para aplicar una aplicación framework, tu heredas de una o mas clases y *override* algunos de los configuras la solución general provista por esta aplicación framework para resolver tu problema especifico. El control framework es un tipo particular de aplicación predominada

por framework por la necesidad.

3 En la otra mano, $ es un meta-carácter en el shell Unix y por eso tendrás a veces problemas cuando listes los ficheros .class. Esto es un poco extraño viniendo de Sun, una compañía basada en Unix. Supongo que ellos no consideraron este principio, pero en vez de pensar que naturalmente se centrará en los ficheros de código fuente.

Para responder a eventos un sistema que preliminarmente responde a eventos se llama un sistema conducido por eventos. Uno de los mas importantes problemas en la programación de aplicaciones es el interface gráfico del usuario (GUI), que es el que es mas dirigido por eventos. Como verás en el capitulo 13, Java 1.1 AWT es un control framework que elegantemente resuelve el problema del GUI usando clases internas.

Para ver como las clases interna permiten la creación simple y el uso de control framework, considerar un control framework aquel cuyo trabajo es ejecutar eventos siempre que esos eventos esten "listos".> Aunque "listos" puede significar nada, en este caso el defecto será basado en el reloj. Que siguido de un control framework que no contiene información especifica acerca de que esta controlando. Primero, aquí esta el interface que describe ningún evento de control. Este es una clase abstracta en vez de un interface, ya que el comportamiento por defecto es un control basado *on time* , por eso algunas de las implementaciones pueden ser incluidas aquí:

```
//: Event.java
// The common methods for any control event
package c07.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
} ///:~
```

El constructor simplemente captura el tiempo cuando quieras que el Evento se ejecute, mientras ready() te dice cuando es tiempo de ejecutarlo. Por supuesto, ready() puede ser *overridden* en una clase derivada basando el evento en algo mas que el tiempo. Action() es el método que es llamado cuando el evento esta

ready(), y description() da información textual a cerca del evento.

El siguiente fichero contiene el actual *control framework* que maneja y fija eventos. La primera clase es realmente un "ayudador" de clases cuyo trabajo es contener objetos eventos. Podrías reemplazarlo con cualquier apropiada colección, y en el capitulo 8 descubrirás otras colecciones que hará el engaño sin requerir que escribas código extra:

```
//: Controller.java
// Along with Event, the generic
// framework for all control systems:
package c07.controller;

// This is just a way to hold Event objects.
class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
    return; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length) && looped)
                        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}

public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
```

```
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
} ///:~
```

EventSet contiene 100 eventos arbitrariamente. (Si una colección "real" del capitulo 8 es usado aquí no necesitas preocuparte a cerca de que es del tamaño máximo, desde que se redimensione él solo). El índice usado para conservar el sitio del siguiente espacio disponible, y el siguiente es usado cuando estas buscando el siguiente Evento en la lista, para ver tanto si has *whether you-ve looped around* . Esto es importante durante una llamada a getNext(), por que los objetos Evento son eliminados de la lista (usando remove Current()) una vez que se esta ejecutando, por eso getNext() encontrará huecos en la lista que se mueven por ella.

Notar que removeCurrent() no establece solamente algunas "banderas" indicando que el objeto no estará en uso durante bastante tiempo. En vez de esto, se le manda al manejador que lo anule. Esto es importante ya que si la basura del *collector sees* un manejador que es todavía en uso entonces no puede *clean up* el objeto. Si piensas que tus manejadores pueden "engancharlo", una buena idea para ponerlos para dar permiso al colector de basura *to clean them up* .

El controlador está donde el actual trabajo está. Este usa un Set de Eventos para contener estos objetos Event, y addEvent() te permite añadir nuevos eventos a esta lista. Pero el método importante es run(). Este método dobla a través de EventSethunting para un objeto Evento que está listo ready() para ejecutarse. Para cada uno que esté listo ready(), se llama al método action(), escribe la descripción description() y entonces elimina el evento de la lista.

Notar que en este diseño no sabemos nada exactamente sobre que hace un evento. Y este es el inconveniente del diseño, como "separar las cosas que cambian de las que continúan igual". O, para usar mi termino, el "vector de cambio" son las diferentes acciones de los varios tipos de objetos. Evento, y expresar diferentes acciones creando diferentes subclases de Eventos.

Aquí es donde las clases internas entran en juego. Ellos permiten dos cosas:

1. Expresar la implementación de una aplicación control-framework en una clase simple, por el encapsulamiento todo lo que es único de esa implementación. Las clases internas son usadas para expresar los diferentes tipos de action() necesarios para resolver el problema. Además, el siguiente ejemplo usa clases internas privadas por lo que la implementación es completamente escondida y puede ser cambiada con impunidad.
2. Las clases internas conservan esta implementación *from becoming awkard* , desde que

puedes fácilmente acceder a cualquiera de los miembros de la clase "padre". Sin que esto habilite que el código pueda *become unpleasant enough that you´d end up* buscando una alternativa.

Considerar una implementación particular del control-framework diseñada para controlar las funciones "greenhouse".[2] Cada acción es completamente diferente: cambiar luces, agua, y encender o apagar termostatos, llamar al timbre, y "restart" el sistema. Pero el control-framework es diseñado para aislar fácilmente este diferente código. Para cada tipo de acción heredas un nuevo evento de la clase interna, y escribe el código de control dentro de action().

Como estipoco con una aplicación framework, la clase GreenhouseControls es heredada de un Controlador:

---

[2]Por alguna razón esto ha tenido siempre un agradable problema para resolverlo; esto viene de C++ Inside&Out, pero java permite una forma mucho mas elegante.

```
//: GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
package c07.controller;

public class GreenhouseControls
     extends Controller {
  private boolean light = false;
                          private boolean water = false;
                          private String thermostat = "Day";
                          private class LightOn extends Event {
                            public LightOn(long eventTime) {
                              super(eventTime);
                            }
                            public void action() {
                              // Put hardware control code here to
                              // physically turn on the light.
                              light = true;
                  }
                  public String description() {
                    return "Light is on";
                  }
                }
                private class LightOff extends Event {
                  public LightOff(long eventTime) {
                    super(eventTime);
                  }
                  public void action() {
                    // Put hardware control code here to
                    // physically turn off the light.
                    light = false;
                  }
                  public String description() {
                    return "Light is off";
                  }
                }
                private class WaterOn extends Event {
                  public WaterOn(long eventTime) {
                    super(eventTime);
                  }
                  public void action() {
                    // Put hardware control code here
                    water = true;
                  }
                  public String description() {
                    return "Greenhouse water is on";
                  }
```

```
      }
      private class WaterOff extends Event {
        public WaterOff(long eventTime) {
          super(eventTime);
        }
        public void action() {
          // Put hardware control code here
          water = false;
        }
        public String description() {
          return "Greenhouse water is off";
        }
      }
      private class ThermostatNight extends Event {
        public ThermostatNight(long eventTime) {
          super(eventTime);
        }
        public void action() {
          // Put hardware control code here
          thermostat = "Night";
        }
        public String description() {
          return "Thermostat on night setting";
        }
      }
   private class ThermostatDay extends Event {
  public ThermostatDay(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Put hardware control code here
    thermostat = "Day";
  }
  public String description() {
    return "Thermostat on day setting";
  }
}
// An example of an action() that inserts a
// new one of itself into the event list:
private int rings;
private class Bell extends Event {
  public Bell(long eventTime) {
    super(eventTime);
  }
  public void action() {
    // Ring bell every 2 seconds, rings times:
    System.out.println("Bing!");
    if(--rings > 0)
      addEvent(new Bell(
        System.currentTimeMillis() + 2000));
```

```
          }
          public String description() {
            return "Ring bell";
          }
        }
        private class Restart extends Event {
          public Restart(long eventTime) {
      super(eventTime);
    }
    public void action() {
    long tm = System.currentTimeMillis();
      // Instead of hard-wiring, you could parse
      // configuration information from a text
      // file here:
      rings = 5;
      addEvent(new ThermostatNight(tm));
      addEvent(new LightOn(tm + 1000));
      addEvent(new LightOff(tm + 2000));
      addEvent(new WaterOn(tm + 3000));
      addEvent(new WaterOff(tm + 8000));
      addEvent(new Bell(tm + 9000));
      addEvent(new ThermostatDay(tm + 10000));
      // Can even add a Restart object!
      addEvent(new Restart(tm + 20000));
    }
    public String description() {
      return "Restarting system";
    }
  }

  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
} ///:~
```

Notar que luz, agua, termostato, y timbres pertenecen todas a la clase de salida GreenhouseControls, y todavía las clases interna no tienen problemas para acceder a esos campos. También, muchas de los métodos action() también envuelven algún tipo de control hardware, que envolverán llamadas a código no-Java.

La mayoría de las clases eventos parecen similares, pero Bell y Restart son especiales. Bell suena, y si esto no se hubiera ejecutado todavía suficientes veces esto añadiría un nuevo objeto Bell a la lista de Eventos, por eso sonará de nuevo mas tarde. Notar como interna clases parece herencia múltiple: Bell tiene todos

los métodos de Event y esto también aparece para tener todos los métodos de la clase de salida GreenhouseControls.

Restart es el responsable de inicializar el sistema, por eso añade los eventos apropiados. Por supuesto, una forma mas flexible de ejemplo esto es *avoid hard-coding* los eventos en vez de leerlos del fichero. (Un ejercicio en el capítulo 10 te pedirá modificar este ejemplo para hacer justamente esto). Desde que Restart() es otro objeto evento puedes tambien añadir un objeto Restart con Restart.action () por eso el sistema regularmente restarts el mismo. Y todo lo que necesitas hacer en main() es crear un objeto GreenhouseControls y añadir un objeto Restart para que funcione.

Este ejemplo te llevará a apreciar el valor de las clases interna, especialmente cuando se usan con un control framework. Aunque, en la segunda parte del capitulo 13 veremos como son usados las clases internas elegantemente para describir las acciones de un interface gráfico de un usuario. Al tiempo que terminas esta sección debería ser completamente convincente.

Constructores y polimorfismo

Como es usual, los contructores son diferentes de los otros tipos de métodos. Esto es también verdad cuando el polimorfismo es envuelto. Incluso cuando los constructores no son polimorficos (aunque puedes tener un tipo de "contructor virtual", como veremos en el capitulo 11), es importante comprender la forma de que trabajan los constructores en jerarquías complejas y con polimorfismo. Esto te ayudará a *avoid unpleasant entanglements* .

Orden de las llamadas a constructores

El orden de las llamadas a constructores fue discutida en el capitulo 4, pero esto fue antes de que la herencia y el polimorfismo fuera introducido.

Un constructor para la clase base es siempre llamado en el constructor para una clase derivada, *chaining upward so that* un constructor para todas las clases base es llamado. Este se asegura que el constructor tiene un trabajo especial: ver que el objeto sea construido correctamente. Una clase derivada tiene acceso solo a estos propios miembros, y no a los de las clases base (esos miembros son típicamente primados). Solo el constructor de la clase-base tiene el conocimiento adecuado y acceso a inicializar sus propios elementos.

Entonces, es esencial que todos los constructores sean llamados, de otro modo el objeto no será construido correctamente. Esto es por que el compilador forza una llamada a un constructor para cada porción de una clase derivada. Esto llamará "silenciosamente" al constructor por defecto si no especificas un constructor de clase-base en el cuerpo del constructor de la clase-derivada. Si no hay constructor por defecto, el compilador protestará. (En el caso de que una clase no tuviera constructores, el compilador automáticamente sintetizará un constructor por

defecto.)

Echemos un vistazo a un ejemplo que enseña los efectos de composición, herencia, y polimorfismo para la construcción:

```
//: Sandwich.java
// Order of constructor calls

class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
} ///:~
```

Este ejemplo crea una clase compleja de salida de otras clases, y cada clase tiene

un constructor que se anuncia a si mismo. La clase importante es Sandwich, que refleja 3 niveles de herencia (4, si contamos la herencia implícita del objeto) y 3 objetos miembros. Cuando un objeto Sandwich es creado en main(), la salida es:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

Esto significa que el orden de llamada al constructor para un objeto complejo como sigue:

1. El constructor de la clase-base se llama. Este paso se repite recursivamente hasta que el raiz de la jerarquía se construye primero, seguido por la siguiente clase-derivada, etc. Hasta que la clase mas derivada sea alcanzada.
2. Los inicializadores son llamados en el orden de declaración.
3. El cuerpo del constructor de la clase-derivada es llamada.

El orden de las llamadas al constructor es importante. Cuando heredas, sabes todo a cerca de la clase base y puede acceder a cualquier miembros públicos y protegidos de la clase base. Esto significa que debes *be able to assume* que todos los miembros de una clase base son validos cuando estas en una clase derivada. En un método normal, la construcción ya ha tenido lugar, por eso todos los miembros de todas partes del objeto que ha sido construido. Dentro del constructor, Aunque, debes asumir que todos los miembros que usa han sido construidos. La única forma de garantizar esto es para los constructores de clase-base para ser llamados primeros. Entonces cuando los miembros puedes acceder en la clase base han sido inicializados. "Sabiendo que todos los miembros son validos" dentro del constructor es también razón de que, siempre que sea posible, debes inicializar todos los objetos miembro (que es, objetos puestos en la clase usando la composición) a su punto de definición en la clase. (e.g.: b, c, y l en el ejemplo de abajo). Si sigues esta practica, ayudarás *ensure* que todos los miembros de la clase base y los miembros objetos del objeto actual ha sido inicializado. Desafortunadamente, esto no se maneja en muchos casos, como verás en la siguiente sección.

Herencia y finalize( )

Cuando usas la composición para crear una clase nueva, no te preocupes nunca por finalizar los objetos miembro de esa clase. Cada miembro es un objeto

independiente y así es basura *collected and finalized regardless of whether* sucede a un miembro de tu clase. Con la herencia, aunque debes *override* finalize() in the clase derivada si no tienes ningún *clanup* especial que deba suceder como parte de la colección de basura. Cuando *override* finalize() en una clase heredada, es importante recordar llamar la versión clase-base de finalize(), de otro modo la finalización de la clase-base no ocurrirá. El siguiente ejemplo provee esto:

```
//: Frog.java
// Testing finalize with inheritance

class DoBaseFinalization {
  public static boolean flag = false;
}

class Characteristic {
  String s;
  Characteristic(String c) {
    s = c;
    System.out.println(
      "Creating Characteristic " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizing Characteristic " + s);
  }
}

class LivingCreature {
  Characteristic p =
    new Characteristic("is alive");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void finalize() {
    System.out.println(
      "LivingCreature finalize");
    // Call base-class version LAST!
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Animal extends LivingCreature {
  Characteristic p =
    new Characteristic("has heart");
  Animal() {
```

```
    System.out.println("Animal()");
  }
  protected void finalize() {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

class Amphibian extends Animal {
  Characteristic p =
    new Characteristic("can live in water");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void finalize() {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
      try {
        super.finalize();
      } catch(Throwable t) {}
  }
}

        public class Frog extends Amphibian {
          Frog() {
            System.out.println("Frog()");
          }
          protected void finalize() {
            System.out.println("Frog finalize");
            if(DoBaseFinalization.flag)
              try {
                super.finalize();
              } catch(Throwable t) {}
          }
          public static void main(String[] args) {
            if(args.length != 0 &&
               args[0].equals("finalize"))
               DoBaseFinalization.flag = true;
            else
              System.out.println("not finalizing bases");
            new Frog(); // Instantly becomes garbage
            System.out.println("bye!");
            // Must do this to guarantee that all
            // finalizers will be called:
            System.runFinalizersOnExit(true);
          }
        } ///:~
```

La clase DoBaseFinalization simplemente contiene un indicador que indica a cada clase en la jerarquía al llamar a super.finalize(). Este indicador esta basado en un argumento de línea de comando, por eso puedes ver el comportamiento con y sin finalización de la clase-base.

Cada clase en la jerarquía también contiene un objeto miembro de la clase Characteristic. Veremos que los *regardless of whether* finalizadores de la clase base son llamados, los objetos miembros de Characteristic son siempre finalizados.

Cada método finalize() *overridden* debe tener acceso al menos a los miembros protegidos desde que el método finalize() en la clase Object esta protegida y el compilador no te permitirá reducir el acceso durante la herencia. ("Amistosamente" es menos accesible que protegido.)

En Frog.main(), el indicador DoBaseFinalization esta configurada y un objeto simple Frog es creado. Recuerde que la colección de basura y en particular la finalización puede no ocurrir para ningún objeto particular por eso para forzar esto, System.runFinalizersOnExit(true) añade el *overhead* extra para garantizar que la finalizar tiene lugar. Sin la finalización de la clase-base, la salida es:

```
not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Puedes ver que, realmente, no son llamados los finalizadores para la clase base de Frog. Pero si añade el argumento "finalize" en la línea de comando, consigues:

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
```

```
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

Aunque el orden de los objetos miembro están finalizados es el mismo orden en que fueron creados, técnicamente el orden de finalización de objetos esta inespecificado. Con las clases base, Aunque, tienes control sobre el orden de finalización. El mejor orden para usar es el que se muestra aquí, que es inverso al orden de inicialización. Siguiendo la forma usada en C++ para destructores, deberías permitir que las clases derivadas en la primera finalización, entonces finaliza la clase base. Esto es por que la finalización de la clase derivada podría llamar a algunos métodos en la clase base que requiere que los componentes de la clase base están todavía "vivas", por eso debes no destruirlos prematuramente.

Funcionamiento de los métodos polimorficos dentro de los constructores

La jerarquía de las llamadas a constructores nos trae un dilema interesante. ¿Qué sucede si estas dentro de un constructor y llamas a un método *dynamically-bound* del objeto que s esta construyendo? Dentro de un método ordinario puedes imaginar que sucederá o la llamada al *dynamically-bound* es resuelta en tiempo de ejecución por que el objeto no puede saber si pertenece a la clase el método está dentro o alguna clase derivada de ella. Puedes pensar esto es que podría suceder dentro de los constructores.

Este no es exactamente el caso. Si llamas a un método *dynamically-bound* dentro de un constructor, la definición *overridden* para ese método es usada aunque, el efecto puede ser inesperado, y puede conllevar algunos errores difíciles de encontrar.

Conceptualmente, el trabajo del constructor es traer el objeto a la existenica (que es duro una ordinaria hazaña). Dentro de cualquier constructor, el objeto puede ser solo parcialmente formado y solo puedes saber que los objetos de la clase-base han sido inicializados, pero no puedes saber que clases son heredados. Una llamada a un método dynamically-bound, Aunque, "adelante", o "detrás" dentro de la jerarquía de herencia. Este llama un método en una clase derivada. Si haces esto dentro de un constructor, llamas a un método que puedas manipular miembros que no hayan sido inicializados todavía y una receta para el desastre.

Puedes ver el problema en el ejemplo siguiente:

```
//: PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = "
      + radius);
  }
  void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}

public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} ///:~
```

En Glyph, el método draw() es abstracto, por eso esta diseñado para ser sobreescrito. Realmente, estas forzado para que lo *sobreescriba* en Round Glyph. Pero el constructor Glyph llama a este método, y la llamada termina, en RoundGlyph.draw() que parecería que hace el intento. Pero mire la salida:

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

Cuando el constructor de Glyph llama a draw(), el valor de radius no está todavía inicializado a 1. Es cero, esto resultará, probablemente o un punto o nada será pintado en la pantalla, y estará *staring* intentando *to figure out* porque el programa no funcionará.

El orden de inicialización descrita en la sección previa *isn´t quite complete* , y que es la llave para resolver el misterio. El actual proceso de inicialización es:

1. El almacenamiento señalado para los objetos es inicializado a 0 binario antes de que nada suceda.
2. Los constructores clase-base son llamados como se describió anteriormente. En este punto, el método *overridden* draw() es llamado (si, antes de que sea llamado el constructor de RoundGlyph) que describe un radio de valor de cero, debido al paso 1.
3. Los inicializadores de miembros son llamados en el orden de declaración.
4. El cuerpo de los constructores de la clase-derivada es llamada.

Hay una razón para este, es que todo es al menos inicializado a cero (o, cero significa para este particular tipo de dato) y no figura como basura. Esto incluye los manejadores de objeto que son *embedbed* dentro de una clase vía composición. Por eso si olvidas inicializar que el manejador conseguirá una excepción en tiempo de ejecución. Todo lo demás tiene 0, que es usualmente el valor *a telltale* cuando *locking at output* .

Por otro lado, deberías extrañarte de este programa. Has hecho unas cosas perfectamente lógicas y su funcionamiento es misteriosamente erroneo, sin protesta del compilador (C++ produce funcionamiento mas racional en esta situación.) Errores como esto, podrían ser fácilmente aburridos y que llevaría tiempo descubrir.

Como resultado, una buena guía en línea para constructores es, hacer lo menos posible para poner el objeto en el estado bueno, y si puede evitarse, no hay ningún método." El único método seguro es llamar dentro un constructor es este que es final en la clase base. (Eso también es aplicable a método privados que son automáticamente finales). Estos no pueden ser *overridden* y así no puede producir este tipo de sorpresa.

Diseñando con herencia

Una vez hemos aprendido a cerca del polimorfismo, puede parecer que todo debe ser heredado por que el polimorfismo es una inteligente herramienta. Esto puede ser una carga para tus diseños; de echo si eliges la herencia primero cuando estas usando una clase existente para hacer nuevas clases con clases puede ser complicado.

Un mejor ejemplo es elegir primero composición, cuando esto no es obvio cual debemos usar la composición no fuerza al diseño en una jerarquía de herencia.

Pero la composición es también mas flexible desde que es posible elegir dinámicamente un tipo (y su funcionamiento) cuando usas composición, como una herencia requiere un tipo exacto conocido en tiempo de compilación. El siguiente ejemplo ilustra esto:

```
//: Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

interface Actor {
  void act();
}

class HappyActor implements Actor {
  public void act() {
    System.out.println("HappyActor");
  }
}

class SadActor implements Actor {
  public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  void change() { a = new SadActor(); }
  void go() { a.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage s = new Stage();
    s.go(); // Prints "HappyActor"
    s.change();
    s.go(); // Prints "SadActor"
  }
} ///:~
```

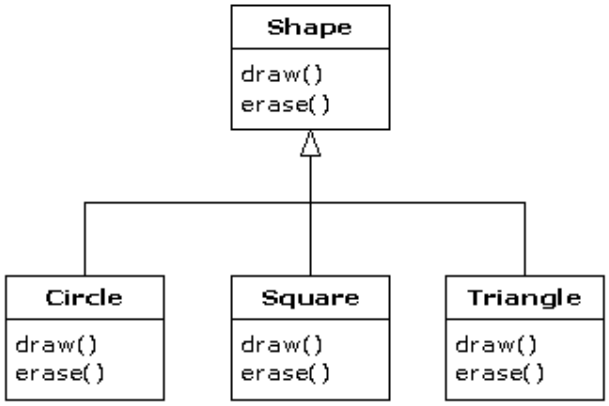
Un objeto Stage contiene un manejador para un Actor, que es inicializado a un objeto HappyActor. Esto significa que go() produce un funcionamiento particular. Pero desde que un manejador puede ser *re-bound* a un objeto diferente en tiempo de compilación, el manejador para el objeto SadActor puede ser sustituido en "a" y entonces el funcionamiento producido por go() cambia. Incluso ganas en flexibilidad dinámica en tiempo de ejecución. En contraste, no puedes decidir en heredar diferentemente en tiempo de ejecución; que debe ser completamente

determinado en tiempo de compilación.

Una guía en línea general es usar la herencia para expresar diferencias en su funcionamiento, y variables miembro para expresar variaciones en el estado." En el ejemplo de abajo, son usados tanto: 2 clases diferentes son heredados para expresar la diferencia en el método act(), y Stage usa composición para permitir que el estado sea cambiando. En este caso, este cambio es el estado sucede para producir un cambio en su funcionamiento.

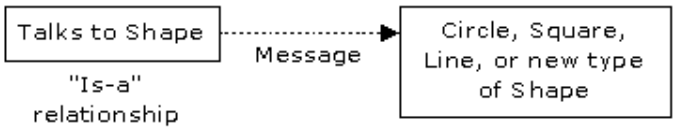
Herencia pura vs. extension

Cuando estudiamos la herencia, podría parecer que la forma mas clara de crear una jerarquía de herencia es tomar el ejemplo puro. Esto es, solo con método que han sido establecidos en la clase base o interface que serán *overridden* en la clase derivada, como se ve en este diagrama:



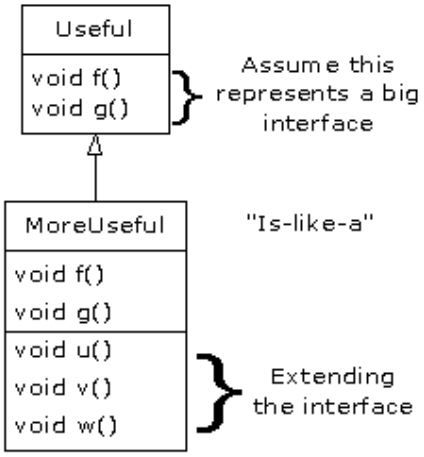
Este puede ser llamado una relación "ia-a" pura por la interface de una clase establecida que es. La herencia garantiza que ninguna clase derivada tendrá el interface de la clase base y nada menos. Si sigues el diagrama de abajo, las clases derivadas tendrán también no mas que el interface de la clase base.

Esto puede a traves de una sustitución pura, por que los objetos de la clase derivada pueden ser perfectamente sustituidos por la clase base, y nunca necesitarás saber ninguna información extra a cerca de las subclases cuando estás usándolas:



Esto es, la clase base puede recibir cualquier mensaje que puedas mandar a la clase derivada ya que las dos tienen exactamente el mismo interface. Todo lo que necesitas hacer es *upcast* de la clase derivada y nunca volver a mirara que tipo exacto de objeto es con el que estas tratando. Todo es manejado a través del polimorfismo.

Cuando ves esta forma, parecerá una relación "is-a" pura es la única forma sensible para hacer cosas, y ningún otro diseño indica pensamiento "barroso" y es por definición "broken". Esto es también una trampa. Tan pronto como empieces a pensar de esta forma, volverás y descubrirás que extendido el interface (con, desafortunadamente, la palabra clave extiende parece promover) es la perfecta solución a un problema particular. Esto podría ser llamado una relacción "is-like-a" por que la clase derivada es como la clase base y esto tiene fundamentalmente el mismo interface y pero esto tiene otras caracteristicas que requieren métodos adicionales para implementar:



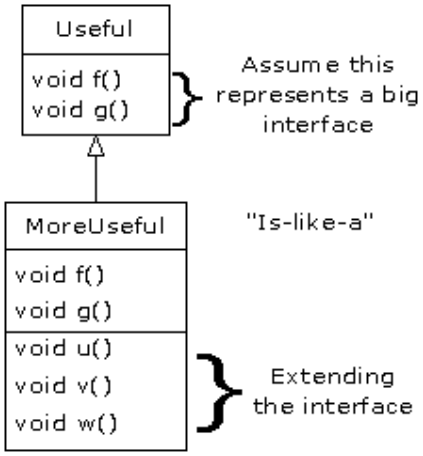
Mientras esto sea también un ejemplo útil y sensible (depende de la situación) esto tiene una desventaja la parte extendida del interface en la clase derivada no esta disponible desde la clase base, por lo que una vez *upcast* no puedes llamar los nuevos métodos:

Si no estas *upcasting* en este caso, no te preocupará pero con frecuencia te encontrarás en una situación en la que necesitas redescubrir el tipo exacto del objeto, por eso puedes acceder a los métodos extendidos de este tipo. La sección

siguiente enseña como se hace esto.

Downcasting y identificación de tipos en tiempo de ejecución

Desde que pierdes la información del tipo especifico por un *upcast* (moviéndose en la jerarquía de herencia), este se asegura que para recuperar la información del tipo y esto es, moverse abajo en la jerarquía de herencia y usa un *downcast* . Como sea, saber que un *upcast* es siempre seguro; la clase base no puede tener un interface mas grande que el de la derivada, entonces todos los mensajes que se mandan a través del interface de la clase base es garantizado que será aceptado. Pero con un *downcast* , no puedes realmente saber que "shape" (por ejemplo) es actualmente un circulo. Esto puede en vez de ser un triángulo o un cuadrado u otro tipo.



Para resolver este problema debe existir alguna forma de garantizar que un *downcast* es correcto, por eso no *cast* accidentalmente a un tipo erróneo y luego mandar un mensaje que el objeto no puede aceptar. Esto no seria seguro.

En algunos lenguajes (como C++) deber permitir una operación especial para conseguir un *downcast* de tipo seguro, pero en Java todos los *cast* son chequeados. Incluso cuando parecen proporcionados por un parentesco ordinario en un *cast* , en tiempo de ejecución este *cast* es chequeado para asegurarse de que es de echo el tipo que pensamos que es. Si no lo es, tendrás una ClassCastException. Este acto de chequear los tipos en tiempo de ejecución es llamando identificación de tipo en tiempo de ejecución (ITTE). El próximo ejemplo demuestra el funcionamiento de (ITTE):

```
//: RTTI.java
// Downcasting & Run-Time Type
```

```
// Identification (RTTI)
import java.util.*;

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();
    // Compile-time: method not found in Useful:
    //! x[1].u();
   ((MoreUseful)x[1]).u(); // Downcast/RTTI
   ((MoreUseful)x[0]).u(); // Exception thrown
  }
} ///:~
```

Como en el diagrama, MoreUseful extiende el interface de Useful. Pero desde que es heredado, puede ser también *upcast* a un Useful. Puedes ver este echo en la inicialización del array x en main(). Desde que ambos objetos en el array son de la clase Useful, puedes mandar los f() y g() a ambos, y si intentas llamar al método u() (que existe solo en MoreUseful) obtendrás un error en tiempo de compilación.

Si quieres acceder al interface extendido del objeto MoreUseful, puedes intentar *downcast* . Si es el tipo correcto, será satisfactorio. De otro modo obtendrás un ClassCastException. No necesitas escribir ningún código especial para esta excepción, desde que indica un error al programador que puede suceder en cualquier sitio de un programa.

Hay mas de ITTE que un simple *cast* . Por ejemplo, hay una forma de ver que tipo es con el que estas tratando antes de intentar *downcast it* . Todo el capitulo 11 es dedicado a estudiar los diferentes aspectos de Java de identificación de tipos en tiempo de ejecución.

Resumen

Polimorfismo significa "diferentes formas". En la programación orientada a objetos, tienes la misma cara (el mismo interface en la clase base) y diferentes formas de usar esa cara: las diferentes versiones de los métodos *dynamically-bound* .

Hemos visto en este capitulo que es imposible comprender o incluso crear un ejemplo de polimorfismo sin usar datos de abstracción y herencia. El polimorfismo es una característica que no puede ser visto aislado (como una orden switch, por ejemplo), pero en vez de funcionar solo como se planeo, como parte de un "gran-cuadro" de clases de relaciones. La gente suele confundirse por otros, caracteristicas de Java no orientadas a objetos, como método *overloading* que son algunas veces presentados como objeto orientado. No se engañe: si no es *late binding* , no es polimorfismo.

Para usar polimorfimso, y así las técnicas orientadas a objetos, efectivamente en tus programas debes expandir tu visión de programar para incluir no solamente miembros y mensajes de clase individuales pero tambien las diferentes clases comunes y sus relaciones con cada otro. Aunque requiere un esfuerzo significante, es un preciado esfuerzo, por que los resultados son mas rápidos que el desarrollo de programas mejores organizados y codificados, programas extensibles, y código mas fácil de mantener.

Ejercicios

1. Añade un nuevo método a la clase base **Shapes.java** que imprima un mensaje, pero no lo sobreescribas en la clase derivada. Explica que ocurre. Ahora sobreescríbelo en una de las clases derivadas pero no en las otra y observa lo que ocurre. Finalmente, sobreescríbelo en todas las clases derivadas.
2. Añade un nuevo tipo de **Shape** a **Shapes.java** y verifica en el **main()** que el polimorfismo funciona para tu nuevo tipo como lo hacía para los viejos.
3. Cambia **Music3.java** para que **what()** sea predominante sobre el método **toString()** de **Object** . Intenta imprimir el objeto **Instrument** usando **System.out.println()** (sin ningún casting).
4. Añade una nuevo tipo de **Instrument** a **Music3.java** y verifica que el polimorfismo funciona para tu nuevo tipo.
5. Modifica **Music3.java** para que cree objetos tipo **Instrument** aleatoriamente de la forma que **Shapes.java** lo hace.
6. Crear una jerarquía de herencia de Rodent: Mouse, Gerbil, Hamster, etc. En la clase base se proveen métodos que son comunes a todos los Rodents y *override* esto en las clase derivadas para permitir diferentes funcionamientos dependiente de un tipo especifico de Rodent. Crear un array de Rodent, llenarlo con diferentes tipo específicos de Rodents, y llame a los métodos de la clase base para ver que sucede.
7. Cambiar el ejercicio 6 parar que **Rodent** sea una clase abstracta. Transforma los

métodos de **Rodent** en **abstract** en la medida que sea posible.

8. Crea una clase como **abstract** sin incluir ningún método abstracto y verifica que no puedes crear ninguna llamada a esa clase.
9. Añade la clase **Pickle** a **Sandwich.java** .
10. Modifica el ejercicio 6 para que muestre el orden de inicialización de las clases bases y derivadas. Ahora, añade un objeto miembro a las clases base y derivadas, y observa el orden en el cual la inicialización ocurre en la construcción de los objetos.
11. Crea una jerarquía de herencia de tercer nivel. Cada clase en la jerarquía debe tener un método **finalize()** y debe llamar debidamente a la versión de **finalize()** de la clase base. Demostrar que tu jerarquía funciona correctamente.
12. Crear una clase base con dos métodos. En el primero, llama al segundo. De ella hereda una clase y sobreescribe el segundo método. Crea un objeto de la clase derivada, haz un "upcast" hacia la clae base y llama al primer método. Explica lo que ocurre.
13. Crea una clase con un método **abstract imprime()** que sea sobreescrito en una clase derivada. La versión sobreescrita del método imprime el valor de un entero definido en la clase derivada. Como valor inicial, dale un valor distinto de cero. En el constructor de la clase base, llama a este método. En **main()** , crea un objeto de la clase derivada y llama a su **imprime()** . Explica los resultados.
14. Siguiendo el ejemplo en **Transmogrify.java** , crea una clase **Starship** que contenga una referencia a **AlertStatus** que pueda indicar tres estados diferentes. Incluye métodos para cambiar los estados.
15. Crea una clase abstracta sin métodos. Deriva de ella una clase y añade un método. Crea un método estático que haga una referencia a la clase base, haz un "downcasts" hacia la clase derivada y llama al método. En el **main()** , demuestra como trabaja. Ahora pon la declaración de abstracta para el método en la clase base, lo que elimina la necesidad de hacer un "downcasts".

8: Interfaces y Clases Internas

Interfaces y clases internas proporcionan formas más sofisticadas para organizar y controlar los objetos en tu sistema.

C++, por ejemplo, no contiene estos mecanismos, aunque un programador inteligente puede simularlos. El hecho de que existan en Java indica que son considerados lo bastante importantes como para proporcionar apoyo directo mediante palabras clave del lenguaje.

En el Capítulo 7 , aprendiste sobre la palabra clave **abstract** , la cual te permite crear uno o más métodos en una clase sin tener definiciones - tú proporcionas parte del interfaz sin proporcionar una implementación correspondiente, la cual es creada por los herederos. La palabra clave **interface** produce completamente una clase abstracta, la cual no proporciona implementación a todo. Aprenderás que la **interface** es más que sólo una clase abstracta llevada a un extremo, ya que te permite realizar una variación sobre la "herencia múltiple" de C++, creando una clase que puede ser upcast a más de un tipo base.

Al principio, las clases internas parecen un mecanismo simple de ocultación de código: tú colocas clases dentro de otras clases. Aprenderás, sin embargo, que la clase interna hace más que eso - conoce y se puede comunicar con la clase circundante - y que el tipo de código que puedes escribir con clases internas es más elegante y claro, aunque sea un nuevo concepto para muchos. Lleva algo de tiempo el sentirse cómodo con el diseño usando clases internas.

Interfaces

La palabra reservada **interface** lleva el concepto **abstract** un paso más allá. PodrÍas pensar en ello como una clase **abstract** "pura". Esto permite al creador establecer la forma de una clase: nombres de métodos, listas de argumentos, y tipos de devolución, pero no los cuerpos de los métodos. Un **interface** puede contener campos, pero éstos son implícitamente **static** y **final** . Un **interface** proporciona sólo una forma, no una implementación.

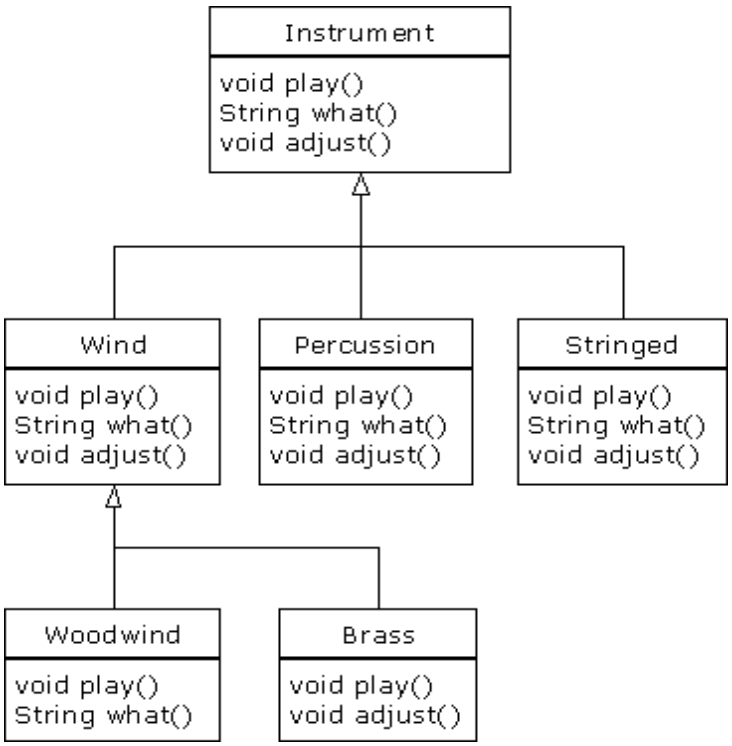
Un **interface** dice: "Así es como parecerán todas las clases que *implementen* este interface en particular."

Aunque cualquier código que use un **interface** en particular sabe qué métodos podrían ser llamados por ese **interface** , y eso es todo. Así que el **interface** se usa para establecer un "protocolo" entre clases. (Algunos lenguajes de programación orientados a objetos tienen una palabra clave llamada **protocol** que

hace lo mismo.)

Para crear un **interface** , utilice la palabra clave **interface** en lugar de la palabra clave **class** . Como en una clase, puedes aÃ±adir la palabra reservada **public** antes de la palabra clave **interface** (pero sólo si ese **interface** está definido en un fichero del mismo nombre) o no ponerle nada para darle un estado "amigo" de forma que sólo se pueda usar dentro del mismo paquete.

Para hacer una clase que se ajuste a un **interface** particular (o grupo de **interfaces**) utilice la palabra reservada **implements** . Estás diciendo "El **interface** es lo que parece, pero ahora voy a decir cómo **trabaja** ."Más que eso, parece herencia. El diagrama para mostrar un ejemplo es:



Una vez que has implementado un **interface** , esa implementación se convierte en una clase ordinaria que puede ser extendida de forma regular.

Puedes elegir declarar de forma explícita las declaraciones de métodos en un **interface** como **public** . Pero son **public** incluso si no lo indicas. De esta manera cuando **implementas** un **interface** , los métodos del **interface** deben ser definidos **public** . De otra manera serîan por defecto "amigos", y estarías reduciendo la accesibilidad de un método durante la herencia, lo cual no es permitido por el compilador de Java.

Puedes ver esto en la versión modificada del ejemplo **Instrument** . Advierte que

cada método en el **interface** es extrictamente una declaración, lo cual es lo único que el compilador permite. En suma, ninguno de los métodos en **Instrument** son declarados como **public** , pero son **public** automáticamente de todas maneras:

```
//: c08:music5:Music5.java
// Interfaces.
import java.util.*;

interface Instrument {
  // Constante en tiempo de compilación:
  int i = 5; // static & final
  // No puedes tener definiciones de métodos:
  void play(); // Automaticamemte público
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
```

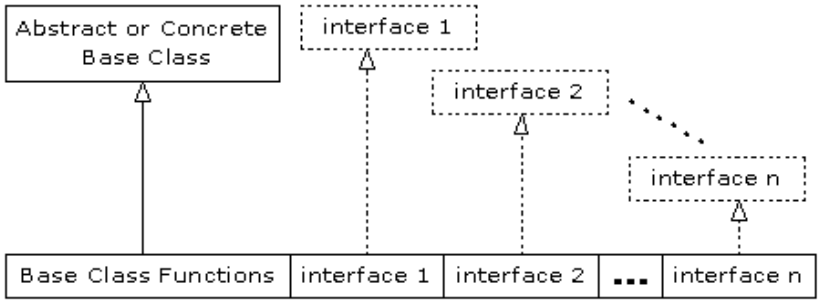
```
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  // No te preocupes del tipo, porque los
  // nuevos tipos son añadidos al sistema
  // todavía funcionan correctamente:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i <e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

El resto del código funciona de la misma manera. No importa si estás lanzando hacia arriba a una clase "regula" llamada **Instrument** , una clase **abstracta** llamada **Instrument** , o a un **interface** llamado **Instrument** . El comportamiento es el mismo. De hecho, puedes ver en el método **tune()** que no hay ninguna evidencia acerca de si **Instrument** es una clase "regular", una clase **abstracta** , o un **interface** . Este es el intento: Cada realización da al programador diferente control sobre la forma en que los objetos son creados y usados.

"Herencia múltiple" en Java

El **interface** no es sólo una forma "más pura" de una clase **abstracta** . Tiene un propósito mayor que ese. Como un **interface** no tiene implementación en absoluto- es decir, no hay almacenamiento asociado con un **interface** - no hay nada que evite que varios **interfaces** puedan combinarse. Esto debe valorarse

porque hay veces que necesitas decir "Un **x** es un **a** ***y*** un **b** ***y*** un **c** ". En C++, este hecho de combinar múltiples interfaces de clases es llamado ***herencia múltiple*** , y aporta bastante equipaje incómodo porque cada clase puede tener una implementación. En Java, puedes aportar el mismo mecanismo, pero sólo una de las clases pueden tener una implementación, de forma que los problemas de C++ no ocurren con Java cuando combines varios interfaces:



En una clase derivada, no estás obligado a tener una clase base que sea tanto una clase **abstracta** o "concreta" (una sin métodos **abstract**). Si ***no*** heredas de un **interface** , pudes heredar sólo de uno. El resto de elementos base deben ser **Interfaces** . Colocas todos los nombres de interfaces tras la palabra reservada **implements** y separadas por comas. Puedes tener tantos **interfaces** como quieras-cada uno será un tipo diferente para lanzar hacia arriba. El siguiente ejemplo muestra una clase concreta combinada con varios **interfaces** para producir una nueva clase:

```
//: c08:Adventure.java

// Multiple interfaces.
import java.util.*;

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 {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  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
  }
} ///:~
```

Puedes ver que **Hero** combina la clase concreta **ActionCharacter** con los interfaces **CanFight, CanSwim,** y **CanFly** . Cuando combinas una clase concreta con interfaces de esta manera, la clase concreta debe aparecer primero, después los interfaces. (El compilador da un error si no lo haces así.)

Advierte que la signatura para **fight()** es la misma en el **interface CanFicht** y en la clase **ActionCharacter** , y que **fight()** ***no*** viene con una definicion en **Hero** . La regla para un **interface** es que puedes heredar de el (como veras pronto), pero entonces tendras otro **interface** . Si quieres crear un objeto del nuevo tipo, debe ser una clase que proporcione todas las definiciones. Incluso aunque **Hero** no proporciona explicitamente una definicion de **fight()** , la definición viene más adelante con **ActionCharacter** aunque es automáticamente proporcionada y es posible crear objetos de **Hero** .

En la clase **Adventure** , puedes ver que hay cuatro metodos que toman como argumentos los distintos interfaces y la clase concreta. Cuando un objeto **Hero** es creado, puede ser pasado a cualquiera de estos metodos, lo cual significa que esta siendo upcast a cada **interface** en turno. Debido a la forma en que los interfaces son diseñados en Java, funciona sin impedimento y sin ningun esfuerzo en particular por parte del programador.

No olvides que la razon principal de los interfaces esta mostrada en el ejemplo anterior: ser capaz de upcast a mas de un tipo tipo base. Sin embargo, una segunda razon para usar interfaces es lo mismo que usar una clase base **abstracta** : evitar al programador cliente hacer un objeto de esta clase y

establecer que es solo un interface. Esto trae una cuestion: ¿Debes usar un **interface** o una clase **abstracta** ? Un **interface** te da los beneficios de una clase **abstracta *y*** los beneficios de un **interface** , luego es posible si quieres crear tu clase base sin ninguna definicion de metodos o variables miembro usar **interfaces** a clases **abstractas** . De hecho, si sabes que algo va a ser una clase base, la primera eleccion sera hacer un **interface** , y solo si estas forzado a tener definiciones de metodos o variables miembro cambiarias aun a clase **abstracta** , o si es necesario a una clase concreta.

Colisiones de nombres al combinar interfaces

Puedes encontrarte con un pequeño problema cuando implementas multiples interfaces. En el ejemplo anterior, tanto **CanFight** como **ActionCharacter** tienen un metodo identico, **void fight()** . Esto no es un problema porque el método sea identico en ambos casos, pero y ¿si no lo es? Aqui tienes un ejemplo:

```
//: c08: InterfaceCollision.java
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C" 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 {
       // Identico, sin problema:
       public int f() { return 1; }
}

// Metodos difieren solo por el tipo de retorno:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

La dificultad ocurre porque la sobreescritura, implementacion, y sobrecarga mezcladas juntas es desagradable, y las funciones sobrecargadas no pueden diferir solo por el tipo de retorno. Cuando las dos ultimas lineas son descomentadas, los mensajes de error dicen:

```
InterfaceCollision.java:23: f() en C no puede
implementar f() en I1; intento de usar
un tipo de retorno incompatible
encontrado: int
requerido: void
InterfaceCollision.java:24: interfaces I3 y I1 son
incompatibles; ambos definen f
(), pero con diferentes tipos de devolucion
```

Usando los mismos nombres de metodo en interfaces diferentes que se intentan combinar generalmenteoriginan confusion en la legibilidad del codigo, ademas. Esfuerzate por evitarlo

Extender un interface con herencia

Puedes facilmente añadir declaraciones de nuevos métodos a un **interface** usando herencia, y puedes tambien combinar varios **interfaces** hacia un nuevo **interface** con herencia. En ambos casos obtienes un nuevo **interface** , como se ve en este ejemplo:

```
//: c08: HorrorShow.java
// Extendiendo un interface con 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 HorrorShow {
```

```
        static void u(Monster b) { b.menace(); }
        static void v(DangerousMonster d) {
                d.menace();
                d.destroy();
        }
        public static void main (String[] args) {
                DragonZilla if2 = new DragonZilla();
                u(if2);
                v(if2);
        }
} ///:~
```

DangerousMonster es una extension simple a **Monster** que produce un nuevo **interface** . Este es implementado en **DragonZilla** .

La sintaxis usada en **Vampire** funciona ***sólo*** cuando hereda interfaces. Normalmente, puedes usar **extends** con una unica clase, pero desde un **interface** puede ser hecho con varios interfaces, **extends** se puede referir a multiples interfaces base cuando construyes un nuevo **interface** . Como puedes ver, los nombres de **interfaces** estan simplemente separados por comas.

Agrupando constantes

Como cualquier campo que pongas en un **interface** es automaticamente **static** y **final** , el **interface** es una herramienta conveniente para crear grupos de valores constantes, tantos com harías con un **enum** en C o C++. Por ejemplo:

```
//: c08:Months.java
// Usando interfaces para crear grupos de constantes.
package c08;

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;
 } ///:~
```

Notar el estilo Java de usar todas las letras mayúsculas (con subrayados para separar varias palabras en un único identificador) para **static final** s que tienen inicializadores constantes.

Los campos en un **interface** son automáticamente **public** , por lo que no es necesario especificarlo.

Ahora puedes usar las constantes desde fuera del paquete importando **c08.*** o **c08.Months** tal como harías con cualquier paquete, y referenciar los valores con expresiones como **Months.JANUARY** . Por supuesto, lo que obtienes es un **int** solo, por lo tanto no hay el tipo extra de seguridad que **enum** de C++ tiene, pero esta técnica (normalmente usada) es en realidad una mejora sobre la copia en sí de números en tu programa. (A esta aproximación se la conoce a menudo como el uso de "números mágicos" y produce código muy difícil de mantener.)

Si quieres una seguridad extra de tipo, puedes construir una clase como ésta[1] :

```
//: c08:Month2.java
// Un sistema de enumeración más robusto
package c08;

public final class Month2 {
	private String name;
	private Month2 (String nm){ name = nm; }
	public String toString() { return name; }
	public final static Month2
		JAN = new Month2("January"),
		FEB = new Month2("February"),
		MAR = new Month2("March"),
		APR = new Month2("April"),
		MAY = new Month2("May"),
		JUN = new Month2("June"),
		JUL = new Month2("July"),
		AUG = new Month2("August"),
		SEP = new Month2("September"),
		OCT = new Month2("October"),
		NOV = new Month2("November"),
		DEC = new Month2("December");
	public final static Month2[] month = {
		JAN, JAN, FEB, MAR, APR, MAY, JUN,
		JUL, AUG, SEP, OCT, NOV, DEC
	;
	public static void main(String[] args) {
		Month2 m = Month2.JAN;
		System.out.println(m);
		m = Month2.month[12];
```

1 Esta aproximación estuvo inspirada por un e-mail de Rich Hoffarth.

```
System.out.println(m);
            System.out.println(m == Month2.DEC);
            System.out.println(m.equals(Month2.DEC));
      }
} ///:~
```

La clase es llamada **Month2** , ya que hay ya una **Month** en la biblioteca estandar de Java. Es una clase **final** con un constructor **privado** luego nadie puede heredar de ella o hacer instancias de ella. Las unicas instancias son las **final static** creadas en la misma clase: **JAN, FEB, MAR** , etc. Estos objetos tambien son usados en el array **month** , el cual te deja elegir meses por el numero en vez de por el nombre. (Notar el extra **JAN** en el array para proporcionar un desplazamiento de uno, asi Diciembre es el mes 12.) En **main()** puedes ver la seguridad de tipo: **m** es un objeto **Month2** de forma que solo puede ser asignada a un **Month2** . El ejemplo previo **Months.java** proporciona solo valores **int** , asÃ que una variable **int** que intente representar un mes podria ser dado por cualquier valor entero, lo cual no era muy seguro.

Esta realización también te permite usar == o **equals()** indistintamente, como se muestra al final del **main().**

Inicializando campos en interfaces

Los campos definidos en interfaces son automáticamente **static** y **final** . Estos no pueden ser "finales vacios", pero pueden ser inicializados con expresiones no constantes. Por ejemplo:

```
//: c08:RandVals.java
      // Inicializar campos en interfaces con
      // inicializadores no-constantes.
      import java.util.*;

      public interface RandVals {
            int rint = (int) (Math.random() * 10);
            long rlong = (long) (Math.random() * 10);
            float rfloat = (float) Math.random() * 10);
            double rdouble = Math.random() * 10;
      } ///:~
```

Como los campos son **static** , son inicializados cuando la clase es cargada por primera vez, lo cual ocurre cuando cualquiera de los campos son accedidos por primera vez. Aqui tiene una simple prueba:

```
//: c08:TestRandVals.java

	public class TestRandVals {
		public static void main (String[] args) {
			System.out.println(RandVals.rint);
			System.out.println(RandVals.rlong);
			System.out.println(RandVals.rfloat);
			System.out.println(RandVals.rdouble);
		}
	}	///:~
```

Los campos, por supuesto, no son parte del interface sino que son almacenados en el area de almacenamiento **static** del interface.

Interfaces anidados

[2]Los interfaces puden estar anidados dentro de clases o fuera de otros interfaces. Esto revela un numero de hechos muy interesantes:

```
//: c08:NestingInterfaces.java
class A {
	interface B {
		void f();
	}
	public class BImp implements B {
		public void f() {}
	}
	private class Bmp2 implements B {
		public void f() {}
	}
	public interface C {
		void f();
	}
```

2 Gracias a Martin Danner por hacer esta pregunta en un seminario.

```
class Cimp implements C {
              public void f() {}
       }
       private class CImp2 implements C {
              public void f() {}
       }
       private interface D {
              void f();
       }
       private 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();
       }
}

interface E {
       interface G {
              void f();
       }
       // "public" redundante:
       public interface H {
              void f();
       }
       void g();
       // No puede ser privado dentro de un interface:
       //! 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 un interface privado excepto
       // dentro de esa clase de definicion de interface:
       //! class DImp implements A.D {
       //!    public void f() {}
       //! }
       class EImp implements E {
```

```
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
    class EImp2 implements E {
        public void g() {}
        class EG implements E.G {
            public void f() {}
        }
    }
    public static void main(String[] args) {
        A a = new A();
        // No puede acceder a A.D:
        //! A.D ad = a.getD();
        // No devuelve nada salvo A.D:
        //! A.DImp2 di2 = a.getD();
        // No puede acceder a miembros del interface:
        //! a.getD().f();
        // Solo otro A puede hacer algo con getD():
        A a2 = new A();
        a2.receiveD(a.getD());
    }
}   ///:~
```

La sintaxis para anidar un interface dentro de una clase es razonablemente obvia, y solo como interfaces no anidados pueden tener visibilidad **public** o "amigo". Puedes observar que los interfaces anidados **public** o "amigo" pueden ser implementados por clases anidadas **public** , "amigas", y **private** .

Como una nueva vuelta de tuerca, los interfaces pueden ser también **private** como se ha visto en **A.D** (la misma sintaxis cualificada es usada para interfaces anidadas como para clases anidadas).

¿Que tiene de bueno un interface anidado **privado** ? Podrías pensar que sólo puede ser implementado por una clase anidada **privada** com en **DImp** , pero **A.DImp2** muestra que también puede ser implementada por una clase **publica** . Sin embargo, **A.DImp2** solo puede ser usada por ella misma. No se te permite mencionar el hecho de que implementa el interface **private** , luego implementar un interface **privado** es una manera de forzar la definicion de los metodos en ese interfae sin añadir ningun tipo de informacion (es decir, sin permitir ningun upcasting).

El metodo **getD()** produce una incertidumbre adicional respecto al interface **privado** : es un metodo **publico** que devuelve una referencia a un interface **privado** . ¿Que puedes hacer con el valor de retorno de este metodo? En **main()** puedes ver varios intentos de usar el valor de retorno, todos los cuales fallan. Lo unico que funciona es si el valor de retorno es asignado a un objeto que tenga

permiso para usarlo - en este caso, otra **A** , a través del metodo **received()** .

El interface **E** muestra que los interfaces pueden ser anidados dentro de otros. Sin embargo, las reglas sobre interfaces - en particular, que todos los elementos de un interface deben ser **publicos** - son estrictamente forzadas aqui, de forma que un interface anidado dentro de otro interface es automaticamente **public** y no pude ser **private** .

NestingInterfaces muestra las distintas formas en que los interfaces anidados pueden ser implementados. En particular, nota que cuando tu implementas un interface, no se te pied implementar cualquier interface anidado dentro de el. Tambien los interfaces **private** no pueden ser implementados fuera de sus clases de definicion.

Inicialmente, estos formas pueden parecer que son añadidos estrictamente para la consistencia sintactica, pero generalmente encuentro que una vez que conoces una forma, a menudo descubres lugares donde es util.

Clases internas

Es posible colocar una definicion de clase dentro de otra definicion de clase. Esta se llama ***clase interna.*** La clase interna es una forma a tener en cuenta porque te permite agrupar clases que pertenecen logicamente juntas y controlar la visibilidad de una dentro de la otra. Sin embargo, es importante comprender que las clases internas son diferentes desde la composicion.

A menudo, mientras estas aprendiendo sobre ellas, la necesidad de clases internas no es obvia inmediatamente. AL final de esta seccion, despues de que toda la sintaxis y semantica de clases internas hayan sido descritas, encontraras ejemplos que te aclararan los beneficios de las clases internas.

Tu creas una clase interna tal como esperarías - colocando la definicion de la clase dentro de una clase circundante:

```
//: c08:Parcel1.java
// Creando clases internas.

public class Parcel1 {
	class Contents {
		private int i = l1;
		public int value() { return i;}
	}
	class Destination {
		private String label;
		Destination(String whereTo) {
			label = whereTo;
		}
```

```
		String readLabel() { return label; }
	}
	// Usar clases internas parece como
	// usar cualquier otra clase, dentro de Parcel1:
	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("Tanzania");
	}
}	///:~
```

Las clases internas, cuando se usan dentro de **ship()** , parece como el uso de cualquier otra clase. Aqui, la unica diferencia practica es que los nombres estan anidados dentro de **Parcel1** . Verás en un momento que esta no es la unica diferencia.

Mas tipicamente, una clase externa tendra un metodo que devuelva una referencia a una clase interna, como esta:

```
//: c08:Parcel2.java
// Devolviendo una referencia a una clase interna.

public class Parcel2 {
	class Contents {
		private int i = l1;
		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 cont() {
		return new Contents();
	}
	public void ship(String dest) {
		Contents c = cont();
		Destination d = to (dest);
```

```
                System.out.println(d.readLabel());
        }
        public static void main (String[] args) {
                Parcel2 p = new Parcel2();
                p.ship("Tanzania");
                Parcel2 q = new PArcel2();
                //Definiendo referencias a clases internas:
                Parcel2.Contents c = q.cont();
                Parcel2.Destination d = q.to("Borneo");
        }
}       ///:~
```

Si quieres hacer un objeto de la clase interna en cualquier lugar excepto dentro de un metodo no- **static** de la clase externa, debes especificar el tipo de ese objeto como ***NombreClaseExterna.NombreClaseInterna*** como has visto en el **main ().**

Clases internas y upcasting

Hasta aqui, las clases internas no parecen tan dramaticas. Despues de todo, si se te ha ocultado antes, Java ya tiene un mecanismo de ocultacion perfectamente bueno - solo permite a la clase ser "amiga" (visible solo dentro del paquete) mas que crearla com una clase interna.

Sin embargo, las clases internas tienen su sentido cuando comienzas un upcasting a una clase base, y en particular a un **interface** . (El efecto de producir una referencia de interface desde un objeto que implementa es esencialmente lo mismo que upcasting a una clase base.) Esto es porque la clase interna - la implementacion del **interface** - puede puede estar oculto y no disponible a nadie, lo cual es conveniente para ocultar la implementacion. Todo lo que obtendras es una referencia a la clase base o **interface** .

Primero, los interfaces comunes seran definidos en sus propios ficheros para que puedan ser usados en todos los ejemplos:

```
//: c08:Destination.java
public interface Destination {
        String readLabel();
}       ///:~

//: c08:Contents.java
public interface Contents {
        int value();
}       ///:~
```

Ahora **Contents** y **Destination** representan interfaces disponibles al programador cliente. (El **interface** , recuerda, hace automaticamente todos sus miembros **publicos** .)

Cuando obtienes una referencia a la clase base o al **interface** , es posible que no puedas encontrar el tipo exacto, como se muestra aqui:

```
//: c08:Parcel3.java
// Devolviendo una referencia a una clase interna.
public class Parcel3 {
	private class PContents implements Contents {
		private int i = l1;
		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 dest (String s) {
		return new PDestination(s);
	}
	public Contents cont() {
		return new PContents();

	}
}

class Test {
	public static void main (String[] args) {
		Parcel3 p = new Parcel3();
		Contents c = p.cont();
		Destination d = p.dest("Tanzania");
		// Ilegal -- no se puede acceder a una clase privada:
		//! Parcel3.PContents pc = p.new PContents();
	}
}	///:~
```

Notar que como **main()** esta en **Test** , cuando quieras correr este programa no ejecutas **Parcel3** , sino:

```
java Test
```

En el ejemplo, **main()** debe estar en una clase aparte para poder demostrar la privacidad de la clase interna **PContents** .

En **Parcel3** , algo nuevo se ha añadido: la clase interna **PContents** es **private** de forma que nadie excepto **Parcel3** tiene acceso a ella. **PDestination** es **protected** , de forma que nadie salvo **Parcel3** , las clases del paquete de **Parcel3** (dado que **protected** también da acceso al paquete - es decir, **protected** tambien es "amigo"), y los herederos de **Parcel3** pueden acceder a **PDestination** . Esto significa que el programador cliente ha restringido el conocimiento y acceso a estos miembros. De hecho, puedes downcast a una clase interna **private** (o a una clase interna **protected** a menos que seas un heredero), porque no puedes acceder al nombre, como puedes ver en la **clase Test** . A pesar de todo, la clase interna **private** proporciona una forma al diseñador de clases para prevenir completamente cualquier dependencia de codificacion de tipo y ocultar completamente detalles sobre la implementacion. En suma, la extension de un **interface** es menos util desde la perspectiva del programador cliente ya que no puede acceder a cualquier metodo adiccional que no sea parte de la clase de **interface publico** . Esto proporciona una oportunidad al compilador Java de generar codigomas eficiente.

Las clases normales (no internas) no pueden ser **privadas** o **protegidas** - solo **publicas** o "amigas".

Clases internas en metodos y ambitos de visibilidad

Lo visto hasta ahora es el uso tipico de las clases internas. En general, el codigo que escribiras y leeras sobre clases internas serán clases internas "planas" que son simples y faciles de comprender. Sin embargo, el diseño de clases internas es bastante completo y hay un numero de otras formas, mas oscuras, de usarlas si lo deseas: clases internas pueden ser creadas dentro de metodos o incluso en un ambito arbitrario. Hay dos razones para hacer esto:

1. Como se vio anteriormente, estas implementando un interface de algun tipo que tu puedas crear y devolver una referencia.
2. Estas resolviendo un problema complicado y quieres crear una clase para ayudar en tu solucion, pero no quieres que este disponible publicamente.

En los siguientes ejemplos, el codigo anterior sera modificado para usar:

1. Un clase definida dentro de un metodo
2. Una clase definida dentro de un ambito en un metodo
3. Una clase anonima implementando un interface
4. Una clase anonima extendiendo una clase sin un constructor por defecto
5. Una clase anonima que realiza la inicializacion de campos

6. Una clase anonima que realiza la construccion usando inicializacion de instancias (clases anonimas internas no pueden tener constructores)

Aunque es una clase normal con implementacion, **Wrapping** tambien esta siendo usada como un "interface" comun para sus clases derivadas:

```
//: c08:Wrapping.java
      public class Wrapping {
            private int i;
            public Wrapping(int i) { i = x; }
            public int value() { return i; }
      }     ///:~
```

Habras notada arriba que **Wrapping** tiene un constructor que necesita un argumento, para hacer las cosas un poco mas interesantes.

El primer ejemplo muestra la creacion de una clase entera dentro del ambito de un metodo (en lugar del ambito de otra clase):

```
//: c08:Parcel4.java
      // Anidando una clase dentro de un metodo.

      public class PArcel4 {
            public Destination dest(String s) {
                  class PDestination implements Destination {
                        private String label;
                        private PDestination(String whereTo) {
                              label = whereTo;
                        }
                        public String readLabel() { return label; }
                  }
                  return new PDestination(s);
            }
            public static void main(String[] args) {
                  Parcel4 p = new PArcel4();
                  Destination d = p.dest("Tanzania");
            }
      }     ///:~
```

La clase **PDestination** es parte de **dest()** mas que ser parte de **Parcel4** . (Date cuenta que tambien podrias usar el identificador de clase **PDestination** para una clase interna dentro de cada clase en el mismo subdirectorio sin un nombre conflictivo.) Sin embargo, **PDestination** no puede ser accedido desde fuera de **dest()** . Nota el upcasting que ocurre en la sentencia return -- nada sale de **dest**

() excepto una referencia a **Destination** , la clase base. Por supuesto, el hecho de que el nombre de la clase **PDestination** este colocada dentro de **dest()** no significa que **PDestination** no sea un objeto valido una vez que regresa **dest()** .

El siguiente ejemplo te muestra como puedes anidar una clase interna dentro de cualquier ambito arbitrario:

```
//: c08:Parcel5.java
       // Anidando una clase dentro de un ambito.

       public class Parcel5 {
              private void internalTracking(boolean b) {
                     if(b) {
                            class TarckingSlip {
                                   private String id;
                                   TarckingSlip(String s) {
                                          id=s;
                                   }
                                   String getSlip() { return id; }
                            }
                            TarckingSlip ts = new TarckingSlip("slip");
                            String s = ts.getSlip();
                     }
                     // Â¿No se puede usar aqui! Fuera del ambito:
                     //! TrackingSlip  ts = new TarckingSlip("x");
              }
              public void track() { internalTarcking(true); }
              public static void main(String[] args) {
                     Parcel5 p = new Parcel5();
                     p.track();
              }
       }      ///:~
```

La clase **TrackingSlip** esta anidada dentro del ambito de una sentencia **if** . Esto no significa que la clase este creada condicionalmente - se compila con todo lo demas. Sin embargo, no esta disponible fuera del ambito en el cual esta definido. Aparte de eso, parece una clase ordinaria mas.

Clases internas anonimas

El siguiente ejemplo parece un poco extraño:

```
//: c08:Parcel6.java
       // UN metodo que devuelve una clase interna anonima.
```

```
public class PArcel6 {
	public Contents cont() {
		return new Contents() {
			private int i = l1;
			public int value() { return i; }
		}; // Necesario el punto y coma en este caso
	}
	public static void main(String[] args) {
		Parcel6 p = new PArcel6();
		Contents c = p.cont();
	}
}	///:~
```

¡El metodo **cont()** combina la creacion del valor de retorno con la definicion de la clase que representa ese valor de retorno! En suma, la clase es anonima - no tiene nombre. Para hacel las cosas un poco peor, parece que estas empezando a crear un objeto **Contents** :

```
return new Contents()
```

Pero entonces, antes del punto y coma, dices, "Pero espera, creo que me metere en una definicion de clase":

```
return new Contents() {
	private int i = l1;
	public int value() { return i; }
};
```

Lo que esta extraña sintaxis quiere decir es: " Crea un objeto de una clase anonima que es heredado de **Contents** ." La referencia devuelta por la expresion **new** es automaticamente upcast a una referencia **Contents** . La sintaxis de la clase interna anonima e sun atajo para:

```
class MyContents implements Contents {
	private int i = l1;
	public int value() { return i; }
}
return new MyContents();
```

En la clase anonima interna, **Contents** es creada usando un constructor por

defecto. El codigo siguiente muestra que hacer si tu clase base necesita un constructor con un argumento:

```
//: c08:Parcel7.java
	// Una clase anonima interna que llama
	// al constructor de la clase base.

	public class Parcel7 {
		public Wrapping wrap(int x) {
			// Llamada al constructor base:
			return new Wrapping(x) {
				public int value() {
					return super.value() * 47;
				}
			};	// Punto y coma necesario
		}
		public static void main(String[] args) {
			Parcel7 p = new PArcel7();
			Wrapping w = p.wrap(10);
		}
	}	///:~
```

Es decir, simplemente pasas el argumento apropiado al constructor de la clase base, visto aqui como la **x** es pasada en **new Wrapping(x)** . Una clase anonima no puede tener un constructor donde tu normalmente llamarías a **super()** .

En los dos ejemplos previos, el punto y coma no marca el fin del cuerpo de la clase (como pasa en C++). En su lugar, marca el fin de la expresion que contiene la clase anonima. Así, es identico al uso del punto y coma en cualquier otro sitio.

¿ Que ocurre si necesitas realizar algun tipo de inicializacion para un objeto de una clase anonima interna? Como es anonima, no hay ningun nombre para dar al constructor - luego no puedes tener un constructor. Puedes, sin embargo, realizar la inicialización en el punto de definicion de tus campos:

```
//: c08:Parcel8.java
// Una clase anonima interna que realiza
// la inicializacion. Una version mas breve
// de Parcel5.java.

public class Parcel8 {
	// El argumento debe ser final para usarlo dentro
	// de la clase interna anonima:
	public Destination dest(final String dest) {
		return new Destination() {
```

```
                private String label = dest;
                public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Destination d = p.dest("Tanzania");
    }
} ///:~
```

Si estas definiendo una clase interna anonima y quieres usar un objeto que esta definido fuera de la clase anonima interna, el compilador necesita que el objeto externo sea **final** . Por esto el argumento de **dest()** es **final** . Si lo olvidas, tendrás un mensaje de error en tiempo de compilacion.

Mientras simplemente asignes un campo, la realizacion de antes es correcta. Pero ¿que pasa si necesitas realizar alguna actividad como un constructor? Con una ***instancia de inicialización*** , puedes, en efecto, crear un constructor para una clase anonima interna:

```
//: c08:Parcel9.java
// Usando "instancia de inicializacion" para realizar
// la construccion de una clase anonima interna.

public class Parcel9 {
    public Destination dest(final String dest, final float price) {
        return new Destination() {
            private int const;
            // Instancia de inicializacion 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) {
        Parcel9 p = new Parcel9();
        Destination d = p.dest("Tanzania", 101.395F);
    }
} ///:~
```

Dentro de la instancia inicializadora puedes ver codigo que no seria ejecutado como parte de un inicializador de campo (es decir, la sentencia **if**). Asi en efecto,

un inicializador de instancia es el constructor para una clase anonima interna. Por supuesto, esta limitado; no pueds sobrecargar inicializadores de instancia por lo que puedes tener solo uno de estos constructores.

El enlace a la clase externa

Hasta aqui, parece que las clases internas son solo una ocultacion de nombre y esquema para organizar el codigo, lo cual es util pero no totalmente forzado. Sin embargo, hay otra vuelta. Cuando creas una clase interna, un objeto de esa clase interna tiene un enlace al objeto circundante que llo hizo, y asi puede acceder a los miembros de ese objeto circundante - **sin** ninguna cualificacion especial. En suma, las clases internas tienen derechos de acceso a todos los elementos de la clase circundante[3]. El siguiente ejemplo demuestra esto:

```
//: c08:Sequence.java
// Mantiene una secuencia de objetos.

interface Selector {
	boolean end();
	Object current();
	void next();
}

public class Sequence {
	private Object[] obs;
	private int next = 0;
	public Sequence (int size) {
		obs = new Object[size];
	}
	public void add(Object x) {
		if (next  <obs.length) {
			obs[next] = x;
			next++;
		}
	}
	private class SSelector implements Selector {
		int i = 0;
		public boolean end() {
			return i == obs.length;
		}
		public Object current() {
			return obs[i];
		}
		public void next() {
			if (i <obs.length) i++;
		}
	}
```

[3]Esto es muy diferente del diseño de ***clases anidadas*** en C++, lo cual es simplemente un mecanismo de ocultacion de nombre. No hay enlace a un objeto circundante ni permisos implicados en C++.

```
public Selector getSelector() {
       return new SSelector();
}
public static void main(String[] args) {
       Sequence s = new Sequence(10);
       for (int i=0;i< 10; i++)
              s.add(Integer.toString(i));
       Selector s1 = s.getSelector();
       while (!s1.end()) {
              System.out.println(s1.current());
              s1.next();
       }
}}     ///:~
```

Sequence es simplemente un array de tamaño fijo de **Object** con una clase envolviéndola. Llamas a **add()** para añadir un nuevo **Object** al final de la secuencia (mientras haya espacio). Para ordenar cada uno de los objetos en una **Secuence** , hay una interfaz llamada **Selector** , la cual te permite ver si estás en **end()** , para ver el **ucrrent() Object** , o para desplazarte al **next() Object** en la **Secuence** . Porque **Selector** es una **interface** , otras clases pueden implementarla la **interface** como un argumento, con la finalidad de crear un código genérico.

Aquí, **SSelector** es una clase **privada** que provee a **Selector** de funcionabilidad. En el **main()** , puedes observar la creación de una **Secuence** , seguida por la adición de un número de objetos tipo **String** . Después, se produce un **Selector** con la llamada a **getSelector()** y esto se usa para moverse a través de **Secuence** y seleccionar cada item

Al principio, la creación de **SSlector** parecía tan sólo otra clase interna. Pero examinémosla más detenidamente. Advierte que cada uno de los métodos **end()** , **current()** y **next()** hacen referencia a **obs** , el cual es una referencia que no forma parte de **SSelector** , pero es, sin embargo, un atributo **private** en la clase circundante. Como siempre, la clase interna puede acceder a métodos y atributos desde las clases circundantes, así como poseerlos. Esto se nos muestra muy efectivo, como puedes ver en el ejemplo anterior.

Por lo anteriormente comentado, una clase interna tiene acceso automático a los miembros de una clase circundantes. ¿Cómo ocurre esto?. La clase interna debe guardar una referencia al objeto particular de la clase circundante que era la responsable de crearlo. Cuando haces referencia a un miembro de la clase circundante, esa referencia (oculta) es usada para seleccionar ese miembro. Afortunadamente, el compilador tiene cuidado de todos esos detalles por tí, pero puedes comprender ahora que un objeto de una clase interna puede ser creado sólo en asociación con un objeto de la clase circundante. La construcción de un objeto de la clase interna requiere la referencia al objeto de la clase circundante y el compilador determinará si no puedes acceder a esa referencia. Muchas veces,

esto ocureee sin ninguna interverción por parte del programador.

static clases internas

Si no necesitas una conexión entre objetos de la clase interna y de clases externas, entonces puedes hacer la clase interna **static** . Para entonder el significado de **static** aplicadas a clases internas, debes recordar que el objeto de una clase interna normal guarda una referencia al objeto de la clase circundada que se ha creado. Esto no es verdad, como siempre, cuando llamas a una clase interna es **static** . Una clase interna **static** implica:

1. No necesitas objetos de clases externas para crear un objeto de una clase interna **static** .
2. No puedes acceder a un objeto de una clase externa desde un objeto de una clase interna **static** .

Existen diferencias notables entre clases internas **static** y no **static** . Los atributos y métodos de una clase interna no **static** sólo pueden estar al nivel externo de una clase, también clases internas no **static** no pueden tener datos **estáticos** , campos **estáticos** ni clases internas **estáticas** . Como siempre, las clases internas **static** pueden estar?? de la siguiente forma:

```
//: c08:Parcel10.java
// Clases internas estáticas.

public class Parcel10 {
       private static class PContents
       implements Contents {
              private int i = 11;
              public int value() { return i; }
}

protected static class PDestination
          implements Destination {
       private String label;
       private PDestination(String wereTo) {
              label = wereTo;
       }
       public String readLabel() { return label; }
       // Clases internas estáticas pueden contener
       // otros elementos estáticos:
       public static void f() {}
       static int x = 10;
       static class AnotherLevel {
              public static void f() {}
              static int x = 10;
```

```
            }
      }
      public static Destination dest(String s) {
            return new PDestination(s);
      }
      public static Contents cont() {
            return new PContents();
      }
      public static void main(String[] args) {
            Contents c = cont();
            Destination d = dest("Tanzania");
      }
} ///:~
```

En el **main()** , no se necesita ningún objeto de **Parcel10** ; en su lugar se usa una sintaxis normal para seleccionar un miembro **static** para llamar a métodos que devuelven referencias a **Contents** y **Destination** .

Como verás muy pronto, en una clase interna ordinaria (no **static**), el enlace la objeto de la clase externa es tratado con una referencia especial: **this** . Una clase interna **static** no tiene esta especial referencia **this** , la cual la hace análoga a un método **static** .

Normalmente, no puede coner ningún código dentro de una **interface** . Desde que una clase interna es **estática** no viola la reglas para interfases - la clase interna **static** es sólo situada dentro del espacio del nombre de la interfase:

```
//: c08:IInterface.java
// Clases internas estáticas dentro
// de interfases.
interface IInterface {
      static class Inner {
            int i, j, k;
            public Inner() {}
            void f() {}
      }
} ///:~
```

Al comienzo del libro, sugerí poner un méotodo **main()** en cada clase como método para testearla fácilmente. Un inconveniente es la cantidad de código extra compilado que debe ser portado. Si esto es un problema, puedes usa una clase interna **static** para colgar tu código de prueba:

```
//: c08:TestBed.java
```

```
// Putting test code in a static inner class.
class TestBed {
	TestBed() {}
	void f() { System.out.println("f()"); }
	public static class Tester {
		public static void main(String[] args) {
		TestBed t = new TestBed();
		t.f();
		}
	}
} ///:~
```

Esto genera una clase separada que se llama **TestBed$Tester** (para ejecutar el programa, deben llamar **>java TestBed$Tester**). Puedes usar esta clase para testear, pero no necesitas incluirla entre los archivos de tu producto.

Referenciando a objetos de clases externas

Si necesitas producir una referencia a un objeto de una clase externa, llamas a la clase externa seguida por un punto y **this** . Por ejemplo, in la clase **Secuence.SSelector** , cualquiera de sus métodos puede devolver la referencia almacenada a la clase externa **Secuence** diciendo **Sequence.this** . La referencia resultante tiene automáticamente el tipo correcto. (Esto es sabido y chequeado en tiempo de compilación, por lo que no hay que molestarse en comprobarlo en tiempo de ejecución.).

A veces, quieres instar a otro objeto para crear un objeto de una de sus clases internas. Para hacer esto, debes proveer la referencia a otro objeto de una clase externa con la expreción **new** , tal como se muestra a continuación:

```
//: c08:Parcel11.java
// Creando instancias clases internas.
public class Parcel11 {
	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) {
		Parcel11 p = new Parcel11();
```

```
		// Debes usar una instancia de
		// una clase externa para crear
		// una instancia de la clase interna:
		Parcel11.Contents c = p.new Contents();
		Parcel11.Destination d =
		p.new Destination("Tanzania");
	}
} ///:~
```

Para crear un objeto de la clase interna directamente, no sigas la misma forma y referencia a la clase externa **Parcel11** como podrías esperar, en su lugar debes usar un *objeto* de la clase externa para hacer un objeto de la clase interna:

```
Parcel11.Contents c = p.new Contents();
```

Lógicamente, no es posible crear un objeto de una clase interna hasta que no hayas creados un objeto de la clase externa. Esto es así porque el objeto de la clase interna está estrechamente conectado al objeto de la clase externa. Como siempre, si haces una clase interna **static** , no necesita una referencia a un objeto de la clase externa.

[3]No importa.

```
//: c08:MultiNestingAccess.java
// Clases encadenadas pueden acceder
// a todos los miembros de cualquier
// nivel de las clases en las que están encadenadas.
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();
        }
} ///:~
```

[3]Gracias de nuevo a Martin Danner.

Puedes observar que en **MNA.A.B** los métodos **f()** y **f()** se pueden llamar sin ningún problema (aunque son **private**). Este ejemplo nos demuestra la sintáxis necesaria para crear objetos de múltiples clases internas encadenadas cuano creas objetos en clases diferentes. La sintáxis "**.new** " prudce el correcto ámbito para que no tengas que cualificar?? el nombre de la clase en la llamada al constructor.

Heredando de clases internas.

Debido a que el constructor de la clase interna debe incluir una referencia al objeto de la clase circundante, cosas que son puntillosas cuando heredas desde una clase interna. El problema es que la referencia "secreta" al objeto de la clase circundante *debe* ser iniializada y que no haya todavía un objeto por defecto al que adjuntarla en la clase derivada. La respuesta es usar una sintáxis específica para crear las asociaciones explícitas:.

```
//: c08:InheritInner.java
// Heredando de una clase interna
class WithInner {
       class Inner {}
       }
public class InheritInner
         extends WithInner.Inner {
       //! InheritInner() {} // ¡No compila!
       InheritInner(WithInner wi) {
              wi.super();
       }
       public static void main(String[] args) {
              WithInner wi = new WithInner();
              InheritInner ii = new InheritInner(wi);
       }
} ///:~
```

Puedebes ver que **InheritInner** extiende sólo la clase interna, no su clase externa. Pero cuando tarda en crear un constructor, el designador por defecto no es bueno y no puedes pasar una referencia a un objeto circundante. Como añadido, debes usar la siguiente sintáxis:.

```
enclosingClassReference.super();
```

dentro del constructor. Esto provee la referencia necesaria y el programa entonces compilará.

¿Pueden ser sobreescritas las clases internas?

¿Qué ocurre cuando creas una clase interna e intentas heredad de la clase circundante y redefinir la clase interna? Es decir, ¿es posible sobreescribir una clase interna?. Esto para que podría ser un concepto muy potente, pero "sobreescribir" una clase interna es como si fuera otro método de la clase externa que no hiciera nada:

```
//: c08:BigEgg.java
// Una clase interna no puede ser
// sobreescrita como un método.
class Egg {
	protected class Yolk {
		public Yolk() {
			System.out.println("Egg.Yolk()");
		}
	}
	private Yolk y;
	public Egg() {
		System.out.println("New Egg()");
		y = new Yolk();
	}
}
public class BigEgg extends Egg {
	public class Yolk {
		public Yolk() {
		System.out.println("BigEgg.Yolk()");
	}
}
public static void main(String[] args) {
	new BigEgg();
	}
} ///:~
```

El constructor por defecto es sintetizado automáticamente por el compilador y su llamada al constructor por defecto de la clase base. Podrías pensar que desde que **BigEgg** es creado, la versión "sobreescrita" de **Yolk** prodría ser usada, pero no es este el caso. La salida es:

```
New Egg()
Egg.Yolk()
```

Este ejemplo simplemente te muestra que no hay ninguna clase interna mágica

estra cuando heredas desde la clase externa. Las dos clases internas son entidades completamente diferentes, cuada cual en su especio. Como siempre, todavía es posible explicar herencia desde la clase interna:

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.
class Egg2 {
	protected class Yolk {
		public Yolk() {
		   System.out.println("Egg2.Yolk()");
		}
		public void f() {
		   System.out.println("Egg2.Yolk.f()");
		}
	}
	private Yolk y = new Yolk();
	public Egg2() {
		System.out.println("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() {
		   System.out.println("BigEgg2.Yolk()");
		}
		public void f() {
		   System.out.println("BigEgg2.Yolk.f()");
		}
	}
	public BigEgg2() { insertYolk(new Yolk()); }
	public static void main(String[] args) {
		Egg2 e2 = new BigEgg2();
		e2.g();
	}
} ///:~
```

Ahora **BigEgg2.Yolk** explicitamente es **extends Egg2.Yolk** y sobreescribe sus métodos. El método **insertYolk()** permite a **BigEgg2** hacer un upcast de sus propios objetos tipo **Yolk** dentro de la referencia **y** en **Egg2** , así cuando **g()** llama a **y.f()** la versión sobreescrita de **f()** es usada. La salida es la siguiente:

```
Egg2.Yolk()
New Egg2()
```

```
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

La segunda llamada a **Egg2.Yolk()** está llamando al constructor de la clase base que llama al constructor de **BigEgg2.Yolk** . Puedes ver que la versión sobreescrita de **f()** es usada cuando **g()** es llamada.

Identificadores de las clases internas

Desde que cada clase produce un archivo **.class** que contiene toda la información sobre cómo crear objetos de ese tipo (esta información prouce "meta-clases" llamadas objetos **Class**), podrías esperar que las clases internas debieran también producir archivos **.class** que contengan la información de *sus* objetos **Class** . Los nombres de estos archivos/clases tienen una forma estricta: el nombre de la clase circundante seguida de '$' seguida por el nombre de la clase interna. Por ejemplo, los archivos **.class** creados por **InheritInner.java** son:

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

Si las clases internas son "anónimas", el compilador simplemente comenzará a generar números como identificados de las clases internas. Si las clases internas están encadenadas con una clase internas, sus nombres simplemente serán añadidos después del '$' y del/ de los identificador/es de las clases externas.

Aunque este esquema de generación de nombres internos es simple y potente, es también robusto y de fácil manejo en la mayoría de las situaciones[3]. Desde que esto es la forma estándar de nombrar esquemas para Java, los archivos generados son automáticamente independiente de la plataforma. (Observa que el compilador Java está cambiando tus clases internas para conseguir que funcionen.)

---

[3]Por otro lado, '$' es un meta-carácter para el shell de Unix y algunas veces habrá problemas al listar los archivos **.class** . Esto es una confrontación de Sun hacia compañías basadas en Unix. Mi opinión es que no están considerando este tema, pero en vez de pensar en ello, deberías fijarte en los archivos fuente.

¿Porqué clases internas?

Llegados a este punto, has visto una gran cantidad de sintaxis y semántica describiéndote la manera como las clases internas funcionan, pero esto no responde a la pregunta de por qué existen. ¿Porqué Sun se complicó la existencia añadiendo esta característica fundamental al lenguaje?.

Normalmente, la clase interna es heredada de una clase o implementación de **interface** , y el código en la clase interna manipula el objeto creado en la clase externa. De esta manera, puedes decir que una clase interna privee un tipo de ventana dentro de la clase externa.

Una pregunta que afecta a las clases internas es esta: si necesito una referencia a una **interface** , ¿porqué no la creo símplemente en la clase externa?. La respuesta es "Si eso es todo lo que necesitas, qué esperas a hacerlo". Pero ¿qué es lo que diferencia una clase interna implementando una **interface** de una clase externa implementando la misma **interface** ?. La respuesta es que no puedes tener siempre lo necesario para **interfaces** - a veces trabajas con implementaciones. Así, la más completa razón para clases internas es:

Cada clase interna puede heredar independientemente de una implementación. Por tanto, una clase interna no está limitado como lo está la clase externa heredando de una implementación.

Sin la habilidad de las clases internas de proveer de herencia -en esencia- desde más de una clase en concreto o **abstracta** , alguno diseños y problemas de programación sería intratables. Por consiguiente, una forma de observar la clase interna es como la conclusión de la solución de los problemas de herencia múltiple. Las interfaces resuelven parte del problema, pero las clases internas realmente permiten "implementar múltiple herencia". Es decir, las clases internas realmente permiten que heredes desde más de una no- **interface** .

Para observar esto con más detenimiento, consideremos una situación donde tienes dos interfaces que deben ser implementadas en una clase como sea. Debido a la flexibilidad de las interfaces, tienes dos opciones: una clase simple o una clase interna:

```
//: c08:MultiInterfaces.java
// Dos maneras para que una clase
// pueda implementar interfaces múltiples.
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());
	}
} ///:~
```

De acuerdo, esto asume que la estructura de tu código realiza un uso lógico de cualquier manera. Como siempre, tendrás algún tipo de guía por la naturaleza del problema sobre si debes usar una clase simple o una clase interna. Pero sin otras restricciones, en el ejemplo anterior la aproximación que haces no se diferencia mucho de una implementación "standpoint". Ambos funcionan.

Como siempre, si tienes clases concretas o **abstractas** en vez de **interfaces** , estás fuermente limitado a usar clases internas si tu clase debe como sea implementar a ambas anteriores.

```
//: c08:MultiImplementation.java
// Con clases concretas o abstractas,
// clases internas son el único modo
// de producir el efecto de
// "implementar múltiple herencia"
class C {}
abstract class D {}
class Z extends C {
	D makeD() { return new D() {}; }
}
public class MultiImplementation {
	static void takesC(C c) {}
	static void takesD(D d) {}
	public static void main(String[] args) {
		Z z = new Z();
		takesC(z);
		takesD(z.makeD());
	}
} ///:~
```

Si no necesitaras resolver problemas de "implementar múltiple herencia", podrías

concebir código sin necesitar para nada clases internas. Pero con clases internas, tienes estas características adicionales:.

1. Las clases internas pueden tener múltiples instancias, cada cual con su información independiente de la información del objeto de la clase externa.
2. En cada clase externa, puedes tener muchas clases internas, cada cual implemente la misma **interface** o heredar de la misma clase de diferentes formas. Un ejemplo de esto será mostrado dentro de poco.
3. El punto de creación de objeto de una clase interna no está atado a la creación de un objeto de la clase externa.
4. No hay confusión potencial en una relación "es-un" con la clase interna: es una entidad separada.

Como ejemplo, si **Sequence.java** no usara clases internas, tendrías que decir "una **Secuence** es un **Selector** ", y sólo serías capaz de tener un **Selector** en existencia para un **Sequence** en particular. También, puedes tener un segundo método, **getRSelector()** , que produce un **Selector** que se desplaza en dirección inversa a la secuencia. Este tipo de flexibilidad es sólo disponible con clases internas.

Closures & Callbacks

Un *closure* es un objeto que se puede llamar y que retiene información del ámbito en el cual fue creado. A partir de esta definición, puedes observar que una clase interna es un "closure" orientado a objetos, porque no sólo contiene cada una de las piezas de información sobre el objeto de la clase externa ("el ámbito en el cual él fue creado"), pues automáticamente porta una referencia sobre la totalidad del objeto de la clase externa, donde tiene permisos para manipular todos los miembros, y, a veces, los **privados** .

Uno de los más obligados argumentos hechos para incluir algún tipo de mecanismo de puntero en Java fue permitir *callbacks* . Con una "callback", a algunos objetos se les da una porción de información que les permite ¿¿llamar a algún último punto en el objeto orginal??. Este es un poderoso concepto, como verás en los capítulos 12 y 16. Si una "callback" es implementada usando un puntero, como siempre, deberás confiar en el programador para proceder y no desperdiciar el puntero. Como has visto hasta ahora, Java tiende a ser más cuidadoso que eso, por lo que los punteros no fueron incluidos en el lenguaje.

El "closure" provehído por la clase interna es una perfecta solución; más flexibilidad y más seguridad que un puntero. Aquí hay un ejemplo sencillo:

```
//: c08:Callbacks.java
// Usando clases internas para "callbacks"
```

```
interface Incrementable {
	void increment();
}
// Implementación muy simple de la interface:
class Callee1 implements Incrementable {
	private int i = 0;
	public void increment() {
		i++;
		System.out.println(i);
	}
}
class MyIncrement {
	public void increment() {
		System.out.println("Other operation");
	}
	public static void f(MyIncrement mi) {
		mi.increment();
	}
}
// Si tu clase debe implementar increment() de
// otra manera, debes usar una clase interna:
class Callee2 extends MyIncrement {
	private int i = 0;
	private void incr() {
		i++;
		System.out.println(i);
	}
	private class Closure implements Incrementable {
		public void increment() { incr(); }
	}
	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();
        }
} ///:~
```

Este ejemplo también provee un adelanto de la diferencia entre implementar una interface en una clase externa vx. hacerlo en una clase interna. **Callee1** es claramente la solución más simple en términos de código. **Callee2** hereda de **MyIncrement** que ha tiene un método **increment()** diferente, el cual no hace nada relacionado con lo esperado por la interface **Incrementable** . Cuando **MyIncrement** es heredado en **Calle2** , **increment()** no puede ser sobreescrita para usar por **Incrementable** , así eres forzado a proveer una implementación separada usando una clase interna. También observa que cuando creas una clase interna no añades ni modificas la inteface de la clase externa.

Date cuenta de que todo, excepto **getCallbackReference()** , en **Callee2** es **privado** . Para permitir *cualquier* conexión con el mundo exterior, la **interface Incrementable** es esencial. Aquí puedes ver cómo las **interfaces** permiten una separación completa de interface e implementación.

La clase interna **Closure** simplemente implementa **Incrementable** para proveer un puente de retorno hacia **Callee2** - pero seguro. Todo aquel que utilice la referencia **Incrementable** puede, de acuerdo, sólo llamar **increment()** y no tiene otras habilidades (diferente a un puntero, el cual te permitiría ejecutarlo salvajemente).

El valor de una "callback" no reside en su flexibilidad - puedes decidir dinámicamente qué funciones debeas llamar en tiempo de ejecución. El beneficio de esto será más evidente en el capítulo 13, donde "callback" son usados en todas partes para implementar la funcionalidad de interfaces gráficas de usuario (IGU).

.

Clases internas & "control frameworks"

Un ejemplo más concreto del uso de clases internas puede ser hallado en algo a lo que me referiré aquí como *control framework* .

Una *aplicación framework* es una clase o consujunto de ellas que son diseñadas para resolver un tipo determinado de problema. Para aplicar una aplicación framework, heredas desde una o más clases y sobreescribes algunos métodos. El código que escribes en los métodos sobreescritos customiza la solución general proveída por la aplicación framework en orden para resolver tu problema específico. El control framework es un tipo particular de aplicación framework dominado por la necesidad de responder a ventos; un sistema que primeramente

responde a un evento es denominado un *sistema event-driven* . Uno de los más importantes problemas en programación de aplicaciones es la interface gráfica de usuario (IGU), la cual es completamente event-driven. Como verás en el capítulo 13, la librería Java Swing es una control framework que resuelve elegantemente el problema IGU y que usa clases internas a duras penas.

Para ver cómo las clases internas permiten la fácil creación y uso del control frameworks, considera un control framework cuyo trabajo sea ejecutar eventos cuando sea, los cuales estén "preparados". Donde "preparados" pueda significar cualquier cosa, en este caso por defecto será basado en un reloj. Lo que sigue es un control framework que contenga ninguna información específica sobre qué está controlando. Primero, aquí está la interface que describe cualquier control de eventos. Es una clase **abstract** en vez de una **interface** actual porque por defecto su comportamiento es manejar el control basado en el tiempo, así alguna de la implementación puede ser incluida aquí:

```
//: c08:controller:Event.java
// Los métodos comunes para cualquier control de eventos
package c08.controller;

abstract public class Event {
       private long evtTime;
       public Event(long eventTime) {
          evtTime = eventTime;
       }
       public boolean ready() {
          return System.currentTimeMillis() >= evtTime;
       }
       abstract public void action();
       abstract public String description();
} ///:~
```

El constructor simplemente captura el tiempo cuando quiere que **Event** se ejecute, mientras, **ready()** te indica cuándo es la hora de ejecutarlo. De acuerdo, **ready()** podría ser sobreescrito en una clase derivada para basar el **Event** en otra cosa que no fuera tiempo.

action() es un método que es llamado cuando el **Event** está **ready()** , y **description()** devuelve información en forma de texto sobre el **Event** .

El siguiente archivo contiene el actuales control framework que maneja y dispara eventos. La primera clase es sólo una clase "ayudante" cuyo trabajo es colgar objetos tipo **Event** . Puedes sustituirla por cualquier contenedor apropiados, y en el capítulo 9 descubrirás otros contenedores que te ayudarán sin necesidad de escribir este código extra:

```
//: c08:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c08.controller;
// This is just a way to hold Event objects.
class EventSet {
   private Event[] events = new Event[100];
   private int index = 0;
   private int next = 0;
   public void add(Event e) {
       if(index >= events.length)
          return; // (In real life, throw exception)
       events[index++] = e;
   }
   public Event getNext() {
       boolean looped = false;
       int start = next;
       do {
          next = (next + 1) % events.length;
          // See if it has looped to the beginning:
          if(start == next) looped = true;
          // If it loops past start, the list
          // is empty:
          if((next == (start + 1) % events.length)
                  && looped)
              return null;
       } while(events[next] == null);
       return events[next];
   }
   public void removeCurrent() {
       events[next] = null;
   }
}
public class Controller {
   private EventSet es = new EventSet();
   public void addEvent(Event c) { es.add(c); }
   public void run() {
       Event e;
       while((e = es.getNext()) != null) {
          if(e.ready()) {
              e.action();
              System.out.println(e.description());
              es.removeCurrent();
          }
       }
   }
} ///:~
```

EvenSet arbitrariamente cuelga 100 **Eventos** . (Si un contenedor "real" del capítulo 9 es usado aquí, no necesitas preocuparte sobre su tamaño máximo, puesto que él se redimensionará por sí mismo). El **index** es usado para localizar el registro del próximo espacio disponible, y **next** es usado cuando estás buscando el próximo **Event** en la lista, para ver si estás alrededor. Esto es importante durante una llamada a **getNetxt()** , porque los objetos **Event** son eliminados de la lista (usando **removeCurrent()**) una vez se están ejecutando, así **getNext()** encontrará agujeros en las lista mientras se desplaza por ella.

Observa que **removeCurrent()** no sólo pone banderas indicadoras de que el objeto no se usará más. En su lugar, pone la referencia a **null** . Esto importante porque si el recolector de basura ve una referencia que todavía se está uso no podrá limpiar el objeto. Si pienzas que tus referencia podrían pender sobre ellas (como están aquí), entonces es mejor idea referenciarlos a **null** que permitir al recolector de basura limpiarlos.

Controller es donde se desarrolla el trabajo actual. Él usa un **EventSet** para colgar su objetos tipo **Event** y **addEvent()** te permite añadir nuevos evento a esta lista. Pero el método importante es **run()** . Este método reincide en **EventSet** , rastreando un objeto tipo **Event** que esté **ready()** para ejecutarse. Por cada uno que encuentre **ready()** , llama al método **action()** , imprime la **description()** y después elimina el **Event** de la lista.

Advierte, sin ir más lejos, que en este diseño no sabes nada acerca de *qué* hace exactamente **Event** . Y esto es lo realmente importante del diseño; como "separas las cosas que cambias de las cosas que permanecen iguales". O, usando mis términos, el "vector de cambio" es las diferentes acciones de los variados tipos de objetos **Event** y como expresas diferentes acciones creando diferentes subclases de **Event** .

Aquí es donde las clases internas entran en juego. Permiten dos cosas:

1. Para crear una implementación completa de una aplicación control framework en una sola clase, en relación con la encapsulación de todo en una única sobre implementación. Las clases internas son usadas como medio de expresión de diferentes tipos de **action()** necesarios para resolver el problema. Como añadido, el siguiente ejemplo usa **private** clases internas, así la implementación es completamente oculta y puede ser cambiada con impunidad.
2. Las clases internas guardan su implementación de miradas obscenas, de este modo, eres capaz de acceder fácilmente a cualquiera de los miembros en la clase externa. Sin esta habilidad, el código sería lo bastante desagradable como para comenzar a buscar una alternativa..

Considera una implementación particular de un control framework diseñado para controlar las funciones de un invernadero[3]. Cada acción es completamente diferente: encender y apagar luces, agua y termostatos, hacer sonar timpres y restaurar el sistema. Pero el control framework es diseñado para aislar fácilmente

los diferentes códigos. Las clases internas te permiten tener múltiples versiones derivadas de la misma clase base, **Event** , en una sola clase. Para cada tipo de acción tu heredas una nueva clase interna tipo **Event** , y escribess el código de control dentro de **action()** .

Como ejemplo típico de una aplicación framework, la clase **GreenHouseControls** es heredada de **Controller** :

```
//: c08:GreenhouseControls.java
// Esto pudce una aplicación específica del
// sistema de control, todas en una sola clase.
// Las clases internas te permiten encapsular
// diferentes funcionalidades para cada tipo de evento.
import c08.controller.*;
public class GreenhouseControls
extends Controller {
   private boolean light = false;
   private boolean water = false;
   private String thermostat = "Day";
   private class LightOn extends Event {
       public LightOn(long eventTime) {
          super(eventTime);
       }
       public void action() {
          // Pon el código de control de harware aquí
          // para encender las luces.
          light = true;
       }
       public String description() {
          return "Light is on";
       }
   }
   private class LightOff extends Event {
       public LightOff(long eventTime) {
          super(eventTime);
       }
       public void action() {
          // Pon el código de control de harware aquí
          // para apagar las luces.
          light = false;
       }
       public String description() {
          return "Light is off";
       }
   }
   private class WaterOn extends Event {
       public WaterOn(long eventTime) {
          super(eventTime);
```

```
    }
    public void action() {
       // Pon el código de control de harware aquí
       water = true;
    }
    public String description() {
       return "Greenhouse water is on";
    }
}
private class WaterOff extends Event {
    public WaterOff(long eventTime) {
       super(eventTime);
    }
    public void action() {
       // Pon el código de control de harware aquí
       water = false;
    }
    public String description() {
       return "Greenhouse water is off";
    }
}
private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
       super(eventTime);
    }
    public void action() {
       // Pon el código de control de harware aquí
       thermostat = "Night";
    }
    public String description() {
       return "Thermostat on night setting";
    }
}
private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
       super(eventTime);
    }
    public void action() {
       // Pon el código de control de harware aquí
       thermostat = "Day";
    }
    public String description() {
       return "Thermostat on day setting";
    }
}
// Un ejemplo de action() que inserta uno
// nuevo de él mismo dentro de la lista event
private int rings;
private class Bell extends Event {
    public Bell(long eventTime) {
```

```
            super(eventTime);
        }
        public void action() {
            // Ring cada 2 segundos, veces 'rings':
            System.out.println("Bing!");
            if(--rings > 0)
                addEvent(new Bell(
                System.currentTimeMillis() + 2000));
        }
        public String description() {
            return "Ring bell";
        }
    }
    private class Restart extends Event {
        public Restart(long eventTime) {
            super(eventTime);
        }
        public void action() {
            long tm = System.currentTimeMillis();
            // En vez de hard-wiring, puedes parse
            // información de configuración desde
            // un archivo de texto aquí:
            rings = 5;
            addEvent(new ThermostatNight(tm));
            addEvent(new LightOn(tm + 1000));
            addEvent(new LightOff(tm + 2000));
            addEvent(new WaterOn(tm + 3000));
            addEvent(new WaterOff(tm + 8000));
            addEvent(new Bell(tm + 9000));
            addEvent(new ThermostatDay(tm + 10000));
            // ¡Puedes includo añadir un objeto Restart!
            addEvent(new Restart(tm + 20000));
        }
        public String description() {
            return "Restarting system";
        }
    }
    public static void main(String[] args) {
        GreenhouseControls gc =
            new GreenhouseControls();
        long tm = System.currentTimeMillis();
        gc.addEvent(gc.new Restart(tm));
        gc.run();
    }
} ///:~
```

Observa que **light** , **water** , **thermostat** y **rings** todas antes de la clase externa **GreenhouseControls** , la clase interna puede acedera esos campos sin cualificaciones o permisos especiales. También, muchos de los métodos **action()**

invocan algunas partes del control de hardware, el cual debería invocar a código no-Java.

Muchas de las clases **Event** parecen similares, pero **Bell** y **Restart** son especiales. **Bell** suena y, si no has tenido suficiente, añade un nuevo objeto **Bell** a la lista de eventos, así sonará otra vés después. Date cuenta de cómo las clases internas *casi* parecen herencia múltipe: **Bell** tiene todos los métodos de **Event** y ésta parece tener todos los métodos de la clase externa **GreenHouseControls** .

Restart es responsable de la inicialización del sistema, así añade todos los eventos apropiados. De acuerdo, un camibo más efecto para cuplir esto es evitar codificar "a pelo" los eventos y en su lugar leerlos de un archivo. (Un ejercicio en el capítulo 11 te pregunta cómo modificar este ejemplo para hacerlo). Como **Restart** es otro objeto **Event** , tu puedes también añadir un objeto **Restart** con **Restart.action()** para que el sistema se reinicie a sí mismo. Y todo lo que necesitas hacer en **main()** es crear un objeto **GreenhouseControls** y añadir un objeto **Restart** para que funcione.

Este ejemplo debería indicarte el modo de pareciar el valor de las clases internas, especialmente cuando las uses con control framework. Como siempre, en el capítulo 13 verás cómo se pueden usar las clases internas para realizar acciones de una interface gráfica de usuario (IGU). Una vez que acabes ese capítulo, estarás completamente convencido.

Resumen

Las interfaces y clases internas son conceptos más sofisticados que los que puedas encontrar en muchos lenguajes POO. Por ejemplo, no hay nada parecido en C++. Al mismo tiempo, solucionan el mismo problema que C++ intenta resolver con su característica de herencia múltime (HM). Como siempre, HM en C++ se vuelve tediosa de utilizar, mienbras las interfaces y clases internas de Java son, por comparación, muchos más accesibles.

Aunque las características por sí mismas son razonablemente potentes, el uso de esas características es una cuestión de diseño, al igual que el polimorfismo. Más adelante, mejorarás en el reconocimiento de situaciones donde debes usar una interface, una clase interna o ambas. Pero en este momento en este libro, debes, al menos, sentirte confortable con la sintatxis y la semántica. Como ves, utilizarás de estas características del lenguaje eventualmente.

Ejercicios

1. Demuestra que los campos en una **interface** son implicitamente **static** y **final** .
2. Crea una **interface** que contenga tres métodos en su propio **package** . Implementa la interface en un **paquete** diferente.

3. Demuestra que todos los métodos en una **interface** son automáticamente **public** .
4. En **c07:Sandwhich.java** , crea una interface llamada **FastFood** (con sus métodos apropiados) y cambia **Sandwitch** para que también implemente **FastFood** .
5. Crea tres **interfaces** , cada cual con dos métodos. Hereda una nueva **interface** de las tres, añadiendo un nuevo método. Crea una clase implementando la nueva **interface** y también hereda de una clase concreta. Ahora escribe cuatro métodos, cada uno de los cuales toma una de las cuatro **interfaces** como un argumento. En el **main()** , crea un objeto de tu calse y pásale cada uno de los métodos.
6. Modifica el ejercicio 5 creando una clase **abstract** y heredándola en la clase derivada.
7. Modifica **Music5.java** añadiendo una **interface** llamada **Playable** . Elimina la declaración **play()** de **Instrument** . Añade **Playable** a las clases derivadas incluyéndola en la lista de **implements** . Cambia **tune()** para que tome un **Playable** en vez de un **Instrument** .
8. Cambia el ejercicio 6 del capítulo 7 para que **Rodent** sea una **interface** .
9. En **Adventure.java** , añade una **interface** llamada **CanClimb** , a continuación de la inicialización de las otras interfaces.
10. Escribe un programa que importe y use **Month2.java** .
11. Siguiendo el ejemplo dado en **Month2.java** , crea una enumeración de los días de la semana.
12. Crea una **interface** con al menos un método, en su propio paquete. Crea una clase en un paquete diferente. Añade una clase interna **protected** que implemente la **interface** . En un tercer paquete, hereda de tu clase y, dentro de un método, devuelve un objeto de la clase interna **protected** , haciendo upcasting a la **interface** durante la devolución.
13. Crea una **interface** con al menos un método, que implemente la **interface** por definición de una clase interna dentro de un método, la cual devuelve una referencia a tu **interface** .
14. Repite el ejercicio 13 pero definiendo la clase interna con un ámbito con un método.
15. Repite el ejercicio 13 usando una clase interna anónima.
16. Crea una clase interna **private** que implemente una **public interface** . Escribe un método que devuelva una referencia a una instancia de la clase interna **private** , haz un upcasting hacia la **interface** . Demuestra que la clase interna está completamente oculta intentando hacerle un downcast.
17. Crea una clase con un constructor que no sea por defecto y sin constructor por defecto. Crea una segunda clase que tenga un método que devuelva una referencia a la primera clae. Crea dicho objeto haciendo una clase interna anónima que herede de la primera clase.
18. Crea una clase con un campo **private** y un método **private** . Crea una clase interna con un método que modifique el campo de la clase externa y llame al método de la clase externa. En un segundo método de la clase externa, crea un objeto de la clase interna y llama a su método, después, muestra el efecto que tiene sobre el objeto de la clase externa.
19. Repite el ejercicio 18 usando una clase interna anónima.
20. Crea una clase que contenga una clase interna **static** . En el **main()** , crea una instancia

de la clase interna.

21. Crea una **interface** que contenga una clase interna **static** y crea una instancia de la clase interna.
22. Crea una clase que contenga una clase interna que, a su vez, contenga una clase interna. Repite esto usando clases internas **static** . Observa el nombre los archivos **.class** producidos por el compilador.
23. Crea una clase con una clase interna. En una clase separada, haz una instancia a la clase interna.
24. Crea una clase con una clase interna que tenga un constructor que no sea el por defecto. Crea una segunda clase con una clase interna que herede de la primera clase interna.
25. Repara el problema en **WindError.java** .
26. Modifica **Secuence.java** añadiendo un método **getRSelector()** que poduzca una diferente implementación de la **interface Selector** que desplace en sentido contrario la secuencia, es decir, desde el fin hacia el principio.
27. Crea una **interface U** con tres métodos. Crea una clase **A** con un método que poduzca una referencia a **U** construyendo una clase interna anónima. Crea una segunda clase **B** que contenga un array de **U** . **B** debería tener un método que aceptara y almacenara una referencia a **U** en el array, un segundo miembro que pusiera una referencia en el array (especificado por el argumento del método) a **null** y un tercer método que se desplazara por el array y llamara a los métodos en **U** . En el **main()** , crea un grupo de objetos de **A** y uno sólo de **B** . Rellena el **B** con referencias **U** proucidas por los objetos **A** . Usa **B** para llamar a los objetos de **A** . Elimina algunas de las referencia de **U** que existen en **B** .
28. En **GreenhouseControls.java** , añade la clase interna **Event** para que encienda o apague los ventiladores.
29. Observa que una clase interna tiene acceso a los elementos **privados** de su clase externa. Determina si la condición inversa es verdadera.

9: El control de errores con Excepciones

La filosofía básica de Java es que "el código mal escrito no será ejecutado."

El momento ideal para capturar un error es en tiempo de compilación, antes incluso de intentar ejecutar el programa. Sin embargo, no todos los errores se pueden detectar en tiempo de compilación. El resto de los problemas deberán ser manejados en tiempo de ejecución, mediante alguna formalidad que permita a la fuente del error pasar la información adecuada a un receptor que sabrá hacerse cargo de la dificultad de manera adecuada.

En C y en otros lenguajes anteriores, podía haber varias de estas formalidades, que generalmente se establecían por convención y no como parte del lenguaje de programación. Habitualmente, se devolvía un valor especial o se ponía un indicador a uno, y el receptor se suponía, que tras echar un vistazo al valor o al indicador, determinaría la existencia de algún problema. Sin embargo, con el paso de los años, se descubrió que los programadores que hacían uso de bibliotecas tendían a pensar que eran invencibles -como en "Sí, puede que los demás cometan errores, pero no en mi código". Por tanto, y lógicamente, éstos no comprobaban que se dieran condiciones de error (y en ocasiones las condiciones de error eran demasiado estúpidas como para comprobarlas) **[1]**. Si uno *fuera* tan exacto como para comprobar todos los posibles errores cada vez que se invocara a un método, su código se convertiría en una pesadilla ilegible. Los programadores son reacios a admitir la verdad: este enfoque al manejo de errores es una grandísima limitación especialmente de cara a la creación de programas grandes, robustos y fácilmente mantenibles.

[1] El programador de C puede fijar el valor de retorno de **printf()** como un ejemplo de esta afirmación.

La solución es extraer del manejo de errores la naturaleza casual de los mismos y forzar la formalidad. Esto se ha venido haciendo a lo largo de bastante tiempo, dado que las implementaciones de *manejo de excepciones* se retornan hasta los sistemas operativos de los años 60, e incluso al *"error goto"* de *BASIC*. Pero el manejo de excepciones de C++ se basaba en Ada, y el de Java está basado fundamentalmente en C++ (aunque se parece incluso más al de Pascal Orientado a Objetos).

La palabra "*excepción*" se utiliza en el sentido de: "*Yo me encargo de la excepción a eso*". En cuanto se da un problema puede desconocerse qué hacer con el mismo, pero se sabe que simplemente no se puede continuar sin más; hay que parar y alguien, en algún lugar, deberá averiguar qué hacer. Pero

puede que no se disponga de información suficiente en el contexto actual como para solucionar el problema. Por tanto, se pasa el problema a un contexto superior en el que alguien se pueda encargar de tomar la decisión adecuada (algo semejante a una cadena de comandos).

El otro gran beneficio de las excepciones es que limpian el código de manejo de errores. En vez de comprobar si se ha dado un error en concreto y tratar con él en diversas partes del programa, no es necesario comprobar nada más en el momento de invocar al método (puesto que la excepción garantizará que alguien la capture). Y es necesario manejar el problema en un solo lugar, el denominado *gestor de excepciones.* Éste salva el código, y separa el código que describe qué se desea hacer a partir del código en ejecución cuando algo sale mal. En general, la lectura, escritura, y depuración de código se vuelve mucho más sencilla con excepciones, que cuando se hace uso de la antigua manera de gestionar los errores.

Debido a que el manejo de excepciones se ve fortalecido con el compilador de Java, hay numerosísimos ejemplos en este libro que permiten aprender todo lo relativo al manejo de excepciones. Este capítulo presenta el código que es necesario escribir para gestionar adecuadamente las excepciones, y la forma de generar excepciones si algún método se mete en problemas.

Excepciones básicas

Una ***condición excepcional*** es un problema que evita la continuación de un método o el alcance actual. Es importante distinguir una condición excepcional de un problema normal, en el que se tiene la suficiente información en el contexto actual como para hacer frente a la dificultad de alguna manera. Con una condición excepcional no se puede continuar el proceso porque no se tiene la información necesaria para tratar el problema, en el ***contexto actual.*** Todo lo que se puede hacer es salir del contexto actual y relegar el problema a un contexto superior. Esto es lo que ocurre cuando se lanza una excepción.

Un ejemplo sencillo es una división. Si se corre el riesgo de dividir entre cero, merece la pena comprobar y asegurarse de que no se seguirá adelante y se llegará a ejecutar la división. Pero ¿qué significa que el denominador sea cero? Quizás se sabe, en el contexto del problema que se está intentando solucionar en ese método particular, cómo manejar un denominador cero. Pero si se trata de un valor que no se esperaba, no se puede hacer frente a este error, por lo que habrá que lanzar una excepción en vez de continuar hacia delante.

Cuando se lanza una excepción, ocurren varias cosas. En primer lugar, se crea el objeto excepción de la misma forma en que se crea un objeto Java: en el montículo, con **new.** Después, se detiene el cauce normal de ejecución (el que no se podría continuar) y *se* lanza la referencia al objeto excepción desde el contexto actual. En este momento se interpone el mecanismo de gestión de excepciones que busca un lugar apropiado en el que continuar ejecutando el

programa. Este lugar apropiado es el *gestor de excepciones*, cuyo trabajo es recuperarse del problema de forma que el programa pueda, o bien intentarlo de nuevo, o bien simplemente continuar.

Como un ejemplo sencillo de un lanzamiento de una excepción, considérese una referencia denominada **t.** Es posible que se haya recibido una referencia que no se haya inicializado, por lo que *se*ría una buena idea comprobarlo antes de que *se* intentara invocar a un método utilizando esa referencia al objeto. Se puede enviar información sobre el error a un contexto mayor creando un objeto que represente la información y "arrojándolo" fuera del contexto actual. A esto se le llama *lanzamiento de una excepción*. Tiene esta apariencia:

```
if(t == null)
  throw new NullPointerException();
```

Esto lanza una excepción, que permite - en el contexto actual - abdicar la responsabilidad de pensar sobre este aspecto más adelante. Simplemente se gestiona automáticamente en algún otro sitio. El dónde se mostrará más tarde.

Parámetros de las excepciones

Como cualquier otro objeto en Java, las excepciones siempre se crean en el montículo haciendo uso de **new**, que asigna espacio de almacenamiento e invoca a un constructor. Hay dos constructores en todas las excepciones estándar: el primero es el constructor por defecto y el segundo toma un parámetro string de forma que se pueda ubicar la información pertinente en la excepción:

```
throw new NullPointerException("t = null");
```

Este string puede extraerse posteriormente utilizando varios métodos, como se verá más adelante.

La palabra clave **throw** hace que ocurran varias cosas relativamente mágicas. Habitualmente, se usará primero **new** para crear un objeto que represente la condición de error. Se da a **throw** la referencia resultante. En efecto, el método "devuelve" el objeto, incluso aunque ese tipo de objeto no sea el que el método debería devolver de forma natural. Una manera natural de pensar en las excepciones es como si se tratara de un mecanismo de retorno alternativo, aunque el que lleve esta analogía demasiado lejos acabará teniendo problemas. También se puede salir del ámbito ordinario lanzando una excepción. Pero se devuelve un valor, y el método o el ámbito finalizan.

Cualquier semejanza con un método de retorno ordinario acaba aquí, puesto que el punto de retorno es un lugar completamente diferente del punto al que se sale en una llamada normal a un método. (Se acaba en un gestor de excepciones adecuado que podría estar a cientos de kilómetros – es decir,

mucho más bajo dentro de la pila de invocaciones – del punto en que se lanzó la excepción.)

Además, se puede lanzar cualquier tipo de objeto lanzable **Throwable** que se desee. Habitualmente, se lanzará una clase de excepción diferente para cada tipo de error. La información sobre cada error se representa tanto dentro del objeto excepción como implícitamente en el tipo de objeto excepción elegido, puesto que alguien de un contexto superior podría averiguar qué hacer con la excepción. (A menudo, la única información es el tipo de objeto excepción, y no se almacena nada significativo junto con el objeto excepción.)

Capturar una excepción

Si un método lanza una excepción, debe asumir que esa excepción será "capturada" y que será tratada. Una de las ventajas del manejo de excepciones de Java es que te permite concentrarte en el problema que se intenta solucionar en un único sitio, y tratar los errores que ese código genere en otro sitio.

Para ver cómo se captura una excepción, hay que entender primero el concepto de región guardada, que es una sección de código que podría producir excepciones, y que es seguida del código que maneja esas excepciones.

El bloque try

Si uno está dentro de un método y lanza una excepción (o lo hace otro método al que se invoque), ese método acabará en el momento en que haga el lanzamiento. Si no se desea que una **excepción** implique abandonar un método, se puede establecer un bloque especial dentro de ese método para que capture la excepción. A este bloque se le denomina el bloque **try** puesto que en él se "intentan" varias llamadas a métodos. El bloque **try** es un ámbito ordinario, precedido de la palabra clave **try:**

```
try {
  // el código que podría generar excepciones
}
```

Si se estuviera comprobando la existencia de errores minuciosamente en un lenguaje de programación que no soporte manejo de excepciones, habría que rodear cada llamada a método con código de prueba de invocación y errores, incluso cuando el mismo método fuese invocado varias veces. Esto significa que el código es mucho más fácil de escribir y leer debido a que no se confunde el objetivo del código con la comprobación de errores.

Manejadores de excepción

Por supuesto, la excepción que se lance debe acabar en algún sitio. Este "sitio" es el *manejador de excepciones* y hay uno por cada tipo de excepción que se desee capturar. Los manejadores de excepciones siguen inmediatamente al bloque **try** y se identifican por la palabra clave **catch:**

```
try {
  // el código que podría generar excepciones
} catch(Type1 id1) {
  // maneja excepciones de Type1
} catch(Type2 id2) {
  // maneja excepciones de Type2
} catch(Type3 id3) {
  // maneja excepciones de Type3
}

// etc...
```

Cada cláusula **catch** (manejador de excepciones) es semejante a un pequeño método que toma uno y sólo un argumento de un tipo en particular. El identificador **(id1, id2,** y así sucesivamente) puede usarse dentro del manejador, exactamente igual que un parámetro de un método. En ocasiones nunca se usa el identificador porque el tipo de la excepción proporciona la suficiente información como para tratar la excepción, pero el identificador debe seguir ahí.

Los manejadores deben aparecer directamente tras el bloque **try**. Si se lanza una excepción, el mecanismo de gestión de excepciones trata de cazar el primer manejador con un argumento que coincida con el tipo de excepción. Posteriormente, entra en esa cláusula **catch**, y la excepción se da por manejada. La búsqueda de manejadores se detiene una vez que se ha finalizado la cláusula **catch**. Sólo se ejecuta la cláusula **catch**; no es como una sentencia **switch** en la que haya que colocar un **break** después de cada **case** para evitar que se ejecute el resto.

Fíjese que, dentro del bloque **try**, varias llamadas a métodos podrían generar la misma excepción, pero sólo se necesita un manejador.

Terminación vs. Reanudación

Hay dos modelos básicos en la teoría de manejo de excepciones. En la terminación (que es lo que soportan Java y C++) se asume que el error es tan crítico que no hay forma de volver atrás a resolver dónde se dio la excepción. Quien quiera que lanzara la excepción decidió que no había forma de resolver la situación, y no quería volver atrás.

La alternativa es el *reanudación*. Significa que se espera que el manejador de excepciones haga algo para rectificar la situación, y después se vuelve a

ejecutar el método que causó el error, presumiendo que a la segunda no fallará. Desear este segundo caso significa que se sigue pensando que la excepción continuará tras el manejo de la excepción. En este caso, la excepción es más como una llamada a un método - que es como deberían establecerse en Java aquellas situaciones en las que se desea este tipo de comportamiento. (Es decir, es mejor llamar a un método que solucione el problema antes de lanzar una excepción.) Alternativamente, se ubica el bloque **try** dentro de un bucle **while** que sigue intentando volver a entrar en el bloque **try** hasta que se obtenga el resultado satisfactorio.

Históricamente, los programadores que usaban sistemas operativos que soportaban el manejo de excepciones reentrantes acababan usando en su lugar código con terminación. Por tanto, aunque la técnica de los reintentos parezca atractiva a primera vista, no es tan útil en la práctica. La razón dominante es probablemente el acoplamiento resultante: el manejador debe ser, a menudo, consciente de dónde se lanza la excepción y contener el código no genérico específico del lugar de lanzamiento. Esto hace que el código sea difícil de escribir y mantener, especialmente en el caso de sistemas grandes en los que la excepción podría generarse en varios puntos.

Crear sus propias excepciones

No hay ninguna limitación que obligue a utilizar las excepciones existentes en Java. Esto es importante porque a menudo será necesario crear sus propias excepciones para indicar un error especial que puede crear su propia biblioteca, pero que no fue previsto cuando se creó la jerarquía de excepciones de Java.

Para crear su propia clase excepción, se verá obligado a heredar de un tipo de excepción existente, preferentemente uno cercano al significado de su nueva excepción (sin embargo, a menudo esto no es posible). La forma más trivial de crear un nuevo tipo de excepción es simplemente dejar que el compilador cree el constructor por defecto, de forma que prácticamente no haya que escribir ningún código:

```
//: c09:SimpleExceptionDemo.java
// heredando tus excepciones.
import com.bruceeckel.simpletest.*;

class SimpleException extends Exception {}

public class SimpleExceptionDemo {
  private static Test monitor = new Test();
  public void f() throws SimpleException {
    System.out.println("Lanza SimpleException desde f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    SimpleExceptionDemo sed = new SimpleExceptionDemo();
```

```
    try {
      sed.f();
    } catch(SimpleException e) {
      System.err.println("Lo atrapé!");
    }
    monitor.expect(new String[] {
      "Atraviesa SimpleException desde f()",
      "Lo atrapé!"
    });
  }
} ///:~
```

Cuando el compilador crea el constructor por defecto, se trata del que llama automáticamente (y de forma invisible) al constructor por defecto de la clase base. Por supuesto, en este caso no se obtendrá un constructor **SimpleException(String),** pero en la práctica esto no se usa mucho. Como se verá, lo más importante de una excepción es el nombre de la clase, por lo que en la mayoría de ocasiones una excepción como la mostrada arriba es plenamente satisfactoria.

Aquí, se imprime el resultado en la consola de *error estándar* escribiendo en **System.err**. Éste suele ser el mejor sitio para enviar información de error, en vez de **System.out**, que podría estar redirigida. Si se envía la salida a **System.err** no estará redireccionada junto con **System.out**, por lo que el usuario tiene más probabilidades de enterarse.

Crear una clase excepción que tenga también un constructor que tome un **String** como parámetro es bastante sencillo:

```
//: c09:FullConstructors.java
import com.bruceeckel.simpletest.*;

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}

public class FullConstructors {
  private static Test monitor = new Test();
  public static void f() throws MyException {
    System.out.println("Lanzando MyException desde f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Lanzando MyException desde g()");
    throw new MyException("Originado en g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
```

```
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "Lanzando MyException desde f()",
      "MyException",
      "%% \thasta FullConstructors.f\\(.*\\)",
      "%% \thasta FullConstructors.main\\(.*\\)",
      "Lanzando MyException desde g()",
      "MyException: Originado en g()",
      "%% \thasta FullConstructors.g\\(.*\\)",
      "%% \thasta FullConstructors.main\\(.*\\)"
    });
  }
} ///:~
```

El código añadido es poco: - la inserción de dos constructores que definen la forma de crear **MyException**. En el segundo constructor, el constructor de clase base con un argumento **String** es explícitamente invocado usando la palabra clave **super**.

En los manipuladores, uno de los métodos **Throwable** (de cuál **Exception** es heredada) es llamado: **PrintStackTrace**( ). Esto produce información acerca de la secuencia de métodos que fueron llamados para ponerse directos donde la excepción ocurrió. Por defecto, la información va al flujo estándar de error, pero las versiones sobrecargadas te permiten enviar los resultados a cualquier otro flujo igualmente.

El proceso de crear tus excepciones puede ser tomado más allá. puedes agregar constructores y miembros extras:

```
//: c09:ExtraFeatures.java
// Futuro embellecimiento de clases de excepción.
import com.bruceeckel.simpletest.*;

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 "Mensaje Detallado: "+ x + " "+ super.getMessage();
  }
```

```
}

public class ExtraFeatures {
  private static Test monitor = new Test();
  public static void f() throws MyException2 {
    System.out.println("Lanzando MyException2 desde f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println("Lanzando MyException2 desde g()");
    throw new MyException2("Originado en g()");
  }
  public static void h() throws MyException2 {
    System.out.println("Lanzando MyException2 desde h()");
    throw new MyException2("Originado en h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace();
      System.err.println("e.val() = " + e.val());
    }
    monitor.expect(new String[] {
      "Lanzando MyException2 desde f()",
      "MyException2: Mensaje Detallado: 0 null",
      "%% \thasta ExtraFeatures.f\\(.*\\)",
      "%% \thasta ExtraFeatures.main\\(.*\\)",
      "Lanzando MyException2 desde g()",
      "MyException2: Mensaje Detallado: 0 Originado en g()",
      "%% \thasta ExtraFeatures.g\\(.*\\)",
      "%% \thasta ExtraFeatures.main\\(.*\\)",
      "Lanzando MyException2 desde h()",
      "MyException2: Mensaje Detallado: 47 Originado en h()",
      "%% \thasta ExtraFeatures.h\\(.*\\)",
      "%% \thasta ExtraFeatures.main\\(.*\\)",
      "e.val() = 47"
    });
  }
} ///:~
```

Un campo i ha sido añadido, junto con un método que lee ese valor y un constructor adicional que lo coloca. Además, **Throwable.getMessage** ( ) ha

sido sobrescrito para producir un mensaje detallado. **getMessage** ( ) es algo como **toString** ( ) para clases de excepción.

Desde que una excepción es simplemente otra clase de objeto, puedes continuar este proceso de embelleciendo el poder de tus clases de excepción. Mantén en mente, sin embargo, tan todo este juego de disfraces podría ser despistado en los programadores del cliente usando tus paquetes, desde que simplemente podrían buscar la excepción para ser lanzados y nada más. (Es el método más de las excepciones de la biblioteca Java que son usados.)

Especificación de excepción

En Java, se pide que se informe al programador cliente, que llama al método, de las excepciones que podría lanzar ese método. Esto es bastante lógico porque el llamador podrá saber exactamente el código que debe escribir si desea capturar todas las excepciones potenciales. Por supuesto, si está disponible el código fuente, el programador cliente podría simplemente buscar sentencias **throw**, pero a menudo las bibliotecas no vienen con sus fuentes. Para evitar que esto sea un problema, Java proporciona una sintaxis (y *fuerza* el uso de la misma) para permitir decir educadamente al programador cliente qué excepciones lanza ese método, de forma que el programador cliente pueda manejarlas. Ésta es la *especificación de excepciones, y* es parte de la declaración del método, y se sitúa justo después de la lista de parámetros.

La especificación de excepciones utiliza la palabra clave **throws**, seguida de la lista de todos los tipos de excepción potenciales. Por tanto, la definición de un método podría tener la siguiente apariencia:

```
void f() throws TooBig, TooSmall, DivZero { //...
```

Si se dice

```
void f() { // ...
```

significa que el método no lanza excepciones. (*Excepto* las excepciones de tipo **RuntimeException**, que puede ser lanzado razonablemente desde cualquier sitio -como se describirá más adelante.)

No se puede engañar sobre una especificación de excepciones - si un método provoca excepciones y no las maneja, el compilador lo detectará e indicará que, o bien hay que manejar la excepción o bien hay que indicar en la especificación de excepciones todas las excepciones que el método puede lanzar. Al fortalecer las especificaciones de excepciones de arriba abajo, Java garantiza que se puede asegurar la corrección de la excepción en *tiempo de compilación* **[2]**.

[2] Esto constituye una mejora significativa frente al manejo de excepciones de C++, que no captura posibles violaciones de especificaciones de excepciones hasta tiempo de ejecución, donde no es ya muy útil.

Sólo hay un lugar en el que se pude engañar: se puede decir que se lanza una excepción que verdaderamente no se lanza. El compilador cree en tu palabra, y fuerza a los usuarios del método a tratarlo como si verdaderamente arrojara la excepción. Esto tiene un efecto beneficioso al ser un objeto preparado para esa excepción, de forma que, de hecho, se puede empezar a lanzar la excepción más tarde sin que esto requiera modificar el código ya existente. También es importante para la creación de clases base abstractas e interfaces cuyas clases derivadas o implementaciones pueden necesitar lanzar excepciones.

Las excepciones que son comprobadas y obligadas en la fase de compilación son llamadas excepciones comprobadas.

Capturar cualquier excepción

Es posible crear un manejador que capture cualquier tipo de excepción. Esto se hace capturando la excepción de clase base **Exception** (hay otros tipos de excepciones base, pero **Exception** es la clase base a utilizar pertinentemente en todas las actividades de programación):

```
catch(Exception e) {
  System.err.println("Capturé una excepción");
}
```

Esto capturará cualquier excepción, de forma que si se usa, habrá que ponerlo al *final* de la lista de manejadores para evitar que los manejadores de excepciones que puedan venir después queden ignorados.

Dado que la clase **Exception** es la base de todas las clases de excepción que son importantes para el programador, no se logra mucha información específica sobre la excepción, pero se puede llamar a los métodos que vienen de su tipo base **Throwable:**

String getMessage ( )

String getLocalizedMessage ( )

Toma el mensaje de detalle, o un mensaje ajustado a este escenario particular.

String toString ( )

Devuelve una breve descripción del objeto **Throwable**, incluyendo el mensaje de detalle si es que lo hay.

void printStackTrace ( )

void printStackTrace (PrintStream)

void printStackTrace (java.io.PrintWriter)

Imprime el objeto y la traza de pila de llamadas lanzada. La pila de llamadas muestra la secuencia de llamadas al método que condujeron al momento en que se lanzó la excepción. La primera versión imprime en el error estándar, la segunda y la tercera apuntan a un flujo de datos de tu elección (en el Capítulo 12, se entenderá por qué hay dos tipos de flujo de datos).

Throwable fillInStackTrace ( )

Registra información dentro de este objeto **Throwable,** relativa al estado actual de las pilas. Es útil cuando una aplicación está relanzando un error o una excepción (en breve se contará algo más al respecto).

Además, se pueden conseguir otros métodos del tipo base de **Throwable, Object** (que es el tipo base de todos). El que podría venir al dedillo para excepciones es **getClass**() que devuelve un objeto que representa la clase de este objeto. Se puede también preguntar al objeto de esta **Clase** por su nombre haciendo uso de **getName**() o **toString**(). Asimismo se pueden hacer cosas más sofisticadas con objetos **Class** que no son necesarios en el manejo de excepciones. Los objetos **Class** se estudiarán más adelante.

He aquí un ejemplo que muestra el uso de los métodos básicos de **Exception:**

```
//: c09:ExceptionMethods.java
// Demostrando los Métodos de Excepción.
import com.bruceeckel.simpletest.*;

public class ExceptionMethods {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      throw new Exception("Mi Excepcion");
    } catch(Exception e) {
      System.err.println("Capture la Excepcion");
      System.err.println("getMessage():" + e.getMessage());
      System.err.println("getLocalizedMessage():" +
        e.getLocalizedMessage());
      System.err.println("toString():" + e);
      System.err.println("printStackTrace():");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "Capture la Excepcion",
      "getMessage():Mi Excepcion",
      "getLocalizedMessage():Mi Excepcion",
      "toString():java.lang.Exception: Mi Excepcion",
      "printStackTrace():",
```

```
      "java.lang.Exception: Mi Excepcion",
      "%% \thasta ExceptionMethods.main\\(.*\\)"
    });
  }
} ///:~
```

Se puede ver que los métodos proporcionan más información exitosamente - cada una es efectivamente, un superconjunto de la anterior.

Relanzar una excepción

En ocasiones se desea volver a lanzar una excepción que se acaba de capturar, especialmente cuando se usa **Exception** para capturar cualquier excepción. Dado que ya se tiene la referencia a la excepción actual, se puede volver a lanzar esa referencia:

```
catch(Exception e) {
  System.err.println("Una excepción fue lanzada");
  throw e;
}
```

Volver a lanzar una excepción hace que la excepción vaya al contexto inmediatamente más alto de manejadores de excepciones. Cualquier cláusula **catch** subsiguiente del mismo bloque **try** seguirá siendo ignorada. Además, se preserva todo lo relativo al objeto, de forma que el contexto superior que captura el tipo de excepción específico pueda extraer toda la información de ese objeto.

Si simplemente se vuelve a lanzar la excepción actual, la información que se imprime sobre esa excepción en **printStackTrace(**) estará relacionada con el origen de la excepción, no al lugar en el que se volvió a lanzar. Si se desea instalar nueva información de seguimiento de la pila, se puede lograr mediante **fillInStackTrace(**), que devuelve un objeto **Throwable** creado rellenando la información de la pila actual en el antiguo objeto excepción. Éste es su aspecto:

```
//: c09:Rethrowing.java
// Demonstrando fillInStackTrace()
import com.bruceeckel.simpletest.*;

public class Rethrowing {
  private static Test monitor = new Test();
  public static void f() throws Exception {
    System.out.println("originando la excepcion en f()");
    throw new Exception("lanzado desde f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
```

```
      System.err.println("En g(),e.printStackTrace()");
      e.printStackTrace();
      throw e; // 17
      // atraviesa e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String[] args) throws Throwable {
    try {
      g();
    } catch(Exception e) {
      System.err.println(
        "Capturado en main, e.printStackTrace()");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "originando la excepcion en f()",
      "En g(),e.printStackTrace()",
      "java.lang.Exception: lanzado desde f()",
      "%% \thasta Rethrowing.f(.*?)",
      "%% \thasta Rethrowing.g(.*?)",
      "%% \thasta Rethrowing.main(.*?)",
      "Capturado en main, e.printStackTrace()",
      "java.lang.Exception: lanzado desde f()",
      "%% \thasta Rethrowing.f(.*?)",
      "%% \thasta Rethrowing.g(.*?)",
      "%% \thasta Rethrowing.main(.*?)"
    });
  }
} ///:~
```

Los números importantes de la línea son marcados como comentarios. Con la línea 17 no comentada (como mostrado), se muestra la salida, así es que el rastro de la pila de excepción siempre recuerda su punto de origen verdadero no importa cuán muchas veces queda relanzado.

Considerando que la línea 17 sea comentario, y no lo sea la línea 18, se usa **fillInStackTrace**(), siendo el resultado:

```
originando la exception in f()
En g(),e.printStackTrace()
java.lang.Exception: lanzado desde f()
        hasta Rethrowing.f(Rethrowing.java:9)
        hasta Rethrowing.g(Rethrowing.java:12)
        hasta Rethrowing.main(Rethrowing.java:23)
Capturado en main, e.printStackTrace()
java.lang.Exception: lanzado desde f()
        hasta Rethrowing.g(Rethrowing.java:18)
        hasta Rethrowing.main(Rethrowing.java:23)
```

(Agregue complementos adicionales desde el método **Test.expect**( ).) Por **fillInStackTrace**(), la línea 18 se convierte en el punto de origen nuevo de la excepción.

La clase **Throwable** debe aparecer en la especificación de excepción para **g**() y **main**() porque **fillInStackTrace**() produce una referencia a un objeto **Throwable**. Desde que **Throwable** es una clase base de **Exception**, es posible obtener un objeto que es un **Throwable** pero no un **Exception**, así es que el manipulador para **Exception** en **main**() le podría hacer falta. Para estar seguro de que todo está en orden, el compilador fuerza una especificación de excepción para **Throwable**. Por ejemplo, la excepción en el siguiente programa no es capturada en **main**():

```
//: c09:ThrowOut.java
// {ThrowsException}
public class ThrowOut {
  public static void
  main(String[] args) throws Throwable {
    try {
      throw new Throwable();
    } catch(Exception e) {
      System.err.println("Capturado en main()");
    }
  }
} ///:~
```

También es posible volver a lanzar una excepción diferente de la capturada. Si se hace esto, se consigue un efecto similar al usar **fillInStackTrace(**) -la información sobre el origen primero de la excepción se pierde, y lo que queda es la información relativa al siguiente **throw:**

```
//: c09:RethrowNew.java
// Relanza un objeto diferente desde el primero que
// fue capturado.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class OneException extends Exception {
  public OneException(String s) { super(s); }
}

class TwoException extends Exception {
  public TwoException(String s) { super(s); }
}

public class RethrowNew {
  private static Test monitor = new Test();
  public static void f() throws OneException {
    System.out.println("originando la excepcion en f()");
    throw new OneException("lanzado desde f()");
  }
  public static void
```

```
  main(String[] args) throws TwoException {
    try {
      f();
    } catch(OneException e) {
      System.err.println(
        "Capturado en main, e.printStackTrace()");
      e.printStackTrace();
      throw new TwoException("desde main()");
    }
    monitor.expect(new String[] {
      "originando la excepcion en f()",
      "Capturado en main, e.printStackTrace()",
      "OneException: lanzado desde f()",
      "\thasta RethrowNew.f(RethrowNew.java:18)",
      "\thasta RethrowNew.main(RethrowNew.java:22)",
      "Excepcion en hilo \"main\" " +
      "TwoException: desde main()",
      "\thasta RethrowNew.main(RethrowNew.java:28)"
    });
  }
} ///:~
```

La excepción final sólo sabe que proviene de **main**(), y no de **f**().

No hay que preocuparse nunca de limpiar la excepción previa, o cualquier otra excepción en este sentido. Todas son objetos basados en el montículo creados con **new**, por lo que el recolector de basura los limpia automáticamente.

Encadenamiento de excepción

A menudo quieres capturar una excepción y lanzar otro, pero todavía mantiene la información acerca de la excepción originaria – esto es llamado *encadenamiento de excepción*. Antes de JDK 1.4, los programadores tuvieron que escribir su código para conservar la información original de excepción, pero ahora todas las subclases **Throwable** pueden llevar un objeto de causa en su constructor. La causa está dirigida a ser la excepción originaria, y pasando eso en ti mantiene el rastro de la pila de regreso a su origen, aun cuando estás generando y lanzando una excepción nueva en este punto.

Es interesante notificar que las únicas subclases **Throwable** que proveen el argumento de causa en el constructor son las tres clases excepción fundamental **Error** (usadas por el JVM para reportar errores de sistema), **Exception**, y **RuntimeException**. Si quieres concatenar cualquier tipo distintos de excepción, lo haces a través del método **initCause**() en vez del constructor.

Aquí hay un ejemplo que te permite dinámicamente añadirle los campos a un objeto **DynamicFields** en el tiempo de ejecución:

```
//: c09:DynamicFields.java
// Una Clase que dinamicamente agrega campos a si mismo.
// Demuestra encadenamiento de excepcion.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
  private static Test monitor = new Test();
  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() {
    StringBuffer result = new StringBuffer();
    for(int i = 0; i < fields.length; i++) {
      result.append(fields[i][0]);
      result.append(": ");
      result.append(fields[i][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;
      }
    // Sin campos vacios. Agrega 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 reursiva 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) {
      // Muchas excepciones no tienen un constructor de
      // "causa". En estos casos debes usar initCause(),
      // disponibles 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); // Obtiene el valor viejo
    } catch(NoSuchFieldException e) {
      // Usa el constructor que toma la "causa":
      throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
  }
  public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    System.out.println(df);
    try {
      df.setField("d", "A value for d");
      df.setField("number", new Integer(47));
      df.setField("number2", new Integer(48));
      System.out.println(df);
      df.setField("d", "A new value for d");
      df.setField("number3", new Integer(11));
      System.out.println(df);
      System.out.println(df.getField("d"));
      Object field = df.getField("a3"); // Excepcion
    } catch(NoSuchFieldException e) {
      throw new RuntimeException(e);
    } catch(DynamicFieldsException e) {
      throw new RuntimeException(e);
    }
    monitor.expect(new String[] {
      "null: null",
      "null: null",
      "null: null",
```

```
      "",
      "d: Un valor para d",
      "numero: 47",
      "numero 2: 48",
      "",
      "d: Un nuevo valor para d",
      "numero: 47",
      "numero2: 48",
      "numero3: 11",
      "",
      "Un valor para d",
      "Excepcion en hilo \"main\" " +
      "java.lang.RuntimeException: " +
      "java.lang.NoSuchFieldException",
      "\thasta DynamicFields.main(DynamicFields.java:98)",
      "Causado por: java.lang.NoSuchFieldException",
      "\thasta DynamicFields.getFieldNumber(" +
      "DynamicFields.java:37)",
      "\thasta DynamicFields.getField(DynamicFields.java:58)",
      "\thasta DynamicFields.main(DynamicFields.java:96)"
    });
  }
} ///:~
```

Cada objeto **DynamicFields** contiene un arreglo de pares de Objeto-Objeto. El primer objeto es el identificador del campo (un **String**), y el segundo es el valor del campo, lo cual puede ser cualquier tipo excepto un primitivo desenvuelto. Cuando usted crea el objeto, usted hace una suposición educada de aproximadamente cuántos campos a usted le hace falta. Cuando usted llama a **setField**(), este o encuentra el campo existente por ese nombre o crea a uno nuevo, y aporta su valor. Si corre fuera de espacio, suma espacio nuevo creando un arreglo de longitud una más largo y copiando los viejos elementos adentro. Si usted trata de aportar un valor nulo, entonces lanza a un **DynamicFieldsException** creando uno y usando a **initCause**() para insertar a un **NullPointerException** como la causa.

Como un valor de retorno, **setField**() también hace salir el valor viejo en esa posición del campo usando a **getField**(), lo cual podría lanzar a un **NoSuchFieldException**. Si el programador cliente llama a **getField**(), luego son responsables de manipular a **NoSuchFieldException**, pero si esta excepción es lanzado dentro de **setField**(), es un error de programación, así es que el **NoSuchFieldException** es convertido a un **RuntimeException** usando al constructor que toma un el argumento de *causa*.

Excepciones estándar Java

La clase **Throwable** de Java describe todo aquello que se pueda lanzar en forma de excepción. Hay dos tipos generales de objetos **Throwable** ("tipos

de" - "heredados de"). **Error** representa los errores de sistema y de tiempo de compilación que uno no se preocupa de capturar (excepto en casos especiales). **Exception** es el tipo básico que puede lanzarse desde cualquier método de las clases de la biblioteca estándar de Java y desde los métodos que uno elabore, además de incidencias en tiempo de ejecución. Por tanto, el tipo base que más interesa al programador es **Exception**.

La mejor manera de repasar las excepciones es navegar por la documentación HTML de Java que se puede descargar de *jaua.sun.com.* Merece la pena hacer esto simplemente para to mar un contacto inicial con las diversas excepciones, aunque pronto se verá que no hay nada especial entre las distintas excepciones, exceptuando el nombre. Además, Java cada vez tiene más excepciones; básicamente no tiene sentido imprimirlas en un libro. Cualquier biblioteca nueva que se obtenga de un tercero probablemente tendrá también sus propias excepciones. Lo que es importante entender es el concepto y qué es lo que se debe hacer con las excepciones.

La idea básica es que el nombre de la excepción representa el problema que ha sucedido; de hecho se pretende que el nombre de las excepciones sea autoexplicativo. Las excepciones no están todas ellas definidas en **java.lang**; algunas están creadas para dar soporte a otras bibliotecas como **util**, **net** e **io**. Así, por ejemplo, todas las excepciones de E/S se heredan de **java.io.IOException**.

El caso especial de RuntimeException

El primer ejemplo en este capítulo fue

```
if(t == null)
  throw new NullPointerException();
```

Puede ser bastante horroroso pensar que hay que comprobar que cualquier referencia que se pase a un método sea **null** o no **null** (de hecho, no se puede saber si el que llama pasa una referencia válida). Afortunadamente, no hay que hacerlo - esto es parte de las comprobaciones estándares de tiempo de ejecución que hace Java, de forma que si se hiciera una llamada a una referencia null, Java lanzaría automáticamente una **NullPointerException**. Por tanto, el fragmento de código de arriba es totalmente superfluo.

Hay un grupo completo de tipos de excepcio nes en esta categoría. Se trata de excepciones que Java siempre lanza automáticamente, y no hay que incluirlas en las especificaciones de excepciones. Además, están convenientemente agrupadas juntas bajo una única clase base denominada **RuntimeException**, que es un ejemplo perfecto de herencia: establece una familia de tipos que tienen algunas características Y comportamientos en común. Tampoco se escribe nunca una especificación de excepciones diciendo que un método podría lanzar una **RuntimeException**, puesto que se asume. Dado que indican fallos, generalmente una **RuntimeException** nunca se captura – se

maneja automáticamente. Si uno se viera forzado a comprobar las excepciones de tipo **RuntimeException**, su código se volvería farragoso. Incluso aunque generalmente las **RuntimeExceptions** no se capturan, en los paquetes que uno construya se podría decidir lanzar algunas **RuntimeExceptions**.

¿Qué ocurre si estas excepciones no se capturan? Dado que el compilador no fortalece las especificaciones de excepciones en este caso, es bastante verosímil que una **RuntimeException** pudiera filtrar todo el camino hacia el exterior hasta el método **main**() sin ser capturada. Para ver qué ocurre en este caso, puede probarse el siguiente ejemplo:

```
//: c09:NeverCaught.java
// Ignorando RuntimeExceptions.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

public class NeverCaught {
  private static Test monitor = new Test();
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
    monitor.expect(new String[] {
      "Excepcion en hilo \"main\" " +
      "java.lang.RuntimeException: desde f()",
      "        hasta NeverCaught.f(NeverCaught.java:7)",
      "        hasta NeverCaught.g(NeverCaught.java:10)",
      "        hasta NeverCaught.main(NeverCaught.java:13)"
    });
  }
} ///:~
```

Ya se puede ver que una **RuntimeException** (y cualquier cosa que se herede de la misma) es un caso especial, puesto que el compilador no exige una especificación de excepciones para ellas.

Por tanto la respuesta es: si una **excepción en tiempo de ejecución** consigue todo el camino hasta el método **main**() sin ser capturada, se invoca a **printStackTrace**() para esa excepción, y el programa finaliza su ejecución.

Debe tenerse en cuenta que sólo se pueden ignorar en un código propio las **excepciones en tiempo de ejecución,** puesto que el compilador obliga a realizar el resto de gestiones. El razonamiento es que una **excepción en tiempo de ejecución** representa un error de programación:

1. Un error que no se puede capturar (la recepción de una referencia **null** proveniente de un programador cliente por parte de un método, por ejemplo).
2. Un error que uno, como programador, debería haber comprobado en su código (como un **ArrayIndexOutOfBoundsException** en la que se debería haber comprobado el tamaño del array) Una excepción que ocurre desde el punto #1 a menudo se convierte en un asunto para el punto #2.

Se puede ver fácilmente el gran beneficio aportado por estas excepciones, puesto que éstas ayudan en el proceso de depuración.

Es interesante saber que no se puede clasificar el manejo de excepciones de Java como si fuera una herramienta de propósito específico. Aunque efectivamente está diseñado para manejar estos errores de tiempo de compilación que se darán por motivos externos al propio código, simplemente es esencial para determinados tipos de fallos de programación que el compilador no pueda detectar.

Limpiando con finally

Hay a menudo algunas pieza de código que usted quiere ejecutar si o no una excepción es lanzada dentro de un bloque **try**. Esto usualmente pertenece para alguna operación aparte de la recuperación de memoria (desde que eso ha tenido cuidado de por el colector de basuras). Para lograr este efecto, usted usa una cláusula **finally [2]** al final de todos los manipuladores de excepción. El esquema completo de una sección de manejo de excepción es por consiguiente:

[3] El manejo de excepciones de C++ no tiene la cláusula **finally** porque confía en los destructores para complementar ese tipo de limpieza.

```
try {
  // La region protegida: Actvidades Peligrosas
  // que podria lanzar A, B, o C
} catch(A a1) {
  // Manipulador para situation A
} catch(B b1) {
  // Manipulador para situation B
} catch(C c1) {
  // Manipulador para situation C
} finally {
  // Actividades que ocurren siempre
}
```

Para demostrar que la cláusula **finally** siempre se ejecuta, pruebe este programa:

```
//: c09:FinallyWorks.java
// La clausula finally es siempre ejecutado.
import com.bruceeckel.simpletest.*;

class ThreeException extends Exception {}

public class FinallyWorks {
  private static Test monitor = new Test();
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // Post-incremento es cero la primera vez:
        if(count++ == 0)
          throw new ThreeException();
        System.out.println("No exception");
      } catch(ThreeException e) {
        System.err.println("ThreeException");
      } finally {
        System.err.println("In finally clause");
        if(count == 2) break; // fuera del "while"
      }
    }
    monitor.expect(new String[] {
      "ThreeException",
      "En clausula finally",
      "Sin exception",
      "En clausula finally"
    });
  }
} ///:~
```

De la salida, usted puede ver que ya sea o no una excepción es lanzada, la cláusula **finally** está todo el tiempo ejecutada.

Este programa también da una orientación para manejar el hecho de que las excepciones de Java (al igual que ocurre en C++) no permiten volver a ejecutar a partir del punto en que se lanzó la excepción, como ya se comentó. Si se ubica el bloque **try** dentro de un bloque, se podría establecer una condición a alcanzar antes de continuar con el programa. También se puede añadir un contador **estático** o algún otro dispositivo para permitir al bucle intentar distintos enfoques antes de rendirse. De esta forma se pueden construir programas de extremada fortaleza.

¿Para qué sirve finaly?

En un lenguaje sin colección de basura y sin llamadas automáticas del destructor, **[4] finally** es importante porque le permite al programador garantizar que la liberación de memoria a pesar de lo que ocurre en el bloque **try**. Pero Java tiene colección de basura, así es que la memoria liberatoria es

virtualmente jamás un problema. También, no tiene destructores para la llamada. ¿Así cuando necesita usted usar a **finally** en Java?

[3] Un destructor es una función a la que se llama siempre que se deja de usar un objeto. Siempre se sabe exactamente donde y cuando llamar al destructor. C++ tiene llamadas automáticas del destructor, y C# (el cual es bastante parecido a Java) tiene una forma en que la destrucción automática puede ocurrir.

La cláusula **finally** es necesaria cuando usted necesita colocar fuera de la memoria de regreso para su estado original. Esto es alguna clase de limpieza total como una la conexión del archivo abierto o de la red, algo que usted ha dibujado en la pantalla, o aun un interruptor en el mundo exterior, tan modelado en el siguiente ejemplo:

```
//: c09:Switch.java
public class Switch {
  private boolean state = false;
  public boolean read() { return state; }
  public void on() { state = true; }
  public void off() { state = false; }
} ///:~

//: c09:OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

//: c09:OnOffException2.java
public class OnOffException2 extends Exception {} ///:~

//: c09:OnOffSwitch.java
// Why use 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();
      // Codigo que puede lanzar excepciones...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
      sw.off();
    }
  }
```

```
} ///:~
```

La meta aquí es asegurarse de que el interruptor está cerrado cuando **main**() es completado, así es que **sw.off**() es colocado al final del bloque **try** y al final de cada manipulador de excepción. Pero es posible que una excepción podría ser lanzada que no es capturado aquí, así es que **sw.off**() haría falta. Sin embargo, con **finally** usted puede colocar el código de limpieza total de un bloque **try** en simplemente un lugar:

```
//: c09:WithFinally.java
// Finally Garantiza limpieza total.

public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Codigo que puede lanzar excepciones...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.err.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.err.println("OnOffException2");
    } finally {
      sw.off();
    }
  }
} ///:~
```

Aquí el **sw.off**() ha sido movido a simplemente un lugar, dónde - se asegura - corre pase lo que pase.

Aun en los casos en los cuales la excepción no queda atrapada en la corriente se sedimenta de cláusulas **catch**, finalmente será ejecutado antes de que el mecanismo de manipulación de excepción continúe su búsqueda para un manipulador en el siguiente nivel más alto:

```
//: c09:AlwaysFinally.java
// Finally es siempre ejecutado.
import com.bruceeckel.simpletest.*;

class FourException extends Exception {}

public class AlwaysFinally {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println("Entrando el primer bloque try");
    try {
      System.out.println("Entrando el segundo bloque try");
      try {
        throw new FourException();
```

```
      } finally {
        System.out.println("finally en el 2do bloque try");
      }
    } catch(FourException e) {
      System.err.println(
        "Capturo FourException en el 1er bloque try");
    } finally {
      System.err.println("finally en el 1er bloque try");
    }
    monitor.expect(new String[] {
      "Entrando el primer bloque try",
      "Entrando el segundo bloque try",
      "finally en el 2do bloque try ",
      "Capturo FourException en el 1er bloque try ",
      "finally en el 1er bloque try "
    });
  }
} ///:~
```

La declaración **finally** también será ejecutada en situaciones en las cuales las declaraciones **break** y **continue** están involucrados. Note que, junto con el **break** y **continue** designado, **finally** elimina la necesidad para una declaración del **goto** en Java.

Peligro: La excepción perdida

En general, la implementación de las excepciones en Java destaca bastante, pero desgraciadamente tiene un problema. Aunque las excepciones son una indicación de una crisis en un programa, y nunca debería ignorarse, es posible que simplemente se pierda una excepción. Esto ocurre con una configuración particular al hacer uso de la cláusula **finally:**

```
//: c09:LostMessage.java
// Como una excepcion puede ser perdida.
// {ThrowsException}
import com.bruceeckel.simpletest.*;

class VeryImportantException extends Exception {
  public String toString() {
    return "Una excepcion muy importante!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "Una excepcion trivial";
  }
}

public class LostMessage {
```

```
  private static Test monitor = new Test();
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) throws Exception {
    LostMessage lm = new LostMessage();
    try {
      lm.f();
    } finally {
      lm.dispose();
    }
    monitor.expect(new String[] {
      "Excepcion en hilo \"main\" Una excepcion trivial",
      "\thasta LostMessage.dispose(LostMessage.java:24)",
      "\thasta LostMessage.main(LostMessage.java:31)"
    });  }
} ///:~
```

Se puede ver que no hay evidencia de la **VeryImportantException**, que simplemente es reemplazada por la **HoHumException** en la cláusula **finally**. Ésta es una trampa bastante seria, puesto que significa que una excepción podría perderse completamente, incluso de forma más oculta y difícil de detectar que en el ejemplo de arriba. Por el contrario, C++ trata la situación en la que se lanza una segunda excepción antes de que se maneje la primera como un error de programación fatal. Quizás alguna versión futura de Java repare este problema (por otro lado, generalmente se envuelve todo método que lance alguna excepción, tal como **dispose**() dentro de una cláusula **try-catch**).

Restricciones a las excepciones

Cuando se sobrescribe un método, sólo se pueden lanzar las excepciones que se hayan especificado en la versión de la clase base del método. Ésta es una restricción útil, pues significa que todo código que funcione con la clase base funcionará automáticamente con cualquier objeto derivado de la clase base (un concepto fundamental en POO, por supuesto), incluyendo las excepciones.

Este ejemplo demuestra los tipos de restricciones impuestas (en tiempo de compilación) a las excepciones:

```
//: c09:StormyInning.java
// Los métodos Sobrescritos que lanzan solo las excepciones
// especificadas en sus versiones de 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 {
    // Actualmente no tiene para lanzar cualquiera
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Atraviesa una excepción no
                        // 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 {
  // OK para agregar nuevas excepciones para constructores,
  // pero usted debe aprobar con las excepciones de la
  // clase base:
  public StormyInning()
    throws RainedOut, BaseballException {}
  public StormyInning(String s)
    throws Foul, BaseballException {}
  // Metodos regulares deben conformar a la clase base:
//! void walk() throws PopFoul {} //Error de Compilacion
  // Interfase NO PUEDEN agregar excepciones a metodos
  // existentes de la clase base:
//! public void event() throws RainedOut {}
  // Si el metodo realmente no existe en la clase base,
  // la excepcion esta OK:
  public void rainHard() throws RainedOut {}
  // Usted puede escoger para no lanzar varias
  // excepciones, siempre que si la version base hace:
  public void event() {}
  // Metodos sobrescritos pueden lanzar excepciones
  // heredadas:
  public void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.err.println("Pop foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
```

```
      System.err.println("Generic baseball exception");
    }
    // Strike no lanza en la version derivada.
    try {
      // Que ocurre si usted upcast?
      Inning i = new StormyInning();
      i.atBat();
      // Usted puedo capturar las excepciones de la version
      // de clase base del metodo:
    } catch(Strike e) {
      System.err.println("Strike");
    } catch(Foul e) {
      System.err.println("Foul");
    } catch(RainedOut e) {
      System.err.println("Rained out");
    } catch(BaseballException e) {
      System.err.println("Generic baseball exception");
    }
  }
} ///:~
```

En **Inning**, se puede ver que tanto el método **event**() como el constructor dicen que lanzarán una excepción, pero nunca lo hacen. Esto es legal porque permite forzar al usuario a capturar cualquier excepción que se pueda añadir a versiones superpuestas de **event**(). Como se ve en **atBat**(), en el caso de métodos abstractos se mantiene la misma idea.

La interfaz **Storm** es interesante porque contiene un método (**event**()) que está definido en **Inning**, y un método que no lo está. Ambos métodos lanzan un nuevo tipo de excepción, **RainedOut**. Cuando **StormyInning** hereda de **Inning** e implementa **Storm**, se verá que el método **event**() de Tormenta no puede cambiar la interfaz de excepciones de **event**() en **Inning**. De nuevo, esto tiene sentido porque de otra forma nunca se sabría si se está capturando lo correcto al funcionar con la clase base. Por supuesto, si un método descrito en una interfaz no está en la clase base, como ocurre con **rainHard**(), entonces no hay problema si lanza excepciones.

La restricción sobre las excepciones no se aplica a los constructores. En **StormyInning** se puede ver que un constructor puede lanzar lo que desee, independientemente de lo que lance el constructor de la clase base. Sin embargo, dado que siempre se llamará de una manera u otra a un constructor de clase base (aquí se llama automáticamente al constructor por defecto), el constructor de la clase derivada debe declarar cualquier excepción del constructor de la clase base en su especificación de excepciones. Fíjese que un constructor de clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

La razón por la que **StormyInning.walk**() no compilará es que lanza una excepción, mientras que **Inning.walk**() no lo hace. Si se permitiera esto, se podría escribir código que llamara a **Inning**.**walk**() y que no tuviera que manejar ninguna excepción, pero entonces, al sustituir un objeto de una clase

derivada de **Inning** se lanzarían excepciones, causando una ruptura del código. Forzando a los métodos de la clase derivada a ajustarse a las especificaciones de excepciones de los métodos de la clase base, se mantiene la posibilidad de sustituir objetos.

El método **event**() superpuesto muestra que una versión de clase derivada de un método puede elegir no lanzar excepciones, incluso aunque lo haga la versión de clase base. De nuevo, esto es genial, puesto que no rompe ningún código escrito -asumiendo que la versión de clase base lanza excepciones. A **atBat**() se le aplica una lógica semejante, pues ésta lanza **PopFoul**, una excepción derivada de **Foul**, lanzada por la versión de clase base de **atBat**( ). De esta forma, si alguien escribe código que funciona con **Inning** y llama a **atBat**(), debe capturar la excepción **Foul**. Dado que **PopFoul** deriva de **Foul**, el manejador de excepciones también capturará **PopFoul**.

El último punto interesante está en el método **main**( ). Aquí se puede ver que si se está tratando con un objeto **StormyInning**, el compilador te fuerza a capturar sólo las excepciones específicas a esa clase, pero si se hace una conversión hacia arriba al tipo base, el compilador te fuerza (correctamente) a capturar las excepciones del tipo base. Todas estas limitaciones producen un código de manejo de excepciones más robusto **[5]**.

[5] La Organización Internacional de Normalización C++ agregó restricciones similares que requieren que excepciones derivados de métodos sea lo mismo como, o derivadas de, las excepciones lanzadas por el método de la clase base. Éste es un caso en el cual C++ puede realmente comprobar especificaciones de excepción en la fase de compilación.

Es útil darse cuenta de que aunque las especificaciones de excepciones se ven reforzadas por el compilador durante la herencia, las especificaciones de excepciones no son parte del tipo de un método, que está formado sólo por el nombre del método y los tipos de parámetros. Además, justo porque existe una especificación de excepciones en una versión de clase base de un método, no tiene por qué existir en la versión de clase derivada del mismo. Esto es bastante distinto de lo que dictaminan las reglas de herencia, según las cuales todo método de la clase base debe existir también en la clase derivada. Dicho de otra forma, "*la interfaz de especificación de excepciones*" de un método particular puede estrecharse durante la herencia y superponerse, pero no puede ancharse - esto es precisamente lo contrario de la regla de la interfaz de clases durante la herencia.

Constructores

Cuando se escribe código con excepciones, es particularmente importante que siempre se pregunte: "Si se da una excepción, ¿será limpiada adecuadamente?" La mayoría de veces es bastante seguro, pero en los constructores hay un problema. El constructor pone el objeto en un estado de

partida seguro, pero podría llevar a cabo otra operación -como abrir un fichero- que no se limpia hasta que el usuario haya acabado con el objeto y llame a un método de limpieza especial. Si se lanza una excepción desde dentro de un constructor, puede que estos comportamientos relativos a la limpieza no se den correctamente. Esto significa que hay que ser especialmente cuidadoso al escribir constructores.

Dado que se acaba de aprender lo que ocurre con **finally**, se podría pensar que es la solución correcta. Pero no es tan simple, puesto que **finally** ejecuta siempre el código de limpieza, incluso en las situaciones en las que no se desea que se ejecute este código de limpieza hasta que acabe el método de limpieza. Por consiguiente, si se lleva a cabo una limpieza en **finally**, hay que establecer algún tipo de indicador cuando el constructor finaliza normalmente, de forma que si el indicador está activado no se ejecute nada en **finally**. Dado que esto no es especialmente elegante (se está asociando el código de un sitio a otro), es mejor si se intenta evitar llevar a cabo este tipo de limpieza en el método **finally**, a menos que uno se vea forzado a ello.

En el ejemplo siguiente, se crea una clase llamada **InputFile** que abre un archivo y permite leer una línea (convertida a **String**) de una vez. Usa las clases **FileReader** y **BufferedReader** de la biblioteca estándar de E/S de Java que se verá en el Capítulo 11, pero que son lo suficientemente simples como para no tener ningún problema en tener su uso básico:

```
//: c09:Cleanup.java
// Prestando atencion a excepciones en constructores.
import com.bruceeckel.simpletest.*;
import java.io.*;

class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
      // Otro codigo que podria lanzar excepciones
    } catch(FileNotFoundException e) {
      System.err.println("Could not open " + fname);
      // No lo abre, asi que no lo cierra
      throw e;
    } catch(Exception e) {
      // Todas las otras excepciones deben cerrarlo
      try {
        in.close();
      } catch(IOException e2) {
        System.err.println("in.close() unsuccessful");
      }
      throw e; // Relanza
    } finally {
      // No lo cierra aqui!!!
    }
  }
  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");
    }
  }
}

public class Cleanup {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      InputFile in = new InputFile("Cleanup.java");
      String s;
      int i = 1;
      while((s = in.getLine()) != null)
        ; // Mejora procesamiento linea por linea aqui…
      in.dispose();
    } catch(Exception e) {
      System.err.println("Caught Exception in main");
      e.printStackTrace();
    }
    monitor.expect(new String[] {
      "dispose() successful"
    });
  }
} ///:~
```

El constructor de **InputFile** toma un parámetro **String,** que es el nombre del archivo que se desea abrir. Dentro de un bloque **try,** crea un **FileReader** usando el nombre de archivo. Un **FileReader** no es particularmente útil hasta que se usa para crear un **BufferedReader** con el que nos podemos comunicar - nótese que uno de los beneficios de **InputFile** es que combina estas dos acciones.

Si el constructor **FileReader** no tiene éxito, lanza una **FileNotFoundException** que podría ser capturada de forma separada porque éste es el caso en el que no se quiere cerrar el archivo, puesto que éste no se abrió con éxito. Cualquier *otra* cláusula de captura debe cerrar el archivo, puesto que *fue* abierto en el momento en que se entra en la cláusula **catch.** (Por supuesto, esto es un truco si **FileNotFoundException** puede ser lanzado por más de un método. En este caso, se podría desear romper todo en varios

bloques **try**.) El método **close**() podría lanzar una excepción, por lo que es probado y capturado incluso aunque se encuentra dentro de otro bloque de otra cláusula **catch** - es simplemente otro par de llaves para el compilador de Java. Después de llevar a cabo operaciones locales, se relanza la excepción, lo cual es apropiado porque este constructor falló, y no se desea llamar al método asumiendo que se ha creado el objeto de manera adecuada o que sea válido.
En este ejemplo, que no usa la técnica de los indicadores anteriormente mencionados, la cláusula **finally** no es definitivamente el lugar en el que **close**() (cerrar) el archivo, puesto que lo cerraría cada vez que se complete el constructor. Dado que queremos que se abra el fichero durante la vida útil del objeto **InputFile** esto no sería apropiado.

El método **getLine**() devuelve un **String** que contiene la línea siguiente del archivo. Llama a **readLine**(), que puede lanzar una excepción, pero esa excepción se captura de forma que **getLine**() no lanza excepciones. Uno de los aspectos de diseño de las excepciones es la decisión de si hay que manejar una excepción completamente en este nivel, o hay que manejarla parcialmente y pasar la misma excepción (u otra), o bien simplemente pasarla. Pasarla, siempre que sea apropiada, puede definitivamente simplificar la codificación. En esta situación, el método **getLine**() convierte la excepción a un **RuntimeException** para indicar un error de programación.

Pero por supuesto, el objeto que realiza la llamada es ahora el responsable de manejar cualquier **IOException** que pudiera surgir.

El usuario debe llamar al método **limpiar**() al acabar de usar el objeto **InputFile.** Esto liberará los recursos del sistema (como los manejadores de archivos) que fueron usados por el **BufferedReader** y/u objetos **FileReader [4]**. Esto no se quiere hacer hasta que se acabe con el objeto **InputFile,** en el momento en el que se le deje marchar. Se podría pensar en poner esta funcionalidad en un método **finalize**(), pero como se mencionó en el Capítulo 4, no se puede estar seguro de que se invoque siempre a **finalize**() (incluso si se puede estar seguro de que se invoque, no se sabe cuándo). Éste es uno de los puntos débiles de Java: toda la limpieza -que no sea la limpieza de memoria- no se da automáticamente, por lo que hay que informar al programador cliente de que es responsable, y debe garantizar que se dé la limpieza usando **finalize**().

[4] En C++ un destructor se encargaría de esto por ti.

En **Cleanup.java** se crea un **InputFile** para abrir el mismo archivo fuente que crea el programa, se lee el archivo de línea en línea, y se añaden números de línea. Se capturan de forma genérica todas las excepciones en el método **main**(), aunque se podría elegir una granularidad mayor.

Uno de los beneficios de este ejemplo es mostrar por qué se presentan las excepciones en este punto del libro -no se puede hacer E/S básica sin usar las excepciones. Las excepciones son tan integrantes de la programación de Java,

especialmente porque el compilador las fortalece, que se puede tener éxito si no se conoce bien cómo trabajar con ellas.

Emparejamiento de Excepciones

Cuando se lanza una excepción, el sistema de manejo de excepciones busca en los manejadores más "cercanos" en el mismo orden en que se escribieron. Cuando hace un emparejamiento, se considera que la excepción ya está manejada y no se llevan a cabo más búsquedas.

Emparejar una excepción no exige un proceso perfecto entre la excepción y su manejador. Un objeto de clase derivada se pue de emparejar con un manejador de su clase base, como se ve en el ejemplo siguiente:

```
//: c09:Human.java
// Capturando jerarquias de excepcion.
import com.bruceeckel.simpletest.*;

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
    monitor.expect(new String[] {
      "Caught Sneeze"
    });
  }
} ///:~
```

La excepción **Sneeze** será atrapada por la primera cláusula **catch** a la que corresponde, el cual es el primero, por supuesto. Sin embargo, si usted quita la primera cláusula **catch**, saliendo sólo:

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    }
```

El código todavía surtirá efecto porque captura la clase base **Sneeze**. Ponga de cualquier otro modo, **catch** (**Annoyance e**) capturará a un **Annoyance** o cualquier clase derivada de eso. Esto es útil porque si usted decide añadirle más excepciones derivadas a un método, luego el código cliente del programador no necesitará cambiar con tal de que el cliente capture las excepciones de clase base.

Si usted trata de "camuflar" excepciones de clase derivada de la clase derivada poniendo la primera parte de cláusula **catch** de la clase base, como esto:

```
try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.err.println("Caught Annoyance");
    } catch(Sneeze s) {
      System.err.println("Caught Sneeze");
}
```

El compilador le dará un mensaje de error, desde que ve que la cláusula **catch** de **Sneeze** nunca puede ser alcanzada.

La alternativa se acerca

Un sistema que manipula excepción es una puerta de trampa que le permite su programa abandonar ejecución de la secuencia normal de declaraciones. La puerta de trampa es usada cuando una "condición excepcional" ocurre, tal esa ejecución normal es ya no posible o deseable. Las excepciones representan condiciones que el método actual es incapaz de manejar. La razón por la que la manipulación de excepción que los sistemas fueron desarrollados lo es porque el acercamiento de ocuparse de cada condición de error posible producida por cada llamada de función fue demasiada onerosa, y las programadoras simplemente no la estaban haciendo. Como consecuencia, ignoraban los errores. Es digno de acechanza que el asunto de conveniencia del programador en manejar errores fue una motivación de primera para excepciones en primer lugar.

Una de las líneas directivas importantes en el manipular excepciones es "no capture una excepción a menos que usted sepa qué hacer con ella." De hecho, una de las metas importantes de manipular excepciones es apartar el código de control de errores del punto donde los errores ocurren. Esto le permite enfocar la atención en lo que usted quiere lograr en una sección de su código, y cómo usted va a ocuparse de problemas en una sección separada bien definida de su código. Como consecuencia, su código de línea principal no es desordenado con lógica de control de errores, y es mucho más fácil entender y mantener.

Las excepciones comprobadas complican este panorama un poco, porque le obligan a añadir cláusulas **catch** en lugares donde usted no puede estar listo a manejar un error. Esto resulta en el problema "dañino si es tragado":

```
try {
  // ... hacer algo util
} catch(ObligatoryException e) {} // Gulp!
```

Los programadores (yo me incluyo, en la primera edición de este libro) simplemente harían la cosa más simple, y se tragarían la excepción – a menudo involuntariamente, pero una vez que usted la hace, el compilador ha quedado satisfecho, así es que a menos que usted se acuerde de volver a visitar y corregir el código, la excepción estará perdida. La excepción ocurre, pero desaparece completamente cuando es tragado. Porque el compilador le obliga a escribir código de inmediato para manejar la excepción, esto tiene la apariencia de la solución más fácil si bien es probablemente lo peor que usted puede hacer.

Horrorizado al darse cuenta de que había hecho esto, en la segunda edición que "arreglé" el problema imprimiendo el rastro de la pila dentro del manipulador (como se ve todavía – apropiadamente – en un número de ejemplos en este capítulo). Mientras esto es útil para rastrear el comportamiento de excepciones, todavía señala que usted realmente no sabe qué hacer con la excepción en ese momento en su código. En este pasaje consideraremos una cierta cantidad de los asuntos y las complicaciones proviniendo de comprobadas excepciones, y opciones que usted tiene cuándo está ocupándose de ellos.

Este tema parece simple. Pero no es sólo complicado, que sea también un asunto de alguna volatilidad. Hay personas que están incondicionalmente arraigadas en ya sea el aspecto de la cerca y quién considera que la respuesta correcta (manifiestamente la de ellos) es obvia. Creo que la razón para uno de estas posiciones es el beneficio bien definido visto en ir de un lenguaje pobremente escrito como ANSI C para un lenguaje fuerte, (es decir, comprobada en la fase de compilación) estáticamente escrito como C++ o Java. Cuando usted hace esa transición (como hice), los beneficios son tan dramáticos que puede parecer que el comprobar tipos fuertemente estático es siempre la mejor respuesta para la mayoría de problemas. Mi esperanza es relatar un poquito de mi evolución, eso ha cuestionado el valor absoluto de comprobación de tipo fuertemente estática; Claramente, es mero útil demasiado del tiempo, pero hay una línea borrosa que cruzamos cuando comienza a meter en la forma y convertirse en un obstáculo (una de mis citas favoritas es: "Todos los modelos están mal. Algunos tiene utilidad.").

La Historia

El manipular excepciones se originó en los sistemas como PL/1 y Mesa, y más tarde apareció en CLU, Smalltalk, Modula-3, Ada, Eiffel, C + +, Python, Java, y los lenguajes Java Ruby y C #. El diseño Java es similar a C + +, excepto en lugares donde los diseñadores Java sintieron que el diseño C++ causó problemas.

Para proveer a programadores de un armazón que más probable usaran para el control de errores y recuperación, el manipular excepciones fue añadido a C++ más bien hacia fines del proceso de estandarización, promovido por Bjarne Stroustrup, el autor original del lenguaje. El modelo para excepciones C++ provino primordialmente de CLU. Sin embargo, otros idiomas existieron en aquel entonces que también soportaron manejo de excepción: Ada, Smalltalk (ambos del cual si tuviesen excepciones pero ninguna de las especificaciones de excepción) y Modula-3 (que incluyó ambas excepciones y especificaciones).

En su papel seminal **[5]** en el tema, Liskov y Snyder notifica que un defecto mayor de lenguajes como C que reporta errores en una moda transitoria está eso:

"... cada invocación debe ser perseguida por una prueba condicional para determinar lo que el resultado fue. Este requisito conduce a los programas que son difíciles para la lectura, y probablemente ineficiente igualmente, así haciéndole a programadores desistir de hacer señales y manejando excepciones."

[5] Barbara Liskov and Alan Snyder: Exception Handling in CLU, IEEE Transactions on Software Engineering, Vol. SE- 5, No. 6, November 1979. Este papel no está disponible en la Internet, sólo en forma estampada así es que usted tendrá que contactar una biblioteca para obtener una copia.

Note que una de las motivaciones originales de manejo de excepción fueron evitar este requisito, pero con excepciones comprobadas en Java que comúnmente vemos exactamente esta clase de código. Proceden a decir:

"... requerir que el texto de un manipulador esté apegado a la invocación que sube la excepción conduciría a los programas ilegibles en los cuales las expresiones fueron rotas arriba con manipuladores."

Después del acercamiento CLU al diseñar excepciones C + +, Stroustrup manifestó que la meta fue reducirse la cantidad de código requerido para recuperarse de errores. Creo que él observaba que los programadores no fueron típicamente escribiendo código de control de errores en C porque la cantidad y colocación de tal código eran atemorizantes y divertidas. Como consecuencia, estaban acostumbradas a hacerle la forma de la C, ignorar errores en código y usar depuradores para seguirle la pista a los problemas.

Para usar excepciones, estos programadores de C tuvieron que quedar convencidos para escribir código “adicional” que normalmente no escribían. Así, para dibujarlos en una mejor forma de manipular errores, la cantidad de código que necesitarían “agregar” no debe ser onerosa. Pienso que es importante recordar esto objetivo al considerar los efectos de excepciones comprobadas en Java.

C++ trajo una idea adicional más de CLU: La especificación de excepción, programáticamente indicar en la firma de método lo que las excepciones pueden resultar de llamar ese método. La especificación de excepción realmente tiene dos propósitos. Puede decir “origino esta excepción en mi código, usted la maneja.” Pero también puede tratar de decir “ignoro esta excepción que puede ocurrir como resultado de mi código, usted lo maneja.” Hemos estado enfocando en lo “que usted maneja eso” en parte al mirar a los mecánicos y sintaxis de excepciones, pero aquí tengo en particular interés en el hecho de que a menudo ignoramos excepciones y eso es lo que la especificación de excepción puede indicar.

En la especificación de excepción C++ no es parte de la información de tipo de una función. La comprobación sólo de fase de compilación es asegurar que las especificaciones de excepción son usadas consistentemente; Por ejemplo, si una función o un método lanza excepciones, entonces las versiones sobrecargadas o derivadas también deben lanzar esas excepciones. A diferencia de Java, sin embargo, ninguna comprobación de fase de compilación ocurre para determinar si o no la función o el método realmente lanzará esa excepción, o ya sea la especificación de excepción es completa (eso es, ya sea exactamente describe todas las excepciones que pueden ser lanzadas). Esa validación ocurre, pero sólo en el tiempo de ejecución. Si una excepción es lanzada que viola la especificación de excepción, el programa C + + llamará la función estándar de la biblioteca **unexpected**().

Es interesante notar, porque del uso de plantillas, las especificaciones de excepción no son usadas en absoluto en la biblioteca estándar C++. Las especificaciones de excepción, luego, pueden tener un impacto significativo en el diseño de Java Generics (la versión de plantillas C++ de Java, esperado para aparecer en JDK 1.5).

Perspectivas

Primero, merita anotar que Java eficazmente inventó la excepción comprobada (claramente inspirado por especificaciones de excepción C++ y el hecho que programadores C++ típicamente no pierde el tiempo en ellos). Ha sido un experimento, cuál ningún lenguaje desde entonces ha escogido para duplicarse.

En segundo lugar, las excepciones comprobadas parecen ser una cosa obvia de

bien estando discernido en ejemplos introductorios y en programas pequeños. Ha sido sugerido que las dificultades sutiles comienz an a aparecer cuando los programas comienzan a agrandarse. Por supuesto, la magnitud usualmente no ocurre de la noche a la mañana; Avanza lentamente. Los lenguajes que no pueden ser adecuados para proyectos de gran escala sirven para proyectos pequeños que crecen, y que en algún punto nos damos cuenta de que las cosas están yendo de manejables a dificultosos. Esto es lo que yo sugiero puede ser el caso con demasiado tipo inspeccionando; En particular, con excepciones comprobadas.

La escala del programa parece ser un asunto significativo. Esto es un problema porque la mayoría de discusiones tienden a usar programas pequeños como comprobaciones. Uno de los diseñadores C# observó eso:

"El examen de programas pequeños conduce a la conclusión que las especificaciones requerientes de excepción ambos pudieron realzar productividad del desarrollador y pudieron realzar calidad de código, pero la experiencia con proyectos grandes del software sugiere un resultado diferente – la productividad disminuida y poco o ningún incremento en la calidad de código." **[6]**

[6]
http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820

En relación a excepciones libres, los creadores CLU indicaron:

"Sentimos que fue poco realista requerir que el programador provea a los manipuladores en situaciones donde ninguna acción significativa puede ser tomada." **[7]**

[7] ibid

Al explicar por qué una declaración de función sin especificación quiere decir que puede lanzar cualquier excepción, en vez de no excepciones, Stroustrup indica:

"Sin embargo, eso requeriría especificaciones de excepción para esencialmente cada función, sería una causa significativa para recompilación, e inhibiría cooperación con software escrito en otros lenguajes. Esto alentaría a los programadores a corromper los mecanismos que manejan excepción y escribir código falso para suprimir excepciones. Le proveería un sentido falso de seguridad a las personas que pasaron por alto la excepción." **[8]**

[8] Bjarne Stroustrup, The C++ Programming Language, 3rd edition, Addison-Wesley 1997, pp 376.

Vemos este mismo comportamiento – subvirtiendo las excepciones – ocurriendo con excepciones comprobadas en Java.

Martin Fowler (el autor de UML Destilado, Refactoring, y los Patrones de Análisis) me escribió lo siguiente para mí:

"... en general pienso que las excepciones son buenas, pero las excepciones comprobadas Java son más molesta que ellos merecen la pena."

Ahora pienso que el paso importante de Java fue unificar el modelo de información de error, a fin de que todos los errores sean reportados usando excepciones. Esto no ocurría con C++, porque para resguardar la compatibilidad con C el modelo viejo de solo ignorar errores estaba todavía disponible. Pero si ustedes tienen información coherente con excepciones, entonces las excepciones pueden ser usadas si desearon, y en caso de que no, se reproducirán fuera para el nivel de más alto (la consola u otro programa del contenedor). Cuándo Java cambió el modelo C++ a fin de que las excepciones fueran la única forma para reportar errores, la ejecución adicional de excepciones comprobadas pudo haberse vuelto menos necesaria.

En el pasado, he sido un creyente fuerte que ambos las excepciones comprobadas y tipos estáticos fuertes comprobados fueron esenciales para el desarrollo robusto de programa. Sin embargo, la experiencia anecdótica y directa **[9]** con lenguajes que son más dinámicos que estáticos me conduce a pensar que los grandes beneficios realmente vienen de:

1. Un modelo de reporte de error unificado por excepciones, no obstante si el programador es forzado el compilador para manipularlos.
2. La comprobación de tipo, a pesar de cuando tiene lugar. Es decir, con tal de que el uso correcto de un tipo es implementado, no tiene importancia si ocurre en tiempo de compilación o el tiempo de ejecución.

[9] Indirectamente con Smalltalk por conversaciones con varios programadores experimentados en este lenguaje; directamente con Python (www.Python.org).

Además de esto, hay beneficios de productividad muy significativos para reducir las restricciones de fases de compilación en el programador. Ciertamente, la reflexión (y eventualmente, genéricos) está obligada a compensar para la naturaleza que sobre-constriñe de mecanografía estática fuerte, como usted verá en el siguiente capítulo y en un número de ejemplos a lo largo del libro.

Ya he sido distinguido por una cierta cantidad que lo que digo aquí constituye blasfemia, y pronunciando estas palabras mi reputación se destruirá, las civilizaciones caerán, y un porcentaje superior de programar proyectos errarán. La creencia que el compilador puede salvar su proyecto señalando errores en la fase de compilación corre fuertemente, pero es importante darse cuenta de la limitación de qué el compilador puede hacer; En el Capítulo 15, enfatizo el valor de un proceso automatizado de la constitución y prueba de unidades, cuál le da lejos más apalancamiento que usted pase inadvertido tratando de convertir todo en un error de sintaxis. Está que vale en vista de eso:

Un buen lenguaje de programación es uno que ayuda a los programadores a escribir buenos programas. Ningún lenguaje de programación les impedirá a sus usuarios escribir programas malos. **[10]**

[10] (Kees Koster, diseñador del lenguaje CDL, citado por Bertrand Meyer, diseñador del Lenguaje Eiffel). http://www.elj.com/elj/v1/n1/bm/right/.

En cualquier caso, la probabilidad de excepciones comprobadas que están en la vida alejados de Java parece oscura. Sería demasiado radical de un cambio de lenguaje, y los proponentes dentro de Sun parecen ser realmente fuertes. Sun tiene una historia y política de absoluto resguardo de compatibilidad – para darle un sentido de esto, virtualmente todo software Sun anda en todo hardware Sun, no importa cuán viejo esté. Sin embargo, si usted se encuentra con que algunas excepciones comprobadas recaudan su forma, o especialmente si usted se encuentra viéndose forzado a percibir excepciones, sino usted no sabe qué hacer con ellos, aquí están algunas alternativas.

Pasando excepciones a la consola

En los programas simples, como muchos de aquellos en este libro, la forma más fácil para conservar las excepciones sin escribir una gran cantidad de código son distribuirlos de **main**() a la consola. Por ejemplo, si usted quiere abrir un archivo para leer (algo que usted aprenderá casi en el detalle en el Capítulo 12), usted debe abrir y debe cerrar a un **FileInputStream**, lo cual lanza excepciones. Para un programa simple, usted puede hacer esto (usted verá este acercamiento usado en numerosos lugares a lo largo de este libro):

```
//: c09:MainException.java
import java.io.*;

public class MainException {
  // Pasa todas las excepciones a la consola:
  public static void main(String[] args) throws Exception {
    // Abre el archivo:
```

```
    FileInputStream file =
      new FileInputStream("MainException.java");
    // Usa el archivo ...
    // Cierra el archivo:
    file.close();
  }
} ///:~
```

Note que **main**() es también un método que puede tener una especificación de excepción, y aquí el tipo de excepción es **Exception**, la clase raíz de todas las excepciones comprobadas. Distribuyéndolo a la consola, usted es eximido de escribir cláusulas **try**-**catch** dentro del cuerpo de **main**(). (Desafortunadamente, el archivo de Entrada/Salida es significativamente más complejo que pareciese estar desde este ejemplo, así es que no se ponga demasiado excitado hasta que usted haya leído el Capítulo 12).

Convirtiendo las excepciones comprobada a no comprobadas

Lanzar una excepción desde **main**() es conveniente cuando usted escribe a un **main**(), pero no es generalmente útil. El problema real es cuando usted escribe un cuerpo común de método, y usted llama otro método y se percata de que “no tengo idea qué hacer con esta excepción aquí, pero no quiero tragarla o imprimir algún mensaje vulgar.” Con excepciones concatenadas del JDK 1.4, una solución nueva y simple se evita a sí mismo. Usted simplemente “envuelve” una excepción comprobada dentro de un RuntimeException, como éste:

```
try {
  // ... haz algo útil
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
  throw new RuntimeException(e);
}
```

Ésta aparenta ser una solución ideal si usted quiere desactivar la excepción comprobada – usted no la traga, y usted no tiene que meterlo en la especificación de excepción de su método, pero por el encadenamiento de excepción usted no pierde cualquier información de la excepción original.

Esta técnica provee la opción para ignorar la excepción y dejarle a ella burbujear arriba de la pila de llamada sin estar obligada a escribir cláusulas **try**-**catch** y/o las especificaciones de excepción. Sin embargo, usted todavía puede capturar y puede manejar la excepción específica usando **getCause**(), como visto aquí:

```
//: c09:TurnOffChecking.java
```

```
// "Apagando" excepciones comprobadas.
import com.bruceeckel.simpletest.*;
import java.io.*;

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) { // Adapta a no comprobado:
      throw new RuntimeException(e);
    }
  }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    WrapCheckedException wce = new WrapCheckedException();
    // Usted puede llamar a f() sin un bloque try, y permite a
    // RuntimeExceptions salir del metodo:
    wce.throwRuntimeException(3);
    // O usted puede escoger caputar las excepciones:
    for(int i = 0; i < 4; i++)
      try {
        if(i < 3)
          wce.throwRuntimeException(i);
        else
          throw new SomeOtherException();
      } catch(SomeOtherException e) {
          System.out.println("SomeOtherException: " + e);
      } catch(RuntimeException re) {
        try {
          throw re.getCause();
        } catch(FileNotFoundException e) {
          System.out.println(
            "FileNotFoundException: " + e);
        } catch(IOException e) {
          System.out.println("IOException: " + e);
        } catch(Throwable e) {
          System.out.println("Throwable: " + e);
        }
      }
    monitor.expect(new String[] {
      "FileNotFoundException: " +
      "java.io.FileNotFoundException",
      "IOException: java.io.IOException",
```

```
      "Throwable: java.lang.RuntimeException: Donde estoy yo?",
      "SomeOtherException: SomeOtherException"
    });
  }
} ///:~
```

WrapCheckedException.throwRuntimeException() contiene código que genera tipos diferentes de excepciones. Estos son atrapados y enrollados dentro de objetos **RuntimeException**, así es que se convierten en la "causa" de esas excepciones.

En **TurnOffChecking**, usted puede ver que cabe llamar **throwRuntimeException**() sin bloque **try** porque el método no lanza varias excepciones comprobadas. Sin embargo, cuando usted está listo para capturar excepciones, usted todavía tiene la capacidad de capturar cualquier excepción que usted quiere poniendo su código dentro de un bloque **try**. Usted comienza por percibir todas las excepciones que usted explícitamente sabe podría emerger del código en su bloque **try** – en este caso, **SomeOtherException** es atrapado primero. Finalmente, usted captura a **RuntimeException** y lanza el resultado de **getCause**() (la excepción envuelta). Esto extrae las excepciones originarias, lo cual luego puede ser manipulado en sus cláusulas **catch**.

La técnica de envolver una excepción comprobada en un **RuntimeException** será usada cuando asignará a lo largo del resto de este libro.

Las directivas de excepción

Use excepciones para:

1. Manejar problemas en el nivel apropiado. (Evite capturar excepciones a menos que se sepa qué hacer con ellas).
2. Arreglar el problema y llamar de nuevo al método que causó la excepción.
3. Arreglar todo y continuar sin volver a ejecutar el método
4. Calcular algún resultado alternativo en vez de lo que se suponía que iba a devolver el método.
5. Hacer lo que se pueda en el contexto actual y relanzar la *misma* excepción a un contexto superior.
6. Hacer lo que se pueda en el contexto actual y lanzar una excepción *diferente* a un contexto superior.
7. Terminar el programa.
8. Simplificar. (Si tu esquema de excepción hace algo más complicado, es una molestia utilizarlo.)
9. Hacer más seguros la biblioteca y el programa. (Ésta es una inversión a corto plazo de cara a la depuración, y también una inversión a largo plazo de cara a la fortaleza de la aplicación.)

Resumen

La recuperación de errores mejorada es una de las formas más poderosas para que usted pueda aumentar la robustez de su código. La recuperación de errores es una preocupación fundamental para cada programa que usted escribe, pero es especialmente importante en Java, dónde una de las metas primarias es crear componentes de programa para otros aprovechar. *Para crear un sistema robusto, cada componente debe ser robusto*. Proveyéndole un consistente modelo de reporte de error con excepciones, Java permite componentes para confiadamente comunicarle los problemas al código del cliente.

Las metas para el manejo de excepción en Java son simplificar la creación de programas grandes, fidedignos usando menos código que actualmente posible, y para hacer eso con más confianza que su aplicación no tiene un error no manejado. Las excepciones no son terriblemente difíciles de aprender, y son una de esas características que le proveen los beneficios inmediatos y significativos a su proyecto.

Ejercicios

Las soluciones para los ejercicios seleccionados pueden ser encontradas en el documento electrónico *The Thinking en Java Annotated Solution Guide*, disponible para una retribución pequeña de *www.BruceEckel.com*.

1. Cree una clase con un **main**() que lanza un objeto de clase **Exception** dentro de un bloque **try**. Déle al constructor para **Exception** un argumento **String**. Capture la excepción dentro de una cláusula **catch** e imprima el argumento **String**. Agregue una cláusula **finally** e imprima un mensaje para probar que usted estaba allí.
2. Cree su clase de excepción usando la palabra clave **extends**. Escriba un constructor para esta clase que toma un argumento **String** y almacena lo dentro del objeto con una referencia **String**. Escriba un método que imprime el String almacenado. Cree una cláusula **try-catch** para ejercitar su excepción nueva.
3. Escriba una clase con un método que lanza una excepción del tipo creada en el Ejercicio 2. Pruebe compilarlo sin una especificación de excepción para ver lo que el compilador dice. Agregue la especificación apropiada de excepción. Pruebe su clase y su excepción dentro de una cláusula **try-catch**.
4. Defina una referencia del objeto e inicialícela a nulo. Trate de llamar un método a través de esta referencia. Ahora envuelva el código en una cláusula **try-catch** para capturar la excepción.
5. Cree una clase con dos métodos, **f**() y **g**(). En **g**(), lance una excepción de un tipo nuevo que usted define. En **f**(), llame a g(), coja su excepción y, en la cláusula **catch**,

lanza una excepción diferente (de un segundo tipo que usted define). Pruebe su código en **main**().

6. Repita el ejercicio previo, pero dentro de la cláusula **catch**, envuelva la excepción de **g**() en un **RuntimeException**.
7. Cree tres tipos nuevos de excepciones. Escriba una clase con un método que lanza todos los tres. En **main**(), llame el método sino único usa una única cláusula **catch** que atrapará todos los tres tipos de excepciones.
8. Escriba código para generar y atrapar a un **ArrayIndexOutOfBoundsException**.
9. Cree su comportamiento como de reanudación usando un bucle **while** que repite hasta que una excepción sea no más lanzada.
10. Cree una jerarquía de tres niveles de excepciones. Ahora cree una clase base **A** con un método que lanza una excepción en la base de su jerarquía. Herede **B** desde **A** y sobrescriba el método conque lanza una excepción en el nivel dos de su jerarquía. Repita heredando la clase **C** desde **B**. En **main**(), cree una **C** y dirigido hacia arriba para **A**, luego llame el método.
11. Demuestre que un constructor de clase derivada no puede percibir excepciones lanzadas por su constructor de clase base.
12. Demuestre que **OnOffSwitch.java** puede fallar lanzando a un **RuntimeException** dentro del bloque **try**.
13. Demuestre que **WithFinally.java** no falla lanzando a un **RuntimeException** dentro del bloque **try**.
14. Modifique el Ejercicio 7 agregando una cláusula **finally**. Compruebe que su cláusula **finally** es ejecutada, aun si un **NullPointerException** es lanzado.
15. Cree un ejemplo donde usted usa una bandera para controlar si el código de limpieza total es llamado, como descrito en el segundo párrafo después del encabezado “Constructores.”
16. Modifique a **StormyInning.java** agregando un tipo de excepción **UmpireArgument** y métodos que lanzan esta excepción. Pruebe la jerarquía modificada.
17. Quite la primera cláusula **catch** en **Human.java** y compruebe que el código todavía compila y corre correctamente.
18. Añádale un nivel de segundo de pérdida de excepción a **LostMessage.java** a fin de que el **HoHumException** es sí mismo reemplazado por una tercera excepción.
19. Añádale un conjunto apropiado de excepciones al **c08:GreenhouseControls.java**.
20. Añádale un conjunto apropiado de excepciones al **c08:Sequence.java**.
21. Cambie la cadena de nombre de archivo en **MainException.java** para nombrar un archivo que no existe. Corra el programa y note el resultado.

10: Identificación de tipo en tiempo de ejecución

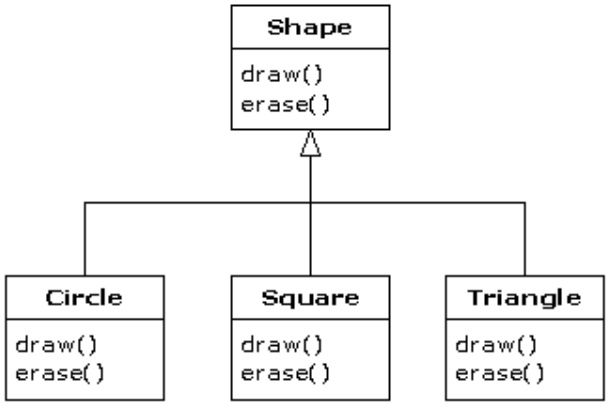
La idea de la identificación de tipo en tiempo de ejecución (RTTI, por run-time type identification) parece medianamente simple al principio: permite encontrar el tipo exacto de un objeto cuando sólo se tiene una referencia al tipo base.

Por supuesto, la *necesidad* de RTTI revela un abundante conjunto de interesantes (y a menudo desconcertantes) cuestiones de diseño OO (Orientado a Objetos), y plantea preguntas fundamentales acerca de cómo debería estructurar sus programas.

Este capítulo apunta a las formas en que Java le permite descubrir información acerca de objetos y clases en tiempo de ejecución. Esto toma dos formas: RTTI "tradicional", el cual asume que se tienen todos los tipos disponibles en tiempo de compilación y en tiempo de ejecución, y el mecanismo " **reflection** ", el cual permite averiguar información de clase únicamente en tiempo de ejecución. El RTTI "tradicional" será cubierto primero, seguido por una discusión acerca de **reflection** .

La necesidad de RTTI

Considere el ahora familiar ejemplo de una jerarquía de clases que usa polimorfismo. El tipo genérico es la clase base **Shape** (forma), y los tipos específicos derivados son **Circle** (círculo), **Square** (cuadrado) y **Triangle** (triángulo):



Este es un diagrama de jerarquía de clases típico, con la clase base en la cima y las clases derivadas creciendo en forma descendiente. El objetivo normal en la programación orientada a objetos es codificar la mayoría de su código haciendo referencia a la clase base (**Shape** , en este caso), de manera que si decide extender el programa agregando una nueva clase (**Rhomboid** , derivada de **Shape** , por ejemplo), el grueso del código no sea afectado. En este ejemplo, el método enlazado dinámicamente en la interfaz **Shape** es **draw( )** (dibujar), de manera que el intento es que el programador cliente llame a **draw( )** por medio de una referencia genérica a **Shape** . **draw( )** está sobrescrito en todas las clases derivadas y, dado que es un método enlazado dinámicamente, tendrá el comportamiento adecuado aunque sea llamado a través de una referencia genérica a **Shape** . Eso es polimorfismo.

De esta manera, generalmente usted crea un objeto de una clase específica (**Circle** , **Square** o **Triangle**), realiza una operación de molde hacia arriba (upcast) a **Shape** (olvidando el tipo específico del objeto), y usa esa referencia anónima a **Shape** en el resto del programa.

Como un breve repaso de polimorfismo y moldeado hacia arriba (upcasting), podría codificar el ejemplo anterior de la siguiente manera:

```
//: c12:Shapes.java
import java.util.*;
class Shape {
   void draw() {
      System.out.println(this + ".draw()");
   }
}
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) {
      ArrayList s = new ArrayList();
      s.add(new Circle());
      s.add(new Square());
      s.add(new Triangle());
      Iterator e = s.iterator();
      while(e.hasNext())
         ((Shape)e.next()).draw();
   }
```

```
} ///:~
```

La clase base contiene un método **draw( )** que usa indirectamente a **toString( )** para imprimir un identificador de la clase pasando **this** a **System.out.println( )** . Si esa función ve un objeto, automáticamente llama al método **toString( )** para producir una representación como cadena (**String**).

Cada una de las clases derivadas sobrescribe el método **toString( )** (de **Object**), de manera que **draw( )** termina imprimiendo algo distinto en cada caso. En **main ( )** , son creados tipos específicos de **Shape** y agregados a un **ArrayList** . Este es el punto en el cual ocurre el molde hacia arriba (upcast) porque **ArrayList** puede contener solamente **Object** s. Ya que todo en Java (con la excepción de los tipos primitivos) es un **Object** , un **ArrayList** puede además contener objetos de tipo **Shape** . Pero en el transcurso de un molde hacia arriba (upcast) a **Object** se pierde además toda información específica, incluyendo el hecho de que los objetos son **Shape** s. Para el **ArrayList** , son sólo **Object** s.

En el momento que usted saca un elemento del **ArrayList** con **next( )** las cosas se complican un poco. Puesto que **ArrayList** contiene sólo **Object** s, naturalmente **next( )** produce una **referencia a Object** . Pero nosotros sabemos que es realmente una referencia a **Shape** y queremos enviar mensajes de **Shape** a ese objeto. De manera que es necesario un molde a **Shape** , usando el molde (cast) tradicional " **(Shape)** ". Esta es la forma más básica de RTTI, puesto que, en Java, todos los moldes (casts) son controlados en tiempo de ejecución. Eso es exactamente lo que RTTI significa: en tiempo de ejecución, el tipo de un objeto es identificado.

En este caso, el molde RTTI es sólo parcial: se hace de **Object** a **Shape** y no hasta el final, a **Circle** , **Square** o **Triangle** . Esto es porque lo único que *sabemos* en este punto es que el **ArrayList** está lleno de **Shape** s. En tiempo de compilación, esto es forzado sólo por reglas autoimpuestas, pero en tiempo de ejecución es asegurado por el molde.

Ahora entra en juego el polimorfismo y el método exacto que es llamado para el **Shape** es determinado de acuerdo a si la referencia es a un **Circle** , **Square** o **Triangle** . Y, en general, esto es como debería ser; usted desea que el grueso de su código sepa tan poco como sea posible acerca de los tipos *específicos* de los objetos y lo justo para tratar con la representación general de una familia de objetos (en este caso **Shape**). Como resultado, su código será más fácil de escribir, leer y mantener y sus diseños serán más fáciles de implementar, entender y cambiar. De manera que el polimorfismo es la meta general de la programación orientada a objetos.

¿Pero que pasa si usted tiene un problema de programación especial que es más fácil de resolver si se conoce el tipo exacto de una referencia genérica? Por ejemplo, suponga que usted desea permitirle a sus usuarios resaltar todas las formas de un tipo particular volviéndolas púrpura. De esta manera, podemos

encontrar todos los triángulos en la pantalla resaltándolos. Esto es lo que RTTI lleva a cabo: usted le puede preguntar a una referencia a **Shape** el tipo exacto al cual está haciendo referencia.

El objeto Class

Para entender cómo trabaja RTTI en Java, debe saber primero cómo es representada la información de tipo en tiempo de ejecución. Esto es llevado a cabo por medio de una clase especial de objeto llamado el *objeto Class* , el cual contiene información acerca de la clase. (Esto es algunas veces llamado *metaclase.*) De hecho, el objeto **Class** es usado para crear todos los objetos "regulares" de su clase.

Hay un objeto **Class** por cada clase que es parte de su programa. Esto es, cada vez que usted escribe y compila una nueva clase, un único objeto **Class** es creado (y almacenado, en un archivo **.class** con el mismo nombre). En tiempo de ejecución, cuando usted crea un objeto de esa clase, la máquina virtual de Java (JVM), que está ejecutando su programa, primero chequea si el objeto **Class** para ese tipo está cargado. Si no, lo carga encontrando el archivo **.class** con ese nombre. De este modo, un programa Java no es cargado completamente antes de comenzar, lo cual es diferente a muchos lenguajes tradicionales.

Una vez que el objeto **Class** para ese tipo está en memoria, es usado para crear todos los objetos de ese tipo.

Si esto le parece sombrío o no lo cree realmente, aquí hay un programa de demostración que puede probarlo:

```
//: c12:SweetShop.java
// Examen de la forma en que trabaja el cargador de
// clases (class loader).
class Candy {
   static {
      System.out.println("Cargando Candy");
   }
}
class Gum {
   static {
   System.out.println("Cargando Gum");
   }
}
class Cookie {
   static {
      System.out.println("Cargando Cookie");
   }
}
public class SweetShop {
```

```
   public static void main(String[] args) {
      System.out.println("dentro de main");
      new Candy();
      System.out.println("Después de crear Candy");
      try {
         Class.forName("Gum");
      } catch(ClassNotFoundException e) {
         e.printStackTrace(System.err);
      }
      System.out.println(
         "Después de Class.forName(\"Gum\")");
      new Cookie();
      System.out.println("Después de crear Cookie");
   }
} ///:~
```

Cada una de las clases Candy (caramelo), Gum (chicle) y Cookie (galleta) tienen una cláusula **static** que es ejecutada cuando la clase es cargada por primera vez. Se imprimirá información para notificarle acerca de cuándo es cargada una clase. En **main( )** , las creaciones de objetos están esparcidas entre sentencias de impresión, para ayudar a detectar el momento de carga.

Una línea particularmente interesante es:

```
Class.forName("Gum");
```

Este método es una miembro estático de **Class** (a la cual pertenecen todos los objeto **Class**). Un objeto **Class** es como cualquier otro, usted puede obtener y manipular una referencia a él. (Eso es lo que el cargador hace.) Una de las maneras de obtener una referencia al objeto **Class** es **forName( )** , el cual toma un **String** conteniendo el nombre textual (cuidado con el deletreo y la capitalización!) de la clase particular de la cual usted quiere una referencia. Retorna una referencia a **Class** .

La salida de este programa para una JVM es:

```
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
```

Puede ver que cada objeto **Class** es cargado sólo cuando se necesita y la inicialización **static** es realizada en la carga de la clase.

Literales de clase

Java provee una segunda forma de producir una referencia al objeto **Class** , usando un literal de clase. En el programa anterior, esto podría verse como:

```
Gum.class;
```

Lo cual no es sólo más simple, sino también más seguro, ya que es controlado en tiempo de ejecución. Debido a que elimina la llamada a un método, también es más eficiente.

Los literales de clase trabajan con clases comunes, así como también con interfaces, arreglos y tipos primitivos. En suma, hay un campo estándar llamado **TYPE** que existe por cada una de las clases envoltura primitivas. El campo **TYPE** produce una referencia al objeto **Class** para el tipo primitivo asociado, de esta forma:

...es equivalente a...	
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

Prefiero usar la versión ". **Class** ", de ser posible, ya que es más consistente con las clases regulares.

Controlando antes de realizar una operación de molde (cast)

Hasta el momento, se ha visto que las formas de RTTI incluyen:

1. El molde clásico, por ejemplo " **(Shape)** ", el cual usa RTTI para asegurar que el molde es correcto y lanza una **ClassCastException** si usted realizó un moldeado malo.
2. El objeto **Class** , representando el tipo de su objeto. El objeto **Class** puede ser

consultado para obtener información útil en tiempo de ejecución.

En C++, el molde clásico " **(Shape)** " no realiza RTTI. Simplemente le dice al compilador que trate el objeto como si fuera del tipo especificado. En Java, donde se realiza control de tipos, este molde es a menudo llamado "molde de tipo hacia abajo seguro" (type safe downcast).

La razón para el término "molde hacia abajo" (downcast) es el ordenamiento histórico del diagrama de jerarquía de clases. Si moldear un **Circle** a un **Shape** es un molde hacia arriba (upcast), entonces moldear un **Shape** a un **Circle** es un molde hacia abajo (downcast). Por supuesto, usted sabe que un **Circle** es además un **Shape** y el compilador permite realizar una asignación de molde hacia arriba libremente, pero no sabe si un **Shape** es necesariamente un **Circle** , de manera que el compilador no permite realizar una asignación de molde hacia abajo sin usar un molde explícito.

Hay una tercera forma de RTTI en Java. Esta es la palabra clave **instanceof** que le dice si un objeto es una instancia de un tipo particular. Retorna un **boolean** , de manera que puede usarlo en forma de pregunta, de esta forma:

```
if(x instanceof Dog)
   ((Dog)x).bark();
```

La sentencia **if** anterior controla si el objeto **x** pertenece a la clase **Dog** *antes* de aplicar a **x** un molde a **Dog** . Es importante usar **instanceof** antes de realizar un molde hacia abajo cuando usted no tiene otra información que le diga el tipo del objeto; de otra manera, terminará con una **ClassCastException** .

Comunmente, podría capturar un tipo (triángulos para volverlos púrpura, por ejemplo), pero también podría fácilmente etiquetar todos los objetos usando **instanceof** .

Suponga que tiene una familia de clases **Pet** :

```
//: c12:Pets.java
class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}
class Counter { int i; } ///:~
```

La clase **Counter** es usada para llevar el rastro de la cantidad de un tipo particular de **Pet** . Podría pensar en él como un **Integer** que puede ser modificado.

Usando **instanceof** se pueden contar todos las mascotas:

```
//: c12:PetCount.java
// Usando instanceof.
import java.util.*;
public class PetCount {
   static String[] typenames = {
      "Pet", "Dog", "Pug", "Cat",
      "Rodent", "Gerbil", "Hamster",
   };
   // Las excepciones son lanzadas a la consola:
   public static void main(String[] args)
   throws Exception {
      ArrayList pets = new ArrayList();
      try {
         Class[] petTypes = {
            Class.forName("Dog"),
            Class.forName("Pug"),
            Class.forName("Cat"),
            Class.forName("Rodent"),
            Class.forName("Gerbil"),
            Class.forName("Hamster"),
         };
         for(int i = 0; i < 15; i++)
         pets.add(
            petTypes[
            (int)(Math.random()*petTypes.length)]
            .newInstance());
      } catch(InstantiationException e) {
         System.err.println("No se puede instanciar");
         throw e;
      } catch(IllegalAccessException e) {
         System.err.println("No se puede acceder");
         throw e;
      } catch(ClassNotFoundException e) {
         System.err.println("No se puede encontrar la clase");
         throw e;
      }
      HashMap h = new HashMap();
      for(int i = 0; i < typenames.length; i++)
         h.put(typenames[i], new Counter());
      for(int i = 0; i < pets.size(); i++) {
         Object o = pets.get(i);
      if(o instanceof Pet)
         ((Counter)h.get("Pet")).i++;
```

```
      if(o instanceof Dog)
         ((Counter)h.get("Dog")).i++;
      if(o instanceof Pug)
         ((Counter)h.get("Pug")).i++;
      if(o instanceof Cat)
         ((Counter)h.get("Cat")).i++;
      if(o instanceof Rodent)
         ((Counter)h.get("Rodent")).i++;
      if(o instanceof Gerbil)
         ((Counter)h.get("Gerbil")).i++;
      if(o instanceof Hamster)
         ((Counter)h.get("Hamster")).i++;
   }
   for(int i = 0; i < pets.size(); i++)
   System.out.println(pets.get(i).getClass());
   for(int i = 0; i < typenames.length; i++)
      System.out.println(
         typenames[i] + " cantidad: " +
         ((Counter)h.get(typenames[i])).i);
   }
} ///:~
```

Hay una restricción más bien ligada a **instanceof** : puede usarlo para comparar con un tipo nombrado solamente y no con un objeto **Class** . En el ejemplo anterior, usted puede sentir que es tedioso escribir todas esas expresiones **instanceof** y estaría en lo correcto. Pero no hay forma de automatizar ingeniosamente **instanceof** creando un **ArrayList** de objetos **Class** y realizando la comparación con ellos, en cambio (manténgase en sintonía - verá una alternativa). Esto no es una gran restricción, como usted podría pensar, porque eventualmente usted entenderá que su diseño es probablemente defectuoso si termina escribiendo muchas expresiones **instanceof** .

Por supuesto, este ejemplo es inventado - probablemente usted ponga un miembro estático en cada tipo y lo incremente en el constructor para seguir el rastro de las cantidades. Usted podría hacer algo como esto *si* tiene control sobre el código fuente de la clase y puede cambiarlo. Ya que este no es siempre el caso, RTTI puede resultar conveniente.

Usando literales de clase

Es interesante ver cómo el ejemplo **PetCount.java** puede ser escrito nuevamente usando literales de clase. El resultado es más claro, en muchos sentidos:

```
//: c12:PetCount2.java
// Usando literales de clase.
import java.util.*;
```

```
public class PetCount2 {
   public static void main(String[] args)
   throws Exception {
      ArrayList pets = new ArrayList();
      Class[] petTypes = {
         // Literales de clase:
         Pet.class,
         Dog.class,
         Pug.class,
         Cat.class,
         Rodent.class,
         Gerbil.class,
         Hamster.class,
      };
      try {
         for(int i = 0; i < 15; i++) {
            // Desplazamiento en 1 para
            // eliminar a Pet.class:
            int rnd = 1 + (int)(
               Math.random() * (petTypes.length - 1));
            pets.add(
               petTypes[rnd].newInstance());
         }
      } catch(InstantiationException e) {
         System.err.println("No se puede instanciar");
         throw e;
      } catch(IllegalAccessException e) {
         System.err.println("No de puede acceder");
         throw e;
      }
      HashMap h = new HashMap();
      for(int i = 0; i < petTypes.length; i++)
         h.put(petTypes[i].toString(),
            new Counter());
      for(int i = 0; i < pets.size(); i++) {
         Object o = pets.get(i);
         if(o instanceof Pet)
            ((Counter)h.get("clase Pet")).i++;
         if(o instanceof Dog)
            ((Counter)h.get("clase Dog")).i++;
         if(o instanceof Pug)
            ((Counter)h.get("clase Pug")).i++;
         if(o instanceof Cat)
            ((Counter)h.get("clase Cat")).i++;
         if(o instanceof Rodent)
            ((Counter)h.get("clase Rodent")).i++;
         if(o instanceof Gerbil)
            ((Counter)h.get("clase Gerbil")).i++;
         if(o instanceof Hamster)
            ((Counter)h.get("clase Hamster")).i++;
```

```
      }
      for(int i = 0; i < pets.size(); i++)
         System.out.println(pets.get(i).getClass());
      Iterator keys = h.keySet().iterator();
      while(keys.hasNext()) {
         String nm = (String)keys.next();
         Counter cnt = (Counter)h.get(nm);
         System.out.println(
            nm.substring(nm.lastIndexOf('.') + 1) +
            " cantidad: " + cnt.i);
      }
   }
} ///:~
```

Aquí, el arreglo **typenames** ha sido removido a favor de obtener las cadenas con nombre de tipo a partir del objeto **Class** . Note que el sistema puede distinguir entre clases e interfaces.

Puede ver además que la creación de **petTypes** no necesita ser rodeada por un bloque **try** ya que es evaluada en tiempo de compilación y de esta forma no lanzará ninguna excepción, a diferencia de **Class.forName( )** .

Cuando los objetos **Pet** son creados dinámicamnete, usted puede ver que el número aleatorio está restringido entre uno y **petTypes.length** y no incluye el cero. Esto es porque cero se refiere a **Pet.class** y probablemente un **Pet** genérico no es interesante. Por supuesto, ya que **Pet.class** es parte de **petTypes** , el resultado es que todas las mascotas son contadas.

Un instanceof dinámico

El método **isInstance** de **Class** provee una forma de llamar dinámicamente al operador **instanceof** . Así, todas esas tediosas sentencias **instanceof** pueden ser removidas en el ejemplo **PetCount** :

```
//: c12:PetCount3.java
// Usando isInstance().
import java.util.*;
public class PetCount3 {
   public static void main(String[] args)
   throws Exception {
      ArrayList pets = new ArrayList();
      Class[] petTypes = {
         Pet.class,
         Dog.class,
         Pug.class,
         Cat.class,
         Rodent.class,
```

```
            Gerbil.class,
            Hamster.class,
         };
         try {
            for(int i = 0; i < 15; i++) {
               // Desplazamiento en 1 para
               // eliminar a Pet.class:
               int rnd = 1 + (int)(
                  Math.random() * (petTypes.length - 1));
               pets.add(
                  petTypes[rnd].newInstance());
            }
         } catch(InstantiationException e) {
            System.err.println("No se puede instanciar");
            throw e;
         } catch(IllegalAccessException e) {
            System.err.println("No se puede acceder");
            throw e;
         }
         HashMap h = new HashMap();
         for(int i = 0; i < petTypes.length; i++)
            h.put(petTypes[i].toString(),
               new Counter());
         for(int i = 0; i < pets.size(); i++) {
            Object o = pets.get(i);
            // Usando isInstance para eliminar
            // expresiones instanceof individuales:
            for (int j = 0; j < petTypes.length; ++j)
               if (petTypes[j].isInstance(o)) {
                  String key = petTypes[j].toString();
                  ((Counter)h.get(key)).i++;
               }
         }
         for(int i = 0; i < pets.size(); i++)
            System.out.println(pets.get(i).getClass());
         Iterator keys = h.keySet().iterator();
         while(keys.hasNext()) {
            String nm = (String)keys.next();
            Counter cnt = (Counter)h.get(nm);
            System.out.println(
               nm.substring(nm.lastIndexOf('.') + 1) +
               " cantidad: " + cnt.i);
         }
      }
} ///:~
```

Usted puede ver que el método **isInstance( )** ha eliminado la necesidad de expresiones **instanceof** . En suma, esto significa que usted puede agregar nuevos tipos de mascotas simplemente cambiando el arreglo **petTypes** ; el resto del

programa no necesita modificación (la cual si necesita cuando se usan expresiones **instanceof**).

Equivalencia instanceof vs. Class

Cuando se consulta acerca de información de tipo hay una diferencia importante entre ambas formas de **instanceof** (esto es, **instanceof** e **isInstance( )** , los cuales producen los mismos resultados) y la comparación directa de los objetos **Class** . Aquí hay un ejemplo que demuestra la diferencia:

```
//: c12:FamilyVsExactType.java
// La diferencia entre instanceof y class
class Base {}
class Derived extends Base {}
public class FamilyVsExactType {
   static void test(Object x) {
      System.out.println("Probando x de tipo " +
         x.getClass());
      System.out.println("x instanceof Base " +
         (x instanceof Base));
      System.out.println("x instanceof Derived " +
         (x instanceof Derived));
      System.out.println("Base.isInstance(x) " +
         Base.class.isInstance(x));
      System.out.println("Derived.isInstance(x) " +
         Derived.class.isInstance(x));
      System.out.println(
         "x.getClass() == Base.class " +
         (x.getClass() == Base.class));
      System.out.println(
         "x.getClass() == Derived.class " +
         (x.getClass() == Derived.class));
      System.out.println(
         "x.getClass().equals(Base.class)) " +
         (x.getClass().equals(Base.class)));
      System.out.println(
         "x.getClass().equals(Derived.class)) " +
         (x.getClass().equals(Derived.class)));
   }
   public static void main(String[] args) {
      test(new Base());
      test(new Derived());
   }
} ///:~
```

El método **test( )** realiza el control de tipo con su argumento usando ambas formas de **instanceof** . Entonces, obtiene la referencia a **Class** y luego usa

equals( ) y == para probar la igualdad de los objetos **Class** . Aquí está la salida:

```
Testing x of type class 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 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
```

De modo tranquilizador, **instanceof** e **isInstance( )** producen exactamente los mismos resultados, como también ocurre con **equals( )** y ==. Pero las pruebas mismas bosquejan distintas conclusiones. Manteniéndose con el concepto de tipo, **instanceof** dice "¿está usted en esta clase o en una clase derivada de esta?". Por otro lado, si usted compara los objetos **Class** actuales usando ==, no hay consideración con la herencia - es el tipo exacto o no lo es.

Sintaxis RTTI

Java ejecuta RTTI usando un objeto **Class** , aún si estamos haciendo algo como un molde. La clase **Class** cuenta con un número de otros caminos en los que podemos usar RTTI.

Primero, debemos hacer referencia a el objeto **Class** apropiado. Una forma de hacer esto, como se muestra en el ejemplo anterior, es usar un **String** y el método **Class.forName()** . Esto es conveniente porque no necesitamos un objeto de ese tipo a fin de hacer referencia a la clase. Por supuesto, si ya se tiene un objeto del tipo que nos interesa, podemos hacer referencia al objeto **Class** llamando a un método que es parte de la clase raíz **Object** : **getClass()** . Este retorna la referencia a **Class** que representa el tipo actual del objeto. **Class** tiene muchos métodos interesantes, como se muestra en el siguiente ejemplo:

```
//: c12:ToyTest.java
```

```
// Probando la clase Class.
interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
   // Comente el siguiente constructor
   // por defecto para ver
   // NoSuchMethodError from (*1*)
   Toy() {}
   Toy(int i) {}
}
class FancyToy extends Toy
implements HasBatteries,
Waterproof, ShootsThings {
   FancyToy() { super(1); }
}
public class ToyTest {
   public static void main(String[] args) throws Exception {
      Class c = null;
      try {
         c = Class.forName("FancyToy");
      } catch(ClassNotFoundException e) {
         System.err.println("No puedo encontrar FancyToy");
         throw e;
      }
      printInfo(c);
      Class[] faces = c.getInterfaces();
      for(int i = 0; i < faces.length; i++)
         printInfo(faces[i]);
      Class cy = c.getSuperclass();
      Object o = null;
      try {
         // Requiere constructor por defecto:
         o = cy.newInstance(); // (*1*)
      } catch(InstantiationException e) {
         System.err.println("No se puede instanciar");
         throw e;
      } catch(IllegalAccessException e) {
         System.err.println("No se puede acceder");
         throw e;
      }
      printInfo(o.getClass());
   }
   static void printInfo(Class cc) {
      System.out.println(
      "Nombre de la clase: " + cc.getName() +
      " es interfaz? [" +
      cc.isInterface() + "]");
   }
} ///:~
```

Podemos observar que la clase **FancyToy** es algo complicada, puesto que esta hereda de **Toy** e **implementa** las **interfaces HasBatteries** , **Waterproof** y **ShootsThings** . En **main( )** , una referencia a **Class** es creada e inicializada al **Class** de **FancyToy** usando **forName( )** dentro de un bloque **try** apropiado.

El método **Class.getInterfaces( )** retorna un arreglo de objetos **Class** que representa las interfaces que están contenidas en el objeto **Class** que nos interesa.

Si tenemos un objeto **Class** podemos preguntarle por la clase base directa usando **gerSuperclass( )** . Esto, por supuesto, retorna una referencia a **Class** a la cual también se puede consultar. Esto significa que, en tiempo de ejecución , podemos descubrir la jerarquía de clases completa de un objeto.

El método **newInstance( )** de **Class** puede, al principio, parecer sólo otra forma de **clone( )** . Sin embargo, podemos crear un nuevo objeto con **newInstance( )** *sin* un objeto existente, como se vió aquí, porque no hay un objeto **Toy** - sólo un objeto **cy** , el cual es una referencia al objeto **Class** de **y** . Esta es una forma de implementar un "constructor virtual", el cual permite que digamos *"no se exactamente de que tipo eres, pero créate a ti mismo de forma apropiada"* . En el ejemplo citado anteriormente, **cy** es sólo una referencia a **Class** sin ningún tipo de información en tiempo de compilación. Y cuando creamos una nueva instancia, obtenemos una referencia a **Object** . Pero esta referencia apunta a un objeto **Toy** . Por supuesto, antes de poder enviar cualquier mensaje, aparte de los aceptados por **Object** , tenemos que investigar un poco y hacer algún tipo de molde. En suma, la clase que está siendo creada con **newInstance( )** debe tener un constructor por defecto. En la siguiente sección, verá cómo crear dinámicamente objetos de clases usando cualquier constructor, con la API de Java **reflection** .

El método final de la lista es **printInfo( )** , que toma una referencia a **Class** y obtiene el nombre con **getName( )** y determina si es una interfaz con **isInterface( )** .

La salida del programa es:

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

De esta manera, con el objeto **Class** podemos encontrar todo lo que queremos conocer acerca de un objeto.

Reflection: Información de la clase en tiempo de ejecución

Si no conocemos el preciso tipo de un objeto, RTTI nos lo puede decir.

Por supuesto, hay una limitación: el tipo debe ser conocido en tiempo de compilación a fin de poder ser detectado usando RTTI y hacer algo útil con la información. Para ponerlo de otra forma, el compilador debe saber acerca de todas las clases con la que estamos trabajando para usar RTTI.

Esto no parece mucha limitación al principio, pero supongamos que debemos dar una referencia a un objeto que no está en el ámbito de nuestro programa. De hecho, ni siquiera la clase del objeto está disponible en tiempo de compilación de nuestro programa. Por ejemplo, supongamos que tomamos un número de bytes de un archivo en el disco o de una conexión de red y decimos que esos bytes representan una clase. Dado que el compilador no conoce la clase mientras compila el código, ¿cómo es posible que haga uso de la misma?

En un ambiente de programación tradicional esto parece un escenario lejano. Pero a medida que nos movemos dentro del amplio mundo de la programación aparecen casos importantes donde esto ocurre. El primero es la programación basada en componentes, en la cual podemos construir proyectos usando *herramientas rápidas de desarrollo de aplicaciones* (Rapid Aplication Development (RAD)) . Esto es una metodología visual para crear un programa (el cual se puede ver en la pantalla como "formulario") moviendo de íconos que representan componentes sobre el formulario. Estos componentes son entonces configurados colocando algunos valores en tiempo de programación. Esta configuración en tiempo de diseño requiere que todo componente sea instanciable, exponga parte de sí mismo y permita que estos valores sean leídos y establecidos. En suma, los componentes que manejan eventos GUI deben dar a conocer información acerca de métodos apropiados, de manera que el entorno RAD pueda asistir al programador en la tarea de sobreescribir estos método que manejan eventos. Reflection provee el mecanismo para detectar los métodos disponibles y producir los nombres de los métodos. Java nos da una estructura basada en programación de componentes a través de JavaBeans (descripta en el capítulo 13).

Otra motivación urgente para descubrir información de las clases en tiempo de ejecución es proveer la capacidad de crear y ejecutar objetos en plataformas remotas a través de la red. Esto es llamado *Invocación Remota de Métodos* (Remote Method Invocation(RMI)) el cual permite a un programa Java tener objetos distribuidos a través de varias máquinas. Esta distribución puede ocurrir por muchas razones: por ejemplo, si estamos haciendo tareas de cálculo intensivo y queremos dividir el trabajo y ponerlo sobre máquinas que están libres con el fin de acelerar las cosas. En algunas situaciones podrímos querer colocar parte del código que maneje tipos particulares de tareas (ej. Reglas de Negocio en una arquitectura de cliente/servidor) en una máquina particular, de manera que la

misma se convierta en un repositorio común que describe esas acciones y puede se cambiado fácilmente, afectando a todo el sistema. (Este es un desarrollo interesante, ya que la máquina existe sólo para hacer cambios al software fácilmente!) En este sentido, la computación distribuida soporta además hardware especializado que podría ser bueno para una tarea en particular -inversión de matrices, por ejemplo- pero inapropiado o muy costoso para programación de propósito general.

La clase **Class** (descripta previamente en este capítulo) soporta el concepto de *reflection* y hay una librería adicional, **java.lang.reflect** , con clases **Field** , **Method** , y **Constructor** (cada una de los cuales implementa la interfaz **Member**). Los objetos de ese tipo son creados por la JVM en tiempo de ejecución para representar el miembro correspondiente de la clase desconocida. Podemos usar luego los **Constructors** para crear nuevos objetos, el método **get( )** y **set( )** para leer y modificar los campos asociados con objetos de tipo **Field** , y el método **Invoke( )** para llamar métodos asociados con objetos del tipo **Method** . En suma, podemos llamar a conveniencia los métodos **getField( )** , **getMethods( )** , **getConstructors( )** , etc., para retornar arreglos de objetos representando los campos, métodos y constructores. (Podemos encontrar más buscando en la documentación en línea de la clase **Class**) De esta manera la información de clase de objetos anónimos puede ser determinada completamente en tiempo de ejecución, y no necesitamos saber nada de ellos en tiempo de compilación.

Esto es importante para caer en cuenta de no hay nada mágico acerca de reflection. Cuando usemos reflection para interactuar con un objeto del cual no conocemos el tipo, la JVM simplemente mirará el objeto y verá que pertenece a una clase particular (como el RTTI común) pero entonces, antes de que pueda hacer algo más, el objeto **Class** debe ser cargado. De esta manera, el archivo **.class** para ese tipo particular todavía debe estar disponible para la JVM, ya sea en el equipo local o a través de la red. De esta forma, la verdadera diferencia entre RTTI y reflection es que con RTTI el compilador abre y examina el archivo **.class** en tiempo de compilación. Para decirlo de otro modo, podemos llamar métodos de un objeto en una forma "normal". Con reflection, el archivo **.class** no está disponible en tiempo de compilación; este es abierto y examinado en tiempo de ejecución.

Un extractor de métodos de clase

Con poca frecuencia necesitará usar las herramientas de reflection directamente, ellos están en el lenguaje para dar apoyo a otras características de Java, como es la serialización de objetos (Capítulo 11), JavaBeans(Capítulo13) y RMI (Capítulo 15). Por supuesto, hay momentos en los que es útil ser capaz de extraer dinámicamente información acerca de una clase. Una herramienta extremadamente útil es el extractor de métodos de clase. Como se mencionó antes, observando la definición de la clase en el código fuente o en la documentación en línea vemos sólo los métodos definidos o sobreescritos *dentro de la definición de esa clase* . Pero podría haber docenas de métodos disponibles

que provienen de las clases bases. Localizarlos es muy tedioso y consume tiempo. Afortunadamente, reflection provee una forma de escribir una herramienta simple que nos mostrará automáticamente la interfaz completa.

Así es como funciona:

```
//: c12:ShowMethods.java
// Usando reflection para mostrar todos los métodos de
// una clase, incluso si los métodos están definidos en
// la clase base.
import java.lang.reflect.*;
public class ShowMethods {
   static final String usage =
   "uso: \n" +
   "ShowMethods qualified.class.name\n" +
   "Para mostrar todos los métodos en una clase o: \n" +
   "ShowMethods qualified.class.name palabra\n" +
   "Para buscar todos lo métodos que contengan 'palabra'";
   public static void main(String[] args) {
      if(args.length < 1) {
         System.out.println(usage);
         System.exit(0);
      }
      try {
         Class c = Class.forName(args[0]);
         Method[] m = c.getMethods();
         Constructor[] ctor = c.getConstructors();
         if(args.length == 1) {
            for (int i = 0; i < m.length; i++)
               System.out.println(m[i]);
            for (int i = 0; i < ctor.length; i++)
               System.out.println(ctor[i]);
         } else {
            for (int i = 0; i < m.length; i++)
               if(m[i].toString().indexOf(args[1])!= -1)
                  System.out.println(m[i]);
            for (int i = 0; i < ctor.length; i++)
               if(ctor[i].toString().indexOf(args[1])!= -1)
                  System.out.println(ctor[i]);
         }
      } catch(ClassNotFoundException e) {
         System.err.println("No existe la clase: " + e);
      }
   }
} ///:~
```

Los métodos de **Class** , **getMethods( )** y **getConstructors( )** retornan un

arreglo de **Method** y **Constructor** , respectivamente. Cada una de estas clases tienen más métodos para disecar los nombres, argumentos y valores que retornan los métodos que representan. Pero también podemos hacer uso de **toString( )** , como se hizo aquí, para producir un **String** con la firma competa del método. El resto de código es sólo para extraer información de la línea de comando, determinando si una firma particular concuerda con el **String** objetivo (usando **indexof( )**) e imprimiendo los resultados.

Esto muestra a reflection en acción, ya que el resultado elaborado por **Class.forName()** no puede ser conocido en tiempo de compilación, y por consiguiente, toda la informació de firma de método es extraída en tiempo de ejecución. Si investigamos la documentación en línea sobre reflection, veremos que hay apoyo suficiente para establecer y hacer llamadas a un método sobre un objeto que es totalmente desconocido en tiempo de compilación (veremos más ejemplos luego en este libro). Otra vez, esto es algo que podríamos no necesitar hacer nunca -el apoyo existe para RMI y para que el entorno de programación pueda manipular JavaBeans- pero es interesante.

Un experimento interesante es correr

```
java ShowMethods ShowMethods
```

Esto produce una lista que incluye un constructor por defecto público, aunque podamos ver desde el código que no hay ningún constructor definido. El constructor que vemos es unos de los que el compilador genera automáticamente. Si después hacemos **ShowMethods** a una clase no pública (que es amigable), el constructor por defecto generado no se muestra más en la salida. El constructor por defecto generado tiene automáticamente el mismo acceso que la clase.

La salida para **ShowMethods** es aún un poco tediosa. Por ejemplo, esta es una porción de la salida producida por invocar a **java ShowMethods java.lang.String** :

```
java.lang.String:
public boolean
java.lang.String.startsWith(java.lang.String,int)
public boolean
java.lang.String.startsWith(java.lang.String)
public boolean
java.lang.String.endsWith(java.lang.String)
```

Podría ser mejor incluso si los calificadores como **java.lang** fueran desmenuzados. La clase **StreamTokenizer** introducida en el capítulo anterior puede ayudar a

crear una herramienta para resolver este problema:

```
//: com:bruceeckel:util:StripQualifiers.java
package com.bruceeckel.util;
import java.io.*;
public class StripQualifiers {
   private StreamTokenizer st;
   public StripQualifiers(String qualified) {
      st = new StreamTokenizer(
         new StringReader(qualified));
      st.ordinaryChar(' '); // Mantenga los espacios
   }
   public String getNext() {
      String s = null;
      try {
         int token = st.nextToken();
         if(token != StreamTokenizer.TT_EOF) {
            switch(st.ttype) {
               case StreamTokenizer.TT_EOL:
                  s = null;
                  break;
               case StreamTokenizer.TT_NUMBER:
                  s = Double.toString(st.nval);
                  break;
               case StreamTokenizer.TT_WORD:
                  s = new String(st.sval);
                  break;
               default: // Caracter simple en ttype
                  s = String.valueOf((char)st.ttype);
            }
         }
      } catch(IOException e) {
         System.err.println("Error en token");
      }
      return s;
   }
   public static String strip(String qualified) {
      StripQualifiers sq =
         new StripQualifiers(qualified);
      String s = "", si;
      while((si = sq.getNext()) != null) {
         int lastDot = si.lastIndexOf('.');
         if(lastDot != -1)
         si = si.substring(lastDot + 1);
         s += si;
      }
      return s;
   }
```

```
} ///:~
```

Para facilitar la reutilización, esta clase es colocada en **com.bruceeckel.util** . Como se puede ver, usa **StreamTokenizer** y manipulación de **String** para hacer su trabajo.

La nueva versión del programa usa la clase anterior para limpiar la salida:

```
//: c12:ShowMethodsClean.java
// ShowMethods con los calificadores quitados
// para hacer los resultados más fáciles de leer.
import java.lang.reflect.*;
import com.bruceeckel.util.*;
public class ShowMethodsClean {
   static final String usage =
      "uso: \n" +
      "ShowMethodsClean qualified.class.name\n" +
      "Para mostrar todos los métodos en una clase o: \n" +
      "ShowMethodsClean qualif.class.name palabra\n" +
      "Para buscar todos lo métodos que contengan 'palabra'";
      public static void main(String[] args) {
         if(args.length < 1) {
            System.out.println(usage);
            System.exit(0);
         }
         try {
            Class c = Class.forName(args[0]);
            Method[] m = c.getMethods();
            Constructor[] ctor = c.getConstructors();
            // Convierte a un arreglo de Strings limpias:
            String[] n =
               new String[m.length + ctor.length];
            for(int i = 0; i < m.length; i++) {
               String s = m[i].toString();
               n[i] = StripQualifiers.strip(s);
            }
            for(int i = 0; i < ctor.length; i++) {
               String s = ctor[i].toString();
               n[i + m.length] =
                  StripQualifiers.strip(s);
            }
            if(args.length == 1)
               for (int i = 0; i < n.length; i++)
                  System.out.println(n[i]);
            else
               for (int i = 0; i < n.length; i++)
                  if(n[i].indexOf(args[1])!= -1)
```

```
                System.out.println(n[i]);
            } catch(ClassNotFoundException e) {
               System.err.println("No existe la clase: " + e);
            }
    }
} ///:~
```

La clase **ShowMethodsClean** es bastante similar a **ShowMethods** , excepto en que toma los arreglos de **Method** y **Constructor** y los convierte en un único arreglo de **String** . Cada uno de esos objetos **String** es entonces pasado a través **StripQualifiers.Strip( )** para remover toda la calificación del método.

Esta herramienta puede realmente ahorrar tiempo mientras se está programando, cuando no pueda recordar si una clase tiene un método particular y no quiere caminar a través de la jerarquía de clases en la documentación en línea, o si no se sabe que se puede hacer con esa clase, por ejemplo, con los objetos **Color** .

El Capítulo 13 contiene una versión GUI de este programa (personalizado para extraer información para componentes Swing) de manera que puede dejarlo corriendo mientras escribe código y permitir un vistazo rápido.

Resumen

RTTI permite descubrir infromación de tipo a partir de una referencia a una clase base anónima. Así, se presta para ser mal usado por parte de los novicios, ya que podría ser utilizado en lugar de las llamadas polimórficas. Para muchas personas provenientes del paradigma procedural, es difícil no organizar sus programas como un conjunto de sentencias **switch** . Estas personas podrían hacer esto con RTTI y perder de esta forma el importante valor del polimorfismo en el desarrollo y mantenimiento de código. La intención de Java es que se use llamadas a métodos polimórficos en su código, y usar RTTI sólo cuando se deba.

Por supuesto, usar llamadas a métodos polimórficos requiere que se tenga el control de la definición de la clase base porque en algún punto de la extensión de su programa podría descubrir que la clase base no incluye el método que necesita. Si la clase base proviene de una librería o es de alguna forma controlada pro otra persona, una solución al problema es RTTI: puede heredar un nuevo tipo y agregar un nuevo método. En el resto del código se puede detectar el tipo particular e invocar al método especial. Esto no destruye el polimorfismo y la extensibilidad del programa porque agregar un tipo nuevo no requerirá buscar sentencias **switch** en su código. Por supuesto, cuando agregue código en el cuerpo principal que requiera las características nuevas, se deberá usar RTTI.

Poner una característica en una clase base podría significar que, para el beneficio de una clase particular, todas las otras clases derivadas de esa clase base deberán tener algún método sin sentido. Esto hace que la interfaz sea menos clara y molesta a quienes deben sobrescribir métodos abtractos cuando derivan de esta

clase base. Por ejemplo, considere una jerarquía de clases que representa insturmentos musicales. Suponga que quiere limpiar las válvulas de todos los instrumetos apropiados en su orquesta. Una opción es colocar un método **clearSpitValve( )** en la case base **Instrument** , pero esto es confuso porque implica que los instrumentos **Percussion** y **Electronic** también tienen válvulas. RTTI provee una solución mucho más razonable en este caso porque se puede colocar el método en la clase específica (**Wind** en este caso), donde es apropiado. Por supuesto, una solución más apropiada es colocar un método **prepareInstrument( )** en la clase base, pero podría no verlo de esta forma cuando resuelve el problema por primera vez y erróneamente asumir que se debe usar RTTI.

Finalmente, RTTI resolverá a veces problemas de eficiencia. Si su código usa bien el polimorfismo, pero ocurre que uno de sus objetos reaccionan ante este código de propósito general en una forma horriblemente ineficiente, usted puede detectar ese tipo usando RTTI y escribir código específico para el caso a fin de mejorar la eficiencia. Es una trampa seductora. Es mejor hacer que el programa funcione primero y, entonces, decidir si está corriendo lo suficientemente rápido y, sólo entonces, se debería atacar las cuestiones de eficiencia.

Ejercicios

Las soluciones a los ejercicios elegidos pueden encontrarse en el documento electrónico The Thinking in Java Annotated Solution Guide, disponibles por un pequeño monto en www.BruceEckel.com.

1. Agregue **Rhomboid** a **Shapes.java** . Construya un **Rhomboid** , aplíquele un molde hacia arriba a **Shape** , luego aplíquele un molde hacia abajo a **Rhomboid** . Intente aplicar un molde hacia abajo a **Circle** y vea que pasa.
2. Modifique el ejercicio 1 para que use **instanceof** para controlar el tipo antes de realizar el molde hacia abajo.
3. Modifique **Shapes.java** para que se pueda "remarcar" (establecer una bandera) en todas las figuras de un tipo particular. El método **toString( )** para cada **Shape** derivada debería indicar si la figura está "remarcada".
4. Modifique **SweetShop.java** para que cada tipo de creación de objeto sea controlada por un argumento de la línea de comando. Esto es, si su línea de comando es "java SweetShop Candy", entonces sólo el objeto **Candy** es creado. Note cómo usted puede controlar que objetos **Class** son cargados vía el argumento de la línea de comando.
5. Agregue un nuevo tipo de **Pet** a **PetCount3.java** . Verifique que es creado y contado de forma correcta en **main( )** .
6. Escriba un método que tome un objeto e impima recursivamente todas las clases en la jerarquía del dicho objeto
7. Modifique el ejercicio 6 para que use **Class.getDeclaredFields( )** para mostrar además información acerca de los campos en una clase.

8. En **ToyTest.java** , comente el constructor por defecto de **Toy** y explique qué ocurre.
9. Incorpore un nuevo tipo de interfaz en **ToyTest.java** y verifique es es detectada y mostrada en forma correcta
10. Cree un nuevo tipo de contenedor que use un **ArrayList** privado para contener los objetos. Capture el tipo del primer objeto que colocó y luego permítale al usuario insertar objetos sólo de este tipo.
11. Escriba un programa para detemrinar si un arreglo de **char** es un tipo primitivo o un objeto verdadero.
12. Implemente **clearSpitValve( )** como se describió en el resumen.
13. Implemente el método **rotate(Shape)** descripto en este capítulo, de manera tal que controle para ver si está rotando un **Circle** (y, de ser así, no realice la operación).
14. Modifique el ejercicio 6 para que use reflection en lugar de RTTI
15. Modifique el ejercicio 7 para que use reflection en lugar de RTTI
16. En **ToyTest.java** , use reflection para crear un objeto **Toy** usando un constructor que no sea por defecto.
17. Busque la interfaz de **java.lang.Class** en la documentación HTML de Java de *java.sun.com* . Escriba un programa que tome el nombre de una clase como un argumento de la línea de comando, luego use los métodos de **Class** para volcar toda la información disponible para esa clase. Pruebe su programa con una clase de la librería estándar y con una clase creada por usted.

11: Almacenando sus objetos

Un programa realmente simple es el que tiene sólo una cantidad fija de objetos con tiempos de vida conocidos.

En general, nuestros programas podrán crear nuevos objetos basados en algunos criterios que podrán ser conocidos solo en tiempo de ejecución. Hasta el tiempo de ejecución no se sabrá la cantidad exacta ni el tipo de objetos que necesitaremos. Para resolver este problema general de la programación, necesitamos crear un numero de objetos en algún momento y en algún lugar. Por tanto, podemos fiarnos de crear un manejador para almacenar cada uno de nuestros objetos

```
MyObject myHandle;
```

ya que, después de esto, nunca sabremos cuantos de estos objetos necesitaremos.

Para resolver este problema esencial, Java tiene varios métodos para almacenar objetos (o mejor dicho, manejadores de objetos). El tipo construido es el array, que ha sido comentado antes y cuyo conocimiento ampliaremos en este capitulo. Asimismo, las utilidades de la librería de Java poseen *clases colecciones* (también conocidas como *contenedores* de clases aunque, como este termino se maneja en AWT, nos referiremos a ellas con la palabra "colección") que proporcionan métodos más sofisticados para almacenar y nivelar la manipulación de sus objetos. Esto puede resumir el objetivo de este capitulo.

Arrays

La mayor parte de la introducción a los arrays se encuentra en la ultima sección del capitulo 4, que mostraba como definir e inicializar un array. El almacenamiento de objetos es el enfoque de este capitulo, y un array es justamente un camino para almacenar objetos. Pero existen otros métodos para almacenar objetos, por tanto, ¿qué hace a un array tan especial?.

Hay 2 motivos que distinguen a los arrays de otros tipos de colecciones: eficiencia y tipo. El array es el método más eficiente que Java proporciona para almacenar y acceder a una secuencia de objetos (realmente, manejadores de objetos). El array es una simple secuencia lineal, que hace el acceso de elementos rápido, pero hemos de pagar por esta velocidad: cuando creamos un array de objetos su

tamaño es fijo y no podemos cambiarlo durante la vida del array.

Usted podría sugerir crear un array de un tamaño exacto y luego, si le falta espacio, crear uno nuevo y mover todos los manejadores desde el viejo hasta el nuevo. Este es el comportamiento de la clase **Vector** , que estudiaremos más tarde en este mismo capitulo. Sin embargo, aun con esta aparente flexibilidad de tamaño, un **Vector** es considerablemente menos eficiente que un array.

La clase **vector** de C++ conoce el tipo de objetos que posee, pero tiene un inconveniente cuando se compara con los arrays en Java: el operador [] de los vectores de C++ no comprueba los límites en las consultas, por lo que podríamos acceder mas allá del limite final. (Es posible, sin embargo, preguntar como de grande es el vector, y que el método **at()** chequee su limite). En Java, usted podemos obtener el limite chequeando independientemente de dónde estemos usando el array o la colección, y Java nos contestara con una **RuntimeException** si excedemos dicho límite. Como estudiaremos en el capítulo 9, este tipo de excepciones indican un error de programación, de manera que no necesitamos buscarlo en el código. Como un caso aparte, la razón de que el **vector** de C++ no chequee el limite en cada acceso es la rapidez. En Java tenemos la seguridad de que constantemente se lleva a cabo la comprobación de los limites tanto en los arrays como en las colecciones.

Las otras clases colección genérica que estudiaremos en este capitulo, **Vector** , **Stack** , y **Hashtable** , tratan todas los objetos como si no se especificaran sus tipos. Ellos los tratan como de tipo **Object** , la clase padre de todas las clases en Java. Este trabajo es elegante desde un punto de vista: si necesitamos construir una colección, podremos incluir cualquier objeto de Java en esa colección. (Excepto para primitivas, que pueden ser sustituidas en las colecciones como constantes usando primitivas Java envolviendo clases, o como valores cambiables para envolver sus propias clases.) Este es el segundo motivo en el que un array es superior a las colecciones genéricas: cuando creamos un array, lo creamos para almacenar un tipo especifico. Esto significa que en tiempo de compilación se conoce el tipo para evitar la introducción de tipos erróneos, dando errores cuando usemos tipos erróneos. Por supuesto, Java puede prevenirnos de enviar mensajes inapropiados a un objeto, ya sea en tiempo de compilación o en tiempo de ejecución. Por tanto no es un método arriesgado, es mas preciso si los puntos de compilación están fuera de nuestro alcance, mas rápido en tiempo de ejecución, y hay menor probabilidad de que el usuario final se vea sorprendido por una excepción.

Por eficiencia y comprobación de tipo siempre merece la pena intentar usar un array si se puede. Sin embargo, cuando intentemos resolver un problema mas general, los arrays también pueden ser restrictivos. Después de ver los arrays, el resto de este capitulo estará dedicado a las clases colección proporcionadas por Java.

Los arrays son objetos de primera clase

Indiferentemente del tipo de array con el que trabaje, el identificador de array es realmente un manejador de un verdadero objeto que se crea en la pila. El objeto de la pila puede ser creado implícitamente, como parte de la sintaxis de inicialización del array, o bien explícitamente con una expresión **new** . Parte del objeto pila (en realidad, el único campo o método al que podemos acceder) es la variable miembro **length** que nos indica cuantos elementos pueden ser almacenados en un objeto array. La sintaxis '[]' es el otro único modo de acceso de que usted dispone en el objeto array.

Los siguientes ejemplos muestran distintas formas en las que un array puede ser incializado, y como podemos asignar el manejador del array a diferentes objetos del array. También muestran que los arrays de objetos y los arrays de primitivas son también idénticos en su uso. La única diferencia es que los arrays de objetos almacenan los manejadores mientras que los arrays de primitivas almacenan los valores de las primitivas directamente. (Vea la pagina 94 si tiene problemas al ejecutar este programa.)

```
//: ArraySize.java
// Initialization & re-assignment of arrays
package c08;

class Weeble {} // A small mythical creature

public class ArraySize {
  public static void main(String[] args) {
  // Arrays of objects:
    Weeble[] a; // Null handle
    Weeble[] b = new Weeble[5]; // Null handles
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      c[i] = new Weeble();
    Weeble[] d = {
      new Weeble(), new Weeble(), new Weeble()
    };
    // Compile error: variable a not initialized:
    //!System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // The handles inside the array are
    // automatically initialized to null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);
    // Java 1.1 initialization syntax:
    a = new Weeble[] {
```

```
      new Weeble(), new Weeble()
    };
    System.out.println("a.length = " + a.length);

    // Arrays of primitives:
    int[] e; // Null handle
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Compile error: variable e not initialized:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // The primitives inside the array are
    // automatically initialized to zero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    // Java 1.1 initialization syntax:
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
  }
} ///:~
```

A continuación se muestra la salida del programa:

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
```

```
h.length = 3
e.length = 3
e.length = 2
```

El array **a** es iniciado a un manejador **null** , y el compilador se encarga de que dicho manejador no sea utilizado hasta que no haya sido correctamente iniciado. El array **b** es iniciado para almacenar un conjunto de 5 elementos de manejadores del tipo **Weeble** , pero dichos elementos aun no han sido introducidos en el array. Sin embargo, ahora podemos preguntar por la longitud del array, ya que este esta reservado para un tipo concreto de objeto. Esto conlleva un pequeño inconveniente: nosotros no podemos conocer cuántos elementos hay actualmente en el array, ya que lo que indica **length** es el tamaño del objeto array, no el número de elementos que actualmente posee. Sin embargo, cuando se crea un objeto array, sus manejadores son automáticamente inicializados a **null** por lo que podremos ver si un elemento particular de un array (un slot, una posición del array) contiene un elemento comprobando si es o no **null** . De forma parecida, un array de primitivas es automáticamente iniciado a cero para los tipos numéricos, **null** para los **char** y **false** para los **boolean** .

El array **c** muestra la creación de un objeto array seguido por la asignación de sus elementos (que en este caso son objetos **Weeble**) para todas sus posiciones o slots. El array **d** muestra la sintaxis de "inicialización agregada" que provoca que el objeto array sea creado (implícitamente con **new** en la cabecera, tal como el array c) e inicializado con objetos Weeble, todo ello en una sola línea.

La expresión

```
a = d;
```

muestra cómo podemos obtener un manejador que esta ligado a un objeto array y le asignamos otro objeto array, del mismo modo a como lo hacemos con cualquier otro tipo de manejador de objeto. Ahora, ambos, **a** y **d** , son apuntan al mismo objeto array en la cabecera.

Java 1.1 añade una nueva sintaxis para la iniciación de arrays, la cual puede ser conocida como una "inicialización dinámica agregada". Dicha iniciación usada por el array **d** debe ser usada en el punto de definición de **d** porque, con la sintaxis de Java 1.1, podemos crear e iniciar un objeto array en cualquier lugar. Por ejemplo, supongamos que **hide()** es un método que toma un array de objetos **Weeble** . Podríamos invocarlo de la forma siguiente:

```
hide (d);
```

Pero en Java 1.1 también podemos crear dinámicamente el array que deseemos pasar como parámetro:

```
hide (new Weeble[] { new Weeble(), new Weeble() });
```

Esta nueva sintaxis proporciona una forma más conveniente de escribir código en muchas situaciones.

La segunda parte del ejemplo anterior muestra que los arrays de primitivas funcionan como objetos array excepto que los arrays de primitivas poseen los valores primitivos directamente.

Colecciones de primitivas

La clases colección sólo pueden tener manejadores de objetos. Un array, sin embargo, puede ser creado para poseer primitivas directamente, así como manejadores de objetos. Es posible usar las clases "wrapper" como datos **Integer** , **Double** , etc. para almacenar los valores primitivos dentro de una colección pero, como veremos más tarde en este capítulo, en el ejemplo **WordCount.java** , las clases wrapper son solamente útiles en algunos casos. Si introducimos primitivas en arrays o las envolvemos en una clase que esta colocada en una colección es una cuestión de eficiencia. Es mucho más eficiente crear y acceder a un array de primitivas que a una colección de primitivas encubiertas.

Por supuesto, si estamos usando un tipo primitivo y necesitamos la flexibilidad de una colección que automáticamente se expande cuando se necesita mas espacio, el array no funcionará y nos veremos forzados a usar una colección de primitivas encubiertas. Tal vez piense que debería haber un tipo especializado de tipo Vector para cada uno de los tipos primitivos, pero Java no le proporciona esto. Tal vez algún día se proporcione algún mecanismo de plantillas que proporcione una mejor forma para que Java controle este problema. [1]

---

[1]Este es uno de los casos donde C++ es superior a Java, ya que C++ soporta tipos parametrizados con la palabra clave **template** .

Devolviendo un array

Supongamos que estamos escribiendo un método y no queremos devolver una sola cosa, sino un grupo de cosas al mismo tiempo. Lenguajes como C y C++ hace esto dificultoso porque en ellos no se puede devolver fácilmente un array, solo un puntero a un array. Esto introduce problemas porque puede generar complicaciones a la hora de controlar el tiempo de vida de un array, lo cual puede provocar fácilmente pérdidas de direcciones de memoria.

Java proporciona una aproximación similar, pero nos permite "devolver un array" directamente. Realmente, lo que se devuelve es un manejador de un array, pero con Java nunca tendremos que preocuparnos sobre la responsabilidad para los arrays. Este podrá ser tan grande como necesite, y el recolector de basura podrá limpiarlo cuando usted haya acabado.

Como ejemplo, consideremos devolver un array de objetos de tipo String:

```
//: IceCream.java
// Returning arrays from methods
public class IceCream {
  static String[] flav = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  static String[] flavorSet(int n) {
  // Force it to be positive & within bounds:
    n = Math.abs(n) % (flav.length + 1);
    String[] results = new String[n];
    int[] picks = new int[n];
    for(int i = 0; i < picks.length; i++)
      picks[i] = -1;
    for(int i = 0; i < picks.length; i++) {
      retry:
      while(true) {
        int t =
          int)(Math.random() * flav.length);
        for(int j = 0; j < i; j++)
          if(picks[j] == t) continue retry;
        picks[i] = t;
        results[i] = flav[t];
        break;
      }
    }
    return results;
  }
```

```
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String[] fl = flavorSet(flav.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

El método flavorSet() crea un array de String llamado results. El tamaño de este array es n, determinado por el argumento pasado al método. Después se procede a seleccionar los sabores aleatoriamente del array flav y se colocan en el array de resultados, que es finalmente devuelto. Devolver un array es como devolver otro objeto cualquiera, ya que es un manejador. No es importante que el array se creó con flavorSet(), o dónde se creó. El recolector de basura tiene cuidado de limpiar el array cuando hayamos terminado de usarlo, y el array permanecerá tanto tiempo como lo necesite.

Como caso aparte, observe que cuando flavorSet() selecciona sabores aleatoriamente, garantiza que un elemento seleccionado no ha sido elegido anteriormente. Esto se realiza de forma parecida a un **while** infinito que selecciona elementos aleatorios hasta que encuentra uno que no haya sido introducido anteriormente. (Por supuesto, podríamos también realizar una comparación de **String** para ver si la selección aleatoria se encuentra ya en el array de resultados, pero la comparación entre **String** s es ineficiente.) Si esto se realiza correctamente se añade la entrada y se selecciona el siguiente (se incrementa **i**). Pero si **t** es un numero que ya ha sido introducido, se usa una etiqueta de continuación para saltar dos elementos hacia atrás para generar un nuevo **t** a seleccionar. Es recomendable especialmente ver lo que aquí ocurre usando una ejecución paso a paso con el depurador.

La función **main()** escribe 20 tipos de sabores, como puede ver que **flavorSet()** selecciona los sabores en orden aleatorio cada vez. Es mas fácil de ver si redirecciona la salida a un fichero. Mientras observa el fichero, recuerde que no tiene hambre. (Quiere un helado, pero no lo necesita.)

Colecciones

Para resumir lo que hemos visto antes, sus principios, recordemos que elecciones más eficientes de objetos pueden almacenarse como un grupo en un array, en el que controlamos los elementos que introduciremos mediante primitivas. En el resto del capítulo vamos a ver que, en la mayoría de los casos, cuando estemos empezando a escribir un programa, no sabremos cuántos objetos vamos a necesitar, o si necesitaremos un medio mas sofisticado para almacenar y tratar su objetos. Java proporciona cuatro tipos de clases colección para resolver este

problema: **Vector** , **BitSet** , **Stack** y **Hashtable** . Aunque comparado a otros lenguajes que proveen colecciones esto es un escaso suministro, puede no obstante resolver un impresionante número de problemas utilizando estas herramientas.

Entre otras características, **Stack** , por ejemplo, implementa una secuencia LIFO (Last-In, First-Out), y **HashTable** es un array asociativo que nos permite asociar un objeto con algún otro objeto. Las colecciones de clases de Java pueden cambiarse de tamaño a ellas mismas. Es decir, usted introducirá un numero de objetos y no tendrá que preocuparse de la cantidad de espacio que será necesaria mientras está escribiendo el programa.

Inconveniente: tipo desconocido

El inconveniente de usar las colecciones de Java es que al introducir un objeto en la colección perdemos información. Esto ocurre porque, cuando la colección fue creada, el programador no tenía ni idea de qué tipo de objetos necesitaríamos introducir en ella, y escribir una colección sólo para un tipo concreto haría que no fuese una herramienta de uso general Por tanto, en lugar de eso, las colecciones mantienen manejadores de objetos de tipo **Object** , a la que pertenecen todos los objetos de Java, ya que es la clase padre o raíz de todas las demás. (Por supuesto, esto no incluye tipos primitivos, ya que estos no son heredados por nadie.) Esto es una gran solución, excepto por estas razones:

1. Como la información de tipo se pierde cuando se introduce un manejador **object** en la colección, podríamos introducir en la colección algunos tipos de objeto no deseados. Alguien puede introducir un perro en una colección en la que sólo queremos que haya gatos.
2. Como se pierde la información de tipo, la única cosa que conoce la colección es el manejador de un objeto **Object** . Por tanto, tendremos que crear un molde para comprobar el tipo antes de usarlo.

En el lado opuesto (ventajas), Java no nos permite hacer un mal uso de los objetos que introducimos en una colección. Si introducimos un perro en una colección de gatos y seguimos adelante, intentando tratar a ese elemento como un gato, obtendremos una excepción cuando trabaje con el perro. De forma parecida, si intenta hacer un molde al manejador de perros, puede detener la introducción de éste en la colección de gatos, obteniendo una excepción en tiempo de ejecución.

Aquí tiene un ejemplo:

```
//: CatsAndDogs.java
// Simple collection example (Vector)
import java.util.*;
```

```
class Cat {
  private int catNumber;
  Cat(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}

class Dog {
  private int dogNumber;
  Dog(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}

  public class CatsAndDogs {
    public static void main(String[] args) {
    Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.elementAt(i)).print();
    // Dog is detected only at run-time
  }
} ///:~
```

Como podemos comprobar, usar un vector es sencillo: se crea uno, se le poner objetos usando **addElement()** y sacarlos usando **elementAt()** . (Observe que **Vector** tiene el método **size()** para indicarle cuántos elementos han sido introducidos, por lo que tenemos que preocuparnos por el número de elementos que podemos introducir en el vector. Cuando se sobrepase se producirá una excepción).

Las clases **Cat** (gato) y **Dog** (perro) son distintas. No tienen nada en común excepto que son **Objects** . (Si no se especifica de qué clase se hereda, se hereda automáticamente de **Object** .) La clase **Vector** , que proviene de **java.util** , almacena **Objects** , por lo que, para almacenar sólo datos de la clase Cat en su interior no es suficiente con usar **addElement()** , ya que también se podrían introducir objetos **Dog** tanto en tiempo de ejecución como de compilación. Cuando vamos a extraer lo que supone que es un objeto **Cat** de la clase **Vector** usando el

método **elementAt()** , obtendremos un manejador de **Object** al que deberemos hacer un casting a **Cat** . Para ello debemos envolver toda la expresión entre paréntesis para forzar la evaluación del cast o molde antes de llamar al método **print()** de la clase **Cat** , ya que de otra forma podría recibir un mensaje de error de sintaxis. Luego, en tiempo de ejecución, cuando se intenta hacer un molde (cast) a un objeto **Dog** con un objeto **Cat** , obtendríamos una excepción.

Esto es más que un inconveniente. Es algo más que crear alguna dificultad para encontrar bugs. Si una parte (o varias) de un programa introducen objetos en una colección, y descubrimos una excepción de que un objeto incorrecto se ha introducido en la colección en una única parte del programa, debemos encontrar donde se produjo la introducción incorrecta. Para ello, debemos examinar el código, la peor herramienta de depuración que existe. Por el contrario, es conveniente comenzar con algunas clases estandarizadas para programación, a pesar de su escasez y poca eficacia.

Algunas veces, sin embargo, funciona

En algunos casos las cosas parecen funcionar correctamente sin hacer casting al tipo original. El primer caso es completamente especial: la clase **String** posee ayuda extra del compilador para hacerla mas llevadera. Siempre que el compilador espera un objeto **String** y no lo recibe, llama automáticamente al método **toString()** que es definido en la clase **Object** y puede ser sustituido por una clase Java. Este método provoca el deseo del objeto String, que es luego usado cada vez que este es buscado.

Por tanto, todo lo que necesitamos hacer para que los objetos de nuestra clase se escriban es sobreescribir el método **toString()** , como se muestra en el siguiente ejemplo:

```
//: WorksAnyway.java
// In special cases, things just seem
// to work correctly.
import java.util.*;

class Mouse {
private int mouseNumber;
  Mouse(int i) {
    mouseNumber = i;
  }
  // Magic method:
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  void print(String msg) {
    if(msg != null) System.out.println(msg);
    System.out.println(
```

```
    "Mouse number " + mouseNumber);
  }
}

class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // Cast from Object
    mouse.print("Caught one!");
  }
}

public class WorksAnyway {
  public static void main(String[] args) {
    Vector mice = new Vector();
    for(int i = 0; i < 3; i++)
      mice.addElement(new Mouse(i));
    for(int i = 0; i < mice.size(); i++) {
      // No cast necessary, automatic call
      // to Object.toString():
      System.out.println(
        "Free mouse: " + mice.elementAt(i));
      MouseTrap.caughtYa(mice.elementAt(i));
    }
  }
} ///:~
```

Como puede observar **toString** se redefine en **Mouse** . En el segundo bucle for en main() puede encontrar la línea siguiente:

```
System.out.println("Free mouse: " + mice.elementAt(i));
```

Después del signo "+" el compilador espera encontrar un objeto **String** . **elementAt()** produce un **Object** . por lo que para obtener el deseado **String** , el compilador hace una llamada implícita a **toString()** . Desgraciadamente, solo puede trabajar de esta forma con los **String** ; esta técnica no se encuentra disponible para otros tipos.

Dentro de **Mousetrap** se ha colocado un segundo enfoque a la ocultación del casting. El método **caughtYa()** acepta no un Mouse, sino un **Object** , que es el casting de un **Mouse** . Esto es bastante presuntuoso, de acuerdo, ya que aceptando un Object, cualquiera puede ser pasado al método. Sin embargo, si el casting es incorrecto, si pasamos un tipo incorrecto, obtendremos una excepción en tiempo de ejecución. Esto no es tan bueno como el chequeo en tiempo de compilación, pero aún es robusto. Obsérvelo en el uso de este método:

```
MouseTrap.caughtYa(mice.elementAt(i));
```

No es necesario un casting.

Haciendo un Vector consciente del tipo

Aún no hemos terminado esta cuestión. Una solución más es crear una nueva clase usando **Vector** , tal que él sólo aceptará y producirá su tipo:

```
//: GopherVector.java
// A type-conscious Vector
import java.util.*;

class Gopher {
private int gopherNumber;
  Gopher(int i) {
    gopherNumber = i;
  }
  void print(String msg) {
    if(msg != null) System.out.println(msg);
    System.out.println(
      "Gopher number " + gopherNumber);
  }
}

class GopherTrap {
  static void caughtYa(Gopher g) {
    g.print("Caught one!");
  }
}

class GopherVector {
private Vector v = new Vector();
  public void addElement(Gopher m) {
    v.addElement(m);
  }
  public Gopher elementAt(int index) {
    return (Gopher)v.elementAt(index);
  }
  public int size() { return v.size(); }
  public static void main(String[] args) {
  GopherVector gophers = new GopherVector();
    for(int i = 0; i < 3; i++)
      gophers.addElement(new Gopher(i));
    for(int i = 0; i < gophers.size(); i++)
      GopherTrap.caughtYa(gophers.elementAt(i));
```

```
  }
} ///:~
```

Esto es similar al ejemplo anterior, excepto que la nueva clase GopherVector tiene una cabecera privada de tipo **Vector** (heredar de **Vector** puede ser frustrante, por motivos que estudiaremos más adelante), y métodos como los de **Vector** . Sin embargo, no acepta ni produce **Objects** genéricos, solo objetos **Gopher** .

Como **GopherVector** sólo aceptara un **Gopher** si tuviéramos que decir:

```
gophers.addElement(new Pigeon());
```

podríamos obtener un mensaje de error en tiempo de compilación. Esta aproximación, mientras que es más tediosa desde el punto de vista del código, puede decirle inmediatamente si esta usando un tipo incorrectamente.

Observe que no es necesario el casting cuando usa **elementAt()** , es siempre un Gopher.

Tipos parametrizados

Este tipo de problema no es un caso aislado, hay numerosos casos en los que necesitara crear nuevos tipos basados en otros tipos, y en los cuales es útil poseer información especifica en tiempo de compilación. Este es el concepto de tipo parametrizados. En C++, esto es directamente soportado por el lenguaje en plantillas. En un punto, Java ha reservado la palabra clave **generic** para algún día soportar tipos parametrizados, pero no se sabe si esto ocurrirá.

Enumerators (iteradores)

En algunas colecciones de clases, debemos conocer la forma de colocar las cosas en un orden adecuado para luego recuperarlas. Después de todo, este es el trabajo principal de una colección, almacenar cosas. En la clase Vector, addElement() es el camino para insertar objetos, y elementAt() es una de las posibilidades para sacarlos. La clase Vector es totalmente flexible, en la medida en que usted puede seleccionar una cosa cada vez, y seleccionar múltiples elementos de una vez utilizando diferentes índices.

Si deseamos pensar a un nivel superior, tenemos un inconveniente: hay que conocer el tipo exacto de la colección para poder usarla. Esto puede no parecer malo en principio, pero si acabamos por usar un **Vector** , y más tarde decidimos, por eficiencia, ¿qué ocurre si quisiéramos cambiarlo por una lista (objeto **List**) que es parte de la librería de colecciones de Java 1.2?. O si desearais escribir un bloque de código que no conozca o se preocupe de con que colección está

trabajando.

El concepto de un iterador puede ser usado para realizar el siguiente nivel de abstracción. Este es un objeto cuyo trabajo es moverse a través de una secuencia de objetos y seleccionar cada objeto siguiendo esta secuencia sin que el programador tenga que conocer o tener en cuenta la estructura sobre la que se trabaja. Además, un iterador es normalmente denominado objeto de "peso ligero"; es decir, barato de crear. Por esta razón, con frecuencia encontraremos extraños comportamientos en los iteradores; por ejemplo, algunos iteradores pueden moverse sólo en una dirección.

El Java **Enumeration** [2] es un ejemplo de iterador con este tipo de comportamientos. No se puede hacer muchas cosas con ellos excepto:

1. Preguntar a una colección para devolverle un **Enumeration** usando un método llamado **elements()**. Este **Enumeration** estará listo para devolverle su primer elemento en cuanto usted haga su primera llamada al método **nextElement()**.
2. Obtener el siguiente objeto en la secuencia con **nextElement()**.
3. Ver si hay algún objeto más en la secuencia con **hasMoreElements()**.

Esto es todo. Es una implementación simple, pero potente, de un iterador. Para ver cómo funciona, examinaremos el programa **CatsAndDogs.java** visto antes en este capitulo. En la versión original, utilizamos el método **elementAt()** para seleccionar cada elemento, pero en la siguiente versión modificada se usaremos una enumeración:

---

[2] El termino iterador es común en C++ y en la OOP (Object Oriented Programming o Programación Orientada a Objetos), por lo que es difícil de entender como el equipo de desarrollo de Java le ha dado un nombre tan extraño. La librería de colecciones de Java 1.2 soluciona este problema además de muchos otros.

```
//: CatsAndDogs2.java
// Simple collection with Enumeration
import java.util.*;

class Cat2 {
private int catNumber;
  Cat2(int i) {
    catNumber = i;
  }
  void print() {
    System.out.println("Cat number " +catNumber);
  }
}

class Dog2 {
private int dogNumber;
  Dog2(int i) {
    dogNumber = i;
  }
  void print() {
    System.out.println("Dog number " +dogNumber);
  }
}

public class CatsAndDogs2 {
  public static void main(String[] args) {

  Vector cats = new Vector();
    for(int i = 0; i < 7; i++)
      cats.addElement(new Cat2(i));
    // Not a problem to add a dog to cats:
    cats.addElement(new Dog2(7));
    Enumeration e = cats.elements();
    while(e.hasMoreElements())
      ((Cat2)e.nextElement()).print();
    // Dog is detected only at run-time
  }
} ///:~
```

Como puede ver, el único cambio está en las últimas líneas. En vez de:

```
for(int i = 0; i < cats.size(); i++)
  ((Cat)cats.elementAt(i)).print();
```

se usa una enumeración para pasar a la siguiente secuencia:

```
while(e.hasMoreElements())
  ((Cat2)e.nextElement()).print();
```

Con una enumeración no tenemos que preocuparnos del número de elementos que hay en la colección. Esto se hace con los métodos **hasMoreElements()** y **nextElement()** .

Como otro ejemplo, se considera la creación de un método de impresión de propósito general:

```
//: HamsterMaze.java
// Using an Enumeration
import java.util.*;

class Hamster {
private int hamsterNumber;
  Hamster(int i) {
    hamsterNumber = i;
  }
  public String toString() {
    return "This is Hamster #" + hamsterNumber;
  }
}

class Printer {
  static void printAll(Enumeration e) {
    while(e.hasMoreElements())
      System.out.println(
    e.nextElement().toString());
  }
}

public class HamsterMaze {
  public static void main(String[] args) {
  Vector v = new Vector();
    for(int i = 0; i < 3; i++)
      v.addElement(new Hamster(i));
    Printer.printAll(v.elements());
  }
} ///:~
```

Mire atentamente el método de impresión:

```
static void printAll(Enumeration e) {
```

```
  while(e.hasMoreElements())
    System.out.println(e.nextElement().toString());
}
```

Observe que aquí no hay información sobre el tipo de secuencia. Todo lo que tenemos es una enumeración, y es todo lo que necesitamos saber sobre la secuencia es que podemos coger el siguiente objeto, y que podemos saber dónde esta el final. Esta idea de coger una colección de objetos y pasar a través de ella para realizar una operación con cada uno de sus elementos es muy potente y la revisaremos a lo largo de todo este libro.

Este ejemplo en particular es con frecuencia más genérico, en la medida en que usa el omnipresente método **toString()** (omnipresente sólo porque es parte de la clase **Object**). Una forma alternativa de llamar al método de impresión (aunque probablemente algo menos eficiente, si detecta las diferencias) es:

```
System.out.println(""+e.nextElement());
```

que usa la "conversión automática a **String** " que esta implementada en Java. Cuando el compilador ve un **String** , seguido por un '+', espera otro **String** a continuación y llama al método **toString()** automáticamente. (En Java 1.1, el primer **String** es innecesario; cualquier objeto será convertido a un **String**). También podemos realizar una conversión explícita, casting, que tiene el efecto de llamar a **toString()** :

```
System.out.println((String)e.nextElement());
```

En general, sin embargo, querremos hacer algo más que llamar a los métodos **Object** , por lo que ejecutaremos de nuevo la conversión explícita de tipos. Debemos suponer que hemos conseguido una enumeración para una secuencia de un tipo particular en el que estamos interesado, y le hacemos un casting al resultado de este tipo de objetos (obteniendo una excepción en tiempo de ejecución si hay algún error).

Tipos de colecciones

La librería estándar de Java 1.0 y 1.1 viene con una cantidad mínima de colecciones de clases, aunque es suficiente para realizar la mayoría de nuestros proyectos de programación. (Como veremos al final de este capítulo, Java 1.2 incluye una librería de colecciones radicalmente rediseñada y perfeccionada).

Vector

La clase **Vector** , como ya ha visto, es bastante simple de usar. Aunque la mayoría de las veces usaremos **addElement()** para insertar objetos, **elementAt ()** para obtener uno en un momento dado, y **Elements()** para obtener una enumeración para una secuencia, existen también otra serie de métodos que pueden ser útiles. Como es usual con las librerías de Java, no las usaremos ni hablaremos de ellas al completo. Si alguien quiere conocer algún detalle adicional, puede consultar la documentación electrónica que viene con la distribución.

Rompiendo Java

Las colecciones estándar de Java contienen un método **toString()** , por lo que pueden producir una representación de **String** propias, incluidos los objetos que almacenan. Dentro de un **Vector** , por ejemplo, **toString()** pasa a través de los elementos del vector y realiza llama a **toString()** para cada uno de ellos. Supongamos que desea imprimir la dirección de su clase. Puede tener sentido hacer referencia a **this** (en particular, los programadores de C++ son propensos a este aproximación):

```
//: CrashJava.java
// One way to crash Java
import java.util.*;

public class CrashJava {
  public String toString() {
    return "CrashJava address: " + this + "\n";
  }
  public static void main(String[] args) {
  Vector v = new Vector();
    for(int i = 0; i < 10; i++)
      v.addElement(new CrashJava());
    System.out.println(v);
  }
} ///:~
```

Si lo que se hace es crear un objeto **CrashJava** e imprimirlo, lo que obtendremos es una secuencia encadenada de finalización de excepciones. Sin embargo, si almacenamos el objeto **CrashJava** en un **vector** y mostramos el vector como se ha mostrado, éste puede manejarlo y no obtendremos ninguna excepción; Java habrá quebrado. (Pero al menos no habrá colgado su sistema.) Esto fue comprobado con Java 1.1.

Lo que esta ocurriendo es que hay una conversión automática a tipo **String** . Cuando decimos:

```
"CrashJava address: " + this
```

El compilador ve un **String** seguido de un signo "+" y algo que no es un **String** , por lo que intenta convertirlo a **String** . Hace esta conversión llamando a **toString ()** , que produce llamadas recursivas. Cuando esto ocurre dentro de un vector, aparecerá el famoso mensaje de desbordamiento de la pila sin que el mecanismo de control de excepciones pueda dar una respuesta.

Si realmente desea imprimir la dirección del objeto en este caso, la solución es llamar al método **toString()** del objeto Object, el cual hace exactamente eso. Por tanto, en lugar de poner **this** , habrá que poner **super.toString()** . (Esto sólo funciona si estamos heredando directamente de un **Object** o si ninguna de sus clases padre han sobrescrito el método **toString()**).

BitSet

Un **BitSet** es realmente un **Vector** de bits, y se usa cuando deseamos almacenar eficientemente mucha información del tipo on-off (verdadero-falso). Es eficiente solo desde el punto de vista de su tamaño; si lo que deseamos es un acceso eficiente, hay que considerar que **BitSet** es un poco más lento que usar un array de un tipo nativo. Además, el tamaño mínimo de un BitSet tiene una longitud: 64 bits. Esto implica que vamos a almacenar algo pequeño, por ejemplo de 8 bits, un BitSet será un gasto inútil, y crear su propia clase de almacenamiento de sus datos sería la solución acertada. En un **vector** normal, la colección puede expandirse añadiendo mas elementos. El **BitSet** no permite esto tampoco. Es decir, a veces funciona y a veces no, lo que hace parecer que el **BitSet** de Java 1.0 no esta bien implementado. (Esta corregido en Java 1.1.) El siguiente ejemplo muestra como trabaja el **BitSet** y demuestra el bug de la versión 1.0:

```
//: Bits.java
// Demonstration of BitSet
import java.util.*;

public class Bits {
  public static void main(String[] args) {
  Random rand = new Random();
  // Take the LSB of 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);
    System.out.println("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);
      System.out.println("short value: " + st);
      printBitSet(bs);
      int it = rand.nextInt();
      BitSet bi = new BitSet();
      for(int i = 31; i >=0; i--)
        if(((1 << i) &  it) != 0)
          bi.set(i);
        else
          bi.clear(i);
        System.out.println("int value: " + it);
        printBitSet(bi);

    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    // Without the following, an exception is thrown
    // in the Java 1.0 implementation of BitSet:
    //    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
  static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
} ///:~
```

El generador de números aleatorios se utiliza para crear un byte aleatorio, corto, y entero, y cada uno de ellos es transformado mediante patrones de bits para almacenarlos en un **BitSet** . Esto funciona porque un **BitSet** es de 64 bits, y ninguno de ellos supera este tamaño. Pero en Java 1.0, cuando el BitSet es mayor de 64 bits, ocurre algo extraño. Si establecemos un conjunto más grande que el

último mas recientemente acumulado, se expandirá correctamente. Pero si intentamos fijar los bits de las direcciones superiores a éstas sin comprobar primero el limite, obtendremos una excepción, ya que el **BitSet** no se expandirá correctamente en Java 1.0. El ejemplo muestra un **BitSet** de 512 bits creándose. El constructor asigna memoria para el doble del numero de bits. Entonces, si intentamos fijar el bit 1024 o superior sin fijar primero el bit 1023, Java 1.0 lanzará una excepción. Afortunadamente, esto esta solucionado en Java 1.1, pero evite usar el BitSet si va a escribir codigo en Java 1.0.

Stack

Un **Stack** es a veces denominado "colección LIFO" ("last-in, first-out"). Esto significa que lo que introduzcamos (push) en el final de la pila, será la primera que podamos sacar de ella (pop). Como todas las demás colecciones de Java, lo que introducimos y sacamos de la cola son **Objects** , por lo que deberemos hacer un casting a lo que saquemos.

Es curioso es que en vez de usar un **Vector** como constructor de bloques para crear un **Stack** , el **Stack** es heredado del **Vector** . Por tanto tiene todas las características y comportamiento de un Vector y alguna conducta extra propia del **Stack** . Es difícil saber si los diseñadores explícitamente decidieron que esta era una forma especialmente útil para hacer cosas, o si fue solo un diseño <na've>.

Aqui tiene una simple demostración de Stack que lee cada línea de un array y la introduce en el como un String:

```
//: Stacks.java
// Demonstration of Stack Class
import java.util.*;

public class Stacks {
static String[] months = {
     "January", "February", "March", "April",
     "May", "June", "July", "August", "September",
     "October", "November", "December" };
  public static void main(String[] args) {
  Stack stk = new Stack();
    for(int i = 0; i < months.length; i++)
      stk.push(months[i] + " ");
    System.out.println("stk = " + stk);
    // Treating a stack as a Vector:
    stk.addElement("The last line");
    System.out.println("element 5 = " + stk.elementAt(5));
    System.out.println("popping elements:");
    while(!stk.empty())
      System.out.println(stk.pop());
  }
```

```
} ///:~
```

Cada línea del array months es insertada en el Stack con push(), y después es sacado de ella con pop(). Para hacer un puntero, las operaciones de los Vector están también implementadas en los objetos Stack. Esto es posible porque, por la virtud de la herencia, un Stack es un Vector. De este modo, todas las operaciones que pueden ser realizadas en un Vector pueden ser también realizadas sobre un Stack, tal como elementAt().

Hashtable

Un Vector le permite seleccionar un objeto de una lista usando un numero, por lo que en este sentido asocia números a objetos. Pero, ¿qué ocurriría si usted desea seleccionar un objeto de entre una secuencia de ellos usando cualquier otro criterio?. Un Stack es un ejemplo: su criterio de selección es "el ultimo elemento introducido en el stack." Una potente modificación de esta idea de "selección en una secuencia" es llamada alternativamente un mapa, un diccionario o un array asociativo. Conceptualmente, es como un vector, pero en vez de acceder a los objetos usando un numero, accede a través de otro objeto. Esto es frecuentemente utilizado en un programa. El concepto mostrado, en Java es la clase abstracta Dictionary. El interfaz para esta clase se muestra a continuación: size() le indica cuantos elementos caben, isEmpty() es verdadero si no hay elementos, put(Object key, Object value) añade un objeto (el que usted desee) y lo asocia con una clave (key).get(Object key) obtiene la correspondiente clave, y remove(Object key) elimina la pareja valor-clave de la lista. Aquí hay enumeraciones: keys() produce una Enumeration de las claves, y elements() produce una Enumeration de todos los valores. Todo esto es un diccionario (Dictionary).

Un Dictionary no es terriblemente dificultoso de implementar. Aquí tiene una simple aproximación, que usa dos vectores, uno para teclas y otro para valores.

```
//: AssocArray.java
// Simple version of a Dictionary
import java.util.*;

public class AssocArray extends Dictionary {
private Vector keys = new Vector();
private Vector values = new Vector();
  public int size() { return keys.size(); }
  public boolean isEmpty() {
    return keys.isEmpty();
  }
  public Object put(Object key, Object value) {
    keys.addElement(key);
```

```
    values.addElement(value);
    return key;
  }
  public Object get(Object key) {
  int index = keys.indexOf(key);
  // indexOf() Returns -1 if key not found:
    if(index == -1) return null;
    return values.elementAt(index);
  }
  public Object remove(Object key) {
  int index = keys.indexOf(key);
    if(index == -1) return null;
    keys.removeElementAt(index);
    Object returnval = values.elementAt(index);
    values.removeElementAt(index);
    return returnval;
  }
  public Enumeration keys() {
    return keys.elements();
  }
  public Enumeration elements() {
    return values.elements();
  }
  // Test it:
  public static void main(String[] args) {
  AssocArray aa = new AssocArray();
    for(char c = 'a'; c <= 'z'; c++)
      aa.put(String.valueOf(c),
    String.valueOf(c).toUpperCase());
    char[] ca = { 'a', 'e', 'i', 'o', 'u' };
    for(int i = 0; i < ca.length; i++)
      System.out.println("Uppercase:" +
                         aa.get(String.valueOf(ca[i])));
  }
} ///:~
```

Lo primero que usted ve en la definición de AssocArray es "extends Dictionary". Esto significa que AssocArray es un tipo de diccionario, por lo que podrá hacer las mismas peticiones que a un Dictionary. Si usted crea su propio diccionario, como se ha hecho aquí, todo lo que necesita hacer es completar todos los métodos del diccionario. (Y debe sustituir todos los métodos porque todos ellos -a excepción del constructor- son abstractos.)

Los Vectores keys (claves) y values (valores) están asociados mediante un numero índice común. Esto significa que si usted llama a put() con una clave de "roof" y un valor de "blue" (suponiendo que ha asociado varias partes de una casa con los colores en los que serán pintados) y ya hay 100 elementos en AssocArray, "roof" será el elemento 101 de claves y "blue" será el elemento 101 de valores. Y si observa al get(), cuando le pasa "roof" como argumento key, este produce el

elemento índice con keys.indexOf(), y usa este índice numérico para obtener el valor asociado en el vector values.

El test en main() es simple: es solo un conversor de mapa de caracteres en minúscula a caracteres en mayúscula, que puede ser obviamente realizado de formas mas eficientes. Pero de esta forma se muestra que AssocArray es funcional.

La librería estándar de Java contiene solo una incorporación de un Dictionary, llamada Hashtable.[3] Hashtable de Java tienen el mismo interfaz básico y el mismo AssocArray (ya que todas heredan el Dictionary), pero se diferencian en un sentido distinto: eficiencia. Si usted observa lo que hace un get(), parece bastante lento para buscar a través de un Vector por la clave. Aquí donde Hashtable muestra su velocidad. En lugar de hacer una tediosa búsqueda lineal de la clave, usa un especial valor llamado un hash code. El hash code es una forma de obtener alguna información del objeto en cuestión y convertirlo en un entero "relativamente único" para ese objeto. Todos los objetos tendrán un hash code, y hashCode() es un método de la clase padre Object. Un Hashtable obtiene el hashCode() del objeto y lo usa para obtener rápidamente la clave. Esto resulta en una realización improvista.[1] La forma en que funciona un Hashtable esta mas allá del objetivo de este libro[2] Todo lo que necesita saber es que un Hashtable es un diccionario rápido, y que un diccionario es una potente herramienta.

---

[3] Si usted planea usar RMI (descrito en el Capitulo 15), debería saber que habría un problema cuando introduzca objetos remotos en una Hashtable. (Ver Core Java, por Cornell & Horstmann, Prentice-Hall 1997). [1] Si esta velocidad aun no satisface sus necesidades, usted puede acelerar mas el acceso a la tabla escribiendo su propia rutina para la tabla hash. Esto evita pausas para casting a y de Objects y la sincronización construida en las rutinas de tablas hash de Java Class Library. Para alcanzar mayores niveles de desarrollo, los entusiastas de la velocidad pueden usar el libro de Donald Knuth's, The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition, para reemplazar el desbordamiento de listas con arrays que tienen dos beneficios adicionales: pueden ser optimizados para las características del almacenamiento en disco y pueden encargarse la mayoría del tiempo de la creación y recolección de basura de registros individuales. [2] La mejor referencia que yo conozco es Practical Algorithms for Programmers, de Andrew Binstock y John Rex, Addison-Wesley 1995.

Como ejemplo del uso de un Hashtable, considere un programa para chequear la aleatoriedad del método Math.random() de Java. Idealmente, produciría una distribución perfecta de números aleatorios, pero para comprobar esto usted debería generar una gran cantidad de números aleatorios y meterlos en varios rangos. Un Hashtable es perfecto para esto, ya que asocia objetos con objetos (e este caso, los valores producidos porMaht.random() con el numero de veces que estos valores aparecen):

```
//: Statistics.java
// Simple demonstration of Hashtable
import java.util.*;

class Counter {
  int i = 1;
  public String toString() {
    return Integer.toString(i);
  }
}

class Statistics {
  public static void main(String[] args) {
    Hashtable ht = new Hashtable();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(ht.containsKey(r))
        ((Counter)ht.get(r)).i++;
      else
        ht.put(r, new Counter());
    }
    System.out.println(ht);
  }
} ///:~
```

En main(), cada vez que se genera un numero aleatorio, este es cubierto dentro de un objeto Integer por lo que ese manejador puede ser usado con la Hashtable. (Usted no puede usar una primitiva con una colección, solo un manejador de objeto.) El método containsKey() chequea para ver si esta clave se encuentra ya en la colección. (Es decir, ¿tiene el numero que ha sido encontrado ya?). Si es así, el método get() obtiene el valor asociado a la clave, que en este caso en un objeto Counter. El valor i que esta dentro del contado es luego incrementado para indicar que uno mas de este numero aleatorio concreto ha sido encontrado.

Si la clave no se ha encontrado todavía, el método put() introduce un nuevo par clave-valor en la Hashtable. Entonces Counter inicializa automáticamente su

variable i a 1 cuando es creado, lo que indica la primera ocurrencia de ese particular numero aleatorio.

Es muy fácil mostrar la Hashtable. Su método toString() se mueve a través de todos los pares clave-valor. El toString() de los Integer esta predefinido, y usted puede ver el toString() para el Counter. La salida de una ejecución es:

```
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
 13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
  7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
  0=505}
```

Usted quedara encantado con la necesidad de la clase Counter, que parece como si no tuviera todas la funcionalidad de las clases envolventes de Integer. ¿Por qué no usar int o Integer?. Bien, usted no puede usar un int porque todas las colecciones solo pueden almacenar manejadores de Object. Después de ver las colecciones envolventes, deberían comenzar a tomar un mayor sentido para usted, ya que no podrá escribir tipos primitivos de colecciones. Si embargo, lo único que usted puede hacer con las envolturas Java es inicializarlas a un valor y leer ese valor. Es decir, no hay forma de modificar un valor una vez que ha sido creado. Esto hace a la envoltura Integer totalmente ineficiente para resolver nuestro problema, por lo que nos veremos obligados a crear una nueva clase que satisfaga la necesidad.

Creando clases "key" (clases clave)

En el ejemplo anterior, se ha usado una clase de la librería estándar (Integer) como clave para una Hashtable. Esto ha funcionado bien como clave, porque tiene todo lo necesario para que trabaje correctamente como clave. Pero un insospechado peligro ocurre cuando se usan Hashtables en las que usted crea sus propias clases para usarlas como claves. Por ejemplo, considere un sistema de predicción meteorológica que asocia objetos Groundhog a objetos Prediction. Esto parece bastante bueno: usted crea las dos clases y usa Groundhog como la clave y Prediction como el valor.

```
//: SpringDetector.java
// Looks plausible, but doesn't work right.
import java.util.*;

class Groundhog {
int ghNumber;
  Groundhog(int n) { ghNumber = n; }
}
```

```
class Prediction {
boolean shadow = Math.random() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
}

public class SpringDetector {
  public static void main(String[] args) {
  Hashtable ht = new Hashtable();
    for(int i = 0; i < 10; i++)
      ht.put(new Groundhog(i), new Prediction());
    System.out.println("ht = " + ht + "\n");
    System.out.println(
                "Looking up prediction for groundhog #3:");
    Groundhog gh = new Groundhog(3);
    if(ht.containsKey(gh))
      System.out.println((Prediction)ht.get(gh));
  }
} ///:~
```

A cada Groundhog se le da un numero identificador, por lo que usted podrá ver una predicción en la Hashtable diciendo "Dame la predicción asociada con la marmota (groundhog) numero 3." La clase Predicition contiene un booleano (o lógico) que es inicializado usando Math.random(), y un método toString() que interpreta los resultados para usted. En main(), se rellena una Hashtable con Groundhog y sus Predictions asociadas. La Hashtable se imprime, de forma que usted puede comprobar que efectivamente se ha rellenado. Luego, un Groundhog con un numero identificador de 3, es usados para ver la predicción para Groundhog #3.

It seems simple enough, but it doesn't work. The problem is that Groundhog is inherited from the common root class Object (which is what happens if you don't specify a base class, thus all classes are ultimately inherited from Object). It is Object's hashCode( ) method that is used to generate the hash code for each object, and by default it just uses the address of its object. Thus, the first instance of Groundhog(3) does not produce a hash code equal to the hash code for the second instance of Groundhog(3) that we tried to use as a lookup.

Usted debe pensar que todo lo que necesita es escribir una apropiada sustitución para hashCode().Pero esto aun no funcionara hasta que no haga mas cosas: sobreescribir equals() que es también parte del Object. Este método es usado por Hashtabla cuando intenta determinar si su clave es igual a cualquier otra clave de la tabla. De nuevo, el defecto de Object.equals() es que simplemente compara direcciones de objetos, por lo que un Groundhog(3) no es lo mismo que otro

Groundhog(3).

Asi, para usar sus propias clases como claves en Hashtables, usted debe sobreescribir los métodos hashCode() y equals(), como se muestra en la siguiente solución al problema anterior:

```
//: SpringDetector2.java
// If you create a class that's used as a key in
// a Hashtable, you must override hashCode()
// and equals().
import java.util.*;

class Groundhog2 {
int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    if ((o != null) && (o instanceof Groundhog2))
      return ghNumber == ((Groundhog2)o).ghNumber;
    else return false;
  }
}

public class SpringDetector2 {
  public static void main(String[] args) {
  Hashtable ht = new Hashtable();
    for(int i = 0; i < 10; i++)
    ht.put(new Groundhog2(i),new Prediction());
    System.out.println("ht = " + ht + "\n");
    System.out.println(
                  "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
    if(ht.containsKey(gh))
      System.out.println((Prediction)ht.get(gh));
  }
} ///:~
```

Observe que se usa la clase Prediction del ejemplo anterior, por lo que SpringDetector.java debe ser compilado primero o de lo contrario usted obtendra un error en tiempo de compilacoin cuando intente compilar SpringDetector2.java.

Groundhog2.hashCode( ) devuelve el numero de ground hog como un identificador. (En este ejemplo, el programador es el responsable de asegurarse de que no existan dos números de ground hog con el mismo identificador). hasCode() no es necesario para devolver un único identificador, pero el método equals() debería determinar estrictamente si dos objetos son equivalentes.

El método equals() hace dos chequeos: para ver si el objeto es null, y sino, para ver si es una instancia del Groundhog2 (usando la palabra reservada <keyword>, que esta completamente explicada en el Capitulo 11). Debería ser un Groundhog2 para poder continuar ejecutando equals( ). La comparación, como puede ver, esta basada en los ghNumbers actuales. Esta vez, cuando ejecute el programa, usted vera que produce la salida correcta. (Muchas de las clases Java sobreescriben los métodos hashcode() y equals() para hacer que se basen en sus contenidos).

Properties: un tipo de Hashtable

En el primer ejemplo de este libro, se uso un tipo de Hashtable llamado Properties. En ese ejemplo, las líneas:

```
Properties p = System.getProperties();
p.list(System.out);
```

llamadas como el método estático getProperties() para obtener propiedades especiales de los objetos que describen las características del sistema. El método list() es un método de Properties que envía el contenido de una cadena a la salida que usted seleccione. Hay también un método save() para mostrarle a escribir su lista de propiedades en un fichero de forma que pueda ser recuperado mas tarde con el método load().

Aunque la clase Properties es heredada de Hashtable, también contiene una segunda Hashtable para mantener la lista de propiedades por defecto ("default properties"). Por tanto, si una propiedad no se encuentra en la lista primaria, se le busca su valor por defecto.

La clase Properties esta también disponible para uso en sus programas (un ejemplo es ClassScanner.java en el Capitulo 17). Puede encontrar mas detalles en la documentación de la librería de Java.

Revision de las enumeraciones

Ahora vamos a demostrar el verdadero poder de una Enumeration: la habilidad para separar la operación de recorrer una secuencia de la estructura base de esa secuencia. En el siguiente ejemplo, la clase PrintData usa una enumeración para moverse a través de una secuencia y llamar al método toString() para cada object. Se han creado dos tipos distintos de colecciones, un Vector y una Hashtable, y ambas son rellenadas, respectivamente, con objetos Mouse y Hamster. (Estas clases están definidas anteriormente en el capitulo; observe que usted debe haber compilado HamsterMaze.java y WorksAnyway.java para poder compilar el siguiente ejemplo.) Ya que una enumeración oculta la estructura de la colección subyacente, PrintData no sabe ni tiene en cuenta de que tipo de colección viene la

enumeración:

```
//: Enumerators2.java
// Revisiting Enumerations
import java.util.*;

class PrintData {
  static void print(Enumeration e) {
    while(e.hasMoreElements())
      System.out.println(e.nextElement().toString());
  }
}

class Enumerators2 {
  public static void main(String[] args) {
  Vector v = new Vector();
  for(int i = 0; i < 5; i++)
    v.addElement(new Mouse(i));

    Hashtable h = new Hashtable();
    for(int i = 0; i < 5; i++)
      h.put(new Integer(i), new Hamster(i));

    System.out.println("Vector");
    PrintData.print(v.elements());
    System.out.println("Hashtable");
    PrintData.print(h.elements());
  }
} ///:~
```

Observe que PrintData.print() saca partido de el hecho de que los objetos de esas colecciones no son de la clase Object y pueden llamar a toString(). Es mas probable que en su problema, usted deba suponer que su enumeración esta moviéndose a través de una colección de algún tipo especifico. Por ejemplo, usted debe asumir que todo en la colección es una pieza (Shape) con un método draw(). Luego, usted debe hacer un cast del Object que Enumeration.nextElement() devuelve para producir un Shape.

Ordenando

Una de las cosas que faltan en las librerías de Java 1.0 y 1.1 son las operaciones algoritmicas, incluso una simple ordenación. Por tanto, esto da sentido a crear un Vector que se ordene a si mismo usando el clásico Quicksort.

Un problema con el codigo genérico de ordenación es que para ordenar debe realizar comparaciones basadas en los tipos actuales de objetos. Por supuesto,

una aproximación es escribir un método de ordenación diferente para cada tipo diferente, pero usted debería reconocer que esto no produce codigo fácil de mantener y rehusar para nuevos tipos.

Un primero objetivo del diseño de programación es "separar cosas que cambian de cosas que permanecen igual", y aquí, el codigo que permanece igual es el algoritmo genérico de ordenación. Pero en vez de escribir cada codigo de comparación en diferentes rutinas, se puede usar la técnica de la recursividad. Con una llamada recursiva, la parte del codigo que varia de caso a caso es encapsulada dentro de su propia clase, y la parte de codigo que siempre permanece igual en cada llamada puede ser llamada por el codigo que cambia. De esta forma usted podrá crear diferentes objetos para expresar diferentes tipos de comparaciones y mantenerlos con el mismo codigo de ordenación.

El siguiente interface describe como comparar dos objetos, y así encapsular "las cosas que cambian" para este problema en particular:

```
//: Compare.java
// Interface for sorting callback:
package c08;

interface Compare {
  boolean lessThan(Object lhs, Object rhs);
  boolean lessThanOrEqual(Object lhs, Object rhs);
} ///:~
```

Para ambos métodos, lhs representa la parte izquierda ("left hand") y rhs la parte derecha ("right hand") del objeto en la comparacion. Se puede implementar una subclase de la clase Vector que implemente el Quicksort utilizando comparaciones. El algoritmo, que es famoso por su velocidad, no será explicado aqui. Para mas detalles, vea Practical Algorithms for Programmers, por Binstock y Rex, Addison-Wsley 1995.

```
//: SortVector.java
// A generic sorting vector
package c08;
import java.util.*;

public class SortVector extends Vector {
private Compare compare; // To hold the callback
  public SortVector(Compare comp) {
    compare = comp;
  }
  public void sort() {
    quickSort(0, size() - 1);
```

```
  }
  private void quickSort(int left, int right) {
    if(right > left) {
      Object o1 = elementAt(right);
      int i = left - 1;
      int j = right;
      while(true) {
        while(compare.lessThan(
          elementAt(++i), o1));
          while(j > 0)
            if(compare.lessThanOrEqual(elementAt(--j), o1))
              break; // out of while
            if(i >= j) break;
              swap(i, j);
      }
      swap(i , right);
      quickSort(left, i-1);
      quickSort(i+1, right);
    }
  }
  private void swap(int loc1, int loc2) {
    Object tmp = elementAt(loc1);
    setElementAt(elementAt(loc2), loc1);
    setElementAt(tmp, loc2);
  }
} ///:~
```

Usted puede ahora ver la razón del termino "llamada recursiva", ya que el método quickSort() se llama a si mismo en la comparación (Compare). También puede ver como esta técnica ha producido un codigo genérico y reusable.

Para usar SortVector, usted debe crear una clase que implemente Compare para los objetos que esta ordenando. Colocar esto en el interior de una clase no es esencial, pero puede ser bueno para la organización del codigo. Aquí tiene un ejemplo para objetos String:

```
//: StringSortTest.java
// Testing the generic sorting Vector
package c08;
import java.util.*;

public class StringSortTest {
static class StringCompare implements Compare {
  public boolean lessThan(Object l, Object r) {
    return ((String)l).toLowerCase().compareTo(
            ((String)r).toLowerCase()) < 0;
  }
```

```
  public boolean lessThanOrEqual(Object l, Object r) {
    return ((String)l).toLowerCase().compareTo(
            ((String)r).toLowerCase()) <= 0;
  }
}
  public static void main(String[] args) {
  SortVector sv = new SortVector(new StringCompare());
    sv.addElement("d");
    sv.addElement("A");
    sv.addElement("C");
    sv.addElement("c");
    sv.addElement("b");
    sv.addElement("B");
    sv.addElement("D");
    sv.addElement("a");
    sv.sort();
    Enumeration e = sv.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
  }
} ///:~
```

La clase interior es estática porque no necesita un enlace a otras clases para poder funcionar.

Como puede ver, una vez que esta parte del trabajo es correcta, es fácil rehusarla en un diseño como este -usted simplemente escribirá la clase que encapsula "las cosas que cambian" y manejara un objeto SortVector.

La comparación fuerza a los strings a ponerse en minúsculas, por lo que las Aes mayúsculas se convertirán en aes minúsculas y no en otra cosa diferente. Este ejemplo muestra, sin embargo, una pequeña deficiencia en esta aproximación, ya que el codigo del test anterior pone las letras mayúsculas y minúsculas en el mismo orden en que aparecen: A a b B c C d D. Es no es usual en muchos problemas, porque usted normalmente trabajara con cadenas largas y en esa situación el efecto no es el mostrado aquí. (Las colecciones de Java 1.2 proporcionan funcionalidades para la ordenación que resuelven estos problemas.)

La herencia (extendida) es usada aquí para crear un nuevo tipo de Vector -es decir, SortVector es un Vector con algunas funcionalidades añadidas-. El uso de la herencia es muy potente pero presenta problemas. Se supone que algunos métodos son definitivos (descrito en el Capitulo 7), por lo que usted no podrá sobreescribirlos. Si usted quiere crear un Vector ordenado que acepte y produzca métodos que usted necesita sobreescribir, ellos solo aceptaran y producirán objetos String. No tendrá suerte.

En cambio, considere una composición: colocar un objeto dentro de una nueva clase. Mas bien de rescribir el codigo anterior para lograr esto, nosotros podremos

simplemente usar un SortVector dentro de una nueva clase. En este caso, la case interior para implementar la interface Compare será creada anónimamente:

```
//: StrSortVector.java
// Automatically sorted Vector that
// accepts and produces only Strings
package c08;
import java.util.*;

  public class StrSortVector {
  private SortVector v = new SortVector(
  // Anonymous inner class:
    new Compare() {
      public boolean
      lessThan(Object l, Object r) {
      return ((String)l).toLowerCase().compareTo(
              ((String)r).toLowerCase()) < 0;
    }
    public boolean lessThanOrEqual(Object l, Object r) {
      return ((String)l).toLowerCase().compareTo(
              ((String)r).toLowerCase()) <= 0;
    }
  }
);
  private boolean sorted = false;
  public void addElement(String s) {
    v.addElement(s);
    sorted = false;
  }
  public String elementAt(int index) {
    if(!sorted) {
      v.sort();
      sorted = true;
    }
    return (String)v.elementAt(index);
  }
  public Enumeration elements() {
    if(!sorted) {
      v.sort();
      sorted = true;
    }
    return v.elements();
  }
  // Test it:
  public static void main(String[] args) {
  StrSortVector sv = new StrSortVector();
    sv.addElement("d");
    sv.addElement("A");
```

```
    sv.addElement("C");
    sv.addElement("c");
    sv.addElement("b");
    sv.addElement("B");
    sv.addElement("D");
    sv.addElement("a");
    Enumeration e = sv.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
  }
} ///:~
```

Esta rehusa rápidamente el codigo de SortVector para crear la funcionalidad deseada. Sin embargo, no todos los métodos públicos de SortVector y Vector aparecen en StrSortVector. Cuando se rehusa codigo de esta forma, usted puede crear una definición en la nueva clase para cada una de las clases contenidas, o puede comenzar solo con algunas y periódicamente añadir lo que necesite. Eventualmente, la nueva clase diseña como se establecerá.

La ventaja de esto es que se pueden tomar solo objetos String y producir solo objetos String, y el chequeo ocurre en tiempo de compilación en lugar de en tiempo de ejecución. Por supuesto, esto solo es cierto para addElement() y elementAt(); elements() aun produce una Enumeration que no tiene tipo en tiempo de compilación. El chequeo de tipo para la Enumeration y en StrSortVector aun sucede, de acuerdo, ocurre en tiempo de ejecución a través de excepciones si usted hace algo mal. Es como un retroceso: ¿se ha enterado usted de algo con seguridad en tiempo de compilación o probablemente en tiempo de ejecución? (Es decir, "probablemente no mientras testea el codigo", y "probablemente cuando el usuario del programa intente algo que usted no ha testeado".) Una vez vistas las ventajas y los inconvenientes, es mas fácil usar la herencia y apretar los dientes mientras se hace el casting -de nuevo, si los tipos parametrizados nunca se han añadido en Java, ellos resolverán estos problemas.

Usted puede ver que aquí hay un flag o bandera llamada sorted en esta clase. Usted podría ordenar el vector cada vez que llama a addElement(), y constantemente manteniéndolo en un estado ordenado. Pero usualmente la gente añade elementos a un vector antes de comenzar a leerlo. Por tanto, ordenarlo cada vez que se llama a addElment() seria menos eficiente que esperar hasta que alguien quiera leer el vector y entonces ordenarlo, que es lo mejor aquí. La técnica de retrasar el proceso hasta que es absolutamente necesario es llamada "lazy evaluation" (evaluación perezosa). (Hay una técnica análoga llamada lazy initialization, la cual espera a un valor de un campo que es necesario antes de inicializarlo.)

La librería general de colecciones

Hemos visto en este capitulo que la librería estándar de Java tiene muchas

colecciones útiles, pero lejos de ser una amplia variedad. Además, algoritmos como los de ordenación no están soportados en absoluto. Una de las ventajas de C++ es que sus librerías, en particular la Standard Template Library (STL), es que proporciona un completo juego de colecciones así como muchos algoritmos como ordenación y búsqueda que trabajan con estas colecciones. Basados en este modelo, la compañía ObjectSpace se inspiro para crear la Generic Collection Library for Java (formalmente llamada la Librería Genérica de Java o Java Generic Libray, aunque se suele usar la abreviatura JGL -el viejo nombre infringía un copyright de Sun-), la cual sigue el diseño de la STL tanto como le es posible (salvando las diferencias entre los dos lenguajes). JGL parece cumplir muchos, si no todos, las necesidades de una librería de colecciones, llegando tan lejos como puede en la dirección de los mecanismos usados en C++. JGL incluye listas de enlace, juegos, colas, mapas, pilas, secuencias, e iteradores que son más funcionales que Enumeration, así como un completo set de algoritmos tales como la búsqueda y la ordenación. ObjectSpace también hace, en algunos casos, un diseño mas inteligente que los diseños de la librería de Sun. Por ejemplo, los métodos en las colecciones de JGL no son definitivos, por tanto es posible heredarlos y sobreescribirlos.

JGL ha sido incluida en algunos proveedores de las distribuciones de Java y ObjectSpace ha creado el JGL freely disponible para todos los usuarios, incluyendo el uso comercial, en http://ObjectSpace.com. La documentación en línea que viene en el paquete de JGL es bastante buena y debería ser suficiente para que usted comenzara a utilizarla.

Las nuevas colecciones

Las nuevas clases de colecciones son una de las más potentes herramientas para la programación. Usted debe pensar que yo estoy un poco desilusionado con las colecciones incluidas en Java version 1.1. Como resultado, es un tremendo placer ver que las colecciones han tomado importancia en Java 1.2, y han sido rediseñadas (por Joshua Bloch de Sun). Yo considero que las nuevas colecciones son una de las mayores características de Java 1.2 (la otra es la librería Swing, estudiada en el Capitulo 13), porque éstas incrementan significativamente su potencia de programación y reaniman a Java en para madurar con respecto a otro sistemas de programación.

Algunos de las cosas rediseñadas son seguras y más sensibles. Por ejemplo, muchos nombres son acertados, limpiados, y más fáciles de comprender, así como de clasificar. Algunos nombres han cambiado para adecuarse a una terminología aceptada: uno de los mas acertados a mi gusto es "iterator" en vez de "enumeration".

El rediseño también cubre la funcionalidad de la librería de colecciones. Usted puede conocer ahora el comportamiento de listas de enlace, encolar y desencolar (colas con doble fin, "decks" pronunciados).

El diseño de una librería de colecciones es dificultoso (la mayoría de los diseños de librerías conllevan problemas).

En C++, STL cubre las bases con muchas clases diferentes. Esto mejoraba lo que estaba disponible con las versiones anteriores de STL (nada), pero no se translada bien a Java. El resultado fue mas bien confuso en muchas clases. En el otro extremo, yo he visto una librería de colecciones que consiste en clases simples, "collection", que funciona como un Vector y una Hashtable al mismo tiempo. Los diseñadores de la nueva librería de colecciones querían unir el balance: la funcionalidad completa que usted encuentra en las librerías de colecciones maduras, pero mas fácil de aprender y usar que STL y otras librerías similares. El resultado puede parecer extraño en muchos lugares. A diferencia de algunas de las decisiones tomadas en la temprana librería de Java, estas rarezas no son accidentales, sino decisiones cuidadosamente consideradas basadas en una solución de compromiso con la complejidad. Deberá llevarle un tiempo el acomodarse con algunos aspectos de la librería, pero pienso que se encontrara más a gusto y adquiera rapidez usando estas nuevas herramientas.

La nueva librería de colecciones toma su emisión de "almacenando sus objetos" y se divide en dos conceptos distintos:

1. Colección: un grupo de elementos individuales, con frecuencia con ciertas reglas que se les aplican. Una lista debe almacenar los elementos en una secuencia particular, y un Set no puede tener elementos duplicados. (Una bolsa (bag), que no esta implementada en la nueva librería de colecciones ya que las listas le permiten esta funcionalidad, no tienen tales reglas.)
2. Mapa: un grupo de parejas de objetos clave-valor (de lo que ha visto hasta ahora, es como una Hashtable).

A primera vista, esto parece como si fuese una colección de parejas, pero cuando usted intenta implementarla de esta forma, el diseño será malo, por tanto, esta claro que hay que diferenciar este nuevo concepto. Por otro lado, es conveniente ver las porciones de un mapa creando una colección para representar una porción. Así, un Map puede devolver un Set de claves,, una List de valores, o una List de parejas. Los mapas, como los arrays, pueden expandirse fácilmente a múltiples dimensiones sin añadir nuevos conceptos: usted simplemente crea un mapa cuyos valores son mapas (y los valores de estos mapas, pueden ser otros mapas, etc.)

Las colecciones y los mapas pueden ser implementados de muchas formas distintas, según sus necesidades de programación. Es muy útil ver un esquema de las nuevas colecciones: Este capitulo fue escrito mientras Java 1.2 era todavía beta, por lo que los esquemas no muestran la clase TreeSet que fue añadida mas tarde.

Este esquema puede ser algo aplastante en principio, pero a lo largo de todo lo que queda de este capitulo, usted podrá observar que se trata realmente de una colección de tres de componentes: Map, List y Set, y solo dos o tres implementaciones de cada uno (normalmente con una version preferente). Cuando usted vea esto, las nuevas colecciones no parecerán intimidarle.

Las cajas <dashed> representan interfaces, las cajas punteadas representas clases abstractas, y las cajas sólidas son clases concretas. Las flechas <dashed> indican que una clase particular esta implementada en un interface (o en el caso de una clase abstracta, parcialmente implementado en este interface). Las flechas de doble línea muestran que la clase puede producir objetos de la clase a la que la flecha apunta. Por ejemplo, cualquier colección puede producir un Iterator, mientras una lista puede producir un ListIterator (así como un Iterator simple, ya que List es heredada de Collection).

Los interfaces que interesan para almacenar objetos son Collection, List, Set y Map. Normalmente, usted escribirá la mayoría de su codigo para comunicarse con estas interfaces, y el único lugar en el que especificara el tipo preciso que esta usando es en el punto de creación. Por tanto usted podrá crear una lista como esta:

```
List x = new LinkedList();
```

Por supuesto, usted puede también decidir hacer a x un LinkedList (una lista de enlace, en lugar de una lista genérica), y llevar la información del tipo preciso al utilizar x. Lo bonito (y el intento) de usar el interface es que usted decide lo que quiere cambiar en su implementación, y todo lo que necesita es hacer este cambio en el punto de la creación, como aquí:

```
List x = new ArrayList();
```

El resto de su codigo puede quedar intacto.

En la clase heredada, usted puede ver un numero de clases cuyos nombres comienzan con "Abstract", y estos pueden parecer un poco confusos en principio. Son simples herramientas que implementan parcialmente un interface particular. Si usted estuviera creando su propio Set, por ejemplo, no debería comenzar con el interface de Set e implementar todos los métodos, en vez de esto, usted heredaría de AbstractSet y haría el mínimo trabajo necesario para crear su nueva clase. Sin embargo, la nueva librería de colecciones contiene suficiente funcionalidad para satisfacer sus necesidades virtualmente todo el tiempo. Por tanto, para nuestros propósitos, usted puede ignorar cualquier clase que comience con "Abstract".

Por consiguiente, cuando usted mire un esquema, solamente estará interesado en las interfaces de la parte de arriba de este y las clases concretas (aquellas de caja sólida alrededor de ellas). Normalmente creará un objeto de una clase concreta, <upcast> al correspondiente interface, y luego usara el interface durante el resto de su codigo. Aquí tiene un simple ejemplo, que rellena una colección con objetos String y luego imprime cada elemento de la colección:

```
//: SimpleCollection.java
// A simple example using the new Collections
package c08.newcollections;
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
  Collection c = new ArrayList();
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
      while(it.hasNext())
        System.out.println(it.next());
  }
```

```
} ///:~
```

Todos los ejemplos de codigo de las nuevas librerías de colecciones pueden ser almacenados en el subdirectorio newcollections, de forma que pueda recordar que ese trabajo solo funciona con Java 1.2. Como resultado, usted debe invocar al programa escribiendo:

```
java 08.newcollections.SimpleCollection
```

con una sintaxis similar para el resto de los programas del paquete. Como puede ver las nuevas colecciones son parte de la librería java.util, por lo que no necesitara añadir codigo externo para poder usarlas.

La primera línea de main() crea un objeto ArrayList y lo <upcasts> en una colección. Ya que este ejemplo solo usa métodos de Collection, cualquier objeto de una clase heredada de Collection funcionara, pero ArrayList es el típico caballo de batalla de Collection y toma el lugar de Vector.

El método add(), como sugiere su nombre, introduce un nuevo elemento en la colección. Sin embargo, la documentación dice de add() que "asegura que esta colección contenga el elemento especificado". Esto viene a demostrar el significado de Set, el cual añade el elemento solo si no se ha hecho antes. Con un ArraList, o cualquier clase de List, add() siempre significa "introducir".

Todas las colecciones pueden producir un iterador mediante el método iterator(). Un iterador es como una enumeración, que ha sido reemplazada, con estas características:

1. Usa un nombre (iterador) que es conocido históricamente y aceptado en la comunidad de la OOP.
2. Usa nombres de métodos mas cortos que en las enumeraciones: hasNext() en vez de hasMoreElements(), y next() en vez de nextElement().
3. Añade un nuevo método, remove(), que elimina el ultimo elemento producido por el iterador. Por tanto, usted puede llamar a remove() solo una vez por cada llamada a next ().

En SimpleCollection.java, usted puede ver que se crea un iterador y se usa para atravesar la colección, imprimiendo cada elemento.

Usando colecciones

La siguiente tabla muestra todo lo que usted puede hacer con una colección, y de esta forma, todo lo que puede hacer con un Set o una lista. (List también tiene

funcionalidades añadidas). Los mapas no son heredados de Collection, y serán tratados de forma separada.

boolean add (Object)	Asegura que la colección contenga el argumento. Devuelve false si no se puede añadir el argumento.
boolean addAll (Collection)	Añade todos los elementos del argumento. Devuelve verdadero si hay elementos que ya han sido añadidos.
void clear()	Borra todos los elementos de la colección.
boolean contains (Object)	Verdadero si la colección contiene al argumento.
boolean containsAll (Collection)	Verdadero si la colección contiene todos los elementos del argumento.
boolean isEmpty()	Verdadero si la colección no tiene elementos.
Iterator iterator()	Devuelve un iterador que puede usar para moverse a través de los elementos de la colección.
boolean remove(Object)	Si el argumento esta en la colección, una instancia de ese elemento es borrada. Devuelve verdadero si ocurre un borrado.
boolean removeAll (Collection)	Borra todos los elementos contenidos en el argumento. Devuelve verdadero si ocurre algún borrado.
boolean retainAll (Collection)	Retiene solo los elementos que están contenidos en el argumento (una intersección en la teoría de conjuntos). Devuelve verdadero si ocurre algún cambio.
int size()	Devuelve el numero de elementos de una colección.
Object[]	Devuelve un array que contiene todos los elementos de la colección.
	Es un método "opcional", lo que significa que no puede ser implementado para una colección en particular. Si no, el método lanza

toArray()	una UnsupportedOperationException. Las excepciones serán cubiertas en el Capitulo 9.

El siguiente ejemplo demuestra todos estos métodos. De nuevo, esto funciona con cualquier herencia de Collection; un ArrayList se usa como un tipo de "mínimo denominador común".

```
//: Collection1.java
// Things you can do with all Collections
package c08.newcollections;
import java.util.*;

public class Collection1 {
// Fill with 'size' elements, start
// counting at 'start':
  public static Collection fill(Collection c, int start, int size) {
    for(int i = start; i < start + size; i++)
      c.add(Integer.toString(i));
    return c;
  }
// Default to a "start" of 0:
  public static Collection fill(Collection c, int size) {
    return fill(c, 0, size);
  }
// Default to 10 elements:
  public static Collection fill(Collection c) {
    return fill(c, 0, 10);
  }
// Create & upcast to Collection:
                public static Collection newCollection() {
                  return fill(new ArrayList());
                  // ArrayList is used for simplicity, but it's
                  // only seen as a generic Collection
                  // everywhere else in the program.
                }
                // Fill a Collection with a range of values:
                public static Collection
                newCollection(int start, int size) {
                  return fill(new ArrayList(), start, size);
                }
                // Moving through a List with an iterator:
                public static void print(Collection c) {
                  for(Iterator x = c.iterator(); x.hasNext();)
                    System.out.print(x.next() + " ");
```

```
    System.out.println();
  }
  public static void main(String[] args) {
    Collection c = newCollection();
    c.add("ten");
    c.add("eleven");
    print(c);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Add a Collection to another Collection
    c.addAll(newCollection());
    print(c);
    c.remove("3"); // Removes the first one
    print(c);
    c.remove("3"); // Removes the second one
    print(c);
    // Remove all components that are in the
    // argument collection:
    c.removeAll(newCollection());
    print(c);
    c.addAll(newCollection());
    print(c);
    // Is an element in this Collection?
    System.out.println(
      "c.contains(\"4\") = " + c.contains("4"));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(newCollection()) = " +
      c.containsAll(newCollection()));
    Collection c2 = newCollection(5, 3);
    // Keep all the elements that are in both
    // c and c2 (an intersection of sets):
    c.retainAll(c2);
    print(c);
    // Throw away all the elements in c that
    // also appear in c2:
    c.removeAll(c2);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = newCollection();
    print(c);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():");
    print(c);
  }
```

```
} ///:~
```

Los primeros métodos muestran una forma de llenar una colección con datos de test, en esta caso son enteros convertidos en Strings. El segundo método será usado frecuentemente durante el resto de este capitulo.

El método print() será usado durante el resto de esta sección. Ya que este se mueve a través de una colección usando un iterador, el cual puede producir cualquier colección, funcionara con listas y Sets y cualquier colección que produzca un mapa.

main() usa ejercicios simples para mostrar todos los métodos de la colección.

Las secciones siguientes comparan varias implementaciones de List, Set, y Map, e indican en cada caso (con un asterisco) cual debe ser la elección por defecto.. Usted observara el <legacy> de las clases Vector, Stack y Hashtable no han sido incluidas porque en todos los casos son preferibles las clases dentro de las colecciones.

Usando listas

List (interface)	El orden es la característica mas importante de una lista; estas aseguran mantener los elementos en una secuencia particular. Las listas añaden un número de métodos de Collection que permiten la inserción y borrado de elementos en la mitad de la lista. (Esto es recomendado solo para una LinkedList.) Una lista puede producir un ListIterator, y usándolo usted puede atravesar la lista en ambas direcciones, así como insertar y borrar elementos en la mitad de la lista (de nuevo, es recomendado solo para las LinkedList.)
ArrayList	Una lista <backed> por un array. Se usa en vez de un Vector como almacenador de objetos de propósito general. Permite un rápido acceso a los elementos, pero es lento insertando y borrando en el medio de la lista. ListIterator debería ser usado solo para recorridos hacia adelante y hacia detrás de un ArrayList, pero no para insertar o borrar elementos, ya que es mucho más lento comparado con las LinkedList.
	Proporciona acceso secuencial optimo, sin coste excesivo en la inserción y borrado en mitad de la lista. Es relativamente lento

LinkedList	para acceso aleatorio (usar ArrayList para este caso.) También tiene los métodos addFirst(), addLast(), getFirst(), getLast(), removeFirst() y removeLast() (que no están definidos en cualquier interface o clase base) para permitir ser usado como una pila (stack), cola (queue), y dequeue.

Cada método del siguiente ejemplo cubre un grupo diferente de actividades: cosas que siempre puede hacer una lista (basicTest()), moverse alrededor de un iterador (iterMotion()) contra el cambio de cosas con un iterador (iterManipulation()), viendo los efectos de la manipulación de una lista (testVisual()), y operaciones disponibles solo para LinkedLists.

```
//: List1.java
// Things you can do with Lists
package c08.newcollections;
               import java.util.*;

               public class List1 {
                 // Wrap Collection1.fill() for convenience:
                 public static List fill(List a) {
                   return (List)Collection1.fill(a);
                 }
                 // You can use an Iterator, just as with a
                 // Collection, but you can also use random
                 // access with get():
                 public static void print(List a) {
                   for(int i = 0; i < a.size(); i++)
                     System.out.print(a.get(i) + " ");
                   System.out.println();
                 }
                 static boolean b;
                 static Object o;
                 static int i;
                 static Iterator it;
                 static ListIterator lit;
                 public static void basicTest(List a) {
                   a.add(1, "x"); // Add at location 1
                   a.add("x"); // Add at end
                   // Add a collection:
                   a.addAll(fill(new ArrayList()));
                   // Add a collection starting at location 3:
                   a.addAll(3, fill(new ArrayList()));
                   b = a.contains("1"); // Is it in there?
                   // Is the entire collection in there?
                   b = a.containsAll(fill(new ArrayList()));
```

```
            // Lists allow random access, which is cheap
            // for ArrayList, expensive for LinkedList:
            o = a.get(1); // Get object at location 1
            i = a.indexOf("1"); // Tell index of object
            // indexOf, starting search at location 2:
            i = a.indexOf("1", 2);
            b = a.isEmpty(); // Any elements inside?
            it = a.iterator(); // Ordinary Iterator
            lit = a.listIterator(); // ListIterator
            lit = a.listIterator(3); // Start at loc 3
            i = a.lastIndexOf("1"); // Last match
            i = a.lastIndexOf("1", 2); // ...after loc 2
            a.remove(1); // Remove location 1
            a.remove("3"); // Remove this object
            a.set(1, "y"); // Set location 1 to "y"
            // Make an array from the List:
            Object[] array = a.toArray();
            // Keep everything that's in the argument
            // (the intersection of the two sets):
            a.retainAll(fill(new ArrayList()));
            // Remove elements in this range:
            a.removeRange(0, 2);
            // Remove everything that's in the argument:
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
  }
  public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element that was just produced:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element that was just produced:
    it.set("47");
  }
  public static void testVisual(List a) {
    print(a);
```

```
List b = new ArrayList();
fill(b);
System.out.print("b = ");
print(b);
a.addAll(b);
a.addAll(fill(new ArrayList()));
print(a);
// Shrink the list by removing all the
// elements beyond the first 1/2 of the list
System.out.println(a.size());
System.out.println(a.size()/2);
a.removeRange(a.size()/2, a.size()/2 + 2);
print(a);
// Insert, remove, and replace elements
// using a ListIterator:
ListIterator x = a.listIterator(a.size()/2);
x.add("one");
print(a);
System.out.println(x.next());
x.remove();
System.out.println(x.next());
x.set("47");
print(a);
// Traverse the list backwards:
x = a.listIterator(a.size());
while(x.hasPrevious())
  System.out.print(x.previous() + " ");
  System.out.println();
  System.out.println("testVisual finished");
}
       // There are some things that only
       // LinkedLists can do:
       public static void testLinkedList() {
         LinkedList ll = new LinkedList();
         Collection1.fill(ll, 5);
         print(ll);
         // Treat it like a stack, pushing:
         ll.addFirst("one");
         ll.addFirst("two");
         print(ll);
         // Like "peeking" at the top of a stack:
         System.out.println(ll.getFirst());
         // Like popping a stack:
         System.out.println(ll.removeFirst());
         System.out.println(ll.removeFirst());
         // Treat it like a queue, pulling elements
         // off the tail end:
         System.out.println(ll.removeLast());
         // With the above operations, it's a dequeue!
         print(ll);
```

```
  }
  public static void main(String args[]) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
  }
} ///:~
```

En basicTest() y iterMotion() las llamadas son simplemente para mostrar la sintaxis correcta, y aunque el valor devuelto es capturado, no es usado. En algunos casos, el valor devuelto no es capturado ya que no suele ser usado. Usted debería mirar el uso completo de cada uno de estos métodos en su documentación online antes de utilizarlos.

Usando Sets

Un Set tiene exactamente el mismo interface que una Collection, por tanto no tienen ninguna funcionalidad extra y es como si hubiese dos clases diferentes de listas. En cambio, el Set es exactamente una colección, que tiene un comportamiento diferente. (Este es el uso ideal de la herencia y el polimorfismo: expresar comportamientos diferentes). Un Set muestra solo una instancia de cada valor de objeto existente (que constituye el "valor" de un objeto es mas complejo, como usted podrá ver).

Set (interface)	Each element that you add to the Set must be unique; otherwise the Set doesn't add the duplicate element. Objects added to a Set must define equals( ) to establish object uniqueness. Set has exactly the same interface as Collection. The Set interface does not guarantee it will maintain its elements in any particular order.
HashSet	Para todos los Sets excepto los muy pequeños. Los Objects deben también definir hashCode().
ArraySet	Un Set <backed> por un array. Diseñado para Sets muy pequeños, especialmente para cuando estos son creados y eliminados. Para Sets pequeños, la creación y la iteración es substancialmente mejor que para un HashSet. Es muy malo obteniendo los datos cuando un Set es muy grande. HashCode()

	no es necesario.
TreeSet	Un Set ordenado <backed> por un árbol rojinegro.[3] De esta forma, usted puede extraer una secuencia ordenada de un Set.

El siguiente ejemplo no muestra nada de lo que usted puede hacer con un Set, ya que el interface es el mismo que el de Collection y por tanto fue resuelto en el primer ejemplo. En vez de eso, este demuestra el extraño comportamiento que tiene un Set:

[3] En el momento de escribir esto, TreeSet solo había sido anunciado y no estaba implementado aun, por lo que aquí no se muestran ejemplos que usen TreeSet

```
//: Set1.java
// Things you can do with Sets
package c08.newcollections;
import java.util.*;

public class Set1 {
  public static void testVisual(Set a) {
                            Collection1.fill(a);
                            Collection1.fill(a);
                            Collection1.fill(a);
                            Collection1.print(a); // No duplicates!
                            // Add another set to this one:
                            a.addAll(a);
                            a.add("one");
                            a.add("one");
                            a.add("one");
                            Collection1.print(a);
                            // Look something up:
                            System.out.println("a.contains(\"one\"): "
                              a.contains("one"));
                          }
                          public static void main(String[] args) {
                            testVisual(new HashSet());
                            testVisual(new ArraySet());
                          }
                        } ///:~
```

Valores duplicados son añadidos al Set, pero cuando se imprime usted puede ver que el Set ha aceptado solo una ocurrencia de cada valor.

Cuando usted ejecuta un programa puede observa que el mantenimiento en HashSet es diferente del de ArraySet, ya que cada uno tiene una forma diferente de almacenar elementos para que puedan ser localizados mas tarde. (ArraySet los mantiene ordenados, mientras que HashSet usa una función de hasing, que esta diseñada específicamente para búsquedas rápidas.) Cuando crea sus propios tipos, este seguro de que un Set necesita una forma de mantener el orden de almacenamiento, como se mostraba en los ejemplos de los "groundhog" anteriormente en este capitulo. Aquí tiene un ejemplo:

```
//: Set2.java
// Putting your own type in a Set
package c08.newcollections;
import java.util.*;

class MyType {
private int i;
```

```
  public MyType(int n) { i = n;}
  public boolean equals(Object o) {
    if ((o != null) && (o instanceof MyType))
      return i == ((MyType)o).i;
    else return false;
  }
  // Required for HashSet, not for ArraySet:
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
}

public class Set2 {
  public static Set fill(Set a, int size) {
    for(int i = 0; i < size; i++)
      a.add(new MyType(i));
    return a;
  }
  public static Set fill(Set a) {
    return fill(a, 10);
  }
  public static void test(Set a) {
    fill(a);
    fill(a); // Try to add duplicates
    fill(a);
    a.addAll(fill(new ArraySet()));
    Collection1.print(a);
  }
  public static void main(String[] args) {
    test(new HashSet());
    test(new ArraySet());
  }
} ///:~
```

Las definiciones para equals() y hashCode() siguen la forma usada en los ejemplos de los "groundhog". Usted debe definir equals() en ambos casos, pero el hashCode () es necesario solo si la clase será colocada en un HashSet (lo cual es probable, ya que será generalmente su primera elección en la implementación de un Set.)

Usando mapas

Map (interface)	Mantiene asociaciones (parejas) clave-valor, por lo que usted puede obtener un valor usando una clave.
HashMap	Implementación basada en una tabla hash (usado en vez de Hashtable.) Proporciona desarrollo constante para insertar y localizar parejas. Puede ser ajustado vía constructores para permitirle fijar la capacidad y el factor de carga de la tabla hash.
	Mapa <backed> por un ArrayList. Obtiene el control

ArrayMap	preciso sobre el orden de la iteración. Diseñado para mapas muy pequeños, especialmente para los que son normalmente creados y destruidos. Para mapas muy pequeños, la creación y la iteracion es substancialmente mejor que para HashMap. Muy malo en la obtencion cuando el mapa es grande.
TreeMap	Implementación basada en un árbol rojinegro. Cuando usted ve las claves o las parejas, estas estarán en orden (determinado por Comparable o Comparator, que se discuten mas tarde.) La ventaja del TreeMapes que usted puede obtener los resultados de manera ordenada. TreeMap es el único mapa con el métodosubMap(), que le permite devolver una porción del árbol.

El siguiente ejemplo contiene dos set de datos de test y un método fill() que le muestra como rellenar un mapa con cualquier array bidimensional de objetos Object. Estas herramientas serán usados en otros ejemplos de mapas.

```
//: Map1.java
// Things you can do with Maps
package c08.newcollections;
import java.util.*;

public class Map1 {
  public final static String[][] testData1 = {
                { "Happy", "Cheerful disposition" },
                { "Sleepy", "Prefers dark, quiet places" },
                { "Grumpy", "Needs to work on attitude" },
                { "Doc", "Fantasizes about advanced degree"},
                { "Dopey", "'A' for effort" },
                { "Sneezy", "Struggles with allergies" },
                { "Bashful", "Needs self-esteem workshop"},
                 };
  public final static String[][] testData2 = {
                { "Belligerent", "Disruptive influence" },
                { "Lazy", "Motivational problems" },
                { "Comatose", "Excellent behavior" }
                 };
  public static Map fill(Map m, Object[][] o) {
    for(int i = 0; i < o.length; i++)
      m.put(o[i][0], o[i][1]);
    return m;
  }
  // Producing a Set of the keys:
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
```

```
  System.out.print("Keys: ");
  Collection1.print(m.keySet());
}
// Producing a Collection of the values:
public static void printValues(Map m) {
  System.out.print("Values: ");
  Collection1.print(m.values());
}
// Iterating through Map.Entry objects (pairs):
public static void print(Map m) {
Collection entries = m.entries();
Iterator it = entries.iterator();
  while(it.hasNext()) {
    Map.Entry e = (Map.Entry)it.next();
    System.out.println("Key = " + e.getKey() +
                       ", Value = " + e.getValue());
  }
}
public static void test(Map m) {
  fill(m, testData1);
  // Map has 'Set' behavior for keys:
  fill(m, testData1);
  printKeys(m);
  printValues(m);
  print(m);
  String key = testData1[4][0];
  String value = testData1[4][1];
  System.out.println("m.containsKey(\"" + key +
                     "\"): " + m.containsKey(key));
  System.out.println("m.get(\"" + key + "\"): "
                     + m.get(key));
  System.out.println("m.containsValue(\""
                     + value + "\"): " +
                     m.containsValue(value));
  Map m2 = fill(new ArrayMap(), testData2);
  m.putAll(m2);
  printKeys(m);
  m.remove(testData2[0][0]);
  printKeys(m);
  m.clear();
  System.out.println("m.isEmpty(): "
                     + m.isEmpty());
  fill(m, testData1);
  // Operations on the Set change the Map:
  m.keySet().removeAll(m.keySet());
  System.out.println("m.isEmpty(): "
                     + m.isEmpty());
}
public static void main(String args[]) {
  System.out.println("Testing ArrayMap");
```

```
    test(new ArrayMap());
    System.out.println("Testing HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
  }
} ///:~
```

Los métodos printKeys(), printValues() y print() no son solo utilidades generales,, ya que también demuestran la producción de vistas de colecciones de un mapa. El método keySet() produce un Set <backed> por las claves del mapa; aquí, es tratado como una colección. Similar tratamiento se le da a values(), que produce una lista que contiene todos los valores del mapa. (Observe que las claves deben ser únicas, mientras que los valores pueden duplicarse.) Ya que las colecciones son <backed> por el mapa, cualquier cambio en una colección se vera reflejado en el mapa asociado.

El método print() toma el iterador producido por las entradas y lo usa para imprimir la clave y el valor para cada pareja. El resto del programa muestra ejemplos simples de cada operación con el mapa, y testea cada tipo de mapa. Cuando usted crea su propia clase para usarla como clave en un mapa, usted debe distribuir en el mismo orden expuesto previamente para los Sets.

Iterators revisitados

Pueden verificar el verdadero poder del **Iterator** : la habilidad para separar la operación de cruzar una secuencia desde la estructura fundamental de esa secuencia. En el siguiente ejemplo, la clase **PrintData** usa un **Iterator** para moverse a través de una secuencia y llamar al método **toString()** para cada objeto. Dos diferentes tipos de contenedores son creados-un **ArrayList** y un **HashMap** -y están cada uno rellenados con objetos **Mouse** y **Hamster** , respectivamente. (Esas clases están definidas con anterioridad en este capítulo). Porqué un **Iterator** oculta la estructura del contenedor fundamental, **PrintData** no sabe o no se interesa de que tipo de contenedor proviene el **Iterator** :

```
//: c09:Iterators2.java
// Iterators otra vez.
import java.util.*;

class PrintData {
  static void print(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}
```

```
class Iterators2 {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 5; i++)
    v.add(new Mouse(i));
    HashMap m = new HashMap();

    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));

    System.out.println("ArrayList");
    PrintData.print(v.iterator());
    System.out.println("HashMap");
    PrintData.print(m.entrySet().iterator());
  }
} ///:~
```

Para el **HashMap** , el método **entrySet()** produce un **Set** de objetos **Map.entry** , el cual contiene tanto la clave como el valor para cada entrada, así verá los dos impresos.

Apuntar que **PrintData.print()** toma ventaja ante el hecho de que los objetos en ese contenedores son de clase **Object** así la llamada a **toString()** por **System.out.println()** es automática. Es más probable que en su problema, deba hacer la suposición que su **Itetator** esté caminando a través de un contenedor de algún tipo específico. Por ejemplo, puede suponer que todo en el contenedor es una **Forma** con un método **draw()** . Entonces debe hacer un cast a un tipo inferior desde el **Object** que devuelve **Iterator.next()** para generar una **Forma** .

Eligiendo una implementación

En el diagrama de la pagina 297 usted puede ver que solo hay tres componentes de la colección: Map, List y Set, y solo dos o tres implementaciones de cada interface. Si usted necesita usar la funcionalidad dada por un interface en particular, ¿cómo decide que implementación particular utilizar?.

Para responder a esta pregunta, usted debe estar seguro de que cada implementación tienes sus propias características, ventajas e inconvenientes. Por ejemplo, usted puede ver en el diagrama que la "característica" de Hashtable, Vector y Stack es que son clases <legacy>, por lo que su codigo no se romperá. Por otra parte, esto es mejor si no usa esto para el nuevo codigo (Java 1.2).

La distinción entre las otras colecciones con frecuencia se viene abajo para ver lo que han "<backed by>"; es decir, las estructuras de datos que físicamente implementa su diseño de interface. Esto significa que, por ejemplo, ArrayList, LinkedList y Vector (el cual es aproximadamente una equivalencia a ArrayList), implementan la interface de la List, por lo que sus programas producirán los

mismos resultados independientemente del que use. Sin embargo, ArrayList (y Vector) esta <backed> por un array, mientras que LinkedList esta implementada en la forma usual para una lista doblemente enlazada, como objetos individuales que contienen datos con manejadores o punteros a los elementos anterior y posterior en la lista. Debido a esto, si usted quiere hacer inserciones y borrados en la mitad de una lista, una LinkedList es la elección apropiada. (LinkedList también añade funcionalidad adicional que se establece en AbstractSequientialList.) Si no, un ArrayList es probablemente mas rápido.

Como otro ejemplo, un Set puede implementarse como un ArraySet o un HashSet. Un ArraySet es <backed> por un ArrayList y esta diseñado para soportar solo un pequeño numero de elementos, especialmente en situaciones en las que usted crea y destruye muchos objetos Set.

Sin embargo, si usted va a tener grandes cantidades en su Set, la utilizacion de ArraySet puede ser muy malo, o muy rapida. Cuando usted escribe un programa que necesita un Set, debe elegir HashSet por defecto, y cambiarlo a ArraySet solo en casos especiales en los que los requerimientos indican que es necesario.

Eligiendo entre listas

La forma mas convincente para ver las diferencias entre las implementacines de una lista es con un test. El siguiente codigo establece una base interna para usarse como un medidor de test, luego crea una clase interna anonima para cada test diferente. Cada una de estas es llamada por el método test(). Esta aproximacion le permite facilmente añadir y eliminar nuevos tipos de tests.

```
//: ListPerformance.java
// Demonstrates performance differences in Lists
package c08.newcollections;
import java.util.*;

public class ListPerformance {
private static final int REPS = 100;
  private abstract static class Tester {
  String name;
  int size; // Test quantity
    Tester(String name, int size) {
      this.name = name;
      this.size = size;
    }
    abstract void test(List a);
  }
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a) {
        for(int i = 0; i < REPS; i++) {
```

```
        for(int j = 0; j < a.size(); j++)
          a.get(j);
        }
      }
    },
  new Tester("iteration", 300) {
    void test(List a) {
      for(int i = 0; i < REPS; i++) {
        Iterator it = a.iterator();
        while(it.hasNext())
          it.next();
      }
    }
  },
  new Tester("insert", 1000) {
    void test(List a) {
      int half = a.size()/2;
      String s = "test";
      ListIterator it = a.listIterator(half);
      for(int i = 0; i < size * 10; i++)
        it.add(s);
    }
  },
  new Tester("remove", 5000) {
    void test(List a) {
      ListIterator it = a.listIterator(3);
      while(it.hasNext()) {
        it.next();
        it.remove();
      }
    }
  },
};
public static void test(List a) {
// A trick to print out the class name:
  System.out.println("Testing " +
                       a.getClass().getName());
  for(int i = 0; i < tests.length; i++) {
    Collection1.fill(a, tests[i].size);
    System.out.print(tests[i].name);
    long t1 = System.currentTimeMillis();
    tests[i].test(a);
    long t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
  }
}
public static void main(String[] args) {
  test(new ArrayList());
  test(new LinkedList());
}
```

```
} ///:~
```

La clase interna Tester es abstracta, para proporcinar una clase base para los tests especificos. Contiene un String que será impreso cuando el test comienza, un parametro size para ser usado por el test para cuantificar los elementos o repeticines de los tests, un constructor para incializar los campos, y un método abstracto test() que hace el trabajo. Todos los diferentes tipos de tests son almacenados en un lugar, el array tests, que es inicializado con diferentes clase internas anonimas que heredan de Tester. Para añador o eliminar tests, simplemente añada o elimine la definicion de una clase interna en el array, y todo ocurrira automaticamente.

La lista que maneja a test() es primero llenada con elementos, luego cada test del array se va ejecutando. El resultado puede variar de maquina a maquina; estas intentan obtener solo un orden de comparacion de magnitud entre las diferentes colecciones. Aqui tiene un sumario despues de una ejecucion:

Type	Get	Iteration	Insert	Remove
ArrayList	110	270	1980	4780
LinkedList	1870	7580	170	110

Usted puede ver que el acceso aleatorio (get()) y las iteracoines son menos costosas para los ArrayList y mas costosas para las LinkedLists. En cambio, las insercines y eliminaciones en la mitad de una lista son significativamente menjores en una LinkedList que en un ArrayList. La mejor aproximacion es probablemente la eleccion de un ArrayList como lista por defecto y cambiarla a una LinkedList si usted descubre problemas usando inserciones y eliminaciones en la mitad de la lista.

Eligiendo entre Sets

Usted puede elegir entre un ArraySet y un HashSet, dependiendo del tamaño del Set (si necesita producir una secuencia ordenada, use un TreeSet[1]). El siguiente programa testeador obtienen una indicacion de este <tradeoff>:

[1]TreeSet no estaba disponible en el momento de escribir esto, pero usted puede añadir un test para el facilmente.

```
//: SetPerformance.java
// Demonstrates performance differences in Sets
package c08.newcollections;
import java.util.*;

public class SetPerformance {
private static final int REPS = 100;
  private abstract static class Tester {
  String name;
    Tester(String name) { this.name = name; }
    abstract void test(Set s, int size);
  }
  private static Tester[] tests = {
    new Tester("add") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS; i++) {
          s.clear();
          Collection1.fill(s, size);
        }
      }
    },
    new Tester("contains") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS; i++)
          for(int j = 0; j < size; j++)
            s.contains(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS * 10; i++) {
          Iterator it = s.iterator();
            while(it.hasNext())
              it.next();
        }
      }
    },
  };
  public static void test(Set s, int size) {
  // A trick to print out the class name:
    System.out.println("Testing " +
          s.getClass().getName() + " size " + size);
    Collection1.fill(s, size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(s, size);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
```

```
            ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    // Small:
    test(new ArraySet(), 10);
    test(new HashSet(), 10);
    // Medium:
    test(new ArraySet(), 100);
    test(new HashSet(), 100);
    // Large:
    test(new HashSet(), 1000);
    test(new ArraySet(), 500);
  }
} ///:~
```

El ultimo test del ArraySet tiene solo 500 elementos en vez de 1000 porque seria mas lento

Type	Test size	Add	Contains	Iteration
ArraySet	10	5.0	6.0	11.0
	100	24.2	23.1	4.9
	500	100.18	97.12	4.5
HashSet	10	5.0	6.0	16.0
	100	5.5	5.0	6.0
	1000	6.1	6.09	5.77

HashSet es claramente superior a ArraySet para add() y contains(), y su calidad es efectiva independientemente del tamaño. Usted nunca querra usar un ArraySet para su programacion habitual.

Eligiendo entre mapas

Cuando hay que elegir entre varias implementaciones de un mapa, su tamaño es lo que mas fuertemente afecta a su calidad, y el siguiente programa testea una indicacion de esta solución de compromiso:

```
//: MapPerformance.java
// Demonstrates performance differences in Maps
package c08.newcollections;
import java.util.*;

public class MapPerformance {
private static final int REPS = 100;
  public static Map fill(Map m, int size) {
```

```
  for(int i = 0; i < size; i++) {
    String x = Integer.toString(i);
    m.put(x, x);
  }
  return m;
}
private abstract static class Tester {
String name;
  Tester (String name) { this.name = name; }
  abstract void test(Map m, int size);
}
private static Tester[] tests = {
  new Tester("put") {
    void test(Map m, int size) {
      for(int i = 0; i < REPS; i++) {
        m.clear();
        fill(m, size);
      }
    }
  },
  new Tester("get") {
    void test(Map m, int size) {
      for(int i = 0; i < REPS; i++)
        for(int j = 0; j < size; j++)
          m.get(Integer.toString(j));
    }
  },
  new Tester("iteration") {
    void test(Map m, int size) {
      for(int i = 0; i < REPS * 10; i++) {
        Iterator it = m.entries().iterator();
        while(it.hasNext())
          it.next();
      }
    }
  },
};
public static void test(Map m, int size) {
// A trick to print out the class name:
  System.out.println("Testing " +
                     m.getClass().getName() + " size " + size);
  fill(m, size);
  for(int i = 0; i < tests.length; i++) {
    System.out.print(tests[i].name);
    long t1 = System.currentTimeMillis();
    tests[i].test(m, size);
    long t2 = System.currentTimeMillis();
    System.out.println(": " +
                       ((double)(t2 - t1)/(double)size));
  }
```

```
  }
  public static void main(String[] args) {
  // Small:
    test(new ArrayMap(), 10);
    test(new HashMap(), 10);
    test(new TreeMap(), 10);
  // Medium:
    test(new ArrayMap(), 100);
    test(new HashMap(), 100);
    test(new TreeMap(), 100);
  // Large:
    test(new HashMap(), 1000);
  // You might want to comment these out since
  // they can take a while to run:
    test(new ArrayMap(), 500);
    test(new TreeMap(), 500);
  }
} ///:~
```

Ya que el tamaño de los mapas es mostrado, usted vera que el tiempo de los tests divide el tiempo por el tamaño para normalizar cada medida. Aqui tiene un juego de estos resultados (los suyos probablemente sean diferentes):

Type	Test size	Put	Get	Iteration
ArrayMap	10	22.0	44.0	17.0
	100	68.7	118.6	8.8
	500	155.22	259.36	4.84
TreeMap	10	17.0	16.0	11.0
	100	18.1	70.3	8.3
	500	11.22	148.4	4.62
HashMap	10	11.0	11.0	33.0
	100	9.9	10.4	12.1
	1000	13.18	10.65	5.77

Para tamaños iguales a 10, la potencia del ArrayMap es peor que la del HashMap -excepto para la iteracion, que no es normal utilizarla cuando se usan mapas-. (get () es generalmente el lugar donde usted empleara la mayoria de su tiempo.) El TreeMap tiene respetables put() y tiempos de iteracion, pero get() no es demasiado bueno. ¨Porque no usa usted un TreeMap si este tiene un buenos put() y tiempos de iteracion?. Asi usted no usaria un Map, pero es una forma de crear una lista ordenada. La ventaja de un arbol es que como este siempre esta en orden, no necesita ser especialmente ordenado. (La forma en que este ordena será discutida mas adelante.) Una vez que rellena el mapa, usted puede llamar a keySet() para obtener una vista de las claves del Set, y luego a toArray() para producir un array de estas claves. Usted puede usar el método estatico Array.binarySearch() (discutido despues) para encontrar rapidamente objetos en

su array ordenado. Por supuesto, usted probablemente solo hara esto si, por alguna razon, la ventaja del HashMap fuera inaceptable, ya que HashMap esta diseñado para la busqueda rapida. En definitiva, cuando usted use un Map su primera eleccion deberia ser un HashMap, y solo en raros casos necesitara investigar alternativas.

Existe otra ventaja importante, y es que las tablas descritas antes no tienen direccion, lo que agiliza su creacion. El siguiente programa de test de creacion mide la velocidad para diferentes tipos de Map:

```
//: MapCreation.java
// Demonstrates time differences in Map creation
package c08.newcollections;
import java.util.*;

public class MapCreation {
  public static void main(String[] args) {
  final long REPS = 100000;
    long t1 = System.currentTimeMillis();
    System.out.print("ArrayMap");
    for(long i = 0; i < REPS; i++)
      new ArrayMap();

      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
      t1 = System.currentTimeMillis();
      System.out.print("TreeMap");
      for(long i = 0; i < REPS; i++)
        new TreeMap();
      t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
      t1 = System.currentTimeMillis();
      System.out.print("HashMap");
      for(long i = 0; i < REPS; i++)
        new HashMap();
      t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
  }
} ///:~
```

En el momento en que se escribio el programa, la velocidad de creacion de TreeMap fue increiblementa mas rapida que las de los otros dos tipos. (Aunque usted deberia intentarlo, ya que esto fue comentado como un inconveniente de los ArrayMap.) Esto, junto con un aceptable y consistente put() de TreeMap, sugiere una posible estrategia si usted esta creando muchos mapas: Crear y rellenar TreeMaps, y luego comenzar a ver cosas, convertir los TreeMaps importantes en HashMaps usando el constructor HashMap(Map). De nuevo, usted deberia

preocuparse solamente sobre su orden para evitar que se formen cuellos de botella. ("Primero hacer su trabajo,luego aumentar su velocidad -deberia hacer usted-".)

Operaciones no soportadas

Es posible convertir un array en una lista con el método estatico Arrays.toList():

```
//: Unsupported.java
// Sometimes methods defined in the Collection
// interfaces don't work!
package c08.newcollections;
import java.util.*;

public class Unsupported {
private static String[] s = {
        "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten",
        };
static List a = Arrays.toList(s);
static List a2 = Arrays.toList(
new String[] { s[3], s[4], s[5] });
  public static void main(String[] args) {
    Collection1.print(a); // Iteration
    System.out.println(
        "a.contains(" + s[0] + ") = " +
         a.contains(s[0]));
    System.out.println(
        "a.containsAll(a2) = " +
         a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
         a.isEmpty());
    System.out.println(
         "a.indexOf(" + s[5] + ") = " +
         a.indexOf(s[5]));
    // Traverse backwards:
    ListIterator lit = a.listIterator(a.size());
    while(lit.hasPrevious())
      System.out.print(lit.previous());
    System.out.println();
    // Set the elements to different values:
    for(int i = 0; i < a.size(); i++)
      a.set(i, "47");
    Collection1.print(a);
    // Compiles, but won't run:
    lit.add("X"); // Unsupported operation
    a.clear(); // Unsupported
    a.add("eleven"); // Unsupported
```

```
    a.addAll(a2); // Unsupported
    a.retainAll(a2); // Unsupported
    a.remove(s[0]); // Unsupported
    a.removeAll(a2); // Unsupported
  }
} ///:~
```

Usted podra descubrir que solo una parte de los interfaces de Collection y List se encuentran actualmente implementados. El resto de los métodos causan apariencia de indeseables debido a algunas llamadas a UnsupportedOperationException. Usted aprendera todo lo necesario sobre las excepciones en el siguiente capitulo, pero la corta historia los interfaces de colecciones, asi como de otros interfaces de la nueva librería de colecciones, contienen métodos "opcionales", que deben ser o no soportados en la clase concreta que implemente la interface. Llamando a un método no soportado se produce una UnsupportedOperationException para indicar un error de programa.

"Que?!?", dice usted incredulo. "El gran mundo de los interfaces y las clases base es que estos aseguran que sus métodos no haran nada no permitido!. Esto rompe esta promesa -dice que no solo el llamar a estos métodos puede no hacer nada, sino que ademas el programa puede pararse!. La seguridad de los tipos se ha corrompido!". Esto no es tan malo. Con una Collection, List, Set o Map, el compilador todavia le restringe la llamada a los métodos que no son de su interface, por lo tanto, no es como en Smalltalk (en el cual usted puede llamar cualquier a método para cualquier objeto, y enterarse de lo que pasa solo cuando ejecute el programa para ver si su llamada hace algo). Ademas, la mayoria de los métodos que toman una colección como argumento solo leen de esa colección -todos los métodos de lectura de colecciones no son opcionales-.

Esta aproximacion evita una explosion de interfaces en el diseño. Otros diseños para de librerías de colecciones siempre parecen acabar con un confuso conjunto de interfaces para describir cada una de las variaciones del teme principal y son asi dificiles de aprender. Incluso no es posible capturar todos los casos especiales de interfaces, porque algunos pueden inventar siempre interfaces nuevos. La aproximacion "unsupported operation" desarrolla un importante avance en la nueva librería de colecciones: es simple de aprender y usar. Para esta ventaja funcione, sin embargo:

1. UnsupportedOperationException debe ser un evento raro. Es decir, para la mayoria de las clases todas las opereciones deberian funcionar, y solo en casos especiales una operacion no será soportada. Esto es cierto en la nueva librería de colecciones, ya que las clases que usted puede user el 99 por ciento de las veces -ArrayList, LinkedList, HashSet y HasMap, asi como las otras implementaciones concretas- soportan todas las operacones. El diseño proporciona una "puerta trasera" si usted quiere crear una nueva colección sin indicar sus definiciones para todos sus métodos en el interface, y asentarlos asi en una librería existente.

2. Cuando una operacoin no es soportada, deberia ser razonable la posibilidad de que apareciera una UsupportedOperationException en tiempo de implementación, mas aun si usted ya ha proporcionado el producto al cliente. Despues de todo, esta indica un error de programacion: usted ha usado una clase incorrecta. Este punto es menos cierto, y es donde la naturaleza experimental del diseño entra en juego. Solo con el tiempo podremos enterarnos de como de bien funciona.

En el ejemplo anterior, Arrays.toList() produce una lista que es <backed> por un array de tamaño fijo. Por tanto, esto hace sensible que las unicas operaciones soportada sean aquellas que no cambian el tamaño del array. Si, por otra parte, fuese requerido un nuevo interface para expresar este tipo de comportamiento diferente (llamado, quizas, "FixedSizeList"), deberia abrir la puerta de la complejidad y pronto usted sabria donde comenzar cuando intente usar la librería.

La documentación para un método que toma como argumento Collection, List, Set o Map, deberia especificar que métodos opcionales deberian ser implementados. Por ejemplo, las ordenaciones requieren los métodos set() e Iterator.set(), pero no add() ni remove().

Ordenando y buscando

Java 1.2 añade utilidades para realizar ordenaciones y busquedas para listas y arrays. Estas utilidades son métodos estaticos para dos nuevas clases: Arrays para ordenaciones y busquedas en arrays y Collections para ordenar y buscar en listas.

Arrays

Las clases de arrays han sobrecargado el uso de sort() y binarySearch() para arrays de tipos primitivos, asi como para String y Object. Aquie tiene un ejemplo que muestra la ordenacion y busqueda en un array de bytes (todas las demas primitivas son iguales) y un array de Strings:

```
//: Array1.java
// Testing the sorting & searching in Arrays
package c08.newcollections;
import java.util.*;

public class Array1 {
static Random r = new Random();
static String ssource =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
  "abcdefghijklmnopqrstuvwxyz";
static char[] src = ssource.toCharArray();
// Create a random String
  public static String randString(int length) {
    char[] buf = new char[length];
```

```
    int rnd;
    for(int i = 0; i < length; i++) {
      rnd = Math.abs(r.nextInt()) % src.length;
      buf[i] = src[rnd];
    }
    return new String(buf);
  }

// Create a random array of Strings:
  public static String[] randStrings(int length, int size) {
    String[] s = new String[size];
    for(int i = 0; i < size; i++)
      s[i] = randString(length);
    return s;
  }
  public static void print(byte[] b) {
    for(int i = 0; i < b.length; i++)
      System.out.print(b[i] + " ");
    System.out.println();
  }
  public static void print(String[] s) {
    for(int i = 0; i < s.length; i++)
      System.out.print(s[i] + " ");
    System.out.println();
  }
  public static void main(String[] args) {
  byte[] b = new byte[15];
    r.nextBytes(b); // Fill with random bytes
    print(b);
    Arrays.sort(b);
    print(b);
    int loc = Arrays.binarySearch(b, b[10]);
    System.out.println("Location of " + b[10] +
                       " = " + loc);
    // Test String sort & search:
    String[] s = randStrings(4, 10);
    print(s);
    Arrays.sort(s);
    print(s);
    loc = Arrays.binarySearch(s, s[4]);
    System.out.println("Location of " + s[4] +
                       " = " + loc);
  }
} ///:~
```

La primera parte de la clase contiene utilidades para generar objetos String aleatorios usando un array de caracteres con letras aleatorias que se van seleccionando. randString() devuelve una cadena de cualquier longitud, y randStrings() crea un array de Strings aleatorios, obteniendo la longitud de cada

String y el tamaño deseado para el array. Los dos métodos print() simplifican el proceso de mostrar los arrays. En main(), Random.nextBytes() rellena el array de argumento con bytes seleccionados aleatoriamente. (Aqui no hay métodos aleatorios para crear arrays de otros tipos primitivos de datos.) Una vez que usted tiene el array, puede ver que solamente se llama a los métodos para realizar ordenaciones -sort()- y busquedas -binarySearch()-. Hay una importante advertencia concerniente a binarySearch(): si usted no llama a sort() antes de llamar a binarySearch(), un comportamiento impresible puede ocurrir, incluyendo bucles infinitos.

La ordenacion y busqueda con Strings parece los mismo, pero cuando usted ejecuta el programa puede observar cosas interasantes: la ordenacion es lexicografica, por lo que las letras mayusculas preceden a las minusculas en el juego de caracteres. Asi, todas las letras mayusculas estan al comienzo de la lista, seguidas por las letras minusculas, por lo que 'Z' precede a 'a'. Esta es la forma en la que usualmente estan ordenados los listines telefonicos.

Comparable y Comparator

¿Que ocurre si esto no es lo que usted quiere?. Por ejemplo, en el indice de este libro esta ordenacion no seria util si usted quiere buscar lo que empieza por 'a', tendria que buscar en dos lugares, 'a' y 'A'.

Cuando usted quiere ordenar un array de objetos Object, esto es un problema. ¨Que determina el orden de dos objetos?. Desafortunadamente, los diseñadores originales de Java no consideraron este importante problema, que deberia haber sido definido en la clase root Object. Como resultado, la ordenacion debe ser impuesta en los Objects desde el exterior, y la nueva librería de colecciones proporciona una forma estandar para hacerlo (que es casi tan buena como si estuviese definida en el Object).

Hay un sort() para arrays de objetos Object (y String, por supuesto, es un Object) que toma un segundo argumento: un objeto que implementa el interface Comparator (parte de la nueva librería de colecciones) y realiza comparaciones con su simple método compare(). Este método toma los dos objetos a comparar como sus argumentos y devuelve un entero negativo si el primer argumento es menor que el segundo, cero si son iguales, y un entero positivo si el primer argumento es mayor que el segundo. Sabiendo esto, la parte String del ejemplo anterior puede ser reimplementada para realizar una ordenacion alfabetica:

```
//: AlphaComp.java
// Using Comparator to perform an alphabetic sort
package c08.newcollections;
import java.util.*;

public class AlphaComp implements Comparator {
```

```
  public int compare(Object o1, Object o2) {
  // Assume it's used only for Strings...
    String s1 = ((String)o1).toLowerCase();
    String s2 = ((String)o2).toLowerCase();
    return s1.compareTo(s2);
  }
  public static void main(String[] args) {
  String[] s = Array1.randStrings(4, 10);
    Array1.print(s);
    AlphaComp ac = new AlphaComp();
    Arrays.sort(s, ac);
    Array1.print(s);
    // Must use the Comparator to search, also:
    int loc = Arrays.binarySearch(s, s[3], ac);
    System.out.println("Location of " + s[3] +
                       " = " + loc);
  }
} ///:~
```

Haciendo un casting a String, el método compare() implicitamente testea para asegurarse de que es usado solo con objetos String -el sistema en tiempo de ejecucion puede conllevar algunas discrepancias-. Despues de convertir a ambos Strings en minusculas, el método String.compareTo() produce los resultados deseados.

Cuando usted use su propio Comparator para realizar una ordenacion, debe utilizar el mismo Comparator cuando use binarySearch().

Las clases de arrays tienen otro método de ordenacion que toma un simple argumetno: un array de objetos Object, pero no con Comparator. Este método sort() debe también tener una forma de comparar dos objetos. Usa el método de comparacion natural que es suministrado por una clase implementando su interface Comparable. Este interface tiene un método simple, compareTo(), el cual compara el objeto de su argumento y devuelve negativo, cero o positivo dependiendo de si es menor, igual o mayor que el argumento. Un ejemplo simple lo demuestra:

```
//: CompClass.java
// A class that implements Comparable
package c08.newcollections;
import java.util.*;

public class CompClass implements Comparable {
private int i;
  public CompClass(int ii) { i = ii; }
  public int compareTo(Object o) {
  // Implicitly tests for correct type:
```

```
    int argi = ((CompClass)o).i;
    if(i == argi) return 0;
    if(i < argi) return -1;
    return 1;
  }
  public static void print(Object[] a) {
    for(int i = 0; i < a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
  public String toString() { return i + ""; }
  public static void main(String[] args) {
    CompClass[] a = new CompClass[20];
    for(int i = 0; i < a.length; i++)
      a[i] = new CompClass((int)(Math.random() *100));
    print(a);
    Arrays.sort(a);
    print(a);
    int loc = Arrays.binarySearch(a, a[3]);
    System.out.println("Location of " + a[3] +
                       " = " + loc);
  }
} ///:~
```

Pro supuesto el método compareTo() puede ser tan complejo como sea neceasrio.

Listas

Una lista puede ordenarse y buscar en ella de la misma forma que en un array. Los métodos estaticos para ordenar y buscar en una lista estan contenidos en la clase Collection, pero estos tienen similares caracteristicas a los de los arrays: sort (List) para ordenar una lista de objetos que implementa Comparable, binarySearch (List, Object) para buscar un objeto en la lista, sort(List, Comparator) para ordenar una lista usando un comparador, y binarySearch(List, Object, Comparator) para buscar un objeto en esa lista. [2]Este ejemplo usa los previamente definidos CompClass y AlphaComp para demostrar las herramientas de ordenacion en las colecciones:

---

[2] En el momento de su escritura, habia sido anunciado un Colelections.stableSort(), para realizar una ordenacion fusionada, pero no estaba disponible para testearlo.

```
//: ListSort.java
// Sorting and searching Lists with 'Collections'
package c08.newcollections;
import java.util.*;

public class ListSort {
  public static void main(String[] args) {
  final int SZ = 20;
  // Using "natural comparison method":
  List a = new ArrayList();
    for(int i = 0; i < SZ; i++)
      a.add(new CompClass((int)(Math.random() *100)));
    Collection1.print(a);
    Collections.sort(a);
    Collection1.print(a);
    Object find = a.get(SZ/2);
    int loc = Collections.binarySearch(a, find);
    System.out.println("Location of " + find +
                       " = " + loc);
    // Using a Comparator:
    List b = new ArrayList();
    for(int i = 0; i < SZ; i++)
      b.add(Array1.randString(4));
    Collection1.print(b);
    AlphaComp ac = new AlphaComp();
    Collections.sort(b, ac);
    Collection1.print(b);
    find = b.get(SZ/2);
    // Must use the Comparator to search, also:
    loc = Collections.binarySearch(b, find, ac);
    System.out.println("Location of " + find +
                                    " = " + loc);
  }
} ///:~
```

El uso de estos métodos es identico que en los Arrays, pero se usan en una lista en lugar de en un array.

TreeMap debe también ordenar sus objetos de acuerdo con Comparable o Comparator.

Utilidades

Aqui tiene otra serie de potentes utilidades de la clase Collection:

enumeration (Collection)	Produce una Enumeration del argumento al estilo antigo.
	Produce el maximo o el minimo

max(Collection)	elemento del argumento usando el método de comparacion natural de los objetos de la colección.
min(Collection)	
max(Collection, Comparator)	Produce el maximo o minimo elemento de la colección Collection usando el comparador Comparator.
min(Collection, Comparator)	
nCopies(int n, Object o)	Devuelve una lista de tamaño n cuyos manejadores pertenecen a o.
SubList(List, int min, int max)	Devuelve una nueva lista <backed> por el argument especificado List, formada por los elementos que hay entre el argumento min y max.

Observe que min() y max() funcionan con objetos Collection, no con listas, por lo que usted no necesita preocuparse sobre si la coleccoin debe ser ordenada o no. (Como se menciono antes, usted no necesita ordenar -sort()- una lista o un array antes de hacer una busqueda binaria -binarySearch().)

Haciendo una colección o un mapa inmodificable

Con frecuencia es conveniente crear un version de una colección o un mapa de solo lectura. La clase Collections le muestra como hacer esto pasando el contenedor original a contenedores en version de solo lectura. Hay cuatroa variantes para este método, uno para cada colección (si es que usted no quiere deleitar a una colección con tipos mas especificos), List, Set y Map. Este ejemplo muestra el camino correcto para construir versiones de solo lectura de cada uno:

```
//: ReadOnly.java
// Using the Collections.unmodifiable methods
package c08.newcollections;
import java.util.*;

public class ReadOnly {
  public static void main(String[] args) {
      Collection c = new ArrayList();
    Collection1.fill(c); // Insert useful data
    c = Collections.unmodifiableCollection(c);
    Collection1.print(c); // Reading is OK
    //! c.add("one"); // Can't change it

    List a = new ArrayList();
    Collection1.fill(a);
```

```
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // Reading OK
    //! lit.add("one"); // Can't change it

    Set s = new HashSet();
    Collection1.fill(s);
    s = Collections.unmodifiableSet(s);
    Collection1.print(s); // Reading OK
    //! s.add("one"); // Can't change it

    Map m = new HashMap();
    Map1.fill(m, Map1.testData1);
    m = Collections.unmodifiableMap(m);
    Map1.print(m); // Reading OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

En cada caso, usted debe llenar el contenedor con datos significativos antes de hacelo de solo lectura. Una vez cargado, la mejor aproximacion es reemplazar el manejador existente con el manejador que es producido por la llamada a "unmodifiable" (no modificable). Por otra parte, esta herramienta también le muestra como mantener un contenedor modificable como privado dentro de una clase y devolver un manejador de solo lectura para este contenedor desde una llamada a un método. Por tanto usted puuede cambiarlo desde dentro de la clase pero cada uno podra solo leerse.

Llamando al método "unmodifiable" para un tipo en particular no causa chequeo en tiempo de compilación, pero una vez que ocurre la transformacion, algunas llamadas a métodos que modifican los contenidos de un contenedor particular produciran una UnsupportedOperationException.

Sincronizando una colección o mapa

La palabra reservada synchronized es una parte importante en el tema de la multitarea, un asunto mas complicado que no será introducido hasta el Capitulo 14. Aqui, tomaremos nota solo de que la clase Collections contiene una forma automatica de sincronizar un contenedor completamente. La sintaxis es similar a los métodos "unmodifiable":

```
//: Synchronization.java
// Using the Collections.synchronized methods
package c08.newcollections;
import java.util.*;
```

```
public class Synchronization {
  public static void main(String[] args) {
    Collection c =
       Collections.synchronizedCollection(
                                new ArrayList());
    List list =
       Collections.synchronizedList(
                                new ArrayList());
       Set s = Collections.synchronizedSet(
                                new HashSet());
    Map m = Collections.synchronizedMap(
                                new HashMap());
  }
} ///:~
```

En este caso, usted inmediatamente pasa el nuevo contenedor a través de el método "synchronized" apropiado; esta forma evita la posibilidad de la exposición accidental de la version no sincronizada.

Las nuevas colecciones también tienen un mecanismo para prevenir mas de un proceso de la modificación de los contenidos de un contenedor. El problema ocurre si usted intenta iterar a través de un contenedor y algún otro proceso pasa por el e inserta, borra o modifica sus objetos. Tal vez usted ya haya pasado por ese objeto, quizás esta mas adelante, quizás el tamaño del contenedor se acorte después de su llamada a size() -hay muchas mas situaciones para el desastre-. La nueva librería de colecciones incorpora un rápido mecanismo de detección de fallos que mira los cambios de un contenedor para ver si su contenedor quedara en un estado correcto. Si detecta que alguien esta modificando el contenedor, inmediatamente produce una ConcurrentModificationExcpetion. Este es el aspecto "fail-fast" (o fallo rápido) -no intenta detectar un problema después de usar complejos algoritmos-.

Resumen

Recuerde las colecciones provistas por la librería estándar de Java (1.0 y 1.1) (BitSet no esta incluido aquí ya que es mas que una clase de propósito general):

1. Un array asocia índices numéricos a objetos. Almacena objetos de un tipo conocido, por lo que usted no tendrá que hacer un casting al resultado cuando quiera ver un objeto. Podrá ser multidimensional, y podrá almacenar primitivas. Sin embargo, su tamaño no puede ser cambiado una vez que se ha creado.
2. Un Vector también asocia índices numéricos a objetos. Puede pensar que los arrays y los objetos son colecciones de acceso aleatorio. El Vector automáticamente se redimensiona cuando usted añade más elementos. Pero un Vector solo puede almacenar manejadores de objetos Object, por lo que no podrá almacenar primitivas y deberá hacer siempre un casting a los resultados cuando saque a un objeto de la colección.

3. Un Hashtable es un tipo de Dictionary, el cual es una forma de asociar, no números, sino objetos a otros objetos. Un Hashtable también soporta acceso aleatorio a objetos, de hecho, su diseño completo esta orientado al acceso rápido.
4. Un Stack es una cola FIFO (last-in -último en entrar-, first-out -primero en salir-).

Si esta familiarizado con las estructuras de datos, debe estar sorprendido de la poca cantidad de colecciones existentes. Desde un punto de vista funcional, necesita realmente un gran numero de colecciones?.

Con un Hashtable, usted puede almacenar y buscar rápidamente, y con un Enumeration, puede iterar a través de una secuencia y realizar operaciones con todos sus elementos. Estas son una potente herramienta, y debe de ser suficiente.

Pero un Hashtable no tiene el concepto de orden. Los vectores y los arrays le proporcionan a usted un orden lineal, pero es muy costoso insertar un elemento entre otros dos existentes. Además, encolar, desencolar, prioridades de colas, y arboles pueden ordenar sus elementos, para luego buscarlos mas rápidamente y moverse por ellos de una forma lineal. Estas estructuras de datos también son útiles, y es por eso por lo que se incluyen en el estándar C++. Por esta razón, usted debe considerar las colecciones de la librería de Java solo como un punto de partida, y si tiene que usar Java 1.0 o 1.1, use JGL cuando necesite mas de esto.

Si puede usar Java 1.2 solo debe usar las nuevas colecciones, las cuales satisfarán todas sus necesidades.

Observe que el grueso de este libro fue creado usando Java 1.1, y por lo que podrá ver durante todo lo que falta de libro, todo se hace a partir de solo las colecciones que esta disponibles en Java 1.1: Vector y Hashtable. Esto es una restricción un poco penosa a veces, pero proporciona mejor compatibilidad con el codigo Java antiguo. Si usted esta escribiendo codigo nuevo en Java 1.2, las nuevas colecciones puede servirle mucho mejor.

Ejercicios

1. Crear una nueva clase llamada Gerbil con el entero gerbilNumer que se inicializa en el constructor (de forma similar el ejemplo del Mouse en este capitulo). Proporcionarle un método llamado hop() que escriba el numero del gerbil, esto es, que salte. Crear un objeto Vector y añadirle un conjunto de objetos Gerbil. Ahora, use el método elementAt() para moverse a través del vector y llame a hop() para cada Gerbil.
2. Modificar el Ejercicio 1 para usar una Enumeration para moverse a través del Vector mientras se llama a hop().
3. En AssocArray.java, cambiar el ejemplo para usar un Hashtable en lugar de un AssocArray.
4. Coja la clase Gerbil del Ejercicio 1 e introdúzcala en una Hashtable, asociando el nombre del Gerbil como un String (la clave o key) para cada Gerbil (el valor) que introduzca en la tabla. Obtenga una Enumeration para las keys() y úsela para moverse a través de la

Hashtable, examinando el Gerbil para cada clave y escribiendo el gerbil a saltar (hop()).

5. Modifique el Ejercicio 1 del Capitulo 6 para usar un Vector para mantener los Rodents y una Enumeration para moverse a través de la secuencia de Rodents. Recuerde que el vector almacena solo Objects, por lo que debe hacer un casting (por ejemplo: RTTI) cuando acceda a Rodents individuales.
6. (Intermedio) En el Capitulo 7, localice el ejemplo del GreenhouseControls, que consiste en tres ficheros. En Controller.java, la clase EventSet es una colección. Modificar su codigo para usar un Stack en lugar de un EvenSet. Esto requerirá mas que solamente reemplazar EvenSet por Stack; también necesitara usar una Enumeration para hacer un ciclo a través del juego de eventos. Probablemente lo encontrara mas fácil si a veces deleita a una colección con un Stack y otras con un Vector.
7. (Avanzado). Encuentre el codigo fuente para Vector en el codigo fuente de la librería de Java que viene con todas las distribuciones de Java. Copie este codigo y crea una version especial llamada intVector que almacene solo enteros. Considere que es lo que cogería para hacer una version especial de Vector para todos los tipos primitivos. Ahora considere que ocurre si quisiera hacer una clase unida a una lista que trabaja con todos los tipos primitivos. Si los tipos parametrizados están siempre implementados en Java, estos podrán proveer un camino para hacer esto automáticamente (así como otros muchos beneficios).

12: El sistema de E/S de Java

Crear un buen sistema de entrada/salida (E/S) es una de las tareas más difíciles para el diseñador de un lenguaje.

Esto es evidente con sólo observar la cantidad de enfoques diferentes. El reto parece estar en cubrir todas las posibles eventualidades. No sólo hay distintas fuentes y consumidores de información de E/S con las que uno desea comunicarse (archivos, la consola, conexiones de red), pero hay que comunicarse con ellos de varias maneras (secuencial, acceso aleatorio, espacio de almacenamiento intermedio, binario, carácter, mediante líneas, mediante palabras, etc.).

Los diseñadores de la biblioteca de Java acometieron este problema creando muchas clases. De hecho, hay tantas clases para el sistema de E/S de Java que puede intimidar en un principio (irónicamente, el diseño de la E/S de Java evita una explosión de clases). También hubo un cambio importante en la biblioteca de Java después de la versión 1.0, al suprimir la biblioteca original orientada a **bytes** por clases de E/S orientadas a **char** basadas en Unicode. Como resultado hay que aprender un número de clases aceptable antes de entender suficientemente un esbozo de la E/S de Java para poder usarla adecuadamente. Además, es bastante importante entender la historia de la biblioteca de E/S, incluso si tu primera reacción es: NO me aburras con esta historia, simplemente dime cómo usarla!" El problema es que sin la perspectiva histórica es fácil confundirse con algunas de las clases, y no comprender cuándo debería o no usarse.

Este capítulo presentará una introducción a la variedad de clases de E/S contenidas en la biblioteca estándar de Java, y cómo usarlas.

La clase File

Antes de comenzar a ver las clases que realmente leen y escriben datos en flujos, se echará un vistazo a una utilidad proporcionada por la biblioteca para manejar aspectos relacionados con directorios de archivos.

La clase **File** tiene un nombre engañoso -podría pensarse que hace referencia a un archivo, pero no es así. Puede representar, o bien el ***nombre*** de un archivo particular, o los ***nombres*** de un conjunto de archivos de un directorio. Si se trata de un conjunto de archivos, se puede preguntar por el conjunto con el método **list**( ), que devuelve un array de **Strings**. Tiene sentido devolver un array en vez

de una de las clases contenedoras flexibles porque el número de elementos es fijo, y si se desea listar un directorio diferente basta con crear un objeto **File** diferente. De hecho, "FilePath" habría sido un nombre mejor para esta clase. Esta sección muestra un ejemplo de manejo de esta clase, incluyendo la **interfaz FilenameFilter** asociada.

Un generador de listados de directorio

Suponga que se desea ver el contenido de un directorio. El objeto **File** puede listarse de dos formas. Si se llama a **list**( ) sin parámetros, se logrará la lista completa de lo que contiene el objeto **File.** Sin embargo, si se desea una lista restringida -por ejemplo, si se desean todos los archivos de extensión **.java** se usará un "filtro de directorio", que es una clase que indica cómo seleccionar los objetos **File** a mostrar.

He aquí el código para el ejemplo. Nótese que el resultado se ha ordenado sin ningún tipo de esfuerzo, de forma alfabética, usando el método **java.utils.Array.sort**( ) y el **ComparadorAlfabetico** definido en el Capítulo 11:

```
//: c11:DirList.java
// Displays directory listing.
import java.io.*;
import java.util.*;
import com.bruceeckel.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,
      new AlphabeticComparator());
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
}

class DirFilter implements FilenameFilter {
  String afn;
  DirFilter(String afn) { this.afn = afn; }
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.indexOf(afn) != -1;
  }
```

```
} ///:~
```

La clase **FiltroDirectorio** "implementa" la **interfaz FilenameFilter.** Es útil ver lo simple que es la **interfaz FilenameFilter:**

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

Dice que este tipo de objeto proporciona un método denominado **accept**( ). La razón que hay detrás de la creación de esta clase es proporcionar el método **accept**( ) al método **list**( ) de forma que **list**( ) pueda "retrollamar" a **accept**( ) para determinar qué nombres deberían ser incluidos en la lista. Por consiguiente, a esta técnica se le suele llamar *retrollamada* o a veces *functor* (es decir, **FiltroDirectorio** es un *functor* porque su único trabajo es albergar un método) o *Patrón Comando.*
Dado que **list**( ) toma un objeto **FilenameFilter** como parámetro, se le puede pasar un objeto de cualquier clase que implemente **FilenameFilter** para elegir (incluso en tiempo de ejecución) cómo se comportará el método **list**( ). El propósito de una retrollamada es proporcionar flexibilidad al comportamiento del código.

FiltroDirectorio muestra que, justo porque una **interfaz** contenga sólo un conjunto de métodos, uno no está restringido a escribir sólo esos métodos. (Sin embargo, al menos hay que proporcionar definiciones para todos los métodos de la interfaz.) En este caso, se crea también el constructor **FiltroDirectorio.**

El método **accept**( ) debe aceptar un objeto **File** que represente el directorio en el que se encuentra un archivo en particular, y un **String** que contenga el nombre de *ese* archivo. Se podría elegir entre utilizar o ignorar cualquiera de estos parámetros, pero probablemente se usará al menos el nombre del archivo. Debe recordarse que el método **list**( ) llama a **accept**( ) por cada uno de los nombres de archivo del objeto directorio para ver cuál debería incluirse -lo que se indica por el resultado **boolean** devuelto por **accept**( ).

Para asegurarse de que el elemento con el que se está trabajando es sólo un nombre de archivo sin información de ruta, todo lo que hay que hacer es tomar el **String** y crear un objeto **File** a partir del mismo, después llamar a **getName**( ), que retira toda la información relativa a la ruta (de forma independiente de la plataforma). Después, **accept**( ) usa el método **indexOf**( ) de la clase **String** para ver si la cadena de caracteres a buscar `afn` aparece en algún lugar del nombre del archivo. Si se encuentra `afn` en el string, el valor devuelto es el índice de comienzo de &, mientras que si no se encuentra, se devuelve el valor -1. Hay que ser conscientes de que es una búsqueda de cadenas de caracteres simple y que no tiene expresiones de emparejamiento de comodines -como por ejemplo "for?.b?*"- lo cual sería más difícil de implementar.

El método **list**( ) devuelve un array. Se puede preguntar por la longitud del mismo y recorrerlo seleccionando sus elementos. Esta habilidad de pasar un array hacia y desde un método es una gran mejora frente al comportamiento de C y C++.

Clases internas anónimas

Este ejemplo es ideal para reescribirlo utilizando una clase interna anónima (descritas en el Capítulo 8). En principio, se crea un método **filtrar**( ) que devuelve una referencia a **FilenameFilter:**

```
//: c11:DirList2.java
// Uses anonymous inner classes.
import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList2 {
  public static FilenameFilter
  filter(final String afn) {
    // Creation of anonymous inner class:
    return new FilenameFilter() {
      String fn = afn;
      public boolean accept(File dir, String n) {
        // Strip path information:
        String f = new File(n).getName();
        return f.indexOf(fn) != -1;
      }
    }; // End of anonymous inner class
  }
  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,
      new AlphabeticComparator());
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
} ///:~
```

Nótese que el parámetro que se pase a **filtrar**( ) debe ser **final.** Esto es necesario para que la clase interna anónima pueda usar un objeto de fuera de su ámbito.

El diseño es una mejora porque la clase **FilenameFilter** está ahora firmemente ligada a **ListadoDirectorio2.** Sin embargo, es posible llevar este enfoque un paso más allá y definir la clase interna anónima como un argumento de **list**( ), en cuyo caso es incluso más pequeña:

```
//: c11:DirList3.java
// Building the anonymous inner class "in-place."
import java.io.*;
import java.util.*;
import com.bruceeckel.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() {
        public boolean
        accept(File dir, String n) {
          String f = new File(n).getName();
          return f.indexOf(args[0]) != -1;
        }
      });
    Arrays.sort(list,
      new AlphabeticComparator());
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
} ///:~
```

El argumento del **main**( ) es ahora **final**, puesto que la clase interna anónima usa directamente
args[0].

Esto muestra cómo las clases anónimas internas permiten la creación de clases rápida y limpia- mente para solucionar problemas. Puesto que todo en Java se soluciona con clases, ésta puede ser una técnica de codificación útil. Un beneficio es que mantiene el código que soluciona un problema en particular aislado y junto en el mismo sitio. Por otro lado, no es siempre fácil de leer, por lo que hay que usarlo juiciosamente.

Comprobando y creando directorios

La clase File es más que una simple representación de un archivo o un directorio existentes.

También se puede usar un objeto File para crear un nuevo directorio o una trayectoria de directorio completa si ésta no existe. También se pueden mirar las características de archivos (tamaño, fecha de la última modificación, lectura/escritura), ver si un objeto File representa un archivo o un directorio, y borrar un archivo. Este programa muestra algunos de los otros métodos disponibles con la clase File (ver la documentación HTML de *http://iava.sun.com* para obtener el conjunto completo) :

```
//: c11:MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;

public class MakeDirectories {
  private final static String usage =
    "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\n";
  private static void usage() {
    System.err.println(usage);
    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; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
      count++;
      del = true;
    }
    for( ; count < args.length; count++) {
      File f = new File(args[count]);
      if(f.exists()) {
        System.out.println(f + " exists");
        if(del) {
          System.out.println("deleting..." + f);
          f.delete();
        }
      }
      else { // Doesn't exist
        if(!del) {
          f.mkdirs();
          System.out.println("created " + f);
        }
      }
      fileData(f);
    }
  }
} ///:~
```

En **datosArchivo**( ) se pueden ver varios métodos de investigación que se usan para mostrar la información sobre la trayectoria de un archivo o un directorio.

El primer método ejercitado por el método **main**( ) es **renameTo**( ), que permite renombrar (o mover) un archivo a una ruta totalmente nueva representada por un parámetro, que es otro objeto **File.**

Esto también funciona con directorios de cualquier longitud. Si se experimenta con el programa, se verá que se pueden construir rutas de cualquier complejidad, pues **mkdirs**( ) hará todo el trabajo.

Entrada y salida

Las bibliotecas de E/S usan a menudo la abstracción del flujo, que representa cualquier fuente o consumidor de datos como un objeto capaz de producir o recibir fragmentos de código. El flujo oculta los detalles de lo que ocurre con los datos en el dispositivo de E/S real.

Las clases de E/S de la biblioteca de Java se dividen en entrada y salida, como ocurre con los datos en el dispositivo de E/S real. Por herencia, todo lo que deriva de las clases **InputStream** o **Reader** tiene los métodos básicos **read**( ) para leer un simple byte o un array de bytes. Asimismo, todo lo que derive de las clases **OutputStream** o **Writer** tiene métodos básicos denominados **write**( ) para escribir un único byte o un array de bytes. Sin embargo, generalmente no se usarán estos métodos; existen para que otras clases puedan utilizarlos -estas otras clases proporcionan una interfaz más útil. Por consiguiente, rara vez se creará un objeto flujo usando una única clase, sino que se irán apilando en capas diversos objetos para proporcionar la funcionalidad deseada. El hecho de crear más de un objeto para crear un flujo resultante único es la razón primaria por la que la biblioteca de flujos de Java es tan confusa. Ayuda bastante clasificar en tipos las clases en base a su funcionalidad. En Java 1.0, los diseñadores de bibliotecas comenzaron decidiendo que todas las clases relacionadas con entrada heredarían de **InputStream,** y que todas las asociadas con la salida heredarían de **OutputStream.**

Tipos de InputStream

El trabajo de **InputStream** es representar las clases que producen entradas desde distintas fuentes. Éstas pueden ser:

1. Un array de bytes.
2. Un objeto **String.**
3. Un archivo.
4. Una "tubería", que funciona como una tubería física: se ponen elementos en un extremo y salen por el otro.
5. Una secuencia de otros flujos, de forma que se puedan agrupar todos en un único flujo.
6. Otras fuentes, como una conexión a Internet (esto se verá al final del capítulo).

Cada una de éstas tiene asociada una subclase de **InputStream.** Además, el **FilterInputStream** es también un tipo de **InputStream,** para proporcionar una clase base para las clases "decoradoras" que adjuntan atributos o interfaces útiles a los flujos de entrada. Esto se discutirá más tarde.

Tabla 1. Tipos de InputStream

Clase	**Función**	**Parámetros del constructor**
		Cómo usarla
ByteArray-InputStream	Permite usar un espacio de	El intermedio del que extraer los bytes.

Clase	Función	Parámetros del constructor / Cómo usarla
	almacenamiento intermedio de memoria como un **InputStream**	Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar un interfaz útil.
StringBuffer-InputStream	Convierte un **String** en un **InputStream**	Un **String**. La implementación subyacente usa, de hecho, un **StringBuffer**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
File-InputStream	Para leer información de un archivo	Un **String** que represente al nombre del archivo, o un objeto **File** o **FileDescriptor**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
Piped-InputStream	Produce los datos que se están escribiendo en el **PipedOutputStream** asociado. Implementa el concepto de "entubar".	**PipedOutputStream**
		Como una fuente de datos en multihilado. Se conecta al objeto FilterInputStream para proporcionar una interfaz útil.
Sequence-InputStream	Convierte dos o más objetos **InputStream** en un **InputStream** único.	Dos objetos **InputStream** o una **Enumeration** para contener objetos **InputStream**.
		Como una fuente de datos. Se conecta al objeto **FilterInputStream** para proporcionar una interfaz útil.
Filter-InputStream	Clase abstracta que es una interfaz para los	Ver Tabla 3.

Clase	Función	Parámetros del constructor
		Cómo usarla
	decoradores que proporcionan funcionalidad útil a otras clases **InputStream**. Ver Tabla 3.	Ver Tabla 3.

Tipos de OutputStream

Esta categoría incluye las clases que deciden dónde irá la salida: un array de **bytes** (sin embargo, no **String;** presumiblemente **se** puede **crear** uno usando un array de bytes), un fichero, o una "tubería".

Además, el **FilterOutputStream** proporciona una clase base para las clases "decorador" que adjuntan atributos o interfaces útiles de flujos de salida. Esto se discute más tarde.

Tabla 2. Tipos de OutputStream

Clase	Función	Parámetros del constructor
		Cómo usarla
ByteArray-OutputStream	Crea un espacio de Almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio de almacenamiento intermedio.	Tamaño opcional inicial del espacio de almacenamiento intermedio.
		Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
File-OutputStream	Para enviar información a un archivo.	Un **String**, que representa el nombre de archivo, o un objeto **File** o un objeto **FileDescriptor**.
		Para designar el destino de los datos. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.

Clase	Función	Parámetros del constructor Cómo usarla
Piped-OutputStream	Cualquier información que se desee escribir aquí acaba automáticamente como entrada del **PipedInputStream** asociado. Implementa el concepto de "entubar".	**PipedInputStream** Para designar el destino de los datos para multihilo. Conectarlo a un objeto **FilterOutputStream** para proporcionar una interfaz útil.
Filter-OutputStream	Clase abstracta que es una interfaz para los decoradores que proporcionan funcionalidad útil a las otras clases **OutputStream**. Ver Tabla 4.	Ver Tabla 4. Ver Tabla 4.

Añadir atributos e interfaces Útiles

Al uso de objetos en capas para añadir dinámica y transparentemente responsabilidades a objetos individuales se le denomina patrón *Decorador.* (Los Patrones **[1]** son el tema central de *Thinking in Patterns with Java,* descargable de *http://www.BruceEckel.com.)* El patrón decorador especifica que todos los objetos que envuelvan el objeto inicial tengan la misma interfaz. Así se hace uso de la transparencia del decorador -se envía el mismo mensaje a un objeto esté o no decorado. Éste es el motivo de la existencia de clases "filtro" en la biblioteca E/S de Java: la clase abstracta "filter" es la clase base de todos los decoradores. (Un decorador debe tener la misma interfaz que el objeto que decora, pero el decorador también puede extender la interfaz, lo que ocurre en muchas de las clases "filter".)

[1] *Desing Patterns,* Erich Gamma *et al.,* Addison-Wesley *1995.*

Los decoradores se han usado a menudo cuando se generan numerosas subclases para satisfacer todas las combinaciones posibles necesarias -tantas clases que resultan poco prácticas. La biblioteca de E/S de Java requiere muchas combinaciones distintas de características, siendo ésta la razón del uso del patrón decorador. Sin embargo este patrón tiene un inconveniente. Los decoradores dan mucha más flexibilidad al escribir un programa (puesto que se pueden mezclar y emparejar atributos fácilmente), pero añaden complejidad al código. La razón por

la que la biblioteca de E/S es complicada de usar es que hay que crear muchas clases -los tipos de E/S básicos más todos los decoradores- para lograr el único objeto de E/S que se quiere.

Las clases que proporcionan la interfaz decorador para controlar un **InputStream** o un **Output**
Stream particular son **FilertInputStream** y **FilterOuputStream** -que no tienen nombres muy intuitivos. Estas dos últimas clases son abstractas, derivadas de las clases base de la biblioteca de E/S, **InputStream** y **OutputStream,** el requisito clave del decorador (de forma que proporciona la interfaz común a todos los objetos que se están decorando).

Leer de un InputStream con un FilterInputStream

Las clases **FilterInputStream** llevan a cabo dos cosas significativamente diferentes. **DataInputStream** permite leer distintos tipos de datos primitivos, además de objetos **String.** (Todos los métodos empiezan sor "read", como **readRyte**( ). **readFloat**( ). etc.) Esto, junto con su compañero **DataOutputStream,** permite mover datos primitivos de un sitio a otro vía un flujo. Estos "lugares" vendrán determinados por las clases de la Tabla 1.

Las clases restantes modifican la forma de comportarse internamente de **InputStream:** haga uso o no de espacio de almacenamiento intermedio, si mantiene un seguimiento de las líneas que lee (permitiendo preguntar por el número de líneas, o establecerlo) o si se puede introducir un único carácter.

Las dos últimas clases se parecen mucho al soporte para la construcción de un compilador (es decir, *se* añadieron para poder construir el propio compilador de Java), por lo que probablemente no se usen en programación en general.

Probablemente se necesitará pasar la entrada por un espacio de almacenamiento intermedio casi siempre, independientemente del dispositivo de E/S al que se esté conectado, por lo que tendría más sentido que la biblioteca de E/S fuera un caso excepcional (o simplemente una llamada a un método) de entrada sin espacio de almacenamiento intermedio, en vez de una entrada con espacio de almacenamiento intermedio.

Tabla 3. Tipos de FilterInputStream

Clase	Función	Parámetros del constructor
		Cómo usarla

Data-InputStream	Usado junto con **DataOutputStream**, de forma que se puedan leer datos primitivos (int, char, long, etc.) de un flujo de forma portable.	**InputStream** Contiene una interfase completa que permite leer tipos primitivos.
Buffered-InputStream	Se usa para evitar una lectura cada vez que se soliciten nuevos datos. Se está diciendo "utiliza un espacio de almacenamiento intermedio".	**InputStream**, con tamaño de espacio de almacenamiento intermedio opcional. No proporciona una interfaz per se, simplemente el requisito de que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.
LineNumber-InputStream	Mantiene un seguimiento de los números de línea en el flujo de entrada; se puede llamar a **getLineNumber**() y a **setLineNumber**(**int**).	**InputStream** Simplemente añade numeración de líneas, por lo que probablemente se adjunte un objeto interfaz.
Pushback-InputStream	Tiene un espacio de almacenamiento intermedio de un byte para el último carácter a leer.	**InputStream** Se usa generalmente en el escáner de un compilador y probablemente se incluyó porque lo necesitaba el compilador de Java. Probablemente prácticamente nadie la utilice.

Escribir en un OutputStream con FiIterOutputStream

El complemento a **DataInputStream** es **DataOutputStream,** que da formato a los tipos primitivos y objetos **String,** convirtiéndolos en un flujo de forma que cualquier **DataInputStream,** de cualquier máquina, bs pueda leer. Todos los

métodos empiezan por "write", como **writeByte**( ), **writeHoat**( ), etc. La intención original para **PrintStream** era que imprimiera todos los tipos de datos primitivos así como los objetos **String** en un formato visible. Esto es diferente de **DataOutputStream,** cuya meta es poner elementos de datos en un flujo de forma que **DataInputStream** pueda reconstruirlos de forma portable.

Los dos métodos importantes de **PrintStream** son **print**( ) y **printin**( ), sobrecargados para imprimir todo los tipos. La diferencia entre **print**( ) y **printin**( ) es que la útlima añade una nueva línea al acabar. **PrintStream** puede ser problemático porque captura todas las **IOExpections.** (Hay que probar explícitamente el estado de error con **checkError**( ), que devuelve **true** si se ha producido algún error.) Además, **PrintStream** no se internacionaliza adecuadamente y no maneja saltos de línea independientemente de la plataforma (estos problemas se solucionan con **PrintWriter).**

BufferedOutputStream es un modificador que dice al flujo que use espacios de almacenamiento intermedio, de forma que no se realice una lectura física cada vez que se escribe en el flujo. Probablemente siempre se deseará usarlo con archivos, y probablemente la E/S de consola.

Tabla 4. Tipos de FilterOutputStream

Clase	**Función**	**Parámetros del constructor** **Cómo usarla**
Data-OutputStream	Usado junto con **DataInputStream**, de forma que se puedan escribir datos primitivos (int, char, long, etc.) de un flujo de forma portable.	**OutputStream** Contiene una interfaz completa que permite escribir tipos de datos primitivos.
PrintStream	Para producir salida formateada. Mientras que **DataOutputStream** maneja el almacenamiento de datos, **PrintStream** maneja su visualización.	**InputStream**, con un **boolean** opcional que indica que se vacía el espacio de almacenamiento intermedio con cada nueva línea. Debería ser la envoltura "final" del objeto **OutputStream**. Probablemente se usará mucho.

Buffered-OutputStream	Se usa para evitar una escritura física cada vez que se envía un fragmento de datos. Se está diciendo "Usar un espacio de almacenamiento intermedio". Se puede llamar a **flush**( ) para vaciar el espacio de almacenamiento intermedio.	**OutputStream** con tamaño del espacio de almacenamiento intermedio.
		No proporciona una interfaz per se, simplemente pide que se use un espacio de almacenamiento intermedio. Adjuntar un objeto interfaz.

Readers & Writers

Java 1.1. hizo algunas modificaciones fundamentales a la biblioteca de flujos de E/S fundamental de Java (sin embargo, Java 2 no aportó modificaciones significativas). Cuando se observan las clases **Reader** y **Writer,** en un principio se piensa (como hicimos) que su intención es reemplazar las clases **InputStream** y **OutputStream.** Pero ése no es el caso. Aunque se desecharon algunos aspectos de la biblioteca de flujos original (si se usan estos aspectos se recibirá un aviso del compilador), las clases **InputStream** y **OutputStream** siguen proporcionando una funcionalidad valiosa en la forma de E/S orientada al **byte,** mientras que las clases **Reader** y **Writer** proporcionan E/S compatible Unicode basada en caracteres. Además:

1. Java 1.1 añadió nuevas clases a la jerarquía **InputStream** y **OutputStream,** por lo que es obvio que no se estaban reemplazando estas clases.
2. Hay ocasiones en las que hay que usar clases de la jerarquía "byte" *en combinación con* clases de la jerarquía "carácter". Para lograr esto hay clases "puente": **InputStreamReader** convierte un **InputStream** en un **Reader,** y **OutputStreamWriter** convierte un **OutputStream** en un **Writer.**

La razón más importante para la existencia de las jerarquías **Reader** y **Writer** es la internacionalización. La antigua jerarquía de flujos de E/S sólo soporta flujos de 8 bits no manejando caracteres Unicode de 16 bits correctamente. Puesto que Unicode se usa con fines de internacionalización (y el **char** nativo de Java es Unicode de 16 bits), se añadieron las jerarquías **Reader** y **Writer** para dar soporte Unicode en todas las operaciones de E/S. Además, se diseñaron las nuevas bibliotecas de forma que las operaciones se llevaran a cabo de forma más rápida que antiguamente.

En la práctica y en este libro, intentaremos proporcionar un repaso de las clases, pero asumimos que se usará la documentación *en línea* para concretar todos los detalles, así como una lista exhaustiva de los métodos.

Fuentes y consumidores de datos

Casi todas las clases de flujos de E/S de Java originales tienen sus correspondientes clases **Reader** y **Writer** para proporcionar manipulación Unicode nativa. Sin embargo, hay algunos lugares en los que la solución correcta la constituyen los **InputStreams** y **OutputStreams** orientados a **byte;** concretamente, las bibliotecas **java.util.zip** son orientadas a **byte** en vez de orientadas a **char.** Por tanto, el enfoque más sensato es *intentar* usar las clases **Reader** y **Writer** siempre que se pueda, y se descubrirán posteriormente aquellas situaciones en las que hay que usar las otras bibliotecas, ya que **el** código no compilará.

He aquí una tabla que muestra la correspondencia entre fuentes y consumidores de información (es decir, de dónde y a dónde van físicamente los datos) dentro de ambas jerarquías:

Fuentes & Cusumidores: Clase Java 1.0	**Clase Java 1.1 correspondiente**
InputStream	**Reader** convertidor: **InputStreamReader**
OutputStream	**Writer** convertidor: **OutputStreamWriter**
FileInputStream	**FileReader**
FileOutputStream	**FileWriter**
StringBufferInputStream	**StringReader**
(sin clase correspondiente)	**StringWriter**
ByteArrayInputStream	**CharArrayReader**
ByteArrayOutputStream	**CharArrayWriter**
PipedInputStream	**PipedReader**
PipedOutputStream	**PipedWriter**

En general, se descubrirá que las interfaces de ambas jerarquías son similares cuando no idénticas.

Modificar el comportamiento del flujo

En el caso de **InputStreams** y **OutputStreams** se adaptaron los flujos para necesidades particulares utilizando subclases "decorador" de **FilterInputStream** y **FilterOutputStream.** Las jerarquías de clases **Reader** y **Writer** continúan usando esta idea -aunque no exactamente.

En la tabla siguiente, la correspondencia es una aproximación más complicada que en la tabla anterior. La diferencia se debe a la organización de las clases:

mientras que **BufferedOutputStream** es una subclase de **FilterOuputStream, BufferedWriter** no es una subclase de **FilterWriter** (que, aunque es **abstract,** no tiene subclases, por lo que parece haberse incluido como contenedor o simplemente de forma que nadie la busque sin fruto). Sin embargo, las interfaces de las clases coinciden 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	Usar **DataInputStream** (Excepto cuando se necesite usar **readLine**(), caso en que debería usarse un **BufferedReader**)
PrintStream	**PrintWriter**
LineNumberInputStream	**LineNumberReader**
StreamTokenizer	**StreamTokenizer** (usar en vez de ello el constructor que toma un **Reader**)
PushBackInputStream	**PushBackReader**

Hay algo bastante claro: siempre que se quiera usar un **readLine**( ), no se debería hacer con un
DataInputStream nunca más (se mostrará en tiempo de compilación un mensaje indicando que se trata de algo obsoleto), sino que debe usarse en su lugar un **BufferedReader. DataInputStream,** sigue siendo un miembro "preferente" de la biblioteca de E/S.
Para facilitar la transición de cara a usar un **PrintWriter,** éste tiene constructores que toman cualquier objeto **OutputStream,** además de objetos **Writer.** Sin embargo, **PrintWriter** no tiene más soporte para dar formato que el proporcionado por **PrintStream;** las interfaces son prácticamente las mismas.

El constructor **PrintWriter** también tiene la opción de hacer vaciado automático, lo que ocurre tras todo **println**( ) si *se* ha puesto a uno *el* ***flag*** del constructor.

Clases no cambiadas

Java 1.1 no cambió algunas clases de Java 1.0:

Clases de Java 1.0 sin clases correspondientes Java 1.1
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, en particular, se usa sin cambios, por tanto, para almacenar y recuperar datos en un formato transportable se usan las jerarquías **InputStream** y **OutputStream.**

Por sí mismo: RandomAccessFile

RandomAccessFile se usa para los archivos que contengan registros de tamaño conocido, de forma que se puede mover de un registro a otro utilizando **seek**( ), para después leer o modificar los registros. Éstos no tienen por qué ser todos del mismo tamaño; simplemente hay que poder determinar lo grandes que son y dónde están ubicados dentro del archivo.

En primera instancia, cuesta creer que **RandomAccessFile** no es parte de la jeraquía **InputStream** o **OutputStream.** Sin embargo, no tiene ningún tipo de relación con esas jerarquías con la excepción de que implementa las interfaces **DataInput** y **DataOutput** (que también están implementados por **DataInputStream** y **DataOutputStream)** . Incluso no usa ninguna funcionalidad de las clases **InputStream** u **OutputStream** existentes -es una clase totalmente independiente, escrita de la nada, con métodos exclusivos (en su mayoría nativos). La razón para ello puede ser que **RandomAccessFile** tiene un comportamiento esencialmente distinto al de otros tipos de E/S, puesto que se puede avanzar y retroceder dentro de un archivo. Permanece aislado, como un descendiente directo de **Object.**

Fundamentalmente, un **RandomAccessFile** funciona igual **que** un **DataInputStream** unido a un **DataOutputStream,** junto con los métodos **getFilePointer**( ) para averiguar la posición actual en el archivo, **seek**( ) para moverse a un nuevo punto del archivo, y **length**( ) para determinar el tamaño máximo del mismo. Además, los constructores requieren de un segundo parámetro (idéntico al de **fopen**( ) en C) que indique si se está simplemente leyendo (**"r")** al azar, o leyendo y escribiendo (**"rw").** No hay soporte para archivos de sólo escritura, lo cual podría sugerir que **RandomAccessFile** podría haber funcionado bien si hubiera heredado de **DataInputStream.**

Los métodos de búsqueda sólo están disponibles en **RandomAccessFile,** que sólo funciona para archivos. **BufferedInputStream** permite **mark**( ) una posición

(cuyo valor se mantiene en una variable interna única) y hacer un **reset**( ) a esa posición, pero no deja de ser limitado y, por tanto, no muy útil.

Usos típicos de flujos de E/S

Aunque se pueden combinar las clases de *flujos* de E/S de muchas formas, probablemente cada uno solo haga uso de unas pocas combinaciones. Se puede usar el siguiente ejemplo como una referencia básica; muestra la creación y uso de las configuraciones de E/S típicas. Nótese que cada configuración empieza con un número comentado y un título que corresponde a la cabecera de la explicación apropiada que sigue en el texto.

```
//: c11:IOStreamDemo.java
// Typical I/O stream configurations.
import java.io.*;

public class IOStreamDemo {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    // 1. Reading input by lines:
    BufferedReader in =
      new BufferedReader(
        new FileReader("IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();

    // 1b. Reading standard input:
    BufferedReader stdin =
      new BufferedReader(
        new InputStreamReader(System.in));
    System.out.print("Enter a line:");
    System.out.println(stdin.readLine());

    // 2. Input from memory
    StringReader in2 = new StringReader(s2);
    int c;
    while((c = in2.read()) != -1)
      System.out.print((char)c);

    // 3. Formatted memory input
    try {
      DataInputStream in3 =
        new DataInputStream(
          new ByteArrayInputStream(s2.getBytes()));
      while(true)
```

```
    System.out.print((char)in3.readByte());
} catch(EOFException e) {
  System.err.println("End of stream");
}

// 4. File output
try {
  BufferedReader in4 =
    new BufferedReader(
      new StringReader(s2));
  PrintWriter out1 =
    new PrintWriter(
      new BufferedWriter(
        new FileWriter("IODemo.out")));
  int lineCount = 1;
  while((s = in4.readLine()) != null )
    out1.println(lineCount++ + ": " + s);
  out1.close();
} catch(EOFException e) {
  System.err.println("End of stream");
}

// 5. Storing & recovering data
try {
  DataOutputStream out2 =
    new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
  out2.writeDouble(3.14159);
  out2.writeChars("That was pi\n");
  out2.writeBytes("That was pi\n");
  out2.close();
  DataInputStream in5 =
    new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
  BufferedReader in5br =
    new BufferedReader(
      new InputStreamReader(in5));
  // Must use DataInputStream for data:
  System.out.println(in5.readDouble());
  // Can now use the "proper" readLine():
  System.out.println(in5br.readLine());
  // But the line comes out funny.
  // The one created with writeBytes is OK:
  System.out.println(in5br.readLine());
} catch(EOFException e) {
  System.err.println("End of stream");
}
```

```
    // 6. Reading/writing random access files
    RandomAccessFile rf =
      new RandomAccessFile("rtest.dat", "rw");
    for(int i = 0; i < 10; i++)
      rf.writeDouble(i*1.414);
    rf.close();

    rf =
      new RandomAccessFile("rtest.dat", "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();

    rf =
      new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
      System.out.println(
        "Value " + i + ": " +
        rf.readDouble());
    rf.close();
  }
} ///:~
```

He aquí las descripciones de las secciones numeradas del programa:

Flujos de entrada

Las Secciones 1-4 demuestran la creación y uso de flujos de entrada. La Sección 4 muestra también el uso simple de un flujo de salida.

1. Archivo de entrada utilizando espacio de almacenamiento intermedio

Para abrir un archivo para entrada de caracteres se usa un **FileInputReader** junto con un objeto **File** o **String** como nombre de archivo. Para lograr mayor velocidad, se deseará que el archivo tenga un espacio de almacenamiento intermedio de forma que se dé la referencia resultante al constructor para un **BufferedReader.** Dado que **BufferedReader** también proporciona el método **readLine**( ), éste es el objeto final y la interfaz de la que se lee. Cuando se llegue al final del archivo, **readLine**( ) devuelve **null,** por lo que es éste el valor que se usa para salir del bucle **while.** El **String s2** se usa para acumular todo el contenido del archivo (incluyendo las nuevas líneas que hay que añadir porque readLine( ) las quita). Después se usa *s2* en el resto de porciones del programa. Finalmente, se invoca a **close**( ) para cerrar el archivo. Técnicamente, se llamará a **close**( ) cuando se ejecute **finalize**( ), cosa que se supone que ocurrirá (se active o no el recolector de basura) cuando se acabe el programa. Sin embargo,

esto se ha implementado inconsistentemente, por lo que el único enfoque seguro es el de invocar explícitamente a **close**() en el caso de manipular archivos.

La sección l b muestra cómo se puede envolver **System.in** para leer la entrada de la consola. **System.in** es un **DataInputStrearn** y **BufferedReader** necesita un parámetro **Reader,** por lo que se hace uso de **InputStreamReader** para llevar a cabo la traducción.

2. Entrada desde memoria

Esta sección toma el **String s2,** que ahora contiene todos los contenidos del archivo y lo usa para crear un **StringReader.** Después se usa **read**( ) para leer cada carácter de uno en uno y enviarlo a la consola. Nótese que **read**( ) devuelve el siguiente **byte** como un **int** por lo que hay que convertirlo en **char** para que se imprima correctamente.

3. Entrada con formato desde memoria

Para leer datos "con formato", se usa un **DataInputStrearn,** que es una clase de E/S orientada a **byte** (en vez de orientada a **char).** Por consiguiente se deben usar todas las clases **InputStream** en vez de clases **Reader.** Por supuesto, se puede leer cualquier cosa (como un archivo) como si de bytes se tratara, usando clases **InputStream,** pero aquí se usa un **String.** Para convertir el **String** en un array de bytes, apropiado para un **ByteArrayInputStream, String** tiene un método **getBytes**( ) que se encarga de esto. En este momento, se tiene un **InputStream** apropiado para manejar **DataInputStream.** Si se leen los caracteres de un **DataInputStream** de uno en uno utilizando **readByte**( ), cualquier valor de byte constituye un resultado legítimo por lo que no se puede usar el valor de retorno para detectar el final de la entrada. En vez de ello, se puede usar el método **available**( ) para averiguar cuántos caracteres más quedan disponibles. He aquí un ejemplo que muestra cómo leer un archivo byte a byte:

```
//: c11:TestEOF.java
// Testing for the end of file
// while reading a byte at a time.
import java.io.*;

public class TestEOF {
  // Throw exceptions to console:
  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());
    }
} ///:~
```

Nótese que **available**( ) funciona de forma distinta en función del tipo de medio desde el que se esté leyendo; literalmente es "el número de bytes que se pueden leer sin bloqueo". Con archivos, esto equivale a todo el archivo pero con un tipo de flujo distinto podría no ser así, por lo que debe usarse con mucho cuidado.

También se podría detectar el fin de la entrada en este tipo de casos mediante una excepción. Sin embargo, el uso de excepciones para flujos de control se considera un mal uso de esta característica.

4. Salida a archivo

Este ejemplo también muestra cómo escribir datos en un archivo. En primer lugar, se crea un
FileWriter para conectar con el archivo. Generalmente se deseará pasar la salida a través de un espacio de almacenamiento intermedio, por lo que se genera un **BufferedWriter** (es conveniente intentar retirar este envoltorio para ver su impacto en el rendimiento -el uso de espacios de almacenamiento intermedio tiende a incrementar considerablemente el rendimiento de las operaciones de E/S). Después, se convierte en un **PrintWriter** para hacer uso de las opciones de dar formato. El archivo de datos que se cree así es legible como un archivo de texto normal y corriente.

A medida que se escriban líneas al archivo, se añaden los números de línea. Nótese que no se usa **LineNumberInputStream,** porque es una clase estúpida e innecesaria. Como se muestra en este caso, es fundamental llevar a cabo un seguimiento de los números de página.

Cuando se agota el flujo de entrada, **readLine**( ) devuelve **null.** Se verá una llamada **close**( ) explícita para **salidal,** porque si no se invoca a **close**( ) para todos los archivos de salida, los espacios de almacenamiento intermedio no se vaciarán, de forma que las operaciones pueden quedar inacabadas.

Flujos de salida

Los dos tipos primarios de flujos de salida se diferencian en la forma de escribir los datos: uno lo hace de forma comprensible para el ser humano, y el otro lo hace para pasárselos a **DataInputStream.** El **RandomAccessFile** se mantiene independiente, aunque su formato de datos es compatible con **DataInputStream** y **DataOutputStream.**

5. Almacenar y recuperar datos

Un **PrintWriter** da formato a los datos de forma que sean legibles por el hombre. Sin embargo, para sacar datos de manera que puedan ser recuperados por otro flujo, se usa un **DataOutputStream** para escribir los datos y un **DataInputStream** para la recuperación. Por supuesto, estos flujos podrían ser cualquier cosa, pero aquí se usa un archivo con espacios de almacenamiento intermedio tanto para la lectura como para la escritura. **DataOutputStream** y **DataInputStream** están orientados a **byte,** por lo que requieren de **InputStreams** y **OutputStreams.**

Si se usa un **DataOutputStream** para escribir los datos, Java garantiza que se pueda recuperar el dato utilizando eficientemente un **DataInputStream** - independientemente de las plataformas sobre las que se lleven a cabo las operaciones de lectura y escritura. Esto tiene un valor increíble, pues nadie sabe quién ha invertido su tiempo preocupándose por aspectos de datos específicos de cada plataforma. El problema se desvanece simplemente teniendo Java en ambas plataformas **[2]**.

[2] XML es otra solución al mismo problema: mover datos entre plataformas de computación diferentes, que en este caso no depende de que haya Java en ambas plataformas. Además, existen herramientas Java para dar soporte a XML.

Nótese que se escribe la cadena de caracteres haciendo uso tanto de **writeChars**( ) como de **writeBytes**( ). Cuando se ejecute el programa, se observará que **writeChars**( ) saca caracteres
Unicode de 16 bits. Cuando se lee la línea haciendo uso de **readline**( ) se verá que hay un espacio entre cada carácter, que es debido al byte extra introducido por Unicode. Puesto que no hay ningún método "**readChars**" complementario en **DataInputStream,** no queda más remedio que sacar esos caracteres de uno en uno con **readChar**( ). Por tanto, en el caso de ASCII, es más sencillo escribir los caracteres como bytes seguidos de un salto de línea; posteriormente se usa **readLine**( ) para leer de nuevo esos bytes en una línea ASCII tradicional.

El **writeDouble**( ) almacena el número **double** en el flujo y el **readDouble**( ) complementario lo recupera (hay métodos semejantes para hacer lo mismo en la escritura y lectura de otros tipos). Pero para que cualquiera de estos métodos de lectura funcione correctamente es necesario conocer la ubicación exacta del elemento de datos dentro del flujo, puesto que sería igualmente posible leer el **double** almacenado como una simple secuencia de bytes, o como un **char,** etc. Por tanto, o bien hay que establecer un formato fijo para los datos dentro del archivo, o hay que almacenar en el propio archivo información extra que será necesario analizar para determinar dónde se encuentra ubicado el dato.

6. Leer y escribir archivos de acceso aleatorio

Como se vio anteriormente, el **RandomAccessFile** se encuentra casi totalmente aislado del resto de la jerarquía de E/S, excepto por el hecho de que implementa las interfaces **DataInput** y **DataOutput.** Por tanto, no se puede combinar con ninguno de los aspectos de las subclases **InputStream** y **OutputStream.** Incluso aunque podría tener sentido tratar un **ByteArrayInputStream** como un elemento de acceso aleatorio, se puede usar **RandomAccessFile** simplemente para abrir un archivo. Hay que asumir que un **RandomAccessFile** tiene sus espacios de almacenamiento intermedio, así que no hay que añadírselos.

La opción que queda e s el segundo parámetro del constructor: se puede abrir un **RandomAccessFile** para leer (**"r"**), o para leer y escribir (**"rw"**).

La utilización de un **RandomAccessFile** es como usar un **DataInputStream** y un **DataOutput**
Stream combinados (puesto que implementa las interfaces equivalentes). Además, se puede ver que se usa **seek**( ) para moverse por el archivo y cambiar algunos de sus valores.

¿Un error?

Si se echa un vistazo a la Sección 6, se verá que el dato se escribe ***antes*** que el texto. Esto se debe a un problema que se introdujo con Java 1.1 y que persiste en Java 2) que parece un error, pero informamos de él y la gente de JavaSoft que trabaja en errores nos informó de que funciona exactamente como se desea que funcione (sin embargo, el problema no ocurría en Java 1.0, y eso nos hace sospechar). El problema se muestra en el ejemplo siguiente:

```
//: c11:IOProblem.java
// Java 1.1 and higher I/O Problem.
import java.io.*;

public class IOProblem {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out =
      new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeBytes("That was the value of pi\n");
    out.writeBytes("This is pi/2:\n");
    out.writeDouble(3.14159/2);
    out.close();

    DataInputStream in =
      new DataInputStream(
```

```
          new BufferedInputStream(
            new FileInputStream("Data.txt")));
      BufferedReader inbr =
        new BufferedReader(
          new InputStreamReader(in));
      // The doubles written BEFORE the line of text
      // read back correctly:
      System.out.println(in.readDouble());
      // Read the lines of text:
      System.out.println(inbr.readLine());
      System.out.println(inbr.readLine());
      // Trying to read the doubles after the line
      // produces an end-of-file exception:
      System.out.println(in.readDouble());
    }
} ///:~
```

Parece que todo lo que se escribe tras una llamada a **writeBytes**( ) no es recuperable. La respuesta es aparentemente la misma que en el viejo chiste: "¡Doctor, cuando hago esto, me duele!" "¡Pues no lo haga!"

Flujos entubados

PipedInputStream, **PipedOutputStream**, **PipedReader** y **PipedWriter** ya se han mencionado anteriormente en este capítulo, aunque sea brevemente. No se pretende, no obstante, sugerir que no sean útiles, pero es difícil descubrir su verdadero valor hasta entender el multihilo, puesto que este tipo de flujos se usa para la comunicación entre hilos. Su uso se verá en un ejemplo del Capítulo 14.

E/S estándar

El término E/S *estándar* proviene de Unix (si bien se ha reproducido tanto en Windows como en otros muchos sistemas operativos). Hace referencia al flujo de información que utiliza todo programa. Así, toda la entrada a un programa proviene de la *entrada estándar,* y su salida "fluye" a través de la *salida estándar,* mientras que todos sus mensajes de error se envían a la *salida de error estándar.* El valor de la E/S estándar radica en la posibilidad de encadenar estos programas de forma sencilla de manera que la salida estándar de uno se convierta en la entrada estándar del siguiente. Esta herramienta resulta extremadamente poderosa.

Leer de la entrada estándar

Siguiendo el modelo de E/S estándar, Java tiene **Sytem.in, System.out** y **System.err. A** lo largo de todo este libro, se ha visto cómo escribir en la salida

estándar haciendo uso de **System,out,** que ya viene envuelto como un objeto **PrintStream. System.err** es igual que un **PrintStream,** pero **System.in** es como un **InputStream** puro, sin envoltorios. Esto significa que, si bien se pueden utilizar **System.out** y **System.err** directamente, es necesario envolver de alguna forma **Systemh** antes de poder leer de él.

Generalmente se desea leer una entrada línea a línea haciendo uso de **readLine**( ), por lo que se deseará envolver **System.in** en un **BufferedReader.** Para ello hay que convertir **System.in** en un **Reader** haciendo uso de **InputStueamReader.** He aquí un ejemplo que simplemente visualiza toda línea que se teclee:

```
//: c11:Echo.java
// How to read from standard input.
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader in =
        new BufferedReader(
          new InputStreamReader(System.in));
    String s;
    while((s = in.readLine()).length() != 0)
      System.out.println(s);
    // An empty line terminates the program
  }
} ///:~
```

La razón que justifica la especificación de la excepción es que **readLine**( ) puede lanzar una **IOException.** Nótese que **System.in** debería utilizar un espacio de almacenamiento intermedio, al igual que la mayoría de flujos.

Convirtiendo System.out en un PrintWriter

System.out es un **PrintStream,** que es, a su vez, un **OutputStream. PrintWriter** tiene un constructor que toma un **OutputStream** como parámetro. Por ello, si se desea es posible convertir **System.out** en un **PrintWriter** haciendo uso de ese constructor:

```
//: c11:ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out =
```

```
      new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
} ///:~
```

Es importante usar la versión de dos parámetros del constructor **PrintWriter** y poner el segundo parámetro a **true** para habilitar el vaciado automático, pues, si no, puede que no se vea la salida.

Redirigiendo la E/S estándar

La clase **System** de Java permite redirigir los flujos de entrada, salida y salida de error estándares simplemente haciendo uso de las llamadas a métodos estáticos:

setIn(**InputStream)**
setOut(**PrintStream)**
setErr(**PrintStream)**

Redirigir la salida es especialmente útil si, de repente, se desea comenzar la creación de mucha información de salida a pantalla, y el desplazamiento de la misma es demasiado rápido como para leer **[3]**. El redireccionamiento de la entrada es útil en programas de línea de comandos en los que se desee probar repetidamente una secuencia de entrada de usuario en particular. He aquí un ejemplo simple que muestra cómo usar estos métodos:

[3] El Capitulo 13 muestra una solución aún más adecuada para esto: un programa de IGU con un área de desplazamiento de texto.

```
//: c11:Redirecting.java
// Demonstrates standard I/O redirection.
import java.io.*;

class Redirecting {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    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!
  }
} ///:~
```

Este programa adjunta la entrada estándar a un archivo y redirecciona la salida estándar y la de error a otro archivo.

El redireccionamiento de la E/S manipula flujos de bytes, en vez de flujos de caracteres, por lo que se usan **InputStreams** y **OutputStreams** en vez de **Readers** y **Writers.**

Nueva E/S

La *nueva* librería de E/S de Java, introducida en JDK 1.4 en los paquetes del **java.nio**, tiene una meta: *velocidad*. De hecho, la paquetes viejos de E/S han sido reimplementados usando **nio** para sacar provecho de este incremento de velocidad, así es que te beneficiarás aun si explícitamente no escribes código con **nio**. El incremento de velocidad ocurre en ambos, E/S de archivos, lo cual es explorado aquí, y E/S en la red, lo cual es cubierto en el *Capítulo 16* y en el libro *Piensa en Java Empresarial*.

La velocidad proviene de usar estructuras que están más cercas a la forma del sistema operativo de funcionar E/S: Canales y búferes. Podrías pensar acerca de eso como una mina de carbón; El canal es la mina conteniendo la costura de carbón (los datos), y el búfer es la carreta que tú envías en la mina. La carreta regresa llena de carbón, y tú obtiene el carbón de la carreta. Es decir, no le interactúas directamente el canal; Le interactúas el búfer y envías el búfer en el canal. El canal o atrae datos del búfer, o coloca datos en el búfer.

La única clase de búfer que se comunica directamente con un canal es un **ByteBuffer** – es decir, un búfer que mantiene **bytes** crudos. Si tú miras la documentación JDK para **java.nio.ByteBuffer**, verás que es medianamente básica: Creas uno diciéndole cuánto almacenamiento ubicar, y hay una selección de métodos para colocar y recibir datos, en ya sea forma de **byte** crudo o como los tipos de datos primitivos. Pero no hay forma para colocar u obtener un objeto, o hasta un **String**. Eso es medianamente de bajo nivel, precisamente porque éste hace uno más eficiente haciendo mapas con la mayoría de sistemas operativos.

Tres de las clases en la vieja E/S han sido modificadas a fin de que produzcan un **FileChannel**: **FileInputStream**, **FileOutputStream**, y, para la lectura y escritura, **RandomAccessFile**. Note que estos son los flujos de manipulación de byte, de acuerdo con la naturaleza de bajo nivel de **nio**. Las clases **Reader** y **Writer** de modo de carácter no producen canales, pero las clases **java.nio.channels.Channels** tienen métodos de utilidad para producir a **Readers** y **Writers** desde canales.

Aquí hay un ejemplo simple que ejercita los tres tipos de flujos para producir canales que son escribibles, leíble / escribible, y leíble:

```
//: c12:GetChannel.java
// Getting channels from streams
// {Clean: data.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class GetChannel {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    // Write a file:
    FileChannel fc =
      new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text ".getBytes()));
    fc.close();
    // Add to the end of the file:
    fc =
      new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Move to the end
    fc.write(ByteBuffer.wrap("Some more".getBytes()));
    fc.close();
    // Read the file:
    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());
  }
} ///:~
```

Para cualquiera de las clases del flujo mostrado aquí, **getChannel**() producirá un **FileChannel**. Un canal es medianamente básico: Le puedes dar a un **ByteBuffer** para leer o escribir, y puedes cerrar regiones del archivo para el acceso exclusivo (éste será descrito más tarde).

Una forma para poner **bytes** en un **ByteBuffer** debe rellenarlos en directamente usar uno de los métodos **put**, para poner uno o más **bytes**, o valores de tipos

primitivos. Sin embargo, como visto aquí, también puedes envolver un arreglo existente de **byte** en un **ByteBuffer** usando el método **wrap**(). Cuando haces esto, el arreglo subyacente no es copiado, pero en lugar de eso es utilizado como el almacenamiento para el **ByteBuffer** generado. Decimos que el **ByteBuffer** tiene respaldo por el arreglo.

El archivo data.txt es reabierto usando a un **RandomAccessFile**. Note que puedes cambiar de un lado para otro al **FileChannel** en el archivo; Aquí, es movido al final así que escritos adicionales serán anexado.

Para el acceso de sólo lectura, explícitamente debes ubicar a un **ByteBuffer** usando el método estático **allocate**(). La meta de **nio** es mover rápidamente cantidades grandes de datos, así es que el tamaño del **ByteBuffer** debería ser significativo – de hecho, el 1K usado aquí es probablemente mucho más pequeño que normalmente querrías usar (tendrás que experimentar con tu aplicación en funciones para encontrar el mejor tamaño).

Cabe también ir por aun más velocidad usando **allocateDirect**() en lugar de **allocate**() para producir un búfer directo que puede tener un acoplador aun más alto con el sistema operativo. Sin embargo, los costos operativos en tal asignación son mayores, y la implementación real se diferencia de un sistema operativo a otro, así otra vez, debes experimentar con tu aplicación en funciones para descubrir si búferes directos adquirirán cualquier ventaja en la velocidad.

Una vez que llamas a **read**() para decirle al **FileChannel** almacenar **bytes** en el **ByteBuffer**, debes llamar a **flip**() en el búfer para decirle que estás en condición de extraer sus **bytes** (sí, esto parece un poco crudo, pero recuerda que es de muy bajo nivel y se hace para máxima velocidad). Y si usáramos el búfer para más operaciones **read**(), también tendríamos que llamar a **clear**() para prepararlo en cada **read**(). Puedes ver esto en un simple programa de copiado de archivo:

```
//: c12:ChannelCopy.java
// Copying a file using channels and buffers
// {Args: ChannelCopy.java test.txt}
// {Clean: test.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

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(); // Prepare for writing
      out.write(buffer);
      buffer.clear();  // Prepare for reading
    }
  }
} ///:~
```

Puedes ver que un **FileChannel** está abierto para leer, y uno para escribir. Un **ByteBuffer** es ubicado, y cuando **FileChannel.read**( ) retorna **-1** (un remanente, sin duda, de Unix y C), quiere decir que has alcanzado al final de la entrada. Después de cada **read**(), que mete datos en el búfer, **flip**() prepara el búfer a fin de que su información puede ser extraída por el **write**(). Después del **write**(), la información todavía está en el búfer, y **clear**() reanuda todos los punteros internos a fin de que está listo para aceptar datos durante otro **read**().

El programa precedente no es la forma ideal para manejar esta clase de operación, no obstante. Métodos especiales **transferTo**() y **transferFrom**() te permiten conectar un canal directamente al otro:

```
//: c12:TransferTo.java
// Using transferTo() between channels
// {Args: TransferTo.java TransferTo.txt}
// {Clean: TransferTo.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

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);
    // Or:
    // out.transferFrom(in, 0, in.size());
  }
} ///:~
```

No harás esta cosa muy a menudo, pero es bueno estar al tanto.

Convirtiendo datos

Si ves hacia atrás en **GetChannel.java**, notarás que, imprimir la información en el archivo, estamos jalando los datos fuera de un **byte** a la vez y emitiendo cada **byte** a un **char**. Éste parece un poco primitivo – si consideras la clase **java.nio.CharBuffer**, verás que tiene un método **toString**() que dice: "Devuelve a un **string** conteniendo los caracteres en este búfer." Desde que un **ByteBuffer** puede ser visto como un **CharBuffer** con el método **asCharBuffer**(), ¿Por qué no usas esto? Como puedes ver desde la primera línea en la declaración **expect**() abajo, esto no funciona:

```
//: c12:BufferToText.java
// Converting text to and from ByteBuffers
// {Clean: data2.txt}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import com.bruceeckel.simpletest.*;

public class BufferToText {
  private static Test monitor = new Test();
  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();
    // Doesn't work:
    System.out.println(buff.asCharBuffer());
    // Decode using this system's default Charset:
    buff.rewind();
    String encoding = System.getProperty("file.encoding");
    System.out.println("Decoded using " + encoding + ": "
      + Charset.forName(encoding).decode(buff));
    // Or, we could encode with something that will print:
    fc = new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap(
      "Some text".getBytes("UTF-16BE")));
    fc.close();
    // Now try reading again:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
```

```
    buff.flip();
    System.out.println(buff.asCharBuffer());
    // Use a CharBuffer to write through:
    fc = new FileOutputStream("data2.txt").getChannel();
    buff = ByteBuffer.allocate(24); // More than needed
    buff.asCharBuffer().put("Some text");
    fc.write(buff);
    fc.close();
    // Read and display:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
    buff.flip();
    System.out.println(buff.asCharBuffer());
    monitor.expect(new String[] {
      "????",
      "%% Decoded using [A-Za-z0-9_\\-]+: Some text",
      "Some text",
      "Some text\0\0\0"
    });
  }
} ///:~
```

El búfer contiene **bytes** simples, y para convertir estos en caracteres que nosotros tampoco los debemos codificar como los pongamos (a fin de que sean significativos cuando salen) o los desciframos como salen del búfer. Esto puede estar consumado usando la clase **java.nio.charset.Charset**, lo cual provee herramientas para codificar en muchos tipos diferentes de conjuntos de caracteres:

```
//: c12:AvailableCharSets.java
// Displays Charsets and aliases
import java.nio.charset.*;
import java.util.*;
import com.bruceeckel.simpletest.*;

public class AvailableCharSets {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    Map charSets = Charset.availableCharsets();
    Iterator it = charSets.keySet().iterator();
    while(it.hasNext()) {
      String csName = (String)it.next();
      System.out.print(csName);
      Iterator aliases = ((Charset)charSets.get(csName))
        .aliases().iterator();
      if(aliases.hasNext())
        System.out.print(": ");
```

```
    while(aliases.hasNext()) {
      System.out.print(aliases.next());
      if(aliases.hasNext())
        System.out.print(", ");
    }
    System.out.println();
  }
  monitor.expect(new String[] {
    "Big5: csBig5",
    "Big5-HKSCS: big5-hkscs, Big5_HKSCS, big5hkscs",
    "EUC-CN",
    "EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, " +
    "eucjp, Extended_UNIX_Code_Packed_Format_for" +
    "_Japanese, x-euc-jp, euc_jp",
    "euc-jp-linux: euc_jp_linux",
    "EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, " +
    "ksc5601-1987, euc_kr, ks_c_5601-1987, " +
    "euckr, csEUCKR",
    "EUC-TW: cns11643, euc_tw, euctw",
    "GB18030: gb18030-2000",
    "GBK: GBK",
    "ISCII91: iscii, ST_SEV_358-88, iso-ir-153, " +
    "csISO153GOST1976874",
    "ISO-2022-CN-CNS: ISO2022CN_CNS",
    "ISO-2022-CN-GB: ISO2022CN_GB",
    "ISO-2022-KR: ISO2022KR, csISO2022KR",
    "ISO-8859-1: iso-ir-100, 8859_1, ISO_8859-1, " +
    "ISO8859_1, 819, csISOLatin1, IBM-819, " +
    "ISO_8859-1:1987, latin1, cp819, ISO8859-1, " +
    "IBM819, ISO_8859_1, l1",
    "ISO-8859-13",
    "ISO-8859-15: 8859_15, csISOlatin9, IBM923, cp923," +
    " 923, L9, IBM-923, ISO8859-15, LATIN9, " +
    "ISO_8859-15, LATIN0, csISOlatin0, " +
    "ISO8859_15_FDIS, ISO-8859-15",
    "ISO-8859-2", "ISO-8859-3", "ISO-8859-4",
    "ISO-8859-5", "ISO-8859-6", "ISO-8859-7",
    "ISO-8859-8", "ISO-8859-9",
    "JIS0201: X0201, JIS_X0201, csHalfWidthKatakana",
    "JIS0208: JIS_C6626-1983, csISO87JISX0208, x0208, " +
    "JIS_X0208-1983, iso-ir-87",
    "JIS0212: jis_x0212-1990, x0212, iso-ir-159, " +
    "csISO159JISC02121990",
    "Johab: ms1361, ksc5601_1992, ksc5601-1992",
    "KOI8-R",
    "Shift_JIS: shift-jis, x-sjis, ms_kanji, " +
    "shift_jis, csShiftJIS, sjis, pck",
    "TIS-620",
    "US-ASCII: IBM367, ISO646-US, ANSI_X3.4-1986, " +
    "cp367, ASCII, iso_646.irv:1983, 646, us, iso-ir-6,"+
```

```
      " csASCII, ANSI_X3.4-1968, ISO_646.irv:1991",
      "UTF-16: UTF_16",
      "UTF-16BE: X-UTF-16BE, UTF_16BE, ISO-10646-UCS-2",
      "UTF-16LE: UTF_16LE, X-UTF-16LE",
      "UTF-8: UTF8", "windows-1250", "windows-1251",
      "windows-1252: cp1252",
      "windows-1253", "windows-1254", "windows-1255",
      "windows-1256", "windows-1257", "windows-1258",
      "windows-936: ms936, ms_936",
      "windows-949: ms_949, ms949", "windows-950: ms950",
    });
  }
} ///:~
```

Entonces, regresando a **BufferToText.java**, si **rewind**() el búfer (para retroceder al principio de los datos) y luego usa el conjunto de caracteres predeterminado de esa plataforma para **decode**() los datos, el **CharBuffer** resultante imprimirá a la consola nada más. Para descubrir el conjunto de caracteres predeterminado, usa a **System.getProperty**("**file.encoding**"), lo cual produce al **string** que nombra el conjunto de caracteres. Pasándole esto a **Charset.forName**() produce el objeto **Charset** que puede usarse para decodificar al **string**.

Otra alternativa está **encode**() usando un conjunto de caracteres que dará como resultado algo imprimible cuando el archivo le sea leído, como verás en la tercera parte de **BufferToText.java**. Aquí, UTF-16BE se usa para escribir el texto en el archivo, y cuando es leído, todo lo que tienes que hacer es convertírselo a un **CharBuffer**, y produce el texto esperado.

Finalmente, para que veas lo que ocurre si le escribes al **ByteBuffer** a través de un **CharBuffer** (aprenderás más acerca de esto más adelante). Note que 24 **bytes** son ubicados para el **ByteBuffer**. Desde cada **char** requiere dos **bytes**, esto es suficiente para 12 **chars**, pero Algún texto sólo tiene 9. Los **bytes** restantes de cero todavía aparecen en la representación del **CharBuffer** producido por su **toString**(), como puedes ver en la salida.

Mandando a llamar primitivas

Aunque un **ByteBuffer** sólo mantiene **bytes**, contiene métodos para producir cada uno de los tipos diferentes de valores primitivos de los **bytes** que contiene. Este ejemplo muestra la inserción y extracción de varios valores usando estos métodos:

```
//: c12:GetData.java
// Getting different representations from a ByteBuffer
```

```
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class GetData {
  private static Test monitor = new Test();
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // Allocation automatically zeroes the ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        System.out.println("nonzero");
    System.out.println("i = " + i);
    bb.rewind();
    // Store and read a char array:
    bb.asCharBuffer().put("Howdy!");
    char c;
    while((c = bb.getChar()) != 0)
      System.out.print(c + " ");
    System.out.println();
    bb.rewind();
    // Store and read a short:
    bb.asShortBuffer().put((short)471142);
    System.out.println(bb.getShort());
    bb.rewind();
    // Store and read an int:
    bb.asIntBuffer().put(99471142);
    System.out.println(bb.getInt());
    bb.rewind();
    // Store and read a long:
    bb.asLongBuffer().put(99471142);
    System.out.println(bb.getLong());
    bb.rewind();
    // Store and read a float:
    bb.asFloatBuffer().put(99471142);
    System.out.println(bb.getFloat());
    bb.rewind();
    // Store and read a double:
    bb.asDoubleBuffer().put(99471142);
    System.out.println(bb.getDouble());
    bb.rewind();
    monitor.expect(new String[] {
      "i = 1025",
      "H o w d y ! ",
      "12390", // Truncation changes the value
      "99471142",
      "99471142",
      "9.9471144E7",
      "9.9471142E7"
```

```
    });
  }
} ///:~
```

Después de que un **ByteBuffer** es ubicado, sus valores son comprobados para ver si la asignación del búfer automáticamente pone en cero el contenido – y lo hace. Todos los 1,024 valores son comprobados (hasta el **limit**() del búfer), y todo son cero.

La forma más fácil para intercalar valores primitivos en un **ByteBuffer** debe colocar la vista apropiada en ese búfer usando **asCharBuffer**(), **asShortBuffer**(), etc., y luego usar el método **put**() de esa vista. Puedes ver que este es el proceso destinado para cada uno de los tipos primitivos de datos. El único de éstos que es un obstáculo pequeño es el **put**() para el **ShortBuffer**, lo cual requiere una señal (nota que la señal trunca y cambia el valor resultante). Todos los demás búferes de vista no requieren señales en sus métodos **put**().

Búferes de Vista

Un búfer de vista te permita mirar a un **ByteBuffer** subyacente a través de la ventana de un tipo primitivo particular. El **ByteBuffer** es todavía el almacenamiento real que respalda la vista, así cualesquier cambios que tú haces a la vista están reflejados en modificaciones para los datos en el **ByteBuffer**. Como visto en el ejemplo previo, esto te permite convenientemente intercalar tipos primitivos en un **ByteBuffer**. Una vista también te permite leer los valores primitivos de un **ByteBuffer**, ya sea uno a la vez (como **ByteBuffer** permite) o en cantidades de cosas (en los arreglos). Aquí hay un ejemplo que manipula **ints** en un **ByteBuffer** por un **IntBuffer**:

```
//: c12:IntBufferDemo.java
// Manipulating ints in a ByteBuffer with an IntBuffer
import java.nio.*;
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;

public class IntBufferDemo {
  private static Test monitor = new Test();
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // Store an array of int:
    ib.put(new int[] { 11, 42, 47, 99, 143, 811, 1016 });
    // Absolute location read and write:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
```

```
    ib.rewind();
    while(ib.hasRemaining()) {
      int i = ib.get();
      if(i == 0) break; // Else we'll get the entire buffer
      System.out.println(i);
    }
    monitor.expect(new String[] {
      "99",
      "11",
      "42",
      "47",
      "1811",
      "143",
      "811",
      "1016"
    });
  }
} ///:~
```

El método sobrecargado **put**() es primero usado para almacenar un montón de int. Las siguientes llamadas de método **get**() y **put**() acceden directamente a una posición del **int** en el **ByteBuffer** subyacente. Note que estos accesos absolutos de la posición están disponibles para tipos primitivos hablando directamente con un **ByteBuffer**, igualmente.

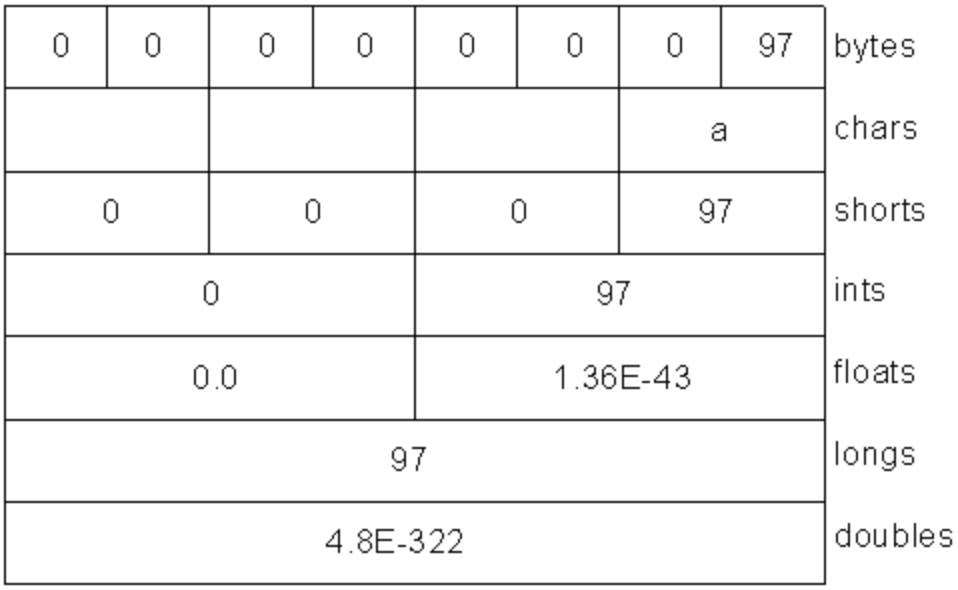
Una vez que el **ByteBuffer** subyacente se llena de **ints** o algún otro tipo primitivo por un búfer de vista, entonces ese **ByteBuffer** puede ser escrito directamente a un canal. Simplemente puedes leer de un canal fácilmente y usar un búfer de vista para convertirlo todo a un tipo particular de primitiva. Aquí está un ejemplo que interpreta la misma secuencia de **bytes** como **short**, **int**, **float**, **long**, y **double** produciendo búferes diferentes de vista en el mismo **ByteBuffer**:

```
//: c12:ViewBuffers.java
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class ViewBuffers {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    System.out.println("Byte Buffer");
    while(bb.hasRemaining())
      System.out.println(bb.position()+ " -> " + bb.get());
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    System.out.println("Char Buffer");
    while(cb.hasRemaining())
```

```
  System.out.println(cb.position()+ " -> " + cb.get());
FloatBuffer fb =
  ((ByteBuffer)bb.rewind()).asFloatBuffer();
System.out.println("Float Buffer");
while(fb.hasRemaining())
  System.out.println(fb.position()+ " -> " + fb.get());
IntBuffer ib =
  ((ByteBuffer)bb.rewind()).asIntBuffer();
System.out.println("Int Buffer");
while(ib.hasRemaining())
  System.out.println(ib.position()+ " -> " + ib.get());
LongBuffer lb =
  ((ByteBuffer)bb.rewind()).asLongBuffer();
System.out.println("Long Buffer");
while(lb.hasRemaining())
  System.out.println(lb.position()+ " -> " + lb.get());
ShortBuffer sb =
  ((ByteBuffer)bb.rewind()).asShortBuffer();
System.out.println("Short Buffer");
while(sb.hasRemaining())
  System.out.println(sb.position()+ " -> " + sb.get());
DoubleBuffer db =
  ((ByteBuffer)bb.rewind()).asDoubleBuffer();
System.out.println("Double Buffer");
while(db.hasRemaining())
  System.out.println(db.position()+ " -> " + db.get());
monitor.expect(new String[] {
  "Byte Buffer",
  "0 -> 0",
  "1 -> 0",
  "2 -> 0",
  "3 -> 0",
  "4 -> 0",
  "5 -> 0",
  "6 -> 0",
  "7 -> 97",
  "Char Buffer",
  "0 -> \0",
  "1 -> \0",
  "2 -> \0",
  "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 **ByteBuffer** se produce envolviendo un arreglo de ocho **bytes**, lo cual es entonces exhibido por búferes de vista de todos los tipos diferentes de primitivas. Puedes ver en el siguiente diagrama la manera que los datos aparecen diferentemente cuando es leída de los tipos diferentes de búferes:

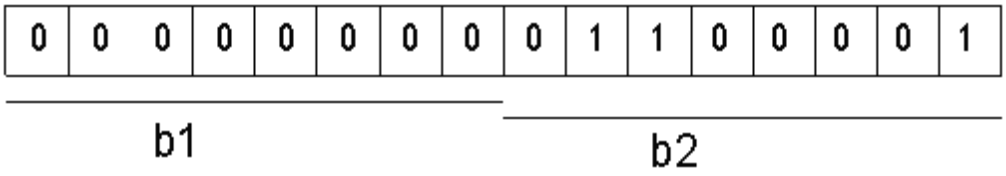


Esto corresponde a la salida del programa.

Endians

Diferentes máquinas pueden usar acercamientos que ordenan **bytes** diferente para almacenar datos. Endian grande coloca el **byte** más significativo en la dirección mínima de memoria, y endian pequeño coloca el **byte** más significativo en la dirección más alta de memoria. Al almacenar una cantidad que es mayor que un **byte**, como **int**, **float**, etc., Puedes necesitar considerar el **byte** haciendo el pedido. Un **ByteBuffer** almacena datos en forma de endian grande, y los datos enviados sobre una red siempre usan una orden de endian grande. Puedes cambiar el endianismo de un **ByteBuffer** usando **order**() con un argumento de **ByteOrder.BIG_ENDIAN** o **ByteOrder.LITTLE_ENDIAN**.

Considera a un **ByteBuffer** conteniendo los siguientes dos **bytes**:



Si lees los datos como un short (**ByteBuffer.asShortBuffer**()), obtendrás el número 97 (00000000 01100001), pero si cambias a endian pequeño, obtendrás el número 24832 (01100001 00000000).

Aquí está un ejemplo que muestra cómo el ordenar **bytes** se varía en caracteres a merced de la configuración del endian:

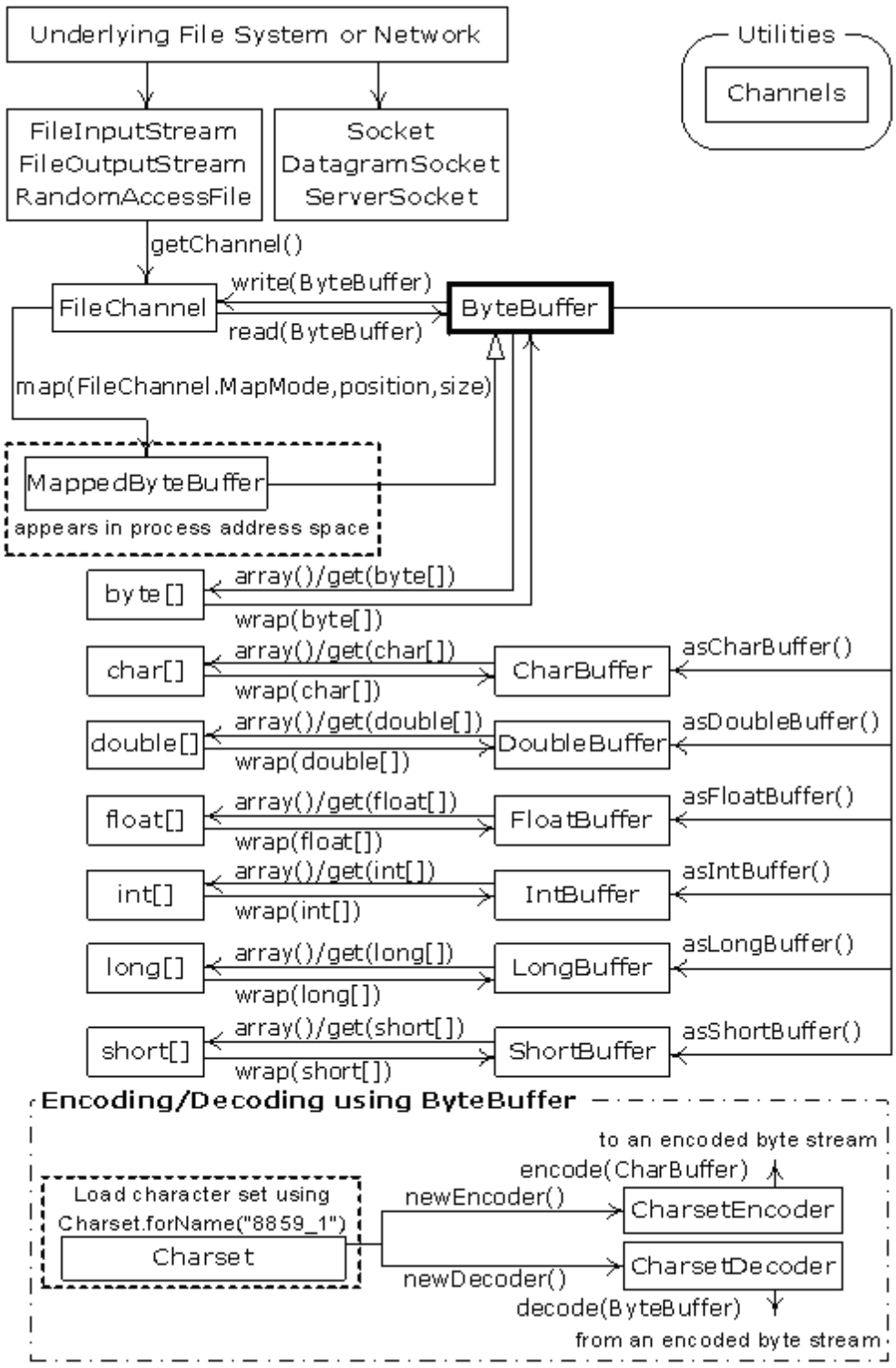
```
//: c12:Endians.java
// Endian differences and data storage.
import java.nio.*;
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;

public class Endians {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays2.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays2.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays2.toString(bb.array()));
    monitor.expect(new String[]{
      "[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]"
    });
  }
} ///:~
```

El **ByteBuffer** recibe espacio suficiente para considerar todos los **bytes** en **charArray** como un búfer externo a fin de que ese método **array**() puede ser llamado para desplegar los **bytes** subyacentes. El método **array**() es opcional, y sólo lo puedes llamar en un búfer que tiene respaldo por un arreglo; De otra manera, tú traerás a un **UnsupportedOperationException**.

charArray es introducido en el **ByteBuffer** por una vista **CharBuffer**. Cuando los **bytes** subyacentes son desplegados, puedes ver que el ordenamiento predeterminado es lo mismo como la subsiguiente orden grande del endian, mientras que la orden pequeña del endian intercambie los **bytes**.

Manipulación de datos con búferes

El diagrama aquí ilustra las relaciones entre las clases del **nio**, a fin de que puedas ver cómo mover y convertir datos. Por ejemplo, si tienes el deseo de escribir un arreglo de byte para un archivo, luego envuelves el arreglo de **byte** usando el método **ByteBuffer.wrap**(), abres un canal en el **FileOutputStream** usando el método **getChannel**(), y luego escribes datos dentro de **FileChannel** de este **ByteBuffer**.



Note que **ByteBuffer** es la única forma para mover información adentro y afuera de canales, y que sólo puedes crear un búfer de tipo primitivo autónomo, o puedes obtener uno de un **ByteBuffer** usando uno como el método. Es decir, no puedes convertir un búfer de tipo primitivo a un **ByteBuffer**. Sin embargo, desde que puedes mover datos primitivos dentro y fuera de un **ByteBuffer** por un búfer de vista, éste no es realmente una restricción.

Detalles del búfer

Un **Buffer** consta de datos y cuatro índices para el acceso y manipula esta información eficazmente: Marca, posición, límite y capacidad. Hay métodos para colocar y reanudar estos índices e consultar su valor.

capacity( )	Retorna la capacidad del búfer
clear( )	Limpia el búfer, pone a cero la *posición*, y limita a la *capacidad*. Llamas este método para sobrescribir un búfer existente.
flip( )	Coloca límite a posición y posición a cero. Este método se usa para preparar el búfer para una lectura después de que los datos han sido escritos en él.
limit( )	Devuelve el valor de límite.
limit(int lim)	Establece el valor de límite.
mark( )	Coloca marca en posición.
position( )	Devuelve el valor de posición.
position(int pos)	Establece el valor de posición.
remaining( )	Retorna (límite - posición).
hasRemaining( )	Retorna verdadero si hay cualquier elemento entre posición y límite.

Los métodos que insertan y extraen datos del búfer actualizan estos índices para reflejar los cambios.

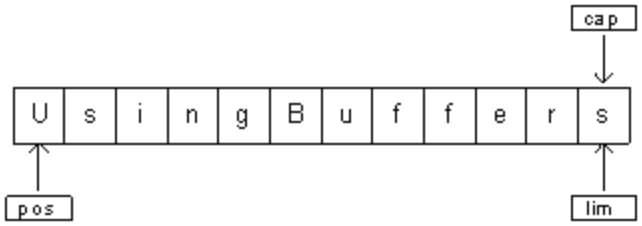
Este ejemplo usa un algoritmo (intercambiando caracteres adyacentes) muy simple para mezclar y descifrar caracteres en un **CharBuffer**:

```
//: c12:UsingBuffers.java
import java.nio.*;
import com.bruceeckel.simpletest.*;

public class UsingBuffers {
  private static Test monitor = new Test();
  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);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
    monitor.expect(new String[] {
      "UsingBuffers",
      "sUniBgfuefsr",
      "UsingBuffers"
    });
  }
} ///:~
```

Aunque podrías producir a un **CharBuffer** directamente llamando a **wrap**( ) con un arreglo **char**, un **ByteBuffer** subyacente es ubicado en su lugar, y un **CharBuffer** es producido como una vista en el **ByteBuffer**. Esto enfatiza ese hecho que la meta está siempre para manipular a un **ByteBuffer**, desde que eso sea lo que le interactúa un canal.

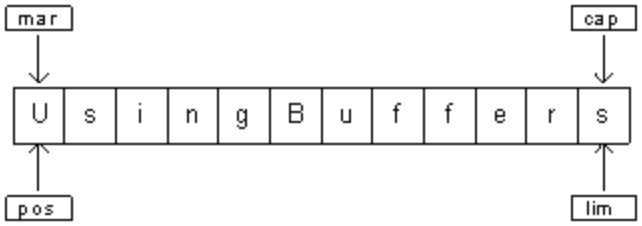
Aquí está lo que el búfer ve después del **put**():



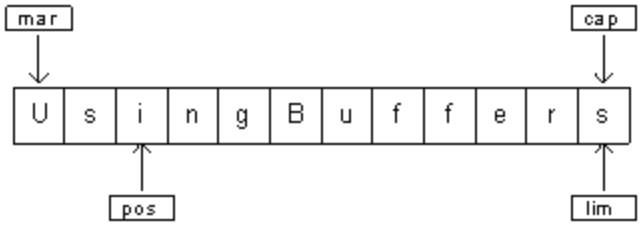
La posición señala el primer elemento en el búfer, y la capacidad y punto del límite para el último elemento.

En **symmetricScramble**(), el bucle **while** itera hasta posición equivalente a límite. La posición del búfer cambia cuando un **get**() relativo o la función **put**() es llamado en ella. También puedes llamar en absoluto a los métodos **get**() y **put**() que incluyen un argumento **index**, lo cual es la posición donde el **get**() o **put**() toma lugar. Estos métodos no modifican el valor de la posición del búfer.

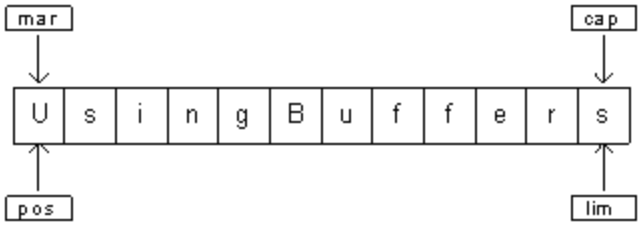
Cuando el control introduce el bucle **while**, el valor de marca es determinado usando llamada **mark**(). El estado del búfer entonces:



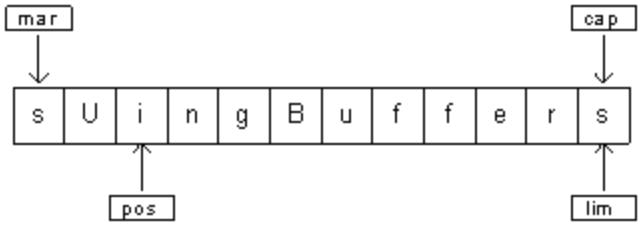
Las dos llamadas relativas **get**() salvan el valor de los primeros dos caracteres en variables **c1** y **c2**. Después de estas dos llamadas, el aspecto general del búfer se ve como este:



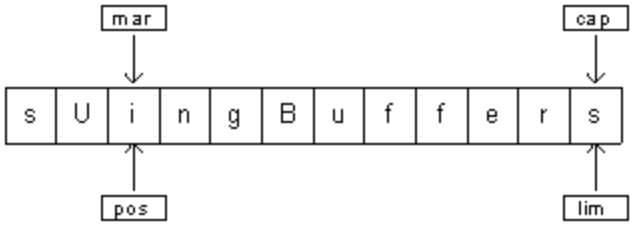
Para realizar el cambio, necesitamos escribir **c2** en posición = 0 y **c1** en posición = 1. Nosotros o podemos usar el método absoluto **put** para lograr esto, o establecer el valor de posición a marcar, lo cual es lo que hace **reset**():



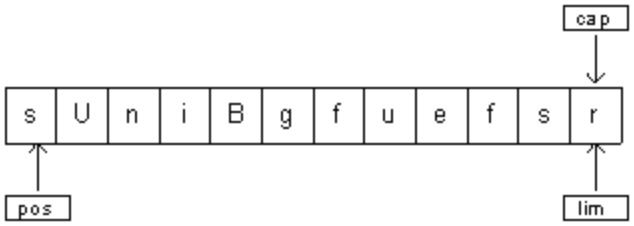
Lo dos métodos **put**() escriben **c2** y luego **c1**:



Durante la siguiente iteración del bucle, marca está lista para el valor actual de posición:



El proceso continúa hasta que el búfer entero sea atravesado. Al final del bucle **while**, la posición está al final del búfer. Si imprimes el búfer, sólo los caracteres entre la posición y el límite son impresos. Así, si quieres mostrar el contenido entero del búfer debes colocar posición al principio del búfer usando **rewind**(). Aquí está el estado de búfer después de la llamada a **rewind**() (el valor de marca se vuelve indefinido):



Cuando la función **symmetricScramble**() es llamado otra vez, el **CharBuffer** experimenta el mismo proceso y es recuperada a su estado original.

Archivos asociados a la memoria

Los archivos asociados a la memoria te permiten crear y modificar archivos que son demasiados grandes para introducir en memoria. Con un archivo asociado a la memoria, puedes pretender que el archivo entero está en memoria y que puedes accederlo por simplemente tratándolo como un arreglo muy grande. Este acercamiento simplifica grandemente el código que escribes para modificar el archivo. Aquí está un ejemplo pequeño:

```
//: c12:LargeMappedFiles.java
// Creating a very large file using mapping.
// {RunByHand}
// {Clean: test.dat}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LargeMappedFiles {
  static int length = 0x8FFFFFF; // 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');
    System.out.println("Finished writing");
    for(int i = length/2; i < length/2 + 6; i++)
      System.out.print((char)out.get(i));
  }
} ///:~
```

Para hacer la escritura y la lectura, empezamos con un **RandomAccessFile**, obtenemos un canal para ese archivo, y luego llamamos a **map**() para producir un **MappedByteBuffer**, lo cual es una clase particular de búfer directo. Note que debes especificar el punto de partida y la longitud de la región que quieres asociar en el archivo; Esto quiere decir que tienes la opción para asociar regiones más pequeñas de un archivo grande.

MappedByteBuffer es heredado de **ByteBuffer**, así tiene todos los métodos de **ByteBuffer**. Sólo los usos muy simples de **put**() y **get**() son mostrados aquí, pero también puedes usar cosas como **asCharBuffer**(), etc.

El archivo creado con el programa precedente es 128 MB de largo, lo cual es probablemente mayor que el espacio que tu sistema operativo permitirá. El archivo parece ser accesible al mismo tiempo porque sólo las porciones de eso son traídas en la memoria, y otras partes son intercambiadas. Así un archivo muy grande (hasta 2 GB) fácilmente puede ser modificado. Noto que las facilidades de asociación de archivo del sistema operativo subyacente se usan para maximizar el desempeño.

Desempeño

Aunque el desempeño del flujo viejo de E/S ha sido mejorado implementándolo con **nio**, el acceso del archivo asociado tiende a ser dramáticamente más rápido. Este programa hace una comparación simple de desempeño:

```
//: c12:MappedIO.java
// {Clean: temp.tmp}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

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 long runTest() {
      System.out.print(name + ": ");
```

```
      try {
        long startTime = System.currentTimeMillis();
        test();
        long endTime = System.currentTimeMillis();
        return (endTime - startTime);
      } 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.put(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.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(int i = 0; i < tests.length; i++)
      System.out.println(tests[i].runTest());
  }
} ///:~
```

Como se vio en anteriores ejemplos en este libro, **runTest**() es el Método de Plantilla que provee la prueba de cuadro de trabajo para implementaciones diversas de **test**() definido en las subclases internas anónimas. Cada una de estas subclases realizan una clase de prueba, así es que los métodos **test**() también te dan un prototipo para realizar las actividades diversas de E/S.

Aunque una escritura asociada parecería usar un **FileOutputStream**, toda salida en archivo asociado debe usar un **RandomAccessFile**, lo mismo que hace la lectura/escritura en el código precedente.

Aquí está la salida de una ejecución:

```
Stream Write: 1719
Mapped Write: 359
Stream Read: 750
Mapped Read: 125
Stream Read/Write: 5188
Mapped Read/Write: 16
```

Note que los métodos **test**() incluyen el tiempo para la inicialización de los objetos diversos de E/S, así es que si bien el esquema para archivos asociados puede ser caro, la ganancia global comparada para E/S del flujo es significativa.

Bloqueo de archivos

El bloqueo de archivos, introducido en JDK 1.4, te permite sincronizar acceso para un archivo como un recurso compartido. Sin embargo, los dos hilos que contienden para el mismo archivo pueden estar en JVMs diferentes, o uno puede ser un hilo Java y el otro algún hilo nativo en el sistema operativo. Los bloqueos del archivo son visibles para otros procesos del sistema operativo porque los cierres de archivo Java asocia directamente a la facilidad de cierre del sistema operativo nativo.

Aquí hay un ejemplo simple de bloqueo de archivos.

```
//: c12:FileLocking.java
// {Clean: file.txt}
import java.io.FileOutputStream;
import java.nio.channels.*;

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");
      Thread.sleep(100);
      fl.release();
      System.out.println("Released Lock");
    }
    fos.close();
  }
} ///:~
```

Pasas un **FileLock** en el archivo entero a través del llamado ya sea **tryLock**() o **lock**() en un **FileChannel**(**SocketChannel**, **DatagramChannel**, y

ServerSocketChannel no necesitan bloquear desde que son entidades de proceso intrínsecamente simple; generalmente no compartes un conector de la red entre dos procesos.) **tryLock**() poco interrumpe. Este trata de agarrar el bloqueo, pero si no lo puede hacer (cuando algún otro proceso ya mantiene el mismo bloqueo y no es compartido), simplemente regresa de los bloques de llamada del método **lock**() hasta que el bloqueo es adquirido, o el hilo que invocó a **lock**() es interrumpido, o el canal en el cual el método **lock**() es llamado es bloqueado. Un bloqueo es soltado usando **FileLock.release**().

Cabe también bloquear una parte del archivo usando:

```
tryLock(long position, long size, boolean shared)
```

o

```
lock(long position, long size, boolean shared)
```

lo cual bloquea la región **(size - position)**. El tercer argumento especifica si este bloqueo es compartido.

Aunque los métodos de bloqueo con argumento cero se adaptan a los cambios en el tamaño de un archivo, los bloqueos con un tamaño fijo no cambian si el tamaño del archivo cambia. Si un bloqueo es adquirido para una región de **posición** hasta **posición + tamaño** y el archivo incrementa más allá de **posición + tamaño**, luego la sección más allá de **posición + tamaño** no es bloqueada. Los métodos de bloqueo con argumento cero bloquean el archivo entero, aun si crece.

El soporte para los bloqueos exclusivos o compartidos debe ser provisto por el sistema operativo subyacente. Si el sistema operativo no soporta bloqueos compartidos y se ha hecho una petición, un bloqueo exclusivo es usado en lugar de eso. El tipo de bloqueo (compartido o exclusivo) puede ser consultado usando **FileLock.isShared**().

Bloqueando porciones de un archivo asociado

Como se mencionó antes, la asociación de archivos sirve típicamente para archivos muy grandes. Una cosa que puedes necesitar para hacer con un archivo tan grande es bloquear porciones de ella a fin de que otros procesos puedan modificar partes desbloqueadas del archivo. Esto es algo que ocurre, por ejemplo, con una base de datos, a fin de que pueda estar disponible para muchos usuarios de inmediato.

Aquí está un ejemplo que tiene dos hilos, cada uno del cual bloquea una porción bien definida de un archivo:

```
//: c12:LockingMappedFiles.java
// Locking portions of a mapped file.
// {RunByHand}
// {Clean: test.dat}
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 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 {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Locked: "+ start +" to "+ end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Released: "+start+" to "+ end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
} ///:~
```

La clase hilo **LockAndModify** configura la región de búfer y crea un **slice**() para ser modificado, y un **run**(), el bloqueo es adquirido en el canal de archivo (no puedes adquirir un bloqueo en el búfer – sólo el canal). La llamada a **lock**() es muy parecida a adquirir un cierre de hilado en un objeto – ahora tienes una "sección crítica" con acceso exclusivo a esa porción del archivo.

Los bloqueos son automáticamente soltados cuando el JVM sale, o el canal en el cual fue adquirido está bloqueado, pero también explícitamente puedes llamar a **release**() en el objeto **FileLock**, como se mostró aquí.

Compresión

La biblioteca de E/S de Java contiene clases que dan soporte a la lectura y escritura de flujos en un formato comprimido. Estas clases están envueltas en torno a las clases de E/S existentes para proporcionar funcionalidad de compresión.

Estas clases no se derivan de las clases **Reader** y **Writer,** sino que son parte de las jerarquías **InputStream** y **OutputStream.** Esto se debe a que la biblioteca de compresión funciona con bytes en vez de caracteres. Sin embargo, uno podría verse forzado en ocasiones a mezclar ambos tipos de flujos. (Recuérdese que se puede usar **InputStreamReader** y **OutputStreamWriter** para proporcionar conversión sencilla entre un tipo y el otro.)

Clase de Compresión	Función
CheckedInputStream	**GetCheckSum()** produce una suma de comprobación para cualquier **InputStream** (no simplemente de descompresión).
CheckedOutputStream	**GetCheckSum()** produce una suma de comprobación para cualquier **OutputStream** (no simplemente de compresión).
DeflaterOutputStream	Clase base de las clases de compresión.
ZipOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos ZIP.
GZIPOutputStream	Una **DeflaterOutputStream** que comprime datos en el formato de archivos GZIP.
InflaterInputStream	Clase base de las clases de descompresión.

Clase de Compresión	Función
ZipInputStream	Una **InflaterInputStream** que descomprime datos almacenados en el formato de archivos ZIP.
GZIPInputStream	Una **InfiaterInputStream** que descomprime datos almacenados en el formato de archivos GZIP.

Aunque hay muchos algoritmos de compresión, los más comúnmente utilizados son probablemente ZIP y GZIP. Por consiguiente, se pueden manipular de forma sencilla los datos comprimidos con las muchas herramientas disponibles para leer y escribir en estos formatos.

Compresión sencilla con GZIP

La interfaz GZIP es simple y por consiguiente la más apropiada a la hora de comprimir un único flujo de datos (en vez de tener un contenedor de datos heterogéneos). He aquí un ejemplo que comprime un archivo:

```
//: c11:GZIPcompress.java
// Uses GZIP compression to compress a file
// whose name is passed on the command line.
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    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);
  }
} ///:~
```

El uso de clases de compresión es directo -simplemente se envuelve el flujo de salida en un **GZIPOutputStream** o en un **ZipOutputStream,** y el flujo de entrada en un **GZIPInputStream** o en un **ZipInputStream.** Todo lo demás es lectura y escritura de E/S ordinarias. Éste es un ejemplo que mezcla flujos orientados a **byte** con flujos orientados a **char: entrada** usa las clases **Reader,** mientras que el constructor de **GZIPOutputStream** sólo puede aceptar un objeto **OutputStream,** y no un objeto **Writer.** Cuando se abre un archivo, se convierte el **GZIPInputStream** en un **Reader.**

Almacenamiento múltiple con ZIP

La biblioteca que soporta el formato ZIP es mucho más completa. Con ella, es posible almacenar de manera sencilla múltiples archivos, e incluso existe una clase separada para hacer más sencillo el proceso de leer un archivo **ZIP** La biblioteca usa el formato ZIP estándar de forma que trabaja prácticamente con todas las herramientas actualmente descargables desde Internet. El ejemplo siguiente tiene la misma forma que el anterior, pero maneja tantos parámetros de línea de comandos como se desee. Además, muestra el uso de las clases **Checksum** para calcular **y** verificar la suma de comprobación del archivo. Hay dos tipos de **Checksum: Adler32** (que es más rápido) y **CRC32** (que es más lento pero ligeramente más exacto).

```
//: c11:ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line.
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class ZipCompress {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f =
      new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(
        f, new Adler32());
    ZipOutputStream out =
```

```
      new ZipOutputStream(
        new BufferedOutputStream(csum));
    out.setComment("A test of Java Zipping");
    // No corresponding getComment(), though.
    for(int i = 0; i < args.length; i++) {
      System.out.println(
        "Writing file " + args[i]);
      BufferedReader in =
        new BufferedReader(
          new FileReader(args[i]));
      out.putNextEntry(new ZipEntry(args[i]));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
    }
    out.close();
    // Checksum valid only after the file
    // has been closed!
    System.out.println("Checksum: " +
      csum.getChecksum().getValue());
    // Now extract the files:
    System.out.println("Reading file");
    FileInputStream fi =
       new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(
        fi, new Adler32());
    ZipInputStream in2 =
      new ZipInputStream(
        new BufferedInputStream(csumi));
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      System.out.println("Reading file " + ze);
      int x;
      while((x = in2.read()) != -1)
        System.out.write(x);
    }
    System.out.println("Checksum: " +
      csumi.getChecksum().getValue());
    in2.close();
    // Alternative way to open and read
    // zip files:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      System.out.println("File: " + ze2);
      // ... and extract the data as before
    }
```

```
    }
} ///:~
```

Para cada archivo que se desee añadir al archivo, es necesario llamar a **putNextEntry**( ) y pasarle un objeto **ZipEntry**. Este objeto contiene una interfaz que permite extraer y modificar todos los datos disponibles en esa entrada particular del archivo ZIP: nombre, tamaño comprimido y descomprimido, fecha, suma de comprobación CRC, datos de campos extra, comentarios, métodos de compresión y si es o no una entrada de directorio. Sin embargo, incluso aunque el formato ZIP permite poner contraseñas a los archivos, no hay soporte para esta faceta en la biblioteca ZIP de Java. Y aunque tanto **CheckedInputStream** como **CheckedOutputStream** soportan ambos sumas de comprobación, Adler32 y CRC32, la clase **ZipEntry** sólo soporta una interfaz para CRC. Esto es una restricción del formato ZIP subyacente, pero podría limitar el uso de la más rápida Adler32.

Para extraer archivos, **ZipInputStream** tiene un método **getNextEntry**( ) que devuelve la siguiente **ZipEntry** si es que la hay. Una alternativa más sucinta es la posibilidad de leer el archivo utilizando un objeto **ZipFile**, que tiene un método **entries**( ) para devolver una **Enumeration** al **ZipEntries**.

Para leer la suma de comprobación hay que tener algún tipo de acceso al objeto **Checksum** asociado. Aquí, se retiene una referencia a los objetos **CheckedOutputStream** y **CheckedInputStream**, pero también se podría simplemente guardar una referencia al objeto **Checksum**.

Existe un método misterioso en los flujos Zip que es **setComment**(). Como se mostró anteriormente, se puede poner un comentario al escribir un archivo, pero no hay forma de recuperar el comentario en el **ZipInputStream**. Parece que los comentarios están completamente soportados en una base de entrada por entrada, eso sí, solamente vía **ZipEntry**. Por supuesto, no hay un número de archivos al usar las bibliotecas GZIP o ZIP -se puede comprimir cualquier cosa, incluidos los datos a enviar a través de una conexión de red.

Archivos Java (JAR)

El formato ZIP también se usa en el formato de archivos JAR (Java ARchive), que es una forma de coleccionar un grupo de archivos en un único archivo comprimido, exactamente igual que el zip. Sin embargo, como todo lo demás en Java, los ficheros JAR son multiplataforma, por lo que no hay que preocuparse por aspectos de plataforma. También se pueden incluir archivos de audio e imagen, o archivo de clases.

Los archivos **JAR** son particularmente útiles cuando se trabaja con Internet. Antes de los archivos JAR, el navegador Web habría tenido que hacer peticiones

múltiples a un servidor web para descargar todos los archivos que conforman un *applet.* Además, cada uno de estos archivos estaba sin comprimir. Combinando todos los archivos de un *applet* particular en un único archivo JAR, sólo es necesaria una petición al servidor, y la transferencia es más rápida debido a la compresión. Y cada entrada de un archivo JAR soporta firmas digitales por seguridad (consultar a la documentación de Java si se necesitan más detalles).

Un archivo JAR consta de un único archivo que contiene una colección de archivos ZIP junto con una "declaración" que los describe. (Es posible crear archivos de declaración; de otra manera el programa **jar** lo hará automáticamente.) Se puede averiguar algo más sobre declaraciones JAR en la documentación del JDK HTML.

La utilidad **jar** que viene con el JDK de Sun comprime automáticamente los archivos que se seleccionen. Se invoca en línea de comandos:

```
jar [options] destination [manifest] inputfile(s)
```

Las opciones son simplemente una colección de letras (no es necesario ningún guión u otro indicador). Los usuarios de Unix/Linux notarán la semejanza con las opciones **tar.** Éstas son:

c	Crea un archivo nuevo o vacío.
t	Lista la tabla de contenidos.
x	Extrae todos los archivos.
x file	Extrae el archivo nombrado.
f	Dice: "Voy a darte el nombre del archivo." Si no lo usas, JAR asume que su entrada provendrá de la entrada estándar, o, si está creando un archivo, su salida irá a la salida estándar.
m	Dice que el primer parámetro será el nombre de un archivo de declaración creado por el usuario.
v	Genera una salida que describe qué va haciendo JAR.
0	Simplemente almacena los archivos; no los comprime (usarlo para crear un archivo JAR que se puede poner en el *classpath*).
M	No crear automáticamente un archivo de declaración.

Si se incluye algún subdirectorio en los archivos a añadir a un archivo JAR, se añade ese subdirec- torio automáticamente, incluyendo también todos sus subdirectorios, etc. También se mantiene la información de rutas.

He aquí algunas formas habituales de invocar a **jar:**

```
jar cf myJarFile.jar *.class
```

Esto crea un fichero JAR llamado **miFicheroJar.jar** que contiene todos los archivos de clase del directorio actual, junto con un archivo declaración creado automáticamente.

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Como en el ejemplo anterior, pero añadiendo un archivo de declaración de nombre **miArchivo Declaracion.mf** creado por el usuario.

```
jar tf myJarFile.jar
```

Añade el indicador que proporciona información más detallada sobre los archivos de **miArchivoJar.jar.**

```
jar tvf myJarFile.jar
```

Añade la bandera “verbose” para dar más información detallada acerca de los archivos en **myJarFile.jar**.

```
jar cvf myApp.jar audio classes image
```

Si se asume que **audio, clases** e **imagen** son subdirectorios, combina todos los subdirectorios en el archivo **miAplicacion.jar.** También se incluye el indicador que proporciona realimentación extra mientras trabaja el programa **jar.**

Si se crea un fichero JAR usando la opción 0, el archivo se puede ubicar en el CLASSPATH:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Entonces, Java puede buscar archivos de clase en **1ibl.jar** y **lib2.jar.**

La herramienta **jar** no es tan útil como una utilidad **zip.** Por ejemplo, no se pueden añadir o actualizar archivos de un archivo JAR existente; sólo se pueden crear archivos JAR de la nada. Además, no se pueden mover archivos a un archivo **JAR,** o borrarlos al moverlos'. Sin embargo, un fichero JAR creado en una plataforma será legible transparentemente por la herramienta **jar** en cualquier otra plataforma (un problema que a veces se da en las utilidades **zip**).

Como se verá en el Capítulo 13, los archivos JAR también se utilizan para empaquetar JavaBeans.

Serialización de objetos

La ***serialización de objetos*** de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits que puede ser posteriormente restaurada para regenerar el objeto original. Esto es cierto incluso a través de una red, lo que significa que el mecanismo de serialización compensa automáticamente las diferencias entre sistemas operativos. Es decir, se puede crear un objeto en una máquina Windows, serializarlo, y enviarlo a través de la red a una máquina Unix, donde será reconstruido correctamente. No hay que preocuparse de las representaciones de los datos en las distintas máquinas, al igual que no importan la ordenación de los bytes y el resto de detalles.

Por sí misma, la serialización de objetos es interesante porque permite implementar ***penitencia ligera.*** Hay que recordar que la persistencia significa que el tiempo de vida de un objeto no viene determinado por el tiempo que dure la ejecución del programa -el objeto vive ***mientras se den*** invocaciones al mismo en el programa. Al tomar un objeto serializable y escribirlo en el disco, y luego restaurarlo cuando sea reinvocado en el programa, se puede lograr el efecto de la persistencia. La razón por la que se califica de "ligera" es porque simplemente no se puede definir un objeto utilizando algún tipo de palabra clave "persistent" y dejar que el sistema se encargue de los detalles (aunque puede que esta posibilidad exista en el futuro). Por el contrario, hay que serializar y deserializar explícitamente los objetos.

La serialización de objetos se añadió a Java para soportar dos aspectos de mayor calibre. La invocación de Procedimientos Remotos *(Remote Method Invocation-RMO* permite a objetos de otras máquinas comportarse como si se encontraran en la tuya propia. Al enviar mensajes a objetos remotos, es necesario serializar los parámetros y los valores de retorno. RMI se discute en el Capítulo 15.

La serialización de objetos también es necesaria en el caso de los JavaBeans, descritos en el Capítulo 13. Cuando se usa un Bean, su información de estado se suele configurar en tiempo de diseño. La información de estado debe almacenarse y recuperarse más tarde al comenzar el programa; la serialización de objetos realiza esta tarea.

La serialización de un objeto es bastante simple, siempre que el objeto implemente la interfaz **Serializable** (la interfaz es simplemente un flag y no tiene métodos). Cuando se añadió la serialización al lenguaje, se cambiaron muchas clases de la biblioteca estándar para que fueran serializables, incluidos todos los envoltorios y tipos primitivos, todas las clases contenedoras, y otras muchas. Incluso los objetos **Class** pueden ser serializados. (Véase el Capítulo 10 para comprender las implicaciones de esto.)

Para serializar un objeto, se crea algún tipo de objeto **OutputStream** y se envuelve en un **Object OutputStream.** En este momento sólo hay que invocar a

writeObject( ) y el objeto se serializa y se envía al **OutputStream.** Para invertir este proceso, se envuelve un **InputStream** en un **ObjectInputStream** y se invoca a **readObject**( ). Lo que vuelve, como siempre, es una referencia a un **Object,** así que hay que hacer una conversión hacia abajo para dejar todo como se debe.

Un aspecto particularmente inteligente de la serialización de objetos es que, no sólo salva la imagen del objeto, sino que también sigue todas las referencias contenidas en el objeto, y salva *esos* objetos, siguiendo además las referencias contenidas en cada uno de ellos, etc. A esto se le suele denominar la "telaraña de objetos" puesto que un único objeto puede estar conectado, e incluir arrays de referencias a objetos, además de objetos miembro. Si se tuviera que mantener un esquema de serialización de objetos propio, el mantenimiento del código para seguir todos estos enlaces sería casi imposible. Sin embargo, la serialización de objetos Java parece encargarse de todo haciendo uso de un algoritmo optimizado que recorre la telaraña de objetos. El ejemplo siguiente prueba el mecanismo de serialización haciendo un "gusano" de objetos enlazados, cada uno de los cuales tiene un enlace al siguiente segmento del gusano, además de un array de referencias a objetos de una clase distinta, **Datos:**

```
//: c11:Worm.java
// Demonstrates object serialization.
import java.io.*;

class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Worm implements Serializable {
  // Generate a random int value:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
    new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
```

```
  }
  Worm() {
    System.out.println("Default constructor");
  }
  public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
      s += d[i].toString();
    s += ")";
    if(next != null)
      s += next.toString();
    return s;
  }
  // Throw exceptions to console:
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    ObjectOutputStream out =
      new ObjectOutputStream(
        new FileOutputStream("worm.out"));
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.close(); // Also flushes output
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    System.out.println(s + ", w2 = " + w2);
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out2 =
      new ObjectOutputStream(bout);
    out2.writeObject("Worm storage");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 =
      new ObjectInputStream(
        new ByteArrayInputStream(
          bout.toByteArray()));
    s = (String)in2.readObject();
    Worm w3 = (Worm)in2.readObject();
    System.out.println(s + ", w3 = " + w3);
  }
} ///:~
```

Para hacer las cosas interesantes, el **array** de objetos **Datos** dentro de **Gusano** se inicializa con números al azar. (De esta forma no hay que sospechar que el

compilador mantenga algún tipo de meta-información.) Cada segmento **Gusano** se etiqueta con un **char** que es automáticamente generado en el proceso de generar recursivamente la lista enlazada de **Gusanos**. Cuando se crea un **Gusano**, se indica al constructor lo largo que se desea que sea. Para hacer la referencia siguiente llama al constructor **Gusano** con una longitud uno menor, etc. La referencia siguiente final se deja a **null** indicando el final del **Gusano**.

El objetivo de todo esto era hacer algo racionalmente complejo que no pudiera ser serializado fácilmente. El acto de serializar, sin embargo, es bastante simple. Una vez que se ha creado el **ObjectOutputStream** a partir de otro flujo, **writeObject**( ) serializa el objeto. Nótese que la llamada a **writeObject**( ) es también para un **String**. También se pueden escribir los tipos de datos primitivos utilizando los mismos métodos que **DataOutputStream** (comparten las mismas interfaces).

Hay dos secciones de código separadas que tienen la misma apariencia. La primera lee y escribe un archivo, y la segunda escribe y lee un **ByteArray**. Se puede leer y escribir un objeto usando la serialización en cualquier **DataInputStream** o **DataOutputStream**, incluyendo, como se verá en el Capítulo 17, una red. La salida de una ejecución fue:

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)
```

Se puede ver que el objeto deserializado contiene verdaderamente todos los enlaces que había en el objeto original.

Nótese que no se llama a ningún constructor, ni siquiera el constructor por defecto, en el proceso de deserialización de un objeto **Serializable.** Se restaura todo el objeto recuperando datos del **InputStream.**

La serialización de objetos está orientada al **byte,** y por consiguiente usa las jerarquías **InputStream** y **OutputStream.**

Encontrar la clase

Uno podría preguntarse qué debe tener un objeto para que pueda ser recuperado de su estado serealizado. Por ejemplo, supóngase que se serializa un objeto y se envía como un archivo o a través de una red a otro máquina. ¿Podría un

programa de la otra máquina reconstruir el objeto simplemente con el contenido del archivo?

La mejor forma de contestar a esta pregunta e s (como siempre) haciendo un experimento. El archivo siguiente se encuentra en el subdirectorio de este capítulo:

```
//: c11:Alien.java
// A serializable class.
import java.io.*;

public class Alien implements Serializable {
} ///:~
```

El archivo que crea y serializa un objeto **Extraterrestre** va en el mismo directorio:

```
//: c11:FreezeAlien.java
// Create a serialized output file.
import java.io.*;

public class FreezeAlien {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException {
    ObjectOutput out =
      new ObjectOutputStream(
        new FileOutputStream("X.file"));
    Alien zorcon = new Alien();
    out.writeObject(zorcon);
  }
} ///:~
```

Más que capturar y manejar excepciones, este programa toma el enfoque ***rápido*** y ***sucio*** de pasar las excepciones fuera del método **main**( ), de forma que serán reportadas en línea de comandos.

Una vez que se compila y ejecuta el programa, copie el fichero **Expediente.X** resultante al directorio denominado **expedientesx,** en el que se encuentra el siguiente código:

```
//: c11:xfiles:ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
import java.io.*;

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

```
  throws IOException, ClassNotFoundException {
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("X.file"));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} ///:~
```

Este programa abre el archivo y lee el objeto **misterio** con éxito. Sin embargo, en cuanto se intenta averiguar algo del objeto -lo cual requiere el objeto **Class** de **Extraterrestre-** la Máquina Virtual Java (JVM) no puede encontrar **Extraterrestre.class** (a menos que esté en el *Classpath*, lo cual no ocurre en este ejemplo). Se obtendrá una **ClassNotFoundException.** (¡De nuevo, se desvanece toda esperanza de vida extraterrestre antes de poder encontrar una prueba de su existencia!)

Si se espera hacer mucho una vez recuperado un objeto serializado, hay que asegurarse de que la JVM pueda encontrar el archivo **.class** asociado en el *path* de clases locales o en cualquier otro lugar en Internet.

Controlar la serialización

Como puede verse, el mecanismo de serialización por defecto tiene un uso trivial. Pero ¿qué pasa si se tienen necesidades especiales? Quizás se tienen aspectos especiales relativos a seguridad y no se desea serializar algunas porciones de un objeto, o quizás simplemente no tiene sentido que se serialice algún subobjeto si esa parte necesita ser creada de nuevo al recuperar el objeto.

Se puede controlar el proceso de serialización implementando la interfaz **Externalizable** en vez de la interfaz **Serializable.** La interfaz **Externalizable** extiende la interfaz **Serializable** añadiendo dos métodos, **writeExternal**( ) y **readExternal**( ), que son invocados automáticamente para el objeto durante la serialización y la deserialización, de forma que se puedan llevar a cabo las operaciones especiales.

El ejemplo siguiente muestra la implementación simple de los métodos de la interfaz
Externalizable. Nótese que **Rastrol** y **Rastro2** son casi idénticos excepto por una diferencia mínima (a ver si la descubres echando un vistazo al código):

```
//: c11:Blips.java
// Simple use of Externalizable & a pitfall.
import java.io.*;
import java.util.*;
```

```
class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    System.out.println("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
  }
}

public class Blips {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    ObjectOutputStream o =
      new ObjectOutputStream(
        new FileOutputStream("Blips.out"));
    System.out.println("Saving objects:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    // Now get them back:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("Blips.out"));
    System.out.println("Recovering b1:");
    b1 = (Blip1)in.readObject();
    // OOPS! Throws an exception:
//! System.out.println("Recovering b2:");
```

```
//! b2 = (Blip2)in.readObject();
  }
} ///:~
```

La salida del programa es:

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
```

La razón por la que el objeto Rastro2 no se recupera es que intentar hacerlo causa una excepción. ¿Se ve la diferencia entre **Rastrol** y **Rastro2**? El constructor de **Rastrol** es **public**, mientras que el constructor de **Rastro2** no lo es, y eso causa la excepción en la recuperación. Puede intentarse hacer **public** el constructor de Rastro2 y retirar los comentarios ***//!*** para ver los resultados correctos.

Cuando se recupera **rl**, se invoca al constructor por defecto de **Rastrol**. Esto es distinto a recuperar el objeto **Serializable**, en cuyo caso se construye el objeto completamente a partir de sus bits almacenados, sin llamadas al constructor. Con un objeto **Externalizable**, se da todo el comportamiento de construcción por defecto normal (incluyendo las inicializaciones del momento de la definición de campos), y ***posteriormente,*** se invoca a **readExternal**( ). Es necesario ser consciente de esto -en particular, del hecho de que siempre tiene lugar toda la construcción por defecto- para lograr el comportamiento correcto en los objetos **Externalizables**.

He aquí un ejemplo que muestra qué hay que hacer para almacenar y recuperar un objeto **Externalizable** completamente:

```
//: c11:Blip3.java
// Reconstructing an externalizable object.
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
```

```
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in nondefault
    // constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    System.out.println("Blip3.writeExternal");
    // You must do this:
    out.writeObject(s);
    out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject();
    i =in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    ObjectOutputStream o =
      new ObjectOutputStream(
        new FileOutputStream("Blip3.out"));
    System.out.println("Saving object:");
    o.writeObject(b3);
    o.close();
    // Now get it back:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("Blip3.out"));
    System.out.println("Recovering b3:");
    b3 = (Blip3)in.readObject();
    System.out.println(b3);
  }
} ///:~
```

Los campos s e i se inicializan solamente en el segundo constructor, pero no en el constructor por defecto. Esto significa que si no se inicializan **s** e **i** en **readExternal**( ), serán **null** (puesto que el espacio de almacenamiento del objeto se inicializa a ceros en el primer paso de la creación del mismo). Si se comentan

las dos líneas de código que siguen a las frases "Hay que hacer esto" y se ejecuta el programa, se verá que se recupera el objeto, **s** es **null**, e **i** es cero.

Si se está heredando de un objeto **Externalizable**, generalmente se invocará a las versiones de clase base de **writeExternal**() y **readExternal**() para proporcionar un almacenamiento y recuperación adecuados de los componentes de la clase base.

Por tanto, para que las cosas funcionen correctamente, no sólo hay que escribir los datos importantes del objeto durante el método **writeExternal**() (no hay comportamiento por defecto que escriba ninguno de los objetos miembro de un objeto **Externalizable**), sino que también hay que recuperar los datos en el método **readExternal**(). Esto puede ser un poco confuso al principio puesto que el comportamiento por defecto de la construcción de un objeto **Externalizable** puede hacer que parezca que tiene lugar automáticamente algún tipo de almacenamiento y recuperación. Esto no es así.

La palabra clave transient

Cuando se está controlando la serialización, puede ocurrir que haya un subobjeto en particular para el que no se desee que se produzca un salvado y recuperación automáticos por parte del mecanismo de serialización de Java. Éste suele ser el caso si ese objeto representa información sensible que no se desea serializar, como una contraseña. Incluso si esa información es **private** en el objeto, una vez serializada es posible que cualquiera acceda a la misma leyendo el objeto o interceptando una transmisión de red.

Una forma de evitar que partes sensibles de un objeto sean serializables es implementar la clase como **Externalizable**, como se ha visto previamente. Así, no se serializa automáticamente nada y se pueden serializar explícitamente sólo las partes de **writeExternal**() necesarias.

Sin embargo, al trabajar con un objeto **Serializable**, toda la serialización se da automáticamente. Para controlar esto, se puede desactivar la serialización en una base campo-a-campo utilizando la palabra clave **transient**, que dice: "No te molestes en salvar o recuperar esto -me encargaré yo".

Por ejemplo, considérese un objeto **InicioSesion,** que mantiene información sobre un inicio de sesión en particular. Supóngase que, una vez verificado el inicio, se desean almacenar los datos, pero sin la contraseña. La forma más fácil de hacerlo es implementar **Serializable** y marcar el campo **contraseña** como **transient.** Debería quedar algo así:

```
//: c11:Logon.java
// Demonstrates the "transient" keyword.
import java.io.*;
```

```
import java.util.*;

class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
    String pwd =
      (password == null) ? "(n/a)" : password;
    return "logon info: \n   " +
      "username: " + username +
      "\n   date: " + date +
      "\n   password: " + pwd;
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    Logon a = new Logon("Hulk", "myLittlePony");
    System.out.println( "logon a = " + a);
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    // Delay:
    int seconds = 5;
    long t = System.currentTimeMillis()
           + seconds * 1000;
    while(System.currentTimeMillis() < t)
      ;
    // Now get them back:
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("Logon.out"));
    System.out.println(
      "Recovering object at " + new Date());
    a = (Logon)in.readObject();
    System.out.println( "logon a = " + a);
  }
} ///:~
```

Se puede ver que los campos **fecha** y **usuario** son normales (no **transient)** y, por consiguiente, se serializan automáticamente. Sin embargo, el campo **contraseña** es **transient,** así que no se almacena en el disco; además el mecanismo de serialización ni siquiera intenta recuperarlo. La salida es:

```
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: myLittlePony
Recovering object at Sun Mar 23 18:25:59 PST 1997
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: (n/a)
```

Cuando se recupera el objeto, el campo **contrasenia** es **null.** Nótese que **toString**( ) debe comprobar si hay un valor **null** en **contraseña** porque si se intenta ensamblar un objeto **String** haciendo uso del operador '+' sobrecargado, y ese operador encuentra una referencia a **null,** se genera una **NullPointerException.** (En versiones futuras de Java puede que se añada código que evite este problema.)

También se puede ver que el campo **fecha** se almacena **y** recupera a partir del disco en vez de regenerarse de nuevo.

Puesto que los objetos **Externalizable** no almacenan ninguno de sus campos por defecto, la palabra clave **transient** es para ser usada sólo con objetos **Serializable.**

Una alternativa a Externalizable

Si uno no es entusiasta de la implementación de la interfaz **Externalizable,** hay otro enfoque. Se puede implementar la interfaz **Serializable** y añadir (nótese que se dice "añadir", y no "superponer" o "implementar") métodos llamados **writeObject**( ) y **readObject**( ), que serán automáticamente invocados cuando se serialice y deserialice el objeto, respectivamente. Es decir, si se proporcionan estos dos métodos, se usarán en vez de la serialización por defecto.

Estos métodos deben tener exactamente las signaturas siguientes:

```
private void
  writeObject(ObjectOutputStream stream)
    throws IOException;

private void
  readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

Desde el punto de vista del diseño, aquí todo parece un misterio. En primer lugar, se podría pensar que, debido a que estos métodos no son parte de una clase base o de la interfaz **Serializable,** deberían definirse en sus propias interface(s). Pero nótese que se definen como **private,** lo que significa que sólo van a ser

invocados por miembros de esa clase. Sin embargo, de hecho no se les invoca desde otros miembros de esta clase, sino que son los métodos **writeObject**( ) y **readobject** ( ) de los objetos **ObjectOutputStream** y **ObjectInputStream** los que invocan a los métodos **writeObject**( ) y **readObject**( ) de nuestro objeto. (Nótese nuestro tremendo temor a no comenzar una larga diatriba sobre el nombre de los métodos a usar aquí. En pocas palabras: todo es confuso.) Uno podría preguntarse cómo logran los objetos **ObjectOutputStream** y **ObjectInputStream** acceso a los métodos **private** de la clase. Sólo podemos asumir que es parte de la magia de la serialización.

En cualquier caso, cualquier cosa que se defina en una **interface** es automáticamente **public,** por lo que si **writeObject**( ) y **readObject**( ) deben ser **private,** no pueden ser parte de una **interface**. Puesto que hay que seguir las signaturas con exactitud, el efecto es el mismo que si se está implementando una **interfaz.**

Podría parecer que cuando se invoca a **ObjectOutputStream.writeObject**( ), se interroga al objeto **Serializable** que se le pasa (utilizando sin duda la reflectividad) para ver si implementa su propio **writeObject**( ). Si es así, se salta el proceso de serialización normal, y se invoca al **writeObject**( ).

En el caso de **readObject**( ) ocurre exactamente igual. Hay otra particularidad. Dentro de tu **writeObject**( ) se puede elegir llevar a cabo la acción **writeObject**( ) por defecto invocando a **defaultWriteObject**( ). De forma análoga, dentro de **readObject**( ) se puede invocar a **defaultReadObject**( ). He aquí un ejemplo simple que demuestra cómo se puede controlar el almacenamiento y recuperación de un objeto **Serializable:**

```
//: c11:SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
  String a;
  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);
    // Now get it back:
    ObjectInputStream in =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf.toByteArray()));
    SerialCtl sc2 = (SerialCtl)in.readObject();
    System.out.println("After:\n" + sc2);
  }
} ///:~
```

En este ejemplo, uno de los campos **String** es normal, y el otro es **transient,** para probar que se salva el campo no **transient** por parte del método **defaultWriteObject**( ), mientras que el campo **transient** se salva y recupera de forma explícita. Se inicializan los campos dentro del constructor en vez de definirlos para probar que no están siendo inicializados por ningún tipo de mecanismo automático durante la deserialización.

Si se va a usar el mecanismo por defecto para escribir las partes no **transient** del objeto, hay que invocar a **defaultWriteObject**( ) como primera operación de **writeObject**( ), y a **defaultReadObject**( ), como primera operación de **readObject**( ). Éstas son llamadas extrañas a métodos. Podría parecer, por ejemplo, que está llamando al método **defaultWriteObject**( ) de un **ObjectOutputStream** sin pasar argumentos, y así de algún modo convierte y conoce la referencia a su objeto y cómo escribir todas las partes no **transient.**

El almacenamiento y recuperación de objetos **transient** usa más código familiar. Y lo que es más: piense en lo que ocurre aquí. En el método **main**( ), se crea un objeto **CtlSerial,** y después se serializa a un **ObjectOutputStream.** (Nótese que en ese caso se usa un espacio de almacenamiento intermedio en vez de un archivo.) La serialización se realiza en la línea:

```
o.writeObject(sc);
```

El método **writeObject**( ) debe examinar **cs** para averiguar si tiene su propio método **writeobject** ( ).
(No comprobando la interfaz -pues no la hay- o el tipo de clase, sino buscando el método haciendo uso de la reflectividad.) Si lo tiene, se usa. Se sigue un enfoque semejante en el caso de **readObject**( ).

Quizás ésta era la única forma, en la práctica, de solucionar el problema, pero es verdaderamente extraña.

Versionar

Es posible que se desee cambiar la versión de una clase serializable (por ejemplo, se podrían almacenar objetos de la clase original en una base de datos). Esto se soporta, pero probablemente se hará sólo en casos especiales, y requiere de una profundidad de entendimiento adicional que no trataremos de alcanzar aquí. La documentación JDK HTML descargable de *http://iava.sun.com* cubre este tema de manera bastante detallada.

También se verá que muchos comentarios de la documentación JDK HTML comienzan por:

***Aviso**: Los objetos serializados de esta clase no serán compatibles con versiones futuras de Swing. El soporte actual para serialización es apropiado para almacenamiento a corto plazo o RMI entre aplicaciones* ...

Esto se debe a que el mecanismo de versionado es demasiado simple como para que funcione correctamente en todas las situaciones, especialmente con JavaBeans. Actualmente se está trabajando en corregir su diseño, y por eso se presentan estas advertencias.

Utilizar la persistencia

Es bastante atractivo usar la tecnología de serialización para almacenar algún estado de un programa de forma que se pueda restaurar el programa al estado actual más adelante. Pero antes de poder hacer esto hay que resolver varias cuestiones. ¿Qué ocurre si se serializan dos objetos teniendo ambos una referencia a un tercero? Cuando se restauren esos dos objetos de su estado serializado ¿se obtiene sólo una ocurrencia del tercer objeto? ¿Qué ocurre si se serializan los dos objetos en archivos separados y se deserializan en partes distintas del código?

He aquí un ejemplo que muestra el problema:

```
//: c11:MyWorld.java
import java.io.*;
import java.util.*;

class House implements Serializable {}

class Animal implements Serializable {
  String name;
  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();
    ArrayList  animals = new ArrayList();
    animals.add(
      new Animal("Bosco the dog", house));
    animals.add(
      new Animal("Ralph the hamster", house));
    animals.add(
      new Animal("Fronk the cat", house));
    System.out.println("animals: " + animals);

    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 =
      new ObjectOutputStream(buf1);
    o1.writeObject(animals);
    o1.writeObject(animals); // Write a 2nd set
    // Write to a different stream:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 =
      new ObjectOutputStream(buf2);
    o2.writeObject(animals);
    // Now get them back:
    ObjectInputStream in1 =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf1.toByteArray()));
    ObjectInputStream in2 =
```

```
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf2.toByteArray()));
    ArrayList animals1 =
      (ArrayList)in1.readObject();
    ArrayList animals2 =
      (ArrayList)in1.readObject();
    ArrayList animals3 =
      (ArrayList)in2.readObject();
    System.out.println("animals1: " + animals1);
    System.out.println("animals2: " + animals2);
    System.out.println("animals3: " + animals3);
  }
} ///:~
```

Una de las cosas interesantes aquí es que es posible usar la serialización de objetos para y desde un array de bytes logrando una "copia en profundidad" de cualquier objeto **Serializable.** (Una copia en profundidad implica duplicar la telaraña de objetos entera, en vez de simplemente el objeto básico y sus referencias). La copia se cubre en detalle en el Apéndice A.

Los objetos **Animal** contienen campos del tipo **Casa.** En el método **main**( ), se crea un **ArrayList** de estos **Animales** y se serializa dos veces en un flujo y de nuevo a otro flujo distinto. Cuando se deserializan e imprimen, se verán en la ejecución los resultados siguientes (en cada ejecución, los objetos estarán en distintas posiciones de memoria):

```
animals: [Bosco the dog[Animal@1cc76c], House@1cc769
, Ralph the hamster[Animal@1cc76d], House@1cc769
, Fronk the cat[Animal@1cc76e], House@1cc769
]
animals1: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals2: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals3: [Bosco the dog[Animal@1cca52], House@1cca5c
, Ralph the hamster[Animal@1cca5d], House@1cca5c
, Fronk the cat[Animal@1cca61], House@1cca5c
]
```

Por supuesto, se puede esperar que los objetos deserializados tengan direcciones distintas a la del original. Pero nótese que en **animalesl** y **animales2** aparecen las mismas direcciones, incluyendo las referencias al objeto **Casa** que ambos comparten. Por otro lado, cuando se recupera **animales3** el sistema no puede saber que los objetos del otro flujo son alias de los objetos del primer flujo, por lo

que construye una telaraña de objetos completamente diferente. Mientras se serialice todo a un único flujo, se podrá recuperar la misma telaraña de objetos que se escribió, sin duplicaciones accidentales de los mismos. Por supuesto, se puede cambiar el estado de los objetos en el tiempo que transcurre entre la escritura del primero y el último, pero eso es responsabilidad de cada uno -los objetos se escribirán en el estado en el que estén (y con cualquier conexión que tengan con otros objetos) en el preciso momento de la serialización.

Lo más seguro si se desea salvar el estado de un sistema es hacer la serialización como una operación "atómica". Si se serializa una parte, se hacen otras cosas, luego se serializa otra parte, etc., no se estará almacenando el sistema de forma segura. Lo que hay que hacer es poner todos los objetos que conforman el estado del sistema en un único contenedor y simplemente se escribe este contenedor en una única operación. Después, es posible restaurarlo también con una única llamada a un método.

El ejemplo siguiente es un sistema de diseño asistido por ordenador (CAD) que demuestra el enfoque. Además, se introduce en el aspecto de los campos **static** - si se echa un vistazo a la documentación se verá que **Class** es **Serializable**, por lo que debería ser fácil almacenar campos **static** simplemente serializando el objeto **Class.** De cualquier forma, este enfoque parece sensato.

```
//: c11:CADState.java
// Saving and restoring the state of a
// pretend CAD system.
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 r = new Random();
  private static int counter = 0;
  abstract public void setColor(int newColor);
  abstract public 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 = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.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 CADState {
  public static void main(String[] args)
  throws Exception {
    ArrayList shapeTypes, shapes;
    if(args.length == 0) {
      shapeTypes = new ArrayList();
      shapes = new ArrayList();
      // Add references to the class objects:
      shapeTypes.add(Circle.class);
      shapeTypes.add(Square.class);
      shapeTypes.add(Line.class);
      // Make some shapes:
      for(int i = 0; i < 10; i++)
        shapes.add(Shape.randomFactory());
      // Set all the static colors to GREEN:
      for(int i = 0; i < 10; i++)
        ((Shape)shapes.get(i))
          .setColor(Shape.GREEN);
      // Save the state vector:
      ObjectOutputStream out =
        new ObjectOutputStream(
          new FileOutputStream("CADState.out"));
      out.writeObject(shapeTypes);
      Line.serializeStaticState(out);
      out.writeObject(shapes);
    } else { // There's a command-line argument
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream(args[0]));
      // Read in the same order they were written:
      shapeTypes = (ArrayList)in.readObject();
      Line.deserializeStaticState(in);
      shapes = (ArrayList)in.readObject();
    }
    // Display the shapes:
    System.out.println(shapes);
  }
```

```
} ///:~
```

La clase **Figura implementa Serializable,** por lo que cualquier cosa que herede de **Figura** es automáticamente también **Serializable.** Cada **Figura** contiene datos, **y** cada clase **Figura** derivada contiene un campo **static** que determina el color de todos esos tipos de **Figuras.** (Colocar un campo **static** en la clase base resultaría en un solo campo, mientras que los campos **static** no se duplican en las clases derivadas). Se pueden superponer los métodos de la clase base para establecer el color de los diversos tipos (los métodos **static** no se asignan dinámicamente, por lo que son método normales). El método **factoriaAleatoria**( ) crea una **Figura** diferente cada vez que se le invoca, utilizando valores al azar para los datos **Figura.**

Círculo y Cuadrado son extensiones directas de **Figura;** la única diferencia radica en que **Círculo** inicializa **color** en el momento de su definición y **Cuadrado** lo inicializa en el constructor. Se dejará la discusión sobre **Línea** para un poco más adelante.

En el método **main**( ), se usa un **ArrayList** para guardar los objetos **Class** y otro para mantener las figuras. Si no se proporciona un parámetro de línea de comandos, se crea el **ArrayList tiposFigura** y se añaden los objetos **Class,** para posteriormente crear el **ArrayList figuras** y añadirle objetos **Figura. A** continuación, se ponen a VERDE todos los valores de **static color,** y todo se serializa al archivo **EstadoCAD.out.**

Si se proporciona un parámetro de línea de comandos (se supone que **EstadoCAD.out)** se abre ese archivo y se usa para restaurar el estado del programa. En ambas situaciones, se imprime el **ArrayList** de **Figuras.** Los resultados de una ejecución son:

```
>java CADState
[class Circle color[3] xPos[-51] yPos[-99] dim[38]
, class Square color[3] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[3] xPos[-70] yPos[1] dim[16]
, class Square color[3] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[3] xPos[-75] yPos[-43] dim[22]
, class Square color[3] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[3] xPos[17] yPos[90] dim[-76]
]

>java CADState CADState.out
[class Circle color[1] xPos[-51] yPos[-99] dim[38]
, class Square color[0] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
```

```
, class Circle color[1] xPos[-70] yPos[1] dim[16]
, class Square color[0] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[1] xPos[-75] yPos[-43] dim[22]
, class Square color[0] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[1] xPos[17] yPos[90] dim[-76]
]
```

Se puede ver que los valores de **xPos**, **yPos** y **dim** se almacenaron y recuperaron satisfactoriamente, pero hay algo que no va bien al recuperar la información **static**. Se están introduciendo todo **"3",** pero no vuelve a visualizarse así. Los **Círculos** tienen valor **1** (**ROJO**, que es la definición), y los **Cuadrados** tienen valor **0** (recuérdese que se inicializan en el constructor). ¡ES como si, de hecho, los **statics** no se serializaran! Esto es así -incluso aunque la clase **Class** sea **Serializable**, no hace lo que se espera. Por tanto si se desea serializar **statics**, hay que hacerlo a mano.

Esto es para lo que sirven los métodos **static serializarEstadoEstatico**( ) y deserializar **EstadoEstatico**( ) de **Línea**. Se puede ver que se invocan explícitamente como parte del proceso de almacenamiento y recuperación. (Nótese que el orden de escritura al archivo serializado y el de lectura del mismo debe ser igual.) Por consiguiente, para que **EstadoCAD.java** funcione correctamente hay que:

Añadir un **senalizarEstadoEstatico**( ) y un **deserializarEstadoEstatico**( ) a los polígonos.
Eliminar el **ArrayList tiposFigura** y todo el código relacionado con él.
Añadir llamadas a los nuevos métodos estáticos para serializar y deserializar en cada figura.

Otro aspecto a tener en cuenta es la seguridad, puesto que la serialización también almacena datos **private**. Si se tiene un problema de seguridad, habría que marcar los campos afectados como **transient**. Pero entonces hay que diseñar una forma segura de almacenar esa información, de forma que cuando se restaure, se puedan poner a cero esas variables **private**.

Identificar símbolos de una entrada

IdentifcarSimbolos es el proceso de dividir una secuencia de caracteres en una secuencia de "símbolos", que son bits de texto delimitados por lo que se elija. Por ejemplo, los símbolos podrían ser palabras, pudiendo delimitarse por un espacio en blanco y signos de puntuación. Las dos clases proporcionadas por la biblioteca estándar de Java y que pueden ser usadas para poner símbolos son: **StreamTokenizer** y **StringTokenizer**.

StreamTokenizer

Aunque **StreamTokenizer** no deriva de **InputStream** ni de **OutputStream**, sólo funciona con objetos **InputStream**, por lo que pertenece a la parte de E/S de la biblioteca.

Considérese un programa que cuenta la ocurrencia de cada palabra en un archivo de texto:

```
//: c11:WordCount.java
// Counts words from a file, outputs
// results in sorted form.
import java.io.*;
import java.util.*;

class Counter {
  private int i = 1;
  int read() { return i; }
  void increment() { i++; }
}

public class WordCount {
  private FileReader file;
  private StreamTokenizer st;
  // A TreeMap keeps keys in sorted order:
  private TreeMap counts = new TreeMap();
  WordCount(String filename)
    throws FileNotFoundException {
    try {
      file = new FileReader(filename);
      st = new StreamTokenizer(
        new BufferedReader(file));
      st.ordinaryChar('.');
      st.ordinaryChar('-');
    } catch(FileNotFoundException e) {
      System.err.println(
        "Could not open " + filename);
      throw e;
    }
  }
  void cleanup() {
    try {
      file.close();
    } catch(IOException e) {
      System.err.println(
        "file.close() unsuccessful");
    }
```

```
  }
  void countWords() {
    try {
      while(st.nextToken() !=
        StreamTokenizer.TT_EOF) {
        String s;
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = new String("EOL");
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = st.sval; // Already a String
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
        if(counts.containsKey(s))
          ((Counter)counts.get(s)).increment();
        else
          counts.put(s, new Counter());
      }
    } catch(IOException e) {
      System.err.println(
        "st.nextToken() unsuccessful");
    }
  }
  Collection values() {
    return counts.values();
  }
  Set keySet() { return counts.keySet(); }
  Counter getCounter(String s) {
    return (Counter)counts.get(s);
  }
  public static void main(String[] args)
  throws FileNotFoundException {
    WordCount wc =
      new WordCount(args[0]);
    wc.countWords();
    Iterator keys = wc.keySet().iterator();
    while(keys.hasNext()) {
      String key = (String)keys.next();
      System.out.println(key + ": "
               + wc.getCounter(key).read());
    }
    wc.cleanup();
  }
} ///:~
```

Es fácil presentar las palabras de forma ordenada almacenando los datos en un **TreeMap**, que organiza automáticamente sus claves en orden (véase Capítulo 10). Cuando se logra un conjunto de claves utilizando **keySet**( ), éstas también estarán en orden.

Para abrir el archivo, se usa un **FileReader**, y para convertir el archivo en palabras se crea un **StreamTokenizer** a partir del **FileReader** envuelto en un **BufferedReader**. En **StreamTokenizer**, hay una lista de separadores por defecto, y se pueden añadir más con un conjunto de métodos. Aquí, se usa **ordinaryChar**( ) para decir: "Este carácter no tiene el significado en el que estoy interesado", por lo que el analizador no lo incluye como parte de las palabras que crea. Por ejemplo, decir **st.ordinaryChar**('.') quiere decir que no se incluirán los periodos como partes de las palabras a analizar. Se puede encontrar más información en la documentación JDK HTML de *http://java.sun.com*.

En **contarPalabras**( ), se sacan los símbolos de uno en uno desde el flujo, y se usa la información
`ttype` para determinar qué hacer con cada símbolo, puesto que un símbolo puede ser un **fin** de 1ínea, un número, una cadena de caracteres, o un único carácter.

Una vez que se encuentra un símbolo, se pregunta al **TreeMap** cuentas para ver si ya contiene el símbolo como clave. Si lo tiene, se incrementa el objeto Contador correspondiente, para indicar que se ha encontrado otra instancia de esa palabra. Si no, se crea un nuevo Contador -puesto que el constructor de Contador inicializa su valor a uno, esto también sirve para contar la palabra. **RecuentoPalabra** no es un tipo de **TreeMap**, por lo que no heredó de éste. Lleva a cabo un tipo de funcionalidad especifico del tipo, por lo que incluso aunque hay que reexponer los métodos **keys**( ) y **values**( ), eso sigue sin querer decir que debería usarse la herencia, puesto que utilizar varios métodos de **TreeMap** sería inadecuado. Además, se usan otros métodos como **obtenercontador**( ), que obtiene el **Contador** de un **String** en particular, y **sortedKeys**( ), que produce un **Iterator**, para finalizar el cambio en la forma de la interfaz de **RecuentoPalabra**.

En el método **main**( ) se puede ver el uso de un **RecuentoPalabra** para abrir y contar las palabras de un archivo -simplemente ocupa dos líneas de código. Después se extrae un **Iterator** a una lista de claves (palabras) ordenadas, y se usa éste para extraer otra clave y su Contador asociado. La llamada a **limpieza**( ) es necesaria para asegurar que se cierre el archivo.

StringTokenizer

Aunque éste no forma parte de la biblioteca de E/S, el **StringTokenizer** tiene funcionalidad lo suficientemente similar a **StrearnTokenizer** como para describirlo aquí.

El **StringTokenizer** devuelve todos los símbolos contenidos en una cadena de caracteres de uno en uno. Estos símbolos son caracteres consecutivos delimitados por tabuladores, espacios y saltos de línea. Por consiguiente, los símbolos de la cadena "¿Dónde está mi gato?" son "¿Dónde", "está", "mi", y "gato?". Al igual que con **StreamTokenizer** se puede indicar a **StringTokenizer** que divida la entrada de la forma que desee, pero con **StringTokenizer** esto se logra pasando un segundo parámetro al constructor, que es un **String** con los delimitadores que se desea utilizar. En general, si se necesita más sofisticación, hay que usar un **StreamTokenizer**.

Para pedir a un **StringTokenizer** que te pase el siguiente *token* de la cadena se usa el método **nextToken**() que o bien devuelve el símbolo, o bien una cadena de caracteres vacía para indicar que no quedan más símbolos. A modo de ejemplo, el programa siguiente lleva a cabo un análisis limitado de una sentencia, buscando secuencias de frases clave para indicar si hay algún tipo de alegría o tristeza implicadas.

```
//: c11:AnalyzeSentence.java
// Look for particular sequences in sentences.
import java.util.*;

public class AnalyzeSentence {
  public static void main(String[] args) {
    analyze("I am happy about this");
    analyze("I am not happy about this");
    analyze("I am not! I am happy");
    analyze("I am sad about this");
    analyze("I am not sad about this");
    analyze("I am not! I am sad");
    analyze("Are you happy about this?");
    analyze("Are you sad about this?");
    analyze("It's you! I am happy");
    analyze("It's you! I am sad");
  }
  static StringTokenizer st;
  static void analyze(String s) {
    prt("\nnew sentence >> " + s);
    boolean sad = false;
    st = new StringTokenizer(s);
    while (st.hasMoreTokens()) {
      String token = next();
      // Look until you find one of the
      // two starting tokens:
      if(!token.equals("I") &&
```

```
          !token.equals("Are"))
        continue; // Top of while loop
      if(token.equals("I")) {
        String tk2 = next();
        if(!tk2.equals("am")) // Must be after I
          break; // Out of while loop
        else {
          String tk3 = next();
          if(tk3.equals("sad")) {
            sad = true;
            break; // Out of while loop
          }
          if (tk3.equals("not")) {
            String tk4 = next();
            if(tk4.equals("sad"))
              break; // Leave sad false
            if(tk4.equals("happy")) {
              sad = true;
              break;
            }
          }
        }
      }
      if(token.equals("Are")) {
        String tk2 = next();
        if(!tk2.equals("you"))
          break; // Must be after Are
        String tk3 = next();
        if(tk3.equals("sad"))
          sad = true;
        break; // Out of while loop
      }
    }
    if(sad) prt("Sad detected");
  }
  static String next() {
    if(st.hasMoreTokens()) {
      String s = st.nextToken();
      prt(s);
      return s;
    }
    else
      return "";
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Por cada cadena de caracteres que se analiza, se entra en un bucle **while** y se extraen de la cadena los símbolos. Nótese la primera sentencia **if**, que dice que se **continúe** (volver al principio del bucle y comenzar de nuevo) si el símbolo no es ni 'Yo" ni "Estás". Esto significa que cogerá símbolos hasta que se encuentre un 'Yo" o un "Estás". Se podría pensar en usar == en vez del método **equals**( ), pero eso no funcionaría correctamente, pues == compara los valores de las referencias, mientras que **equals**( ) compara contenidos.

La lógica del resto del método **analizar**( ) es que el patrón que se está buscando es "Yo estoy triste", "Yo no estoy contento" o "¿Tú estás triste? Sin la sentencia **break,** el código habría sido aún más complicado de lo que ya es. Habría que ser conscientes de que un analizador típico (éste es un ejemplo primitivo de uno) normalmente tiene una tabla de estos símbolos y un fragmento de código que se mueve a través de los estados de la tabla a medida que se leen los nuevos símbolos.

Debería pensarse que **Stringokenizer** sólo es un atajo para un tipo simple y específico de **StrearnTokenizer.** Sin embargo, si se tiene un **String** en el que se desean identificar símbolos, y **StringTokenizer** es demasiado limitado, todo lo que hay que hacer es convertirlo en un **stream** con **StringBufferInputStream** y después usarlo para crear **StrearnTokenizer** mucho más potente.

Comprobar el estilo de escritura de mayúsculas

En esta sección, echaremos un vistazo a un ejemplo más completo del uso de la E/S de Java, que también hace uso de la identificación de símbolos. Este proyecto es directamente útil pues lleva a cabo una comprobación de estilo para asegurarse que el uso de mayúsculas se adecua al estilo de Java, tal y como se relata en *http://java.sun.com/docs/codeconv/index.html.* Abre cada archivo **.java** del directorio actual y extrae todos los nombres de archivos e identificadores, para mostrar después las que no utilicen el estilo Java.

Para que el programa funcione correctamente, hay que construir, en primer lugar, un repositorio de nombres de clases para guardar todos los nombres de clases de la biblioteca estándar de Java. Esto se logra moviendo todos los subdirectorios de código fuente de la biblioteca estándar de Java y ejecutando **ExploradorClases** en cada subdirectorio. Deben proporcionarse como argumentos el nombre del repositorio (usando la misma trayectoria y nombre siempre) y la opción de línea de comandos **-a** para indicar que deberían añadirse los nombres de clases al repositorio.

Al usar el programa para que compruebe el código de cada uno, hay que pasarle la trayectoria y el nombre del repositorio que debe usar. Comprobará todas las clases e identificadores en el directorio actual y dirá cuáles no siguen el ejemplo de uso de mayúsculas típico de Java.

Uno debería ser consciente de que el programa no es perfecto; pocas veces señalará algo que piensa que es un problema, pero que mirando al código se comprobará que no hay que cambiar nada.

Esto es un poco confuso, pero sigue siendo más sencillo que intentar encontrar todos estos casos simplemente mirando cuidadosamente al código.

```
//: c11:ClassScanner.java
// Scans all files in directory for classes
// and identifiers, to check capitalization.
// Assumes properly compiling code listings.
// Doesn't do everything right, but is a
// useful aid.
import java.io.*;
import java.util.*;

class MultiStringMap extends HashMap {
  public void add(String key, String value) {
    if(!containsKey(key))
      put(key, new ArrayList());
    ((ArrayList)get(key)).add(value);
  }
  public ArrayList getArrayList(String key) {
    if(!containsKey(key)) {
      System.err.println(
        "ERROR: can't find key: " + key);
      System.exit(1);
    }
    return (ArrayList)get(key);
  }
  public void printValues(PrintStream p) {
    Iterator k = keySet().iterator();
    while(k.hasNext()) {
      String oneKey = (String)k.next();
      ArrayList val = getArrayList(oneKey);
      for(int i = 0; i < val.size(); i++)
        p.println((String)val.get(i));
    }
  }
}

public class ClassScanner {
  private File path;
  private String[] fileList;
  private Properties classes = new Properties();
  private MultiStringMap
    classMap = new MultiStringMap(),
    identMap = new MultiStringMap();
```

```
  private StreamTokenizer in;
  public ClassScanner() throws IOException {
    path = new File(".");
    fileList = path.list(new JavaFilter());
    for(int i = 0; i < fileList.length; i++) {
      System.out.println(fileList[i]);
      try {
        scanListing(fileList[i]);
      } catch(FileNotFoundException e) {
        System.err.println("Could not open " +
          fileList[i]);
      }
    }
  }
  void scanListing(String fname)
  throws IOException {
    in = new StreamTokenizer(
        new BufferedReader(
          new FileReader(fname)));
    // Doesn't seem to work:
    // in.slashStarComments(true);
    // in.slashSlashComments(true);
    in.ordinaryChar('/');
    in.ordinaryChar('.');
    in.wordChars('_', '_');
    in.eolIsSignificant(true);
    while(in.nextToken() !=
          StreamTokenizer.TT_EOF) {
      if(in.ttype == '/')
        eatComments();
      else if(in.ttype ==
              StreamTokenizer.TT_WORD) {
        if(in.sval.equals("class") ||
           in.sval.equals("interface")) {
          // Get class name:
             while(in.nextToken() !=
                   StreamTokenizer.TT_EOF
                   && in.ttype !=
                   StreamTokenizer.TT_WORD)
               ;
             classes.put(in.sval, in.sval);
             classMap.add(fname, in.sval);
        }
        if(in.sval.equals("import") ||
           in.sval.equals("package"))
          discardLine();
        else // It's an identifier or keyword
          identMap.add(fname, in.sval);
      }
    }
```

```
  }
  void discardLine() throws IOException {
    while(in.nextToken() !=
          StreamTokenizer.TT_EOF
          && in.ttype !=
          StreamTokenizer.TT_EOL)
      ; // Throw away tokens to end of line
  }
  // StreamTokenizer's comment removal seemed
  // to be broken. This extracts them:
  void eatComments() throws IOException {
    if(in.nextToken() !=
       StreamTokenizer.TT_EOF) {
      if(in.ttype == '/')
        discardLine();
      else if(in.ttype != '*')
        in.pushBack();
      else
        while(true) {
          if(in.nextToken() ==
            StreamTokenizer.TT_EOF)
            break;
          if(in.ttype == '*')
            if(in.nextToken() !=
              StreamTokenizer.TT_EOF
              && in.ttype == '/')
              break;
        }
    }
  }
  public String[] classNames() {
    String[] result = new String[classes.size()];
    Iterator e = classes.keySet().iterator();
    int i = 0;
    while(e.hasNext())
      result[i++] = (String)e.next();
    return result;
  }
  public void checkClassNames() {
    Iterator files = classMap.keySet().iterator();
    while(files.hasNext()) {
      String file = (String)files.next();
      ArrayList cls = classMap.getArrayList(file);
      for(int i = 0; i < cls.size(); i++) {
        String className = (String)cls.get(i);
        if(Character.isLowerCase(
             className.charAt(0)))
          System.out.println(
            "class capitalization error, file: "
            + file + ", class: "
```

```
            + className);
      }
    }
  }
  public void checkIdentNames() {
    Iterator files = identMap.keySet().iterator();
    ArrayList reportSet = new ArrayList();
    while(files.hasNext()) {
      String file = (String)files.next();
      ArrayList ids = identMap.getArrayList(file);
      for(int i = 0; i < ids.size(); i++) {
        String id = (String)ids.get(i);
        if(!classes.contains(id)) {
          // Ignore identifiers of length 3 or
          // longer that are all uppercase
          // (probably static final values):
          if(id.length() >= 3 &&
             id.equals(
               id.toUpperCase()))
            continue;
          // Check to see if first char is upper:
          if(Character.isUpperCase(id.charAt(0))){
            if(reportSet.indexOf(file + id)
                == -1){ // Not reported yet
              reportSet.add(file + id);
              System.out.println(
                "Ident capitalization error in:"
                + file + ", ident: " + id);
            }
          }
        }
      }
    }
  }
  static final String usage =
    "Usage: \n" +
    "ClassScanner classnames -a\n" +
    "\tAdds all the class names in this \n" +
    "\tdirectory to the repository file \n" +
    "\tcalled 'classnames'\n" +
    "ClassScanner classnames\n" +
    "\tChecks all the java files in this \n" +
    "\tdirectory for capitalization errors, \n" +
    "\tusing the repository file 'classnames'";
  private static void usage() {
    System.err.println(usage);
    System.exit(1);
  }
  public static void main(String[] args)
  throws IOException {
```

```
    if(args.length < 1 || args.length > 2)
      usage();
    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
      try {
        // Try to open an existing
        // properties file:
        InputStream oldlist =
          new BufferedInputStream(
            new FileInputStream(old));
        c.classes.load(oldlist);
        oldlist.close();
      } catch(IOException e) {
        System.err.println("Could not open "
          + old + " for reading");
        System.exit(1);
      }
    }
    if(args.length == 1) {
      c.checkClassNames();
      c.checkIdentNames();
    }
    // Write the class names to a repository:
    if(args.length == 2) {
      if(!args[1].equals("-a"))
        usage();
      try {
        BufferedOutputStream out =
          new BufferedOutputStream(
            new FileOutputStream(args[0]));
        c.classes.store(out,
          "Classes found by ClassScanner.java");
        out.close();
      } catch(IOException e) {
        System.err.println(
          "Could not write " + args[0]);
        System.exit(1);
      }
    }
  }
}

class JavaFilter implements FilenameFilter {
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

La clase **MapaMultiCadena** es una herramienta que permite establecer una correspondencia entre un grupo de cadenas de caracteres y su clave. Usa un **HashMap** (esta vez con herencia) con la clave como única cadena de caracteres con correspondencias sobre **ArrayList.** El método **add**( ) simplemente comprueba si ya hay una clave en el **HashMap,** y si no, pone una. El método **getArrayList**( ) **produce un ArrayList para una clave en particular, y printValues**( ), que *es* especialmente útil para depuración, imprime todos los valores de **ArrayList** en **ArrayList.**

Para que todo sea sencillo, se ponen todos los nombres de la biblioteca estándar de Java en un objeto **Properties** (de la biblioteca estándar de Java). Recuérdese que un objeto **Properties** es un **HashMap** que sólo guarda objetos **String,** tanto para las entradas de clave como para la de valor.

Sin embargo, se puede salvar y restaurar a disco con una única llamada a un método, por lo que es ideal para un repositorio de nombres. De hecho, sólo necesitamos una lista de nombres, y un **HashMap** no puede aceptar **null,** ni para su entrada clave, ni para su entrada valor. Por tanto, se usará el mismo objeto tanto para la clave como para valor.

Para las clases e identificadores que se descubran para los archivos en un directorio en particular, se usan dos **MultiStringMaps: mapaclases** y **mapaIdent.** Además, cuando el programa empieza carga el repositorio de nombres de clase estándares en el objeto **Properties** llamado **clases,** y cuando se encuentra un nuevo nombre de clase en el directorio local se añade tanto a **clases** como a **mapaclases.** De esta forma, puede usarse **mapaclases** para recorrer todas las clases en el directorio local, y puede usarse **clases** para ver si el símbolo actual es un nombre de clase (que indica que comienza una definición de un objeto o un método).

El constructor por defecto de **ExploradorClases** crea una lista de nombres de archivo usando la implementación **Filtrdava** de **FilenameFilter,** mostrada al final del archivo. Después llama a
explorarlistado( ) para cada nombre de archivo.

Dentro de **explorarlistado**( ) se abre el código fuente y se convierte en **StreamTokenizer.** En la documentación, se supone que pasar **true** a **slashStarCornments**( ) y **slashSlashComrnents**( ) retira estos comentarios, pero parece un poco fraudulento, pues no funciona muy bien. En vez de ello, las líneas se marcan como comentarios que son extraídos por otro método. Para hacer esto, el "/" debe capturarse como un carácter normal, en vez de dejar a **StreamTokenizer** que lo absorba como parte de un comentario, y el método **ordinaryChar**( ) dice al **StreamTokenizer** que lo haga así.

Esto también funciona para los puntos ("."), puesto que se desea retirar las llamadas a métodos en forma de identificadores individuales. Sin embargo, el guión bajo, que suele ser tratado por **StreamTokenizer** como un carácter individual, debería dejarse como parte de los identificadores, pues aparece en valores **static final,** como **TT_EOF,** etc., usados en este mismo programa. El método **wordChars**( ) toma un rango de caracteres que se desee añadir a los ya dejados dentro del símbolo a analizar como palabras. Finalmente, al analizar comentarios de una línea o descartar una 1línea, hay que saber cuándo se produce un fin de línea **[4]**, por lo que se llama a **eolIsSignificant**(**true)** que mostrará los finales de línea en vez de dejar que sean absorbidos por el **StreamTokenizer.**

[4] N. del traductor: en inglés End ***Of Line*** o EOL.

El resto de **explorarlistado**( ) lee y vuelve a actuar sobre los símbolos hasta el fin de fichero, que se encuentra cuando **nextToken**( ) devuelva el valor **final static StreamTokenizer.TT_EOF.**

Si el símbolo es un "/" es potencialmente un comentario, por lo que se llama a **comerComentarios**( ) para que lo maneje. Únicamente la otra situación que nos interesa en este caso es si es una palabra, donde pueden presentarse varios casos.

Si la palabra es **class** o **interfaz,** el siguiente símbolo representa un nombre de clase o interfaz, y se introduce en **clases** y **mapaclases.** Si la palabra es **import** o **package,** entonces no se desea el resto de la línea. Cualquier otra cosa debe ser un identificador (que nos interesa) o una palabra clave (que no nos interesa, pero que en cualquier caso se escriben con minúsculas, por lo que no pasa nada por incluirlas). Éstas se añaden a **mapaIdent.** El método **descartarLínea**( ) es una simple herramienta que busca finales de línea. Nótese que cada vez que se encuentre un nuevo símbolo, hay que comprobar los finales de línea.

El método **comerComentarios**( ) es invocado siempre que se encuentra un "/" en el bucle de análisis principal. Sin embargo, eso no quiere decir necesariamente que se haya encontrado un comentario, por lo que hay que extraer el siguiente comentario para ver si hay otra barra diagonal (en cuyo caso se descarta toda la línea) o un asterisco. Si no estamos ante ninguna de éstas, ¡hay que volver a insertar el símbolo que se acaba de extraer! Afortunadamente, el método **pushBack**( ) permite volver a introducir el símbolo actual en el flujo de entrada, de forma que cuando el bucle de análisis principal llame a **nextToken**( ), se obtendrá el que se acaba de introducir.

Por conveniencia, el método **nombresClases**( ) produce un array de todos los nombres del contenedor **clases**. Este método no se usa en el programa pero es útil en procesos de depuración. Los dos siguientes métodos son precisamente aquéllos en los que de hecho se realiza la comprobación. En

comprobarNombresClases( ), se extraen los nombres de clase de **mapaclases** (que, recuérdese, contiene sólo los nombres de este directorio, organizados por nombre de archivo, de forma que se puede imprimir el nombre de archivo junto con el nombre de clase errante). Esto se logra extrayendo cada **ArrayList** asociado y recorriéndolo, tratando de ver si el primer carácter está en minúsculas. Si es así, se imprimirá el pertinente mensaje de error.

En **comprobarNombresIdent**( ), se sigue un enfoque similar: se extrae cada nombre de identificador de **mapaIdent.** Si el nombre no está en la lista **clases,** se asume que es un identificador o una palabra clave. Se comprueba un caso especial: si la longitud del identificador es tres o más y todos sus caracteres son mayúsculas, se ignora el identificador pues es probablemente un valor **static final** como **TT_EOF.** Por supuesto, éste no es un algoritmo perfecto, pero asume que general- mente los identificadores formados exclusivamente por letras mayúsculas se pueden ignorar.

En vez de informar de todos los identificadores que empiecen con una mayúscula, este método mantiene un seguimiento de aquéllos para los que ya se ha generado un informe en un **ArrayList** denominado **conjuntoInformes**( ). Éste trata al **ArrayList** como un "conjunto" que indica si un elemento se encuentra o no en el conjunto. El elemento se genera concatenando el nombre de archivo y el identificador. Si el elemento no está en el conjunto, se añade y después se emite el informe.

El resto del listado se lleva a cabo en el método **main**( ), que se mantiene ocupado manejando los parámetros de línea de comandos y averiguando si se está o no construyendo un repositorio de nombres de clase a partir de la biblioteca estándar de Java o comprobando la validez del código escrito. En ambos casos hace un objeto **ExploradorClase.**

Se esté construyendo un repositorio o utilizando uno, hay que intentar abrir el repositorio existente. Haciendo un objeto **File** y comprobando su existencia, se puede decidir si abrir un archivo y **load**( ) las **clases** de la lista de **Properties** dentro de **ExploradorClase.** (Las clases del repositorio se añaden, más que sobrescribirse, a las clases encontradas en el constructor **ExploradorClase**) Si se proporciona sólo un parámetro en línea de comandos, se quiere llevar a cabo una comprobación de nombres de clase e identificadores, pero si se proporcionan dos argumentos (siendo el segundo **"-a")** se está construyendo un repositorio de nombres de clase. En este caso, se abre un archivo de salida y se usa el método **Properties.save**( ) para escribir la lista en un archivo, junto con una cadena de caracteres que proporciona información de cabecera de archivo.

Resumen

La biblioteca de flujos de E/S de Java satisface los requisitos básicos: se puede llevar a cabo lectura y escritura con la consola, un archivo, un bloque de memoria o incluso a través de Internet (como se verá en el Capítulo 17). Con la herencia, se pueden crear nuevos tipos de objetos de entrada y salida. E incluso se puede añadir una extensibilidad simple a los tipos de objetos que aceptará un flujo redefiniendo el método **toString**( ) que se invoca automáticamente cuando se pasa un objeto a un método que esté esperando un **String** (la "conversión automática de tipos" limitada de Java).

En la documentación y diseño de la biblioteca de flujos de E/S quedan cuestiones sin contestar. Por ejemplo, habría sido genial si se pudiese decir que se desea que se lance una excepción si se intenta sobrescribir un archivo cuando se abre como salida -algunos sistemas de programación permiten especificar que se desea abrir un archivo de salida, pero sólo si no existe aún. En Java, parece suponerse que uno usará un objeto **File** para determinar si existe un archivo, porque si se abre como un **FileOutputStream** o **FileWriter** siempre será sobrescrito.

La biblioteca de flujos de E/S trae a la mente sentimientos entremezclados; hace gran parte del trabajo y es portable. Pero si no se entiende ya el patrón decorador, el diseño no es intuitivo, por lo que hay una gran sobrecarga en lo que a aprendizaje y enseñanza de la misma se refiere. También está incompleta: no hay soporte para dar formato a la salida, soportado casi por el resto de paquetes de E/S del resto de lenguajes.

Sin embargo, una vez que se entiende el patrón decorador, y se empieza a usar la biblioteca en situaciones que requieren su flexibilidad, se puede empezar a beneficiar de este diseño, punto en el que su coste en líneas extra puede no molestar tanto.

Si no se encuentra lo que se estaba buscando en este capítulo (que no ha sido más que una introducción, que no pretendía ser comprensivo) se puede encontrar información en profundidad en *Java I/O,* de Elliote Rusty Harold (O'Reilly, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide,* disponible a bajo coste en *http://www.BruceEckel.com*.

1. Abrir un archivo de texto de forma que se pueda leer del mismo de línea en línea. Leer cada línea como un **String** y ubicar ese objeto **String** en un **LinkedList.** Imprimir todas las líneas del **LinkedList** en orden inverso.

2. Modificar el Ejercicio 1 de forma que el nombre del archivo que se lea sea proporcionado como un parámetro de línea de comandos.

3. Modificar el Ejercicio 2 para que también abra un archivo de texto de forma que se pueda escribir texto en el mismo. Escribir las líneas del **ArrayList,** junto con los números de línea (no intentar usar las clases "LineNumber"), fuera del archivo.

4. Modificar el Ejercicio 2 para forzar que todas las líneas de **ArrayList** estén en mayúsculas y enviar los resultados a **System.out.**

5. Modificar el Ejercicio 2 para que tome palabras a buscar dentro del archivo como parámetros adicionales de línea de comandos. Imprimir todas las líneas en las que casen las palabras.

6. Modificar **ListadoDirectorio.java** de forma que **FilenameFilter** abra cada archivo y acepte el archivo basado en la existencia de alguno de los parámetros de la línea de comandos en ese archivo.

7. Crear una clase denominada **ListadoDirectorioOrdenado** con un constructor que tome información de una ruta de archivo y construya un listado de directorio ordenado con todos los archivos de esa ruta. Crear dos métodos **listar**( ) sobrecargados que, bien produzcan toda la lista, o bien un subconjunto de la misma basándose en un argumento. Añadir un método **tamaño**( ) que tome un nombre de archivo y produzca el tamaño de ese archivo.

8. Modificar **RecuentoPalabra.java** para que produzca un orden alfabético en su lugar, utilizando la herramienta del Capítulo 10.

9. Modificar **RecuentoPalabra.java** de forma que use una clase que contenga un **String** y un valor de cuenta para almacenar cada palabra, y un **Set** de esos objetos para mantener la lista de palabras.

10. Modificar **DemoFlujoES.java** de forma que use **LineNumberInputStream** para hacer un seguimiento del recuento de líneas. Nótese que es mucho más fácil mantener el seguimiento desde la programación.

11. Basándonos en la Sección 4 de **DemoFlujoES.java,** escribir un programa que compare el rendimiento de escribir en un archivo al usar E/S con y sin espacios de almacenamiento intermedio.

12. Modificar la Sección 5 de **DemoFlujoES.java** para eliminar los espacios en la línea producida por la primera llamada a **entrada5br.readline**( ). Hacerlo utilizando un bucle **while** y **readChar**( ).

13. Reparar el programa **EstadoCAD.java** tal y como se describe en el texto.

14. En **Rastros.java,** copiar el archivo y renornbrarlo a **ComprobarRastro.java,** y renombrar la clase **Rastro2** a **ComprobarRastro** (haciéndola además **public** y retirando el ámbito público de la clase **Rastros** en el proceso). Eliminar las marcas **//!**

del final del archivo y ejecutar el programa incluyendo las líneas que causaban ofensa. A continuación, marcar como comentario el constructor por defecto de **ComprobarRastro.** Ejecutarlo y explicar por qué funciona. Nótese que tras compilar, hay que ejecutar el programa con **"java Rastros"** porque el método **main**( ) sigue en la clase **Rastros.**

15. En **RastroS.java,** marcar como comentarios las dos líneas tras las frases "Hay que hacer esto:" y ejecutar el programa. Explicar el resultado y por qué difiere de cuando las dos líneas se encuentran en el programa.

16. (Intermedio) En el Capítulo 8, localizar el ejemplo **ControlesCasaVerde.java,** que consiste en tres archivos. En **ControlesCasaVerde.java,** la clase interna **Rearrancar**( ) tiene un conjunto de eventos duramente codificados. Cambiar el programa de forma que lea los eventos y sus horas relativas desde un archivo de texto. (Desafío: utilizar un *método factoría* de patrones de diseño para construir los eventos - véase *Thinking in Patterns with Java,* descargable desde *http://www.BruceEckel.com*.)

13: Concurrencia

Los objetos proporcionan una forma de dividir un programa en secciones independientes. A menudo, también es necesario convertir un programa en subtareas separadas que se ejecuten independientemente.

Cada una de estas subtareas independientes recibe el nombre de hilo, y uno programa como si cada hilo se ejecutara por sí mismo y tuviera la UCP para él sólo. Algún mecanismo subyacente divide de hecho el tiempo de UCP entre ellos, pero generalmente el programador no tiene por qué pensar en ello, lo que hace de la programación de hilos múltiples una tarea mucho más sencilla.

Un proceso es un programa en ejecución autocontenido con su propio espacio de direcciones. Un sistema operativo multitarea es capaz de ejecutar más de un proceso (programa) a la vez, mientras hace que parezca como si cada uno fuera el único que se está ejecutando, proporcionándole ciclos de UCP periódicamente. Por consiguiente, un único proceso puede tener múltiples hilos ejecutándose concurrentemente.

Hay muchos usos posibles del multihilo, pero, en general, se tendrá parte del programa vinculado a un evento o recurso particular, no deseando que el resto del programa pueda verse afectado por esta vinculación. Por tanto, se crea un hilo asociado a ese evento o tarea y se deja que se ejecute independientemente del programa principal. Un buen ejemplo es un botón de "salir" -no hay por qué verse obligado a probar el botón de salir en todos los fragmentos de código que se escriban en el programa, aunque sí se desea que el botón de salir responda, como si se estuviera comprobando regularmente. De hecho, una de las razones más importantes para la existencia del multihilo es la existencia de interfaces de usuario que respondan rápidamente.

Interfaces de respuesta de usuario rápida

Como punto de partida, puede considerarse un programa que lleva a cabo alguna operación intensa de UCP y que acaba ignorando la entrada de usuario y por tanto no emite respuestas. Éste, un applet/aplicación, simplemente mostrará el resultado de un contador en ejecución:

```
//: c14:Counter1.java
// A non-responsive user interface.
// <applet code=Counter1 width=300 height=100>
// </applet>
```

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Counter1 extends JApplet {
  private int count = 0;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  private boolean runFlag = true;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public void go() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if (runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter1(), 300, 100);
  }
} ///:~
```

En este punto, el código del *applet* y Swing deberían ser racionalmente familiares, al estar explicados en el Capítulo 13. En el método **comenzar**( ) es donde

permanece ocupado el programa: pone el valor actual de conteo en el **JTextFieldt**, y después incrementa conteo.

Parte del bucle infinito interno a **comenzar**( ) llama a **sleep**( ). Éste debe estar asociado con un objeto **Thread**, y resulta que toda aplicación tiene algún hilo asociado a él. (De hecho, Java se basa en hilos y siempre hay alguna ejecución junto con la aplicación.) Por tanto, independientemente de si se usan o no hilos de forma explícita, se puede producir el hilo actual que usa el programa con **Hilos** y el método **static sleep**( ).

Nótese que **sleep**( ) puede lanzar una **InterruptedException**, aunque lanzar esta excepción se considera una forma hostil de romper un hilo, por lo que no se recomienda. (Una vez más, las excepciones son para condiciones excepcionales, no el flujo normal de control.) La capacidad de interrumpir a un hilo durmiente se ha incluido para soportar una faceta futura del lenguaje.

Cuando se presiona el botón Empezar, se invoca a **comenzar**( ). Al examinar **comenzar**( ), se podría pensar estúpidamente (como hicimos) que debería permitir el multihilo porque se va a dormir. Es decir, mientras que el método está dormido, parece como si la UCP pudiera estar ocupada monitorizando otras presiones sobre el botón. Pero resulta que el problema real es que **comenzar**( ) nunca devuelve nada, puesto que está en un bucle infinito, y esto significa que **actionPerformed**( ) nunca devuelve nada. Puesto que uno está enclavado en **actionPerformed**( ) debido a la primera vez que se presionó el botón, el programa no puede gestionar ningún otro evento. (Para salir, de alguna forma hay que matar el proceso; la forma más sencilla de hacerlo es presionar Control-C en la ventana de la consola, si es que se lanzó desde la consola. Si se empieza vía el navegador, hay que matar la ventana del navegador.)

El problema básico aquí es que **comenzar**( ) necesita continuar llevando a cabo sus operaciones, y al mismo tiempo necesita devolver algo, de forma que **actionPerformed**( ) pueda completar su operación y la interfaz de usuario continúe respondiendo al usuario. Pero en un método convencional como **comenzar**( ) no puede continuar y al mismo tiempo devolver el control al resto del programa. Esto suena a imposible de lograr, como si la UCP debiera estar en dos lugares a la vez, pero ésta es precisamente la ilusión que proporciona el multihilo.

El modelo de hilos (y su soporte de programación en Java) es una conveniencia de programación para simplificar estos juegos malabares y operaciones que se dan simultáneamente en un único programa. Con los hilos, la UCP puede ir rotando y dar a cada hilo parte de su tiempo. Cada hilo tiene impresión de tener la UCP para sí mismo durante todo el tiempo. La excepción se da siempre que el programa se ejecute en múltiples UCP Pero uno de los aspectos más importantes de los hilos es que uno se abstrae de esta capa, de forma que el código no tiene

por qué saber si se está ejecutando en una o en varias UCP Por consiguiente, los hilos son una forma de crear programas transparentemente escalables.

Los hilos pueden reducir algo la eficiencia de computación, pero la mejora en el diseño de programa, el balanceo de recursos y la conveniencia del usuario suelen ser muy valiosos. Por supuesto, si se tiene más de una UCP, el sistema operativo puede dedicar cada UCP a un conjunto de hilos o incluso a un único hilo, logrando que el programa, en su globalidad, se ejecute mucho más rápido. La multitarea y el multihilo tienden a ser las formas más razonables de usar sistemas multiprocesador.

Heredar de **Thread**

La forma más simple de crear un hilo es heredar de la clase **Thread,** que tiene todo lo necesario para crear y ejecutar hilos. El método más importante de **Thread** es **run**( ), que debe ser sobrescrito para hacer que el hilo haga lo que se le mande. Por consiguiente, **run**( ) es el código que se ejecutará "simultáneamente" con los otros hilos del programa.

El ejemplo siguiente crea cualquier número de hilos de los que realiza un seguimiento asignando a cada uno con un único número, generado con una variable **static.** El método **run**( ) de **Thread** se sobrescribe para que disminuya cada vez que pase por el bucle y acabe cuando valga cero (en el momento en que acabe **run**( ), se termina el hilo).

```
//: c14:SimpleThread.java
// Very simple Threading example.

public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread() {
    System.out.println("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.out.println("Thread " +
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread().start();
    System.out.println("All Threads Started");
  }
```

```
} ///:~
```

Un método **run**( ) suele tener siempre algún tipo de bucle que continúa hasta que el hilo deja de ser necesario, por lo que hay que establecer la condición en la que romper el bucle y salir (o, en el caso de arriba, simplemente **return** de **run**( )). **A** menudo, se convierte **run**( ) en forma de bucle infinito, lo que significa que a falta de algún factor externo que haga que **run**( ) termine, continuará para siempre.

En el método **main**( ) se puede ver el número de hilos que se están creando y ejecutando. El método **start**( ) de la clase **Thread** lleva a cabo alguna inicialización especial para el hilo y después llama a **run**( ). Por tanto, los pasos son: se llama al constructor para que construya el objeto, después **start**( ) configura el hilo y llama a **run**( ). Si no se llama a **start**( ) (lo que puede hacerse en el constructor si es apropiado) nunca se dará comienzo al hilo.

La salida de una ejecución de este programa (que será distinta cada vez) es:

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

Se verá que en este ejemplo no se llama nunca a **sleep**( ), y la salida sigue indicando que cada hilo obtiene una porción del tiempo de UCP en el que ejecutarse. Esto muestra que **sleep**( ), aunque descansa en la existencia de un hilo para poder ejecutarse, no está involucrado en la habilitación o deshabilitación de hilos. Es simplemente otro método.

También puede verse que los hilos no se ejecutan en el orden en el que se crean. De hecho, el orden en que la UCP atiende a un conjunto de hilos existente es indeterminado, a menos que se cambien las prioridades haciendo uso del método **setPriority**( ) de **Thread.**

Cuando **main**( ) crea los objetos **Thread** no captura las referencias a ninguno de ellos. Un objeto ordinario debería ser un juego justo para la recolección de basura, pero no un **Thread.**

Cada **Thread** "se registra" a sí mismo de forma que haya una referencia al mismo en algún lugar y el recolector de basura no pueda limpiarlo.

Hilos para una interfaz con respuesta rápida

Ahora es posible solucionar el problema de **Contadorl.java** con un hilo. El truco es colocar la subtarea -es decir, el bucle de dentro de **comenzar**( )- dentro del método **run**( ) de un hilo. Cuando el usuario presione el botón **empezar,** se arranca el hilo, pero después se completa la creación del hilo, por lo que aunque se esté ejecutando el hilo, puede continuar el trabajo principal del programa (estando pendiente y respondiendo a eventos de la interfaz de usuario). He aquí la solución:

```
//: c14:Counter2.java
// A responsive user interface with threads.
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter2 extends JApplet {
  private class SeparateSubTask extends Thread {
    private int count = 0;
    private boolean runFlag = true;
    SeparateSubTask() { start(); }
    void invertFlag() { runFlag = !runFlag; }
    public void run() {
      while (true) {
       try {
```

```
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
         if(runFlag)
           t.setText(Integer.toString(count++));
        }
      }
    }
    private SeparateSubTask sp = null;
    private JTextField t = new JTextField(10);
    private JButton
      start = new JButton("Start"),
      onOff = new JButton("Toggle");
    class StartL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(sp == null)
          sp = new SeparateSubTask();
      }
    }
    class OnOffL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if(sp != null)
          sp.invertFlag();
      }
    }
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(t);
      start.addActionListener(new StartL());
      cp.add(start);
      onOff.addActionListener(new OnOffL());
      cp.add(onOff);
    }
    public static void main(String[] args) {
      Console.run(new Counter2 (), 300, 100);
    }
} ///:~
```

Contador2 es un programa directo, cuya única tarea es establecer y mantener la interfaz de usuario. Pero ahora, cuando el usuario presiona el botón **empezar,** el código de gestión de eventos no llama a un método. En su lugar, se crea un hilo de clase **SubTareaSeparada,** y continúa el bucle de eventos de **Counter2.**

La clase **SubTareaSeparada** es una simple extensión de **Thread** con un constructor que ejecuta el hilo invocando a **start**( ), y un método **run**( ) que esencialmente contiene el código "**comenzar**( )" de **Contador1 .java.**

Dado que **SubtareaSeparada** es una clase interna, puede acceder directamente al **JTextFie ld t** de **Contador2;** se puede ver que esto ocurre dentro de **run**( ). El campo **t** de la clase externa es **private** puesto que **SubTareaSeparada** puede acceder a él sin ningún permiso especial -y siempre es bueno hacer campos "tan **private** como sea posible", de forma que puedan ser cambiados accidentalmente por fuerzas externas a la clase.

Cuando se presiona el botón **onOff** conmuta el **flagEjecutar** de dentro del objeto **SubTareaSeparada.** Ese hilo (cuando mira al *flag)* puede empezar *y* pararse por sí mismo. Presionar el botón **onOff** produce una respuesta aparentemente instantánea. Por supuesto, la respuesta no es verdaderamente instantánea, no como la de un sistema dirigido por interrupciones. El contador sólo se detiene cuando el hilo tiene la UCP y se da cuenta de que el *flag* ha cambiado.

Se puede ver que la clase interna **SubTareaSeparada** es **private**, lo que significa que sus campos y métodos pueden tener el acceso por defecto (excepto en el caso de **run**( ), que debe ser **public**, puesto que es **public** en la clase base). La clase **private** interna no está accesible más que a **Contador2**, y ambas clases están fuertemente acopladas. En cualquier momento en que dos clases parezcan estar fuertemente acopladas entre sí, hay que considerar las mejoras de codificación y mantenimiento que se obtendrían utilizando clases internas.

Combinar el hilo con la clase principal

En el ejemplo de arriba puede verse que la clase hilo está separada de la clase principal del programa. Esto tiene mucho sentido y es relativamente fácil de entender. Sin embargo, hay una forma alternativa que se verá a menudo que no está tan clara, pero que suele ser más concisa (y que es probablemente lo que la dota de popularidad). Esta forma combina la clase principal del programa con la clase hilo haciendo que la clase principal del programa sea un hilo. Puesto que para un programa IGU la clase principal del programa debe heredarse de **Frame** o de **Applet**, hay que usar una interfaz para añadirle funcionalidad adicional. A esta interfaz se le denomina **Runnable**, y contiene el mismo método básico que **Thread**. De hecho, **Thread** también implementa **Runnable**, lo que sólo especifica la existencia de un método **run**( ).

El uso de programa/hilo combinado no es tan obvio. Cuando empieza el programa se crea un objeto que es **Runnable**, pero no se arranca el hilo. Esto hay que hacerlo explícitamente. Esto se puede ver en el programa siguiente, que reproduce la funcionalidad de **Contador2**:

```
//: c14:Counter3.java
// Using the Runnable interface to turn the
// main class into a thread.
// <applet code=Counter3 width=300 height=100>
// </applet>
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter3
    extends JApplet implements Runnable {
  private int count = 0;
  private boolean runFlag = true;
  private Thread selfThread = null;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  public void run() {
    while (true) {
      try {
        selfThread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if(runFlag)
        t.setText(Integer.toString(count++));
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(selfThread == null) {
        selfThread = new Thread(Counter3.this);
        selfThread.start();
      }
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
  public static void main(String[] args) {
    Console.run(new Counter3(), 300, 100);
  }
```

```
} ///:~
```

Ahora el **run**( ) está dentro de la clase, pero sigue estando dormido tras completarse **init**( ). Cuando se presiona el botón **Empezar,** se crea el hilo (si no existe ya) en una expresión bastante oscura:

```
new Thread(Counter3.this);
```

Cuando algo tiene una interfaz **Runnable,** simplemente significa que tiene un método **run**( ), pero no hay nada especial en ello -no produce ninguna habilidad innata a los hilos, como las de una clase heredada de **Thread.** Por tanto, para producir un hilo a partir de un objeto **Runnable,** hay que crear un objeto **Thread** separado como se mostró arriba, pasándole el objeto **Runnable** al constructor **Thread** especial. Después se puede llamar al **start**( ) de ese hilo:

```
selfThread.start();
```

Esta sentencia lleva a cabo la inicialización habitual y después llama a **run**( ).

El aspecto conveniente de la **interface Runnable** es que todo pertenece a la misma clase. Si es necesario acceder a algo, simplemente se hace sin recorrer un objeto separado. Sin embargo, como se vio en el capítulo anterior, este acceso es tan sencillo como usar una clase interna **[1]**.

[1] Runnable ya estaba en Java 1.0, mientras que las clases internas no se introdujeron hasta Java 1.1, que pueda deberse probablemente a la existencia de **Runnable.** También las arquitecturas multihilo tradicionales se centraron en que se ejecutara una función en vez de un objeto. Preferimos heredar de **Thread** siempre que se pueda; nos parece más claro y más flexible.

Construir muchos hilos

Considérese la creación de muchos hilos distintos. Esto no se puede hacer con el ejemplo de antes, por lo que hay que volver hacia atrás, cuando se tenían clases separadas heredadas de **Thread** para encapsular el método **run**( ). Pero ésta es una solución más general y más fácil de entender, por lo que mientras que el ejemplo anterior muestra un estilo de codificación muy abundante, no podemos recomendarlo para la mayoría de los casos porque es un poco más confuso y menos flexible.

El ejemplo siguiente repite la forma de los ejemplos de arriba con contadores y botones de conmutación. Pero ahora toda la información de un contador particular, incluyendo el botón y el campo de texto, están dentro de su propio objeto, heredado de **Thread**. Todos los campos de **Teletipo** son **private**, lo que significa que se puede cambiar la implementación de Teletipo cuando sea

necesario, incluyendo la cantidad y tipo de componentes de datos a adquirir y la información a mostrar. Cuando se crea un objeto **Teletipo**, el constructor añade sus componentes visuales al panel contenedor del objeto externo:

```
//: c14:Counter4.java
// By keeping your thread as a distinct class,
// you can have as many threads as you want.
// <applet code=Counter4 width=200 height=600>
// <param name=size value="12"></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size = 12;
  class Ticker extends Thread {
    private JButton b = new JButton("Toggle");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = true;
    public Ticker() {
      b.addActionListener(new ToggleL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Calls JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ToggleL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
      }
    }
    public void run() {
      while (true) {
        if (runFlag)
          t.setText(Integer.toString(count++));
        try {
          sleep(100);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
```

```
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Get parameter "size" from Web page:
    if (isApplet) {
      String sz = getParameter("size");
      if(sz != null)
        size = Integer.parseInt(sz);
    }
    s = new Ticker[size];
    for (int i = 0; i < s.length; i++)
      s[i] = new Ticker();
    start.addActionListener(new StartL());
    cp.add(start);
  }
  public static void main(String[] args) {
    Counter4 applet = new Counter4();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    if(args.length != 0)
      applet.size = Integer.parseInt(args[0]);
    Console.run(applet, 200, applet.size * 50);
  }
} ///:~
```

Teletipo no sólo contiene su equipamiento como hilo, sino que también incluye la forma de controlar y mostrar el hilo. Se pueden crear tantos hilos como se desee sin crear explícitamente los componentes de ventanas.

En **Contador4** hay un array de objetos **Teletipo** llamado **s.** Para maximizar la flexibilidad, se inicializa el tamaño de este array saliendo a la página web utilizando los parámetros del applet. Esto es lo que aparenta el parámetro **tamanio** en la página, insertado en la etiqueta applet:

```
<param name=size value="20">
```

Las palabras clave **param, name** y **value** pertenecen a **HTML.** La palabra **name** es aquello a lo que se hará referencia en el programa, y **value** puede ser una cadena de caracteres, no sólo algo que desemboca en un número.

Se verá que la determinación del tamaño del array ***S*** se hace dentro de **init**( ), y no como parte de una definición de ***s.*** Es decir, no ***se puede*** decir como parte de la definición de clase (fuera de todo método):

```
int size = Integer.parseInt(getParameter("size"));
Ticker[] s = new Ticker[size];
```

Esto se puede compilar, pero se obtiene una "null-pointer exception" extraña en tiempo de ejecución. Funciona bien si se mueve la inicialización **getParameter**( ) dentro de **init**( ). El marco de trabajo ***applet*** lleva a cabo la inicialización necesaria en los parámetros antes de **init**().

Además, este código puede ser tanto un ***applet*** como una aplicación. Cuando es una aplicación, se extrae el parámetro **tamanio** de la línea de comandos (o se utiliza un valor por defecto).

Una vez que se establece el tamaño del array, se crean nuevos objetos **Teletipo;** como parte del constructor **Teletipo** se añade al ***applet*** el botón y el campo texto de cada **Teletipo.**

Presionar el botón **empezar** implica recorrer todo el array de **Teletipos** y llamar al método **start**( ) de cada uno. Recuérdese que **start**( ) lleva a cabo la inicialización necesaria por cada hilo, invocando al método **run**( ) del hilo.

El oyente **ConmutadorL** simplemente invierte el ***flag*** de **Teletipo,** de forma que cuando el hilo asociado tome nota, pueda reaccionar de forma acorde.

Uno de los aspectos más valiosos de este ejemplo es que permite crear fácilmente conjuntos grandes de subtareas independientes además de monitorizar su comportamiento. En este caso, se verá que a medida que crece el número de tareas, la máquina mostrará mayor divergencia en los números que muestra debido a la forma de servir esos hilos.

También se puede experimentar para descubrir la importancia de **sleep**(**100)** dentro de **teletipo.run**( ). Si se retira el **sleep**( ), todo funcionará correctamente hasta presionar un botón de conmutar. Después, ese hilo particular tendrá un **flagEjecutar** falso, y el **run**( ) se verá envuelto en un bucle infinito y rígido, que parece difícil de romper, haciendo que el grado de respuesta y la velocidad del programa descienda drásticamente.

Hilos demonio

Un hilo "demonio" es aquél que supuestamente proporciona un servicio general en segundo plano mientras se está ejecutando el programa, no siendo parte de la esencia del programa. Por consiguiente, cuando todos los hilos no demonio acaban, se finaliza el programa. Consecuentemente, mientras se siga ejecutando algún hilo no demonio, el programa no acabará. (Por ejemplo, puede haber un hilo ejecutando el método **main**( ).)

Se puede averiguar si un hilo es un demonio llamando a **isDaemon**( ), y se puede activar o desactivar el funcionamiento como demonio de un hilo con **setDaemon**( ). Si un hilo es un demonio, todos los hilos que cree serán a su vez demonios.

El ejemplo siguiente, demuestra los hilos demonio:

```
//: c14:Daemons.java
// Daemonic behavior.
import java.io.*;

class Daemon extends Thread {
  private static final int SIZE = 10;
  private Thread[] t = new Thread[SIZE];
  public Daemon() {
    setDaemon(true);
    start();
  }
  public void run() {
    for(int i = 0; i < SIZE; i++)
      t[i] = new DaemonSpawn(i);
    for(int i = 0; i < SIZE; i++)
      System.out.println(
        "t[" + i + "].isDaemon() = "
        + t[i].isDaemon());
    while(true)
      yield();
  }
}

class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    System.out.println(
      "DaemonSpawn " + i + " started");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}
```

```
public class Daemons {
  public static void main(String[] args)
  throws IOException {
    Thread d = new Daemon();
    System.out.println(
      "d.isDaemon() = " + d.isDaemon());
    // Allow the daemon threads to
    // finish their startup processes:
    System.out.println("Press any key");
    System.in.read();
  }
} ///:~
```

El hilo **Demonio** pone su hilo a true y después engendra otros muchos hilos para mostrar que son también demonios. Después se introduce en un bucle infinito y llama a **yield**( ) para ceder el control a los otros procesos. En una versión anterior de este programa, los bucles infinitos incrementarían contadores **int,** pero eso parecía bloquear todo el programa. Usar **yield**( ) hace que el ejemplo sea bastante picante.

No hay nada para evitar que el programa termine una vez que acabe el método **main**( ), puesto que no hay nada más que hilos demonio en ejecución. Para poder ver los resultados de lanzar todos los hilos demonio se coloca **System.in** para leer, de forma que el programa espere una pulsación de tecla antes de terminar. Sin esto sólo se ve alguno de los resultados de la creación de los hilos demonio. (Puede probarse a reemplazar el código de **read**( ) con llamadas a **sleep**( ) de varias longitudes y observar el comportamiento.)

Compartir recursos limitados

Se puede pensar que un programa de un hilo es una entidad solitaria que recorre el espacio del problema haciendo sólo una cosa en cada momento. Dado que sólo hay una entidad, no hay que pensar nunca que pueda haber dos entidades intentando usar el mismo recurso a la vez, como si fueran dos conductores intentando aparcar en el mismo sitio, o atravesar la misma puerta simultáneamente, o incluso, hablar.

Con la capacidad multihilo, los elementos dejan de ser solitarios, y ahora existe la posibilidad de que dos o más hilos traten de usar el mismo recurso limitado a la vez. Hay que prevenir las colisiones por un recurso o, de lo contrario, se tendrán dos hilos intentando acceder a la misma cuenta bancaria a la vez, o imprimir en la misma impresora o variar la misma válvula, etc.

Acceder a los recursos de forma inadecuada

Considérese una variación de los contadores que se han venido usando hasta ahora en el capítulo.

En el ejemplo siguiente, cada hilo contiene dos contadores que se incrementan y muestran dentro de **run**( ). Además, hay otro hilo de clase **Observador** que vigila los contadores para que siempre sean equivalentes. Ésta parece una actividad innecesaria, puesto que mirando al código parece obvio que los contadores siempre tendrán el mismo valor. Pero es justamente ahí donde aparece la sorpresa. He aquí la primera versión del programa:

```
//: c14:Sharing1.java
// Problems with resource sharing while threading.
// <applet code=Sharing1 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing1 extends JApplet {
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;
  private TwoCounter[] s;
  class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    // Add the display components as a panel:
    public TwoCounter() {
      JPanel p = new JPanel();
      p.add(t1);
```

```
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!started) {
        started = true;
        super.start();
      }
    }
    public void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
    public void synchTest() {
      Sharing1.incrementAccess();
      if(count1 != count2)
        l.setText("Unsynched");
    }
  }
  class Watcher extends Thread {
    public Watcher() { start(); }
    public void run() {
      while(true) {
        for(int i = 0; i < s.length; i++)
          s[i].synchTest();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
```

```
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new JLabel("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing1 applet = new Sharing1();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
} ///:~
```

Como antes, cada contador sólo contiene sus componentes propios a visualizar: dos campos de texto y una etiqueta que inicialmente indica que los contadores son equivalentes. Estos componentes se añaden al panel de contenidos del objeto de la clase externa en el constructor **DosContadores.**

Dado que el hilo **DosContadores** empieza vía una pulsación de tecla por parte del usuario, es posible que se llame a **start**( ) más de una vez. Es ilegal que se

llame a **Thread.start**( ) más de una vez para un mismo hilo (se lanza una excepción). Se puede ver la maquinaria que evita esto en el flag **empezado** y el método **start**( ) superpuesto.

En **run**( ), se incrementan y muestran **conteol** y **conteo2,** de forma que parecen ser idénticos. Después se llama a **sleep**( ); sin esta llamada el programa se detiene bruscamente porque la UCP tiene dificultad para conmutar las tareas.

El método **pruebaSinc**( ) lleva a cabo la aparentemente inútil actividad de comprobar si **conteol** es equivalente a **conteo2;** si no son equivalentes, pone la etiqueta a "Sin Sincronizar" para indicar esta circunstancia. Pero primero llama a un miembro estático de la clase **Compartiendol,** que incrementa y muestra un contador de accesos para mostrar cuántas veces se ha dado esta comprobación con éxito. (La razón de esto se hará evidente en variaciones ulteriores de este ejemplo.)

La clase **Observador** es un hilo cuyo trabajo es invocar a **pruebaSinc**( ) para todos los objetos de **DosContenedores** activos. Lo hace recorriendo el array mantenido en el objeto **Compartiendol.** Se puede pensar que **Observador** está mirando constantemente por encima del hombro de los objetos **DosContadores.**

Compartiendol contiene un array de objetos **DosContenedores** que inicializa en **init**( ) y comienza como hilos al presionar el botón "empezar". Más adelante, al presionar el botón "Vigilar", se crean uno o más vigilantes que se liberan sobre los hilos **DosContadores.**
Nótese que para ejecutar esto como un *applet* en un navegador, la etiqueta *applet* tendrá que contener las líneas:

```
<param name=size value="20">
<param name=watchers value="1">
```

Se puede experimentar variando la anchura, altura y parámetros para satisfacer los gustos de cada uno. Cambiando el **tamaño** y los **observadores,** se puede variar el comportamiento del programa.

Este programa está diseñado para ejecutarse como una aplicación independiente a la que se pasan los parámetros por la línea de comandos (o proporcionando valores por defecto).
He aquí la parte más sorprendente. En **DosContadores.run**( ), se va pasando repetidamente por el bucle infinito recorriendo las líneas siguientes:

```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

(además de dormirse, pero eso ahora no importa). Al ejecutar el programa, sin embargo, se descubrirá que se observarán **conteol** y **conte02** (por parte de los

Observadores) ipara que a veces no sean iguales! Esto se debe a la naturaleza de los hilos -que pueden ser suspendidos en cualquier momento. Por ello, en ocasiones, se da la suspensión justo cuando se ha ejecutado la primera de estas líneas y no la segunda, y aparece el hilo **Observador** ejecutando la comprobación justo en ese momento, descubriendo, por consiguiente, que ambos hilos son distintos.

Este ejemplo muestra un problema fundamental del uso de los hilos. Nunca se sabe cuándo se podría ejecutar un hilo. Imagínese sentado en una mesa con un tenedor, justo a punto de engullir el ú1timo fragmento de comida del plato y justo cuando el tenedor va a alcanzarla, la comida simplemente se desvanece (porque se suspendió el hilo y apareció otro que robó la comida). Éste es el problema con el que se está tratando.

En ocasiones, no importa que un mismo recurso esté siendo accedido a la vez que se está intentado usar (la comida está en algún otro plato). Pero para que el multihilo funcione, es necesario disponer de alguna forma de evitar que dos hilos accedan al mismo recurso, al menos durante ciertos periodos críticos.

Evitar este tipo de colisión es simplemente un problema de poner un bloqueo en el recurso cuando lo esté usando un hilo. El primer hilo que accede al recurso lo bloquea, de forma que posteriormente los demás hilos no pueden acceder a este recurso hasta que éste quede desbloqueado, momento en el que es otro el hilo que lo bloquea y usa, etc. Si el asiento delantero de un coche es un recurso limitado, el primer niño que grite: "¡Para mí!", lo habrá bloqueado.

Cómo comparte Java los recursos

Java tiene soporte integrado para prevenir colisiones sobre cierto tipo de recurso: la memoria de un objeto. Puesto que generalmente se hacen los elementos de datos de clase **private** y se accede a esa memoria sólo a través de métodos, se pueden evitar las colisiones haciendo que un método particular sea **synchronized.** Sólo un hilo puede invocar a un método **synchronized** en cada instante para cada objeto (aunque ese hilo puede invocar a más de un método **synchronized** de varios objetos). He aquí métodos **synchronized** sencillos:

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

Cada objeto contiene un único bloqueo (llamado también monitor) que forma parte del objeto automáticamente (no hay que escribir ningún código especial). Cuando se llama a cualquier método **synchronized,** se bloquea el objeto y no se puede invocar a ningún otro método **synchronized** del objeto hasta que el primero acabe y libere el bloqueo. En el ejemplo de arriba, si se invoca a **f**( ) de un objeto, no se puede invocar a **g**( ) de ese mismo objeto hasta que se complete

f( ) y libere el bloqueo. Por consiguiente, hay un único bloqueo que es compartido por todos los métodos **synchronized** de un objeto en particular, y este bloqueo evita que la memoria en común sea escrita por más de un método en cada instante (es decir, más de un hilo en cada momento).

También hay un único bloqueo por clase (como parte del objeto **Class** de la clase), de forma que los métodos **synchronized static** pueden bloquearse mutuamente por accesos simultáneos a datos **static** en el ámbito de una clase.

Nótese que si se desea proteger algún recurso de accesos simultáneos por parte de múltiples hilos, se puede hacer forzando el acceso a ese recurso mediante métodos **synchronized.**

Sincronizar los contadores

Armado con esta nueva palabra clave, parece que la solución está a mano: simplemente se usará la palabra **synchronized** para los métodos de **DosContadores.** El ejemplo siguiente es igual que el anterior, con la adición de la nueva palabra:

```
//: c14:Sharing2.java
// Using the synchronized keyword to prevent
// multiple access to a particular resource.
// <applet code=Sharing2 width=350 height=500>
// <param name=size value="12">
// <param name=watchers value="15">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Sharing2 extends JApplet {
  TwoCounter[] s;
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;
```

```
class TwoCounter extends Thread {
  private boolean started = false;
  private JTextField
    t1 = new JTextField(5),
    t2 = new JTextField(5);
  private JLabel l =
    new JLabel("count1 == count2");
  private int count1 = 0, count2 = 0;
  public TwoCounter() {
    JPanel p = new JPanel();
    p.add(t1);
    p.add(t2);
    p.add(l);
    getContentPane().add(p);
  }
  public void start() {
    if(!started) {
      started = true;
      super.start();
    }
  }
  public synchronized void run() {
    while (true) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void synchTest() {
    Sharing2.incrementAccess();
    if(count1 != count2)
      l.setText("Unsynched");
  }
}

class Watcher extends Thread {
  public Watcher() { start(); }
  public void run() {
    while(true) {
      for(int i = 0; i < s.length; i++)
        s[i].synchTest();
      try {
        sleep(500);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
```

```
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
  class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new Label("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
  public static void main(String[] args) {
    Sharing2 applet = new Sharing2();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
```

```
        applet.numCounters * 50);
    }
} ///:~
```

Se verá que, *tanto* **run**( ) *como* **pruebaSinc**( ) son **synchronized.** Si se sincroniza sólo uno de los métodos, el otro es libre de ignorar el bloqueo del objeto y accederlo con impunidad. Éste es un punto importante: todo método que acceda a recursos críticos compartidos debe ser **synchronized** o no funcionará correctamente.

Ahora aparece un nuevo aspecto. El **Observador** nunca puede saber qué está ocurriendo exactamente porque todo el método **run**( ) está **synchronized,** y dado que **run**( ) siempre se está ejecutando para cada objeto, el bloqueo siempre está activado y no se puede llamar nunca a **pruebasinc** ( ). Esto se puede ver porque **RecuentoAcceso** nunca cambia.

Lo que nos gustaría de este ejemplo es alguna forma de aislar sólo *parte* del código dentro de **run**( ).

La sección de código que se desea aislar así se denomina una *sección crz'tica* y la palabra clave **synchronized** se usa de forma distinta para establecer una sección crítica. Java soporta secciones críticas con el *bloque sincronizado;* esta vez, **synchronized** se usa para especificar el objeto cuyo bloqueo se usa para sincronizar el código adjunto:

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one thread at a time
}
```

Antes de poder entrar al bloque sincronizado, hay que adquirir el bloqueo en **objetosinc.** Si algún otro hilo ya tiene este bloqueo, no se puede entrar en este bloque hasta que el bloqueo ceda.

El ejemplo **Compartiendo2** puede modificarse quitando la palabra clave **synchronized** de todo el método **run**( ) y pendiendo en su lugar un bloque **synchronized** en torno a las dos líneas críticas.

Pero ¿qué objeto debería usarse como bloqueo? El que ya está involucrado en **pruebaSinc**( ), que es el objeto actual (**this)!** Por tanto, el método **run**( ) modificado es así:

```
  public void run() {
    while (true) {
      synchronized(this) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
```

```
        }
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
```

Éste es el único cambio que habría que hacer a **Compartiendo2.java,** y se verá que, mientras que los dos contadores nunca están fuera de sincronismo (de acuerdo al momento en que **Observador** puede consultar su valor), se sigue proporcionando un acceso adecuado a **Observador** durante la ejecución de **run**( ).

Por supuesto, toda la sincronización depende de la diligencia del programador: todo fragmento de código que pueda acceder a un recurso compartido deberá envolverse en un bloque sincronizado.

Eficiencia sincronizada

Dado que tener dos métodos que escriben al mismo fragmento de información no parece nunca ser una buena idea, podría parecer que tiene sentido que todos los métodos sean automáticamente **synchronized** y eliminar de golpe todas las palabras **synchronized.** (Por supuesto, el ejemplo con un **synchronized run**( ) muestra que esto tampoco funcionaría.) Pero resulta que adquirir un bloqueo no es una operación barata -multiplica el coste de una llamada a un método (es decir, la entrada y salida del método, no la ejecución del método) al menos por cuatro, y podría ser mucho más dependiente de la implementación en sí. Por tanto, si se sabe que un método en particular no causará problemas de contención, es mejor no poner la palabra clave **synchronized.** Por otro lado, dejar de lado la palabra **synchronized** por considerarla un cuello de botella, esperando que no se den colisiones, es una invitación al desastre.

Revisar los JavaBeans

Ahora que se entiende la sincronización, se puede echar un nuevo vistazo a los JavaBeans. Cuando se cree un **Bean**, hay que asumir que se ejecutará en un entorno multihilo. Esto significa que:

1. Siempre que sea posible, todos los métodos **public** de un **Bean** deberían ser **synchronized.** Por supuesto, esto incurre en cierta sobrecarga en tiempo de ejecución. Si eso es un problema, se pueden dejar no **synchronized** los métodos que no causen problemas en secciones críticas, pero hay que tener en cuenta que esto no siempre es obvio. Los métodos encargados de calificar suelen ser pequeños (como es el caso de

getTamanioCirculo() en el ejemplo siguiente) y/o "atómicos", es decir, la llamada al método se ejecuta en una cantidad de código tan pequeña que el objeto no puede variar durante la ejecución. Hacer estos métodos no **synchronized** podría no tener un efecto significativo en la'velocidad de ejecución de un programa. También se podrían hacer **public** todos los métodos de un Bean. También se podrían hacer **synchronized** todos los métodos **public** de un Bean, y eliminar la palabra clave **synchronized** sólo cuando se tiene la total seguridad de que es necesario hacerlo, y que su eliminación surtirá algún efecto.

2. Al disparar un evento multidifusión a un conjunto de oyentes interesados, hay que asumir que se podrían añadir o eliminar oyentes al recorrer la lista.

Es bastante fácil operar con el primer punto, pero el segundo requiere pensar un poco más. Considérese el ejemplo **BeanExplosion.java** del capítulo anterior. Éste eludía el problema del multihilo ignorando la palabra clave **synchronized** (que no se había presentado aún) y haciendo el evento unidifusión. He aquí un ejemplo modificado para que funcione en un evento multihilo y use la "multidifusión" para los eventos:

```
//: c14:BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class BangBean2 extends JPanel
    implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Circle size
  private String text = "Bang!";
  private int fontSize = 48;
  private Color tColor = Color.red;
  private ArrayList actionListeners =
    new ArrayList();
  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);
  }
  // This is a multicast listener, which is
  // more typically used than the unicast
  // approach taken in BangBean.java:
  public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
  }
  public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
  }
  // Notice this isn't synchronized:
  public void notifyListeners() {
    ActionEvent a =
      new ActionEvent(BangBean2.this,
        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Make a shallow copy of the List in case
    // someone adds a listener while we're
    // calling listeners:
    synchronized(this) {
```

```
      lv = (ArrayList)actionListeners.clone();
    }
    // Call all the listener methods:
    for(int i = 0; i < lv.size(); i++)
      ((ActionListener)lv.get(i))
        .actionPerformed(a);
  }
  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();
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BangBean2 bb = new BangBean2();
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("ActionEvent" + e);
      }
    });
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("BangBean2 action");
      }
    });
    bb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        System.out.println("More action");
      }
    });
    Console.run(bb, 300, 300);
  }
} ///:~
```

Añadir **synchronized** a los métodos es un cambio sencillo. Sin embargo, hay que darse cuenta de que en **addActionListener**( ) y **removeActionListener**( ) se añaden y eliminan **ActionListeners** de un **ArrayList**, por lo que se puede tener tantos como se quiera.

Se puede ver que el método **notifyListeners**( ) no es **synchronized**. Puede ser invocado desde más de un hilo simultáneamente. También es posible que **addActionListener**( ) o **removeActionListener**( ) sean invocados en el medio de una llamada a **notifyListeners**( ), lo que supone un problema puesto que recorre el **ArrayList oyentesAccion**. Para aliviar el problema, se clona el **ArrayList** dentro de una cláusula **synchronized** y se recorre el clon (véase Apéndice A para obtener más detalles sobre la clonación). De esta forma se puede manipular el **ArrayList** original sin que esto suponga ningún impacto sobre **notifyListeners**( ).

El método **paintComponent**( ) tampoco es **synchronized**. Decidir si sincronizar o no métodos superpuestos no está tan claro como al añadir métodos propios. En este ejemplo, resulta que **paint**( ) parece funcionar bien esté o no **synchronized**. Pero hay que considerar aspectos como:

1. ¿Modifica el método el estado de variables "críticas" dentro del objeto? Para descubrir si las variables son o no "críticas" hay que determinar si serán leídas o modificadas por otros hilos del programa. (En este caso, la lectura y modificación son casi siempre llevadas a cabo por métodos **synchronized**, con lo que basta con examinar éstos.) En el caso de **paint**( ), no se hace ninguna modificación.

2. ¿Depende el método del estado de estas variables "críticas"? Si un método **synchronized** modifica una variable que usa tu método, entonces es más que deseable hacer que ese método también sea **synchronized**. Basándonos en esto, podría observarse que **tamanioC** se modifica en métodos **synchronized**, y que por consiguiente, **paint**( ) debería ser **synchronized**. Sin embargo, aquí se puede preguntar: ¿qué es lo peor que puede ocurrir si se cambiase **tamanioC** durante un **paint**( )? Cuando se vea que no ocurre nada demasiado malo, se puede decidir dejar **paint**( ) como no **synchronized** para evitar la sobrecarga extra intrínseca a llamadas a este tipo de métodos.

3. Una tercera pista es darse cuenta de si la versión base de **paint**( ) es **synchronized**, que no lo es. Éste no es un argumento sólido, sólo una pista. En este caso, por ejemplo, se ha mezclado un campo que se *modifica* vía métodos **synchronized** (como **tamanioC**) en la fórmula **paint**( ) y podría haber cambiado la situación. Nótese, sin embargo, que el ser **synchronized** no se hereda - es decir, si un método es **synchronized** en la clase base, *no* es automáticamente **synchronized** en la versión superpuesta de la clase derivada.

Se ha modificado el código de prueba de **BeanExplosion2** con respecto al del capítulo anterior para demostrar la habilidad multidifusión de **BeanExplosion2** añadiendo oyentes extra.

Bloqueo

Un hilo puede estar en uno de estos cuatro estados:

1. ***Nuevo****:* se ha creado el objeto hilo pero todavía no se ha arrancado, por lo que no se puede ejecutar.

2. ***Ejecutable****:* Significa que el hilo *puede* ponerse en ejecución cuando el mecanismo de reparto de tiempos de UCP tenga ciclos disponibles para el hilo. Por consiguiente, el hilo podría estar o no en ejecución, pero no hay nada para evitar que sea ejecutado si el planificador así lo dispone; no está ni muerto ni bloqueado.

3. ***Muerto****:* la forma normal de morir de un hilo es que finalice su método **run**( ). También se puede llamar a **stop**( ), pero esto lanza una excepción que es una subclase de **Error** (lo que significa que no hay obligación de poner la llamada en un bloque by). Recuérdese que el lanzamiento de una excepción debería ser un evento especial y no parte de la ejecución normal de un programa; por consiguiente, en Java 2 se ha abolido el uso de **stop**( ). También hay un método **destroy**( ) (que jamás se implementó) al que nunca habría que llamar si puede evitarse, puesto que es drástico y no libera bloqueos sobre los objetos.

4. ***Bloqueado****:* podría ejecutarse el hilo, pero hay algo que lo evita. Mientras un hilo esté en estado bloqueado, el planificador simplemente se lo salta y no le cede ningún tipo de UCP. Hasta que el hilo no vuelva al estado ejecutable no hará ninguna operación.

Bloqueándose

El estado bloqueado es el más interesante, y merece la pena examinarlo más en detalle. Un hilo puede bloquearse por cinco motivos:

1. Se ha puesto el hilo a dormir llamando a **sleep**(milisegundos), en cuyo caso no se ejecutará durante el tiempo especificado.

2. Se ha suspendido la ejecución del hilo con **suspend**( ). No se volverá ejecutable de nuevo hasta que el hilo reciba el mensaje **resume**( ). (Éstos están en desuso en Java 2, y se examinarán más adelante.)

3. Se ha suspendido la ejecución del hilo con **wait(**). No se volverá ejecutable de nuevo hasta que el hilo reciba los mensajes **notify**( ) o **notifyAll**( ). (Sí, esto parece idéntico al caso 2, pero hay una diferencia que luego revelaremos.)

4. El hilo está esperando a que se complete alguna E/S.

5. El hilo está intentando llamar a un método **synchronized** de otro objeto y el bloqueo del objeto no está disponible.

También se puede llamar a **yield**( ) (un método de la clase **Thread**) para ceder voluntariamente la UCP de forma que se puedan ejecutar otros hilos. Sin embargo, si el planificador decide que un hilo ya ha dispuesto de suficiente tiempo ocurre lo mismo, saltándose al siguiente hilo. Es decir, nada evita que el planificador mueva el hilo y le dé tiempo a otro hilo. Cuando se bloquea un hilo, hay alguna razón por la cual no puede continuar ejecutándose.

E1 ejemplo siguiente muestra las cinco maneras de bloquearse. Todo está en un único archivo denominado **Bloqueo.java**, pero se examinará en fragmentos discretos. (Se verán las etiquetas "Continuará" y "Continuación" que permiten a la herramienta de extracción de código componerlo todo junto.)

Dado que este ejemplo demuestra algunos métodos en desuso, se *obtendrán* mensajes de en "desuso" durante la compilación.

Primero, el marco de trabajo básico:

```
//: c14:Blocking.java
// Demonstrates the various ways a thread
// can be blocked.
// <applet code=Blocking width=350 height=550>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import com.bruceeckel.swing.*;

////////////// The basic framework ///////////
class Blockable extends Thread {
  private Peeker peeker;
  protected JTextField state = new JTextField(30);
  protected int i;
  public Blockable(Container c) {
    c.add(state);
    peeker = new Peeker(this, c);
  }
  public synchronized int read() { return i; }
  protected synchronized void update() {
    state.setText(getClass().getName()
      + " state: i = " + i);
  }
  public void stopPeeker() {
    // peeker.stop(); Deprecated in Java 1.2
    peeker.terminate(); // The preferred approach
  }
}
```

```
class Peeker extends Thread {
  private Blockable b;
  private int session;
  private JTextField status = new JTextField(30);
  private boolean stop = false;
  public Peeker(Blockable b, Container c) {
    c.add(status);
    this.b = b;
    start();
  }
  public void terminate() { stop = true; }
  public void run() {
    while (!stop) {
      status.setText(b.getClass().getName()
        + " Peeker " + (++session)
        + "; value = " + b.read());
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
} ///:Continued
```

La clase **Bloqueable** pretende ser la clase base de todas las clases en el ejemplo que muestra el bloqueo. Un objeto **Bloqueable** contiene un **JTextField** de nombre **estado** que se usa para mostrar información sobre el objeto. El método que muestra esta información es **actualizar**( ). Se puede ver que usa **getClass**(**).gemame**( ) para producir el nombre de la clase, en vez de simplemente imprimirlo; esto se debe a que **actualizar**( ) no puede conocer el nombre exacto de la clase a la que se llama, pues será una clase derivada de **Bloqueable.**

El indicador de cambio de **Bloqueable** es un **int i,** que será incrementado por el método **run**( ) de la clase derivada.

Por cada objeto **Bloqueable** se arranca un hilo de clase **Elector,** cuyo trabajo es vigilar a su objeto **Bloqueable** asociado para ver los cambios en **i** llamando a **leer**( ) e informando de ellos en su **JTextField estado.** Esto es importante: nótese que **leer**( ) y **actualizar**( ) son ambos **synchronized,** lo que significa que precisan que el bloqueo del objeto esté libre.

Dormir

La primera prueba del programa se hace con **sleep**( ):

```
///:Continuing
//////////// Blocking via sleep() //////////
```

```
class Sleeper1 extends Blockable {
  public Sleeper1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class Sleeper2 extends Blockable {
  public Sleeper2(Container c) { super(c); }
  public void run() {
    while(true) {
      change();
       try {
        sleep(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
} ///:Continued
```

En **Durmiente1** todo el método **run**( ) es **synchronized.** Se verá que el **Elector** asociado con este objeto se ejecutará alegremente *hasta* que el hilo comience, y después el **Elector** se detiene en seco. Ésta es una forma de bloquear: dado que **Durmientel.run**( ) es **synchronized,** y una vez que el objeto empieza, siempre está dentro de **run**( ), el método nunca cede el bloqueo del objeto, quedando **Elector** bloqueado.

Durmiente2 proporciona una solución haciendo **run**( ) no **synchronized.** Sólo es **synchronized** el método **cambiar**( ), lo que significa que mientras **run**( ) esté en **sleep**( ), el **Elector** puede acceder al método **synchronized** que necesite, en concreto a **leer**( ). Aquí se verá que el **Elector** continúa ejecutándose al empezar el hilo **Durmiente2.**

Suspender y continuar

La siguiente parte del ejemplo presenta el concepto de la suspensión. La clase **Thread** tiene un método **suspend**( ) para detener temporalmente el hilo y **resume**( ) lo continúa en el punto en el que se detuvo. Hay que llamar a **resume**( ) desde otro hilo fuera del suspendido, y en este caso hay una clase separada denominada **Resumidor** que hace exactamente eso. Cada una de las clases que demuestra suspender/continuar tiene un **Resumidor** asociado:

```
///:Continuing
////////// Blocking via suspend() //////////
class SuspendResume extends Blockable {
  public SuspendResume(Container c) {
    super(c);
    new Resumer(this);
  }
}

class SuspendResume1 extends SuspendResume {
  public SuspendResume1(Container c) { super(c);}
  public synchronized void run() {
    while(true) {
      i++;
      update();
      suspend(); // Deprecated in Java 1.2
    }
  }
}

class SuspendResume2 extends SuspendResume {
  public SuspendResume2(Container c) { super(c);}
  public void run() {
    while(true) {
      change();
      suspend(); // Deprecated in Java 1.2
    }
  }
  public synchronized void change() {
      i++;
      update();
  }
}

class Resumer extends Thread {
  private SuspendResume sr;
  public Resumer(SuspendResume sr) {
    this.sr = sr;
    start();
  }
  public void run() {
    while(true) {
```

```
        try {
          sleep(1000);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
        sr.resume(); // Deprecated in Java 1.2
      }
    }
} ///:Continued
```

SuspenderResumir1 también tiene un método **syncronized run**( ). De nuevo, al arrancar este hilo se verá que su **Elector** asociado se bloquea esperando a que el bloqueo quede disponible, lo que no ocurre nunca. Esto se fija como antes en **SuspenderResumir2,** en el que no todo el **run**( ) es **synchronized,** sino que usa un método **synchronized cambiar**( ) diferente.

Hay que ser consciente de que Java 2 ha abolido el uso de **suspend**( ) y **resume**( ) porque **suspend**( ) se guarda el bloqueo sobre el objeto y, por tanto, puede conducir fácilmente a interbloqueos. Es decir, se puede lograr fácilmente que varios objetos bloqueados esperen por culpa de otros que, a su vez, esperan por los primeros, y esto hará que el programa se detenga. Aunque se podrá ver que programas antiguos usan estos métodos, no habría que usarlos. Más adelante, dentro de este capítulo, se describe la solución.

Wait y notify

En los dos primeros ejemplos, es importante entender que, tanto **sleep**( ) como **suspend**( ), no liberan el bloqueo cuando son invocados. Hay que ser consciente de este hecho si se trabaja con bloqueos. Por otro lado, el método **wait**( ) libera el bloqueo cuando es invocado, lo que significa que se puede llamar a otros métodos **synchronized** del objeto hilo durante un **wait**( ). En los dos casos siguientes, se verá que el método **run**( ) está totalmente **synchronized** en ambos casos, sin embargo, el **Elector** sigue teniendo acceso completo a los métodos **synchronized** durante un **wait**( ). Esto se debe a que **wait**( ) libera el bloqueo sobre el objeto al suspender el método en el que es llamado.

También se verá que hay dos formas de **wait**( ). La primera toma como parámetro un número de milisegundos, con el mismo significado que en **sleep**( ): pausar durante ese periodo de tiempo. La diferencia es que en la **wait**( ) se libera el bloqueo sobre el objeto y se puede salir de **wait**( ) gracias a un **notify**( ), o a que una cuenta de reloj expira.

La segunda forma no toma parámetros, y quiere decir que continuará **wait**( ) hasta que venga un **notify**( ), no acabando automáticamente tras ningún periodo de tiempo.

Un aspecto bastante único de **wait**( ) y **notify**( ) es que ambos métodos son parte de la clase base **Object** y no parte de **Thread,** como es el caso de **sleep**( ), **suspend**( ) y **resume**( ). Aunque esto parece un poco extraño a primera vista - tener algo que es exclusivamente para los hilos como parte de la clase base universal- es esencial porque manipulan el bloqueo que también es parte de todo objeto. Como resultado, se puede poner un **wait**( ) dentro de cualquier método **synchronized,** independientemente de si hay algún tipo de hilo dentro de esa clase en particular. De hecho, el único lugar en el que se puede llamar a **wait**( ) es dentro de un método o bloque **synchronized.** Si se llama a **wait**( ) o **notify**( ) dentro de un método que no es **synchronized,** el programa compilará, pero al ejecutarlo se obtendrá una **IllegalMonitorStateException,** con el mensaje no muy intuitivo de "hilo actual no propietario". Nótese que, desde métodos no **synchronized,** sí que se puede llamar a **sleep**( ), **suspend**( ) **y resume**( ) puesto que no manipulan el bloqueo.

Se puede llamar a **wait**( ) o **notify**( ) sólo para nuestro propio bloqueo. De nuevo, se puede compilar código que usa el bloque erróneo, pero producirá el mismo mensaje **IllegalMonitorStateException** que antes. No se puede jugar con el bloqueo de nadie más, pero se puede pedir a otro objeto que lleve a cabo una operación que manipule su propio bloqueo. Por tanto, un enfoque es crear un método **synchronized** que llame a **notify**( ) para su propio objeto. Sin embargo, en **Notificador** se verá la llamada **notify**( ) dentro de un bloque **synchronized:**

```
synchronized(wn2) {
  wn2.notify();
}
```

donde **wn2** es el tipo de objeto **EsperarNotificar2.** Este método, que no es parte de **EsperarNotificar2,** adquiere el bloqueo sobre el objeto **wn2,** instante en el que es legal que invoque al **notify**( ) de **wn2** sin obtener, por tanto, la **IllegalMonitorStateException.**

```
///:Continuing
////////// Blocking via wait() //////////
class WaitNotify1 extends Blockable {
  public WaitNotify1(Container c) { super(c); }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait(1000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
```

```
}

class WaitNotify2 extends Blockable {
  public WaitNotify2(Container c) {
    super(c);
    new Notifier(this);
  }
  public synchronized void run() {
    while(true) {
      i++;
      update();
       try {
        wait();
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}

class Notifier extends Thread {
  private WaitNotify2 wn2;
  public Notifier(WaitNotify2 wn2) {
    this.wn2 = wn2;
    start();
  }
  public void run() {
    while(true) {
       try {
        sleep(2000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      synchronized(wn2) {
        wn2.notify();
      }
    }
  }
}  ///:Continued
```

wait( ) suele usarse cuando se ha llegado a un punto en el que se está esperando alguna otra condición, bajo el control de fuerzas externas al hilo y no se desea esperar ociosamente dentro del hilo.

Por tanto, **wait**( ) permite poner el hilo a dormir mientras espera a que el mundo cambie, y sólo cuando se da un **notify**( ) o un **notifyAll**( ) se despierta el método y comprueba posibles cambios.

Por consiguiente, proporciona una forma de sincronización entre hilos.

Bloqueo en E/S

Si un flujo está esperando a alguna actividad de E/S, se bloqueará automáticamente. En la siguiente porción del ejemplo, ambas clases funcionan con objetos **Reader** y **Writer**, pero en el marco de trabajo de prueba, se establecerá un flujo entubado para permitir a ambos hilos pasarse datos mutuamente de forma segura (éste es el propósito de los flujos entubados).

El Emisor pone datos en el **Writer** y se duerme durante una cantidad de tiempo aleatoria. Sin embargo, **Receptor** no tiene **sleep**( ), **suspend**( ) ni **wait**( ). Pero cuando hace un **read**( ) se bloquea automáticamente si no hay más datos.

```
///:Continuing
class Sender extends Blockable { // send
  private Writer out;
  public Sender(Container c, Writer out) {
    super(c);
    this.out = out;
  }
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          i++;
          out.write(c);
          state.setText("Sender sent: "
            + (char)c);
          sleep((int)(3000 * Math.random()));
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        } catch(IOException e) {
          System.err.println("IO problem");
        }
      }
    }
  }
}

class Receiver extends Blockable {
  private Reader in;
  public Receiver(Container c, Reader in) {
    super(c);
    this.in = in;
  }
  public void run() {
    try {
      while(true) {
        i++; // Show peeker it's alive
```

```
        // Blocks until characters are there:
        state.setText("Receiver read: "
          + (char)in.read());
      }
    } catch(IOException e) {
      System.err.println("IO problem");
    }
  }
} ///:Continued
```

Ambas clases también ponen información en sus campos **estado** y cambian **i**, de forma que el **Elector** pueda ver que el hilo se está ejecutando.

Probar

La clase *applet* principal es sorprendentemente simple, porque se ha puesto la mayoría de trabajo en el marco de trabajo **Bloqueable**. Básicamente, se crea un array de objetos **Bloqueable**, y puesto que cada uno es un hilo, llevan a cabo sus propias actividades al presionar el botón "empezar". También hay un botón y una cláusula **actionPerformed**( ) para detener todos los objetos Elector, que proporcionan una demostración de la alternativa al método **stop**( ) de **Thread**, en desuso (en Java 2).

Para establecer una conexión entre los objetos Emisor y Receptor, se crean un **PipedWriter** y un **PipedReader**. Nótese que el **PipedReader** entrada debe estar conectado al **PipedWriter** salida vía un parámetro del constructor. Después de eso, cualquier cosa que se coloque en salida podrá ser después extraída de entrada, como si pasara a través de una tubería (y de aquí viene el nombre). Los objetos entrada y salida se pasan a los constructores Receptor y Emisor respectivamente, que los tratan como objetos **Reader** y **Writer** de cualquier tipo (es decir, se les aplica un molde hacia arriba).

El array **b** de referencias a **Bloqueable** no se inicializa en este momento de la definición porque no se pueden establecer los flujos entubados antes de que se dé esa definición (lo evita la necesidad del bloque **try**).

```
///:Continuing
////////// Testing Everything //////////
public class Blocking extends JApplet {
  private JButton
    start = new JButton("Start"),
    stopPeekers = new JButton("Stop Peekers");
  private boolean started = false;
  private Blockable[] b;
  private PipedWriter out;
  private PipedReader in;
  class StartL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < b.length; i++)
          b[i].start();
      }
    }
  }
  class StopPeekersL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Demonstration of the preferred
      // alternative to Thread.stop():
      for(int i = 0; i < b.length; i++)
        b[i].stopPeeker();
    }
  }
  public void init() {
     Container cp = getContentPane();
     cp.setLayout(new FlowLayout());
     out = new PipedWriter();
    try {
      in = new PipedReader(out);
    } catch(IOException e) {
      System.err.println("PipedReader problem");
    }
    b = new Blockable[] {
      new Sleeper1(cp),
      new Sleeper2(cp),
      new SuspendResume1(cp),
      new SuspendResume2(cp),
      new WaitNotify1(cp),
      new WaitNotify2(cp),
      new Sender(cp, out),
      new Receiver(cp, in)
    };
    start.addActionListener(new StartL());
    cp.add(start);
    stopPeekers.addActionListener(
      new StopPeekersL());
    cp.add(stopPeekers);
  }
  public static void main(String[] args) {
    Console.run(new Blocking(), 350, 550);
  }
} ///:~
```

Nótese en **init**( ) el bucle que recorre todo el array añadiendo los campos de texto **estado** y **elector.estado** a la página.

Cuando se crean inicialmente los hilos **Bloqueable,** cada uno crea y arranca automáticamente su propio **Elector.** Por tanto, se verán los **Electores** ejecutándose antes de que se arranquen los hilos **Bloqueable.** Esto es importante, puesto que algunos de los **Electores** se bloquearán y detendrán cuando arranquen los hilos **Bloqueable,** y es esencial ver esto para entender ese aspecto particular del bloqueo.

Interbloqueo

Dado que los hilos pueden bloquearse y dado que los objetos pueden tener métodos **synchronized** que evitan que los hilos accedan a ese objeto hasta liberar el bloqueo de sincronización, es posible que un hilo se quede parado esperando a otro, que, de hecho, espera a un tercero, etc. hasta que el último de la cadena resulte ser un hilo que espera por el primero. Se logra un ciclo continuo de hilos que esperan entre sí, de forma que ninguno puede avanzar. A esto se le llama interbloqueo. Es verdad que esto no ocurre a menudo, pero cuando le ocurre a uno es muy frustrante.

No hay ningún soporte de lenguaje en Java que ayude a prevenir el interbloqueo; cada uno debe evitarlo a través de un diseño cuidadoso. Estas palabras no serán suficientes para complacer al que esté tratando de depurar un programa con interbloqueos.

La abolición de **stop(), suspend(**), **resume(**) **y destroy() en** Java **2**

Uno de los cambios que se ha hecho en Java 2 para reducir la posibilidad de interbloqueo es abolir los métodos **stop**( ), **suspend**( ), **resume**( ) y **destroy**( ) de **Thread.**

La razón para abolir el método **stop**( ) es que no libera los bloqueos que haya adquirido el hilo, y si los objetos están en un estado inconsistente ("dañados") los demás hilos podrán verlos y modificarlos en ese estado. Los problemas resultantes pueden ser grandes y además difíciles de detectar.

En vez de usar **stop**( ), habría que seguir el ejemplo de **Bloqueo.java** y usar un indicador (flag) que indique al hilo cuando acabar saliendo de su método **run**( ).

Hay veces en que un hilo se bloquea -como cuando se está esperando una entrada- y no puede interrogar al indicador como ocurre en **Bloqueo.java.** En estos casos, se debería seguir sin usar el método **stop**( ), sino usar el método **interrupt**( ) de **Thread** para salir del código bloqueado:

```
//: c14:Interrupt.java
// The alternative approach to using
```

```
// stop() when a thread is blocked.
// <applet code=Interrupt width=200 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Blocked extends Thread {
  public synchronized void run() {
    try {
      wait(); // Blocks
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
    System.out.println("Exiting run()");
  }
}

public class Interrupt extends JApplet {
  private JButton
    interrupt = new JButton("Interrupt");
  private Blocked blocked = new Blocked();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(interrupt);
    interrupt.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          System.out.println("Button pressed");
          if(blocked == null) return;
          Thread remove = blocked;
          blocked = null; // to release it
          remove.interrupt();
        }
      });
    blocked.start();
  }
  public static void main(String[] args) {
    Console.run(new Interrupt(), 200, 100);
  }
} ///:~
```

El **wait**( ) de dentro de **Bloqueado.run**( ) produce el hilo bloqueado. Al presionar el botón, se pone a **null** la referencia **bloqueado** de forma que será limpiada por el recolector de basura, y se invoca al método **interrupt**( ) del objeto. La primera

vez que se presione el botón se verá que el hilo acaba, pero una vez que no hay hilos que matar, simplemente hay que ver que se ha presionado el botón.

Los métodos **suspend**( ) y **resume**( ) resultan ser inherentemente causantes de interbloqueos. Cuando se llama a **suspend**( ), se detiene el hilo destino, pero sigue manteniendo los bloqueos que haya adquirido hasta ese momento. Por tanto, ningún otro hilo puede acceder a los recursos bloqueados, hasta que el hilo continúe. Cualquier hilo que desee continuar el hilo destino, y que también intente usar cualquiera de los recursos bloqueados, producirá interbloqueo. No se debería usar **suspend**( ) y **resume**( ), sino que en su lugar se pone un indicador en la clase **Thread** para indicar si debería activarse o suspenderse el hilo. Si elflag indica que el hilo está suspendido, el hilo se mete en una espera usando **wait** ( ). Cuando el flag indica que debería continuarse el hilo, se reinicia éste con **notify**( ). Se puede producir un ejemplo modificando **Contador2.java.** Aunque el efecto es similar, se verá que la organización del código es bastante diferente -se usan clases internas anónimas para todos los oyentes y el **Thread** es una clase interna, lo que hace la programación ligeramente más conveniente, puesto que elimina parte de la contabilidad necesaria en **Contador2.java:**

```
//: c14:Suspend.java
// The alternative approach to using suspend()
// and resume(), which are deprecated in Java 2.
// <applet code=Suspend width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Suspend extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspend = new JButton("Suspend"),
    resume = new JButton("Resume");
  private Suspendable ss = new Suspendable();
  class Suspendable extends Thread {
    private int count = 0;
    private boolean suspended = false;
    public Suspendable() { start(); }
    public void fauxSuspend() {
      suspended = true;
    }
    public synchronized void fauxResume() {
      suspended = false;
      notify();
    }
    public void run() {
      while (true) {
```

```
        try {
          sleep(100);
          synchronized(this) {
            while(suspended)
              wait();
          }
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
        t.setText(Integer.toString(count++));
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    suspend.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxSuspend();
        }
      });
    cp.add(suspend);
    resume.addActionListener(
      new ActionListener() {
        public
        void actionPerformed(ActionEvent e) {
          ss.fauxResume();
        }
      });
    cp.add(resume);
  }
  public static void main(String[] args) {
    Console.run(new Suspend(), 300, 100);
  }
} ///:~
```

El indicador **suspendido** de **suspendido** se usa para activar y desactivar la suspensión. Para suspender, se pone el indicador a **true** llamando a **fauxSuspender**( ), y esto se detecta dentro de **run**( ). El método **wait**( ), como se describió anteriormente en este capítulo, debe ser **synchronized,** de forma que tenga el bloqueo al objeto. En **fauxResumir**( ), se pone el indicador **suspendedido** a **false** y se llama a **notifiy**( ) -puesto que esto despierta a **wait**( ) de una cláusula **synchronized,** el método **fauxResumir**( ) también debe ser **syncronized**( ) de forma que adquiera el bloqueo antes de llamar a **notifiy**( ) (por consiguiente, el bloqueo queda disponible para **wait**( )...).

Si se sigue el estilo mostrado en este programa, se puede evitar usar **suspend**( ) y **resume**( ).
El método **destroy**( ) de **Thread**( ) nunca fue implementado; es como un **suspend**( ) que no se puede continuar, por lo que tiene los mismos aspectos de interbloqueo que **suspend**( ). Sin embargo, éste no es un método abolido y puede que se implemente en una versión futura de Java (posterior a la 2) para situaciones especiales en las que el riesgo de interbloqueo sea aceptable.

Uno podría preguntarse por qué estos métodos, ahora abolidos, se incluyeron en Java en primer lugar. Parece admitir un error bastante importante para simplemente eliminarlas (e introduciendo otro agujero más en los argumentos que hablan del excepcional diseño de Java y de su infalibilidad, de los que tanto hablaban los encargados de marketing en Sun). Lo más alentador del cambio es que indica claramente que es el personal técnico y no el de marketing el que dirige el asunto -descubrieron el problema y lo están arreglando. Creemos que esto es mucho más prometedor y alentador que dejar el problema ahí pues "corregirlo supondría admitir un error". Esto significa que Java continuará mejorando, incluso aunque esto conlleve alguna pequeña molestia para los programadores de Java. Preferimos, no obstante, afrontar estas molestias a ver cómo se estanca el lenguaje.

Prioridades

La *prioridad* de un hilo indica al planificador lo importante que es cada hilo. Si hay varios hilos bloqueados o en espera de ejecutarse, el planificador ejecutará el de mayor prioridad en primer lugar.

Sin embargo, esto no significa que los hilos de menor prioridad no se ejecuten (es decir, no se llega a interbloqueo simplemente con la aplicación directa de estos principios). Los hilos de menor prioridad tienden a ejecutarse menos a menudo.

Aunque es interesante conocer las prioridades que se manejan, en la práctica casi nunca hay que establecer a mano las prioridades. Por tanto uno puede saltarse el resto de esta sección si no le interesan las prioridades.

Leer y establecer prioridades

Se puede leer la prioridad de un hilo con **getPriority**( ) y cambiarla con **setPriority**( ). Se puede usar la forma de los ejemplos "contador" anteriores para mostrar el efecto de variar las prioridades.

En este *applet* se verá que los contadores se ralentizan, dado que se han disminuido las prioridades de los hilos asociados:

```
//: c14:Counter5.java
// Adjusting the priorities of threads.
// <applet code=Counter5 width=450 height=600>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class Ticker2 extends Thread {
  private JButton
    b = new JButton("Toggle"),
    incPriority = new JButton("up"),
    decPriority = new JButton("down");
  private JTextField
    t = new JTextField(10),
    pr = new JTextField(3); // Display priority
  private int count = 0;
  private boolean runFlag = true;
  public Ticker2(Container c) {
    b.addActionListener(new ToggleL());
    incPriority.addActionListener(new UpL());
    decPriority.addActionListener(new DownL());
    JPanel p = new JPanel();
    p.add(t);
    p.add(pr);
    p.add(b);
    p.add(incPriority);
    p.add(decPriority);
    c.add(p);
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  class UpL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() + 1;
      if(newPriority > Thread.MAX_PRIORITY)
        newPriority = Thread.MAX_PRIORITY;
      setPriority(newPriority);
    }
  }
  class DownL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int newPriority = getPriority() - 1;
      if(newPriority < Thread.MIN_PRIORITY)
        newPriority = Thread.MIN_PRIORITY;
      setPriority(newPriority);
```

```
    }
  }
  public void run() {
    while (true) {
      if(runFlag) {
        t.setText(Integer.toString(count++));
        pr.setText(
          Integer.toString(getPriority()));
      }
      yield();
    }
  }
}

public class Counter5 extends JApplet {
  private JButton
    start = new JButton("Start"),
    upMax = new JButton("Inc Max Priority"),
    downMax = new JButton("Dec Max Priority");
  private boolean started = false;
  private static final int SIZE = 10;
  private Ticker2[] s = new Ticker2[SIZE];
  private JTextField mp = new JTextField(3);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new Ticker2(cp);
    cp.add(new JLabel(
      "MAX_PRIORITY = " + Thread.MAX_PRIORITY));
    cp.add(new JLabel("MIN_PRIORITY = "
      + Thread.MIN_PRIORITY));
    cp.add(new JLabel("Group Max Priority = "));
    cp.add(mp);
    cp.add(start);
    cp.add(upMax);
    cp.add(downMax);
    start.addActionListener(new StartL());
    upMax.addActionListener(new UpMaxL());
    downMax.addActionListener(new DownMaxL());
    showMaxPriority();
    // Recursively display parent thread groups:
    ThreadGroup parent =
      s[0].getThreadGroup().getParent();
    while(parent != null) {
      cp.add(new Label(
        "Parent threadgroup max priority = "
        + parent.getMaxPriority()));
      parent = parent.getParent();
    }
```

```
  }
  public void showMaxPriority() {
    mp.setText(Integer.toString(
      s[0].getThreadGroup().getMaxPriority()));
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for(int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }
  class UpMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(++maxp > Thread.MAX_PRIORITY)
        maxp = Thread.MAX_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  class DownMaxL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int maxp =
        s[0].getThreadGroup().getMaxPriority();
      if(--maxp < Thread.MIN_PRIORITY)
        maxp = Thread.MIN_PRIORITY;
      s[0].getThreadGroup().setMaxPriority(maxp);
      showMaxPriority();
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter5(), 450, 600);
  }
} ///:~
```

Teletipo2 sigue la forma establecida previamente en este capítulo, pero hay un **JTextField** extra para mostrar la prioridad del hilo y dos botones más para incrementar y disminuir la prioridad.

Hay que tener en cuenta el uso de **yield**( ), que devuelve automáticamente el control al planificador. Sin éste, el mecanismo de multihilos sigue funcionando, pero se verá que funciona lentamente (puede intentarse eliminar la llamada a **yield**( ) para comprobarlo). También se podría llamar a **sleep**( ), pero entonces el ratio de cuenta estaría controlado por la duración de **sleep**( ) en vez de por la prioridad.

El **init**( ) de **Contador5** crea un array de diez **Teletipos2;** sus botones y campos los ubica en el formulario el constructor de **Teletipo2. Contador5** añade botones para dar comienzo a todo además de incrementar y disminuir la prioridad máxima del grupo de hilos. Además, están las etiquetas que muestran las prioridades máxima y mínima posibles para un hilo, y un **JTextField** para mostrar la prioridad máxima del grupo de hilos. (La siguiente sección describirá los grupos de hilos).

Finalmente, también se muestran como etiquetas las prioridades de los grupos de hilos padre.
Cuando se presiona un botón "arriba" o "abajo", se alcanza esa prioridad de **Teletipo2,** que es incrementada o disminuida de forma acorde.

Cuando se ejecute este programa, se apreciarán varias cosas. En primer lugar, la prioridad por defecto del grupo de hilos es cinco. Incluso si se disminuye la prioridad máxima por debajo de cinco antes de que los hilos empiecen (o antes de crearlos, lo que requiere cambiar el código), cada hilo tendrá su prioridad por defecto a cinco.

La prueba más sencilla es tomar un contador y disminuir su prioridad a uno, y observar que cuenta mucho más lentamente. Pero ahora puede intentarse incrementarla de nuevo. Se puede volver a la prioridad del grupo de hilos, pero no a ninguna superior. Ahora puede disminuirse un par de veces la prioridad de un grupo de hilos. Las prioridades de los hilos no varían, pero si se intenta modificar éstas hacia arriba o hacia abajo, se verá que sacan automáticamente la prioridad del grupo de hilos. Además, se seguirá dando a los hilos nuevos una prioridad por defecto, incluso aunque sea más alta que la prioridad del grupo. (Por consiguiente la prioridad del grupo no es una forma de evitar que los hilos nuevos tengan prioridades mayores a las de los existentes.)

Finalmente, puede intentarse incrementar la prioridad máxima del grupo. No puede hacerse. Las prioridades máximas de los grupos de hilos sólo pueden reducirse, nunca incrementarse.

Grupos de hilos

Todos los hilos pertenecen a un grupo de hilos. Éste puede ser, o bien el grupo de hilos por defecto, o un grupo explícitamente especificado al crear el hilo. En el momento de su creación, un hilo está vinculado a un grupo y no puede cambiar a otro distinto. Cada aplicación tiene, al menos, un hilo que pertenece al grupo de hilos del sistema. Si se crean más hilos sin especificar ningún grupo, éstos también pertenecerán al grupo de hilos del sistema.

Los grupos de hilos también deben pertenecer a otros grupos de hilos. El grupo de hilos al que pertenece uno nuevo debe especificarse en el constructor. Si se crea un grupo de hilos sin especificar el grupo de hilos al que pertenezca, se ubicará bajo el grupo de hilos del sistema. Por consiguiente, todos los grupos de hilos de la aplicación tendrán como último padre al grupo de hilos del sistema.

La razón de la existencia de grupos de hilos no es fácil de determinar a partir de la literatura, que tiende a ser confusa en este aspecto. Suelen citarse "razones de seguridad". De acuerdo con Arnold & Gosling **[2]**, "Los hilos de un grupo de hilos pueden modificar a los otros hilos del grupo, incluyendo todos sus descendientes. Un hilo no puede modificar hilos de fuera de su grupo o grupos contenidos." Es difícil saber qué se supone que quiere decir en este caso el término "modificar". El ejemplo siguiente muestra un hilo en un subgrupo "hoja", modificando las prioridades de todos los hilos en su árbol de grupos de hilos, además de llamar a un método para todos los hilos de su árbol.

[2] ***The*** *Java Programming Language,* de Ken Arnold y James Gosling, Addison Wesley 1996 pag. 179.

```
//: c14:TestAccess.java
// How threads can access other threads
// in a parent thread group.

public class TestAccess {
  public static void main(String[] args) {
    ThreadGroup
      x = new ThreadGroup("x"),
      y = new ThreadGroup(x, "y"),
      z = new ThreadGroup(y, "z");
    Thread
      one = new TestThread1(x, "one"),
      two = new TestThread2(z, "two");
  }
}

class TestThread1 extends Thread {
  private int i;
  TestThread1(ThreadGroup g, String name) {
    super(g, name);
  }
  void f() {
    i++; // modify this thread
    System.out.println(getName() + " f()");
  }
}

class TestThread2 extends TestThread1 {
  TestThread2(ThreadGroup g, String name) {
```

```
    super(g, name);
    start();
  }
  public void run() {
    ThreadGroup g =
      getThreadGroup().getParent().getParent();
    g.list();
    Thread[] gAll = new Thread[g.activeCount()];
    g.enumerate(gAll);
    for(int i = 0; i < gAll.length; i++) {
      gAll[i].setPriority(Thread.MIN_PRIORITY);
      ((TestThread1)gAll[i]).f();
    }
    g.list();
  }
} ///:~
```

En el método **main**( ) se crean varios **ThreadGroups**, colgando unos de otros: **x** no tiene más parámetros que su nombre (un **String**), por lo que se ubica automáticamente en el grupo de hilos "del sistema", mientras que y está bajo x, y **z** está bajo y. Nótese que la inicialización se da exactamente en el orden textual, por lo que este código es legal.

Se crean dos hilos y se ubican en distintos grupos de hilos. **PruebaHilo1** no tiene un método **run**( ) pero tiene un **f**( ) que modifica el hilo e imprime algo, de forma que pueda verse que fue invocado. **PruebaHilo2** es una subclase de **PruebaHilol**, y su **run**( ) está bastante elaborado. Primero toma el grupo de hilos del hilo actual, después asciende dos niveles por el árbol de herencia usando **getParent**( ). (Esto se hace así porque ubicamos el objeto **PruebaHilo2** dos niveles más abajo en la jerarquía a propósito.) En este momento, se crea un array de referencias a **Threads** usando el método **activeCount**( ) para preguntar cuántos hilos están en este grupo de hilos y en todos los grupos de hilos hijo. El método **enumerate**( ) ubica referencias a todos estos hilos en el array **gTodos**, después simplemente recorremos todo el array invocando al método **f**( ) de cada hilo, además de modificar la prioridad. Además, un hilo de un grupo de hilos "hoja" modifica los hilos en los grupos de hilos padre.

El método de depuración **list**( ) imprime toda la información sobre un grupo de hilos en la salida estándar y es útil cuando se investiga el comportamiento de los grupos de hilos. He aquí la salida del programa:

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[one,5,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[two,5,z]
one f()
two f()
```

```
java.lang.ThreadGroup[name=x,maxpri=10]
    Thread[one,1,x]
    java.lang.ThreadGroup[name=y,maxpri=10]
        java.lang.ThreadGroup[name=z,maxpri=10]
            Thread[two,1,z]
```

El método **list**( ) no sólo imprime el nombre de clase de **ThreadGroup** o **Thread**, sino que también imprime el nombre del grupo de hilos y su máxima prioridad. En el caso de los hilos, se imprime el nombre del hilo, seguido de la prioridad del hilo y el grupo al que pertenece. Nótese que **list**( ) va indentando los hilos y grupos de hilos para indicar que son hijos del grupo no indentado.

Se puede ver que el método **run**( ) de **PruebaHilo2** llama a **f**( ), por lo que es obvio que todos los hilos de un grupo sean vulnerables. Sin embargo, se puede acceder sólo a los hilos que se ramifican del propio árbol de grupos de hilos sistema, y quizás a esto hace referencia el término "seguridad". No se puede acceder a ningún otro árbol de grupos de hilos del sistema.

Controlar los grupos de hilos

Dejando de lado los aspectos de seguridad, algo para lo que los grupos de hilos parecen ser útiles es para controlar: se pueden llevar a cabo ciertas operaciones en todo un grupo de hilos con un único comando. El ejemplo siguiente lo demuestra, además de las restricciones de prioridades en los grupos de hilos. Los números comentados entre paréntesis proporcionan una referencia para comparar la salida.

```
//: c14:ThreadGroup1.java
// How thread groups control priorities
// of the threads inside them.

public class ThreadGroup1 {
  public static void main(String[] args) {
    // Get the system thread & print its Info:
    ThreadGroup sys =
      Thread.currentThread().getThreadGroup();
    sys.list(); // (1)
    // Reduce the system thread group priority:
    sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
    // Increase the main thread priority:
    Thread curr = Thread.currentThread();
    curr.setPriority(curr.getPriority() + 1);
    sys.list(); // (2)
    // Attempt to set a new group to the max:
    ThreadGroup g1 = new ThreadGroup("g1");
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    // Attempt to set a new thread to the max:
    Thread t = new Thread(g1, "A");
```

```
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (3)
    // Reduce g1's max priority, then attempt
    // to increase it:
    g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
    g1.setMaxPriority(Thread.MAX_PRIORITY);
    g1.list(); // (4)
    // Attempt to set a new thread to the max:
    t = new Thread(g1, "B");
    t.setPriority(Thread.MAX_PRIORITY);
    g1.list(); // (5)
    // Lower the max priority below the default
    // thread priority:
    g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
    // Look at a new thread's priority before
    // and after changing it:
    t = new Thread(g1, "C");
    g1.list(); // (6)
    t.setPriority(t.getPriority() -1);
    g1.list(); // (7)
    // Make g2 a child Threadgroup of g1 and
    // try to increase its priority:
    ThreadGroup g2 = new ThreadGroup(g1, "g2");
    g2.list(); // (8)
    g2.setMaxPriority(Thread.MAX_PRIORITY);
    g2.list(); // (9)
    // Add a bunch of new threads to g2:
    for (int i = 0; i < 5; i++)
      new Thread(g2, Integer.toString(i));
    // Show information about all threadgroups
    // and threads:
    sys.list(); // (10)
    System.out.println("Starting all threads:");
    Thread[] all = new Thread[sys.activeCount()];
    sys.enumerate(all);
    for(int i = 0; i < all.length; i++)
      if(!all[i].isAlive())
        all[i].start();
    // Suspends & Stops all threads in
    // this group and its subgroups:
    System.out.println("All threads started");
    sys.suspend(); // Deprecated in Java 2
    // Never gets here...
    System.out.println("All threads suspended");
    sys.stop(); // Deprecated in Java 2
    System.out.println("All threads stopped");
  }
} ///:~
```

La salida que sigue se ha editado para permitir que entre en una página (se ha eliminado el
java.lang.), y para añadir números que corresponden a los números en forma de comentario del listado de arriba.

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
(2) ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
(3) ThreadGroup[name=g1,maxpri=9]
      Thread[A,9,g1]
(4) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
(5) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
      Thread[B,8,g1]
(6) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,6,g1]
(7) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,3,g1]
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
(10)ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
      ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
        ThreadGroup[name=g2,maxpri=3]
          Thread[0,6,g2]
          Thread[1,6,g2]
          Thread[2,6,g2]
          Thread[3,6,g2]
          Thread[4,6,g2]
Starting all threads:
All threads started
```

Todos los programas tienen al menos un hilo en ejecución, y lo primero que hace el método **main**( ) es llamar al método **static** de **Thread** llamado **currentThread**( ). Desde este hilo, se produce el grupo de hilos y se llama a **list**( ) para obtener el resultado. La salida es:

```
(1) ThreadGroup[name=system,maxpri=10]
      Thread[main,5,system]
```

Puede verse que el nombre del grupo de hilos principal es **system,** y el nombre del hilo principal es **main,** y pertenece al grupo de hilos **system.**

El segundo ejercicio muestra que se puede reducir la prioridad máxima del grupo **system,** y que es posible incrementar la prioridad del hilo **main:**

```
(2) ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
```

El tercer ejercicio crea un grupo de hilos nuevo, **g1,** que pertenece automáticamente al grupo de hilos **system,** puesto que no se especifica nada más. Se ubica en **g1** un nuevo hilo **A.** Después de intentar establecer la prioridad máxima del grupo y la prioridad de **A** al nivel más alto el resultado es:

```
(3) ThreadGroup[name=g1,maxpri=9]
      Thread[A,9,g1]
```

Por consiguiente, no es posible cambiar la prioridad máxima del grupo de hilos para que sea superior a la de su grupo de hilos padre.

El cuarto ejercicio reduce la prioridad máxima de g l por la mitad y después trata de incrementarla hasta **Thread.MAX_PRIORITY.** El resultado es:

```
(4) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
```

Puede verse que no funcionó el incremento en la prioridad máxima. La prioridad máxima de un grupo de hilos sólo puede disminuirse, no incrementarse. También, nótese que la prioridad del hilo **A** no varió, y ahora es superior a la prioridad máxima del grupo de hilos. Cambiar la prioridad máxima de un grupo de hilos no afecta a los hilos existentes.

El quinto ejercicio intenta establecer como prioridad de un hilo la prioridad máxima:

```
(5) ThreadGroup[name=g1,maxpri=8]
      Thread[A,9,g1]
      Thread[B,8,g1]
```

No se puede cambiar el nuevo hilo a nada superior a la prioridad máxima del grupo de hilos.
La prioridad por defecto para los hilos de este programa es seis; esa es la prioridad en la que se crearán los hilos nuevos y en la que éstos permanecerán mientras no se manipule su prioridad.

El Ejercicio 6 disminuye la prioridad máxima del grupo de hilos por debajo de la prioridad por defecto para ver qué ocurre al crear un nuevo hilo bajo esta condición:

```
(6) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,6,g1]
```

Incluso aunque la prioridad máxima del grupo de hilos es tres, el hilo nuevo se sigue creando usando la prioridad por defecto de seis. Por consiguiente, la prioridad máxima del grupo de hilos no afecta a la prioridad por defecto. (De hecho, parece no haber forma de establecer la prioridad por defecto de los hilos nuevos.)

Después de cambiar la prioridad, al intentar disminuirla en una unidad, el resultado es:

```
(7) ThreadGroup[name=g1,maxpri=3]
      Thread[A,9,g1]
      Thread[B,8,g1]
      Thread[C,3,g1]
```

La prioridad máxima de los grupos de hilos sólo se ve reforzada al intentar cambiar la prioridad.

En (8) y (9) se hace un experimento semejante creando un nuevo grupo de hilos **g2** como hijo de **g1** y cambiando su prioridad máxima. Puede verse que es imposible que la prioridad máxima de **g2** sea superior a la de **g1**:

```
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
```

Nótese que la prioridad de **g2** se pone automáticamente en la prioridad máxima del grupo de hilos **gl** en el momento de su creación.

Después de todos estos experimentos, se imprime la totalidad del sistema de grupos de hilos e hilos:

```
(10)ThreadGroup[name=system,maxpri=9]
      Thread[main,6,system]
      ThreadGroup[name=g1,maxpri=3]
        Thread[A,9,g1]
        Thread[B,8,g1]
        Thread[C,3,g1]
        ThreadGroup[name=g2,maxpri=3]
          Thread[0,6,g2]
```

```
        Thread[1,6,g2]
        Thread[2,6,g2]
        Thread[3,6,g2]
        Thread[4,6,g2]
```

Por tanto, debido a las reglas de los grupos de hilos, un grupo hijo debe tener siempre una prioridad máxima inferior o igual a la prioridad máxima de su padre.

La última parte de este programa demuestra métodos para grupos completos de hilos. En primer lugar, el programa recorre todo el árbol de hilos y pone en marcha todos los que no hayan empezado. Después se suspende y finalmente se detiene el grupo **system.** (Aunque es interesante ver que **suspend**( ) y **stop**( ) afectan a todo el grupo de hilos, habría que recordar que estos métodos se han abolido en Java 2.) Pero cuando se suspende el grupo **system** también se suspende el hilo **main,** apagando todo el programa, por lo que nunca llega al punto en el que se detienen todos los hilos. De hecho, si no se detiene el hilo **main,** éste lanza una excepción **ThreadDeath,** lo cual no es lo más habitual. Puesto que **ThreadGroup** se hereda de **Object,** que contiene el método **wait**( ), también se puede elegir suspender el programa durante unos segundos invocando a **wait**(**segundos* 1000).** Por supuesto éste debe adquirir el bloqueo dentro de un bloqueo sincronizado.

La clase **ThreadGroup** también tiene métodos **suspend**( ) y **resume**( ) por lo que se puede parar y arrancar un grupo de hilos completo y todos sus hilos y subgrupos con un único comando. (De nuevo, **suspend**( ) y **resume**( ) están en desuso en Java 2.)

Los grupos de hilos pueden parecer algo misteriosos a primera vista, pero hay que tener en cuenta que probablemente no se usarán a menudo directamente.

Volver a visitar Runnable

Anteriormente en este capítulo, sugerimos que se pensara detenidamente antes de hacer un ***applet*** o un **Frame** principal como una implementación de **Runnable.** Por supuesto, si hay que heredar de una clase y se desea añadir comportamiento basado en hilos a la clase, la solución correcta es **Runnable.** El ejemplo final de este capítulo explota esto construyendo una clase **Runnable JPanel** que pinta distintos colores por sí misma. Esta aplicación toma valores de la línea de comandos para determinar cuán grande es la rejilla de colores y cuán largo es el **sleep**( ) entre los cambios de color.

Jugando con estos valores se descubrirán algunas facetas interesantes y posiblemente inexplicables de los hilos:

```
//: c14:ColorBoxes.java
```

```
// Using the Runnable interface.
// <applet code=ColorBoxes width=500 height=400>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

class CBox extends JPanel implements Runnable {
  private Thread t;
  private int pause;
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) {
    this.pause = pause;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
      cColor = newColor();
      repaint();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}
```

```
public class ColorBoxes extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  private int pause = 50;
  public void init() {
    // Get parameters from Web page:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for (int i = 0; i < grid * grid; i++)
      cp.add(new CBox(pause));
  }
  public static void main(String[] args) {
    ColorBoxes applet = new ColorBoxes();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

CajasColores es la aplicación/applet habitual con un **hit**( ) que establece el IGU. Éste crea **GridLayout,** de forma que tenga celdas **rejilla** en cada dimensión. Después, añade el número apropiado de objetos **CajaC** para rellenar la rejilla, pasando el valor **pausa** a cada uno. En el método **main**( ) puede verse que **pausa** y **rejilla** tienen valores por defecto que pueden cambiarse si se pasan parámetros de línea de comandos, o usando parámetros del applet.
Todo el trabajo se da en **CajaC.** Ésta se hereda de **JPanel** e implementa la interfaz **Runnable** de forma que cada **JPanel** también puede ser un **Thread.** Recuérdese que al implementar **Runnable** no se hace un objeto **Thread,** sino simplemente una clase con un método **run**( ). Por consiguiente, un objeto **Thread** hay que crearlo explícitamente y pasarle el objeto **Runnable** al constructor, después llamar a **start**( ) (esto ocurre en el constructor). En **CajaC** a este hilo se le denomina **t.**

Fijémonos en el array **colores** que es una enumeración de todos los colores de la clase **Color.** Se usa en **nuevoColor**( ) para producir un color seleccionado al azar. El color de la celda actual es **celdaColor.**

El método **paintComponent**( ) es bastante simple -simplemente pone el color a **color** y rellena todo el **JPanel** con ese color. En **run**( ) se ve el bucle infinito que establece el **Color** a un nuevo color al azar y después llama a **repaint**( ) para mostrarlo. Después el hilo va a **sleep**( ) durante la cantidad de tiempo especificada en la línea de comandos.

Precisamente porque este diseño es flexible y la capacidad de hilado está vinculada a cada elemento **JPanel,** se puede experimentar construyendo tantos hilos como se desee. (Realmente, hay una restricción impuesta por la cantidad de hilos que puede manejar cómodamente la JVM.)

Este programa también hace una medición interesante, puesto que puede mostrar diferencias de rendimiento drásticas entre una implementación de hilos de una JVM y otra.

Demasiados hilos

En algún momento, se verá que **CajasColores** se colapsa. En nuestra máquina esto ocurre en cualquier lugar tras una rejilla de 10 + 10. ¿Por qué ocurre esto?

Uno sospecha, naturalmente, que Swing debería estar relacionado con esto, por lo que hay un ejemplo que prueba esa premisa construyendo menos hilos. El siguiente código se ha reorganizado de forma que un **ArrayList implemente Runnable** y un **ArrayList** guarde un número de bloques de colores y elija al azar los que va a actualizar. Después, se crea un número de estos objetos **ArrayList,** dependiendo de la dimensión de la rejilla que se pueda elegir. Como resultado, se tienen bastantes menos hilos que bloques de color, por lo que si se produce un incremento de velocidad se sabrá que se debe a que hay menos hilos que en el ejemplo anterior:

```
//: c14:ColorBoxes2.java
// Balancing thread use.
// <applet code=ColorBoxes2 width=600 height=500>
// <param name=grid value="12">
// <param name=pause value="50">
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

class CBox2 extends JPanel {
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
```

```
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  void nextColor() {
    cColor = newColor();
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
}

class CBoxList
  extends ArrayList implements Runnable {
  private Thread t;
  private int pause;
  public CBoxList(int pause) {
    this.pause = pause;
    t = new Thread(this);
  }
  public void go() { t.start(); }
  public void run() {
    while(true) {
      int i = (int)(Math.random() * size());
      ((CBox2)get(i)).nextColor();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
  public Object last() { return get(size() - 1);}
}

public class ColorBoxes2 extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  // Shorter default pause than ColorBoxes:
  private int pause = 50;
  private CBoxList[] v;
```

```
  public void init() {
    // Get parameters from Web page:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    v = new CBoxList[grid];
    for(int i = 0; i < grid; i++)
      v[i] = new CBoxList(pause);
    for (int i = 0; i < grid * grid; i++) {
      v[i % grid].add(new CBox2());
      cp.add((CBox2)v[i % grid].last());
    }
    for(int i = 0; i < grid; i++)
      v[i].go();
  }
  public static void main(String[] args) {
    ColorBoxes2 applet = new ColorBoxes2();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
} ///:~
```

En **CajaColores2** se crea un array de **ListaCajaC** inicializándose para guardar la **rejilla ListascajaC,** cada uno de los cuales sabe durante cuánto tiempo dormir. Posteriormente se añade un número igual de objetos **CajaC2** a cada **ListaCajaC,** y se dice a cada lista que **comenzar**( ), lo cual pone en marcha el hilo.

CajaC2 es semejante a **CajaC:** se pinta ListaCajaC a sí misma con un color elegido al azar. Pero esto es todo lo que hace un **CajaC2.** Toda la gestión de hilos está ahora en **ListaCajaC.**
ListaCajaC también podría haber heredado **Thread** y haber tenido un objeto miembro de tipo
ArrayList. Ese diseño tiene la ventaja de que los métodos **add**( ) y **get**( ) podrían recibir posteriormente un argumento específico y devolver tipos de valores en vez de **Objects** genéricos. (También se podrían cambiar sus nombres para que sean más cortos.) Sin embargo, el diseño usado aquí parecía a primera vista requerir menos código. Además, retiene automáticamente todos los demás

comportamientos de un **ArrayList** Con todas las conversiones y paréntesis necesarios para **get**( ), éste podría no ser el caso a medida que crece el cuerpo del código.

Como antes, al implementar **Runnable** no se logra todo el equipamiento que viene con **Thread,** por lo que hay que crear un nuevo **Thread** y pasárselo explícitamente a su constructor para tener algo en **start**( ), como puede verse en el constructor **ListaCajaC** y en **comenzar**( ). El método **run**( ) simplemente elige un número de elementos al azar dentro de la lista y llama al **siguientecolor**( ) de ese elemento para que elija un nuevo color seleccionado al azar. Al ejecutar este programa se ve que, de hecho, se ejecuta más rápido y responde más rápidamente (por ejemplo, cuando es interrumpido, se detiene más rápidamente) y no parece saturarse tanto en tamaños de rejilla grandes. Por consiguiente, se añade un nuevo factor a la ecuación de hilado: hay que vigilar para ver que no se tengan "demasiados hilos" (sea lo que sea lo que esto signifique para cada programa y plataforma en particular -aquí, la ralentización de **CajasColores** parece estar causada por el hecho de que sólo hay un hilo que es responsable de todo el pintado, y que se colapsa por demasiadas peticiones). Si se tienen demasiados hilos hay que intentar usar técnicas como la de arriba para "equilibrar" el número de hilos del programa.

Si se aprecian problemas de rendimiento en un programa multihilo, hay ahora varios aspectos que examinar:

1. ¿Hay suficientes llamadas a **sleep(**), **yield(**) y/o **wait(**)?
2. ¿Son las llamadas a **sleep(**) lo suficientemente rápidas?
3. ¿Se están ejecutando demasiados hilos?
4. ¿Has probado distintas plataformas y JVMs?

Aspectos como éste son la razón por la que a la programación multihilo se le suele considerar un arte.

Resumen

Es vital aprender cuándo hacer uso de capacidades multihilo y cuándo evitarlas. La razón principal de su uso es gestionar un número de tareas que al entremezclarse hagan un uso más eficiente del ordenador (incluyendo la habilidad de distribuir transparentemente las tareas a través de múltiples UCP), o ser más conveniente para el usuario. El ejemplo clásico de balanceo de recursos es usar la UCP durante las esperas de E/S. El ejemplo clásico de conveniencia del usuario es monitorizar un botón de "detención" durante descargas largas.

Las desventajas principales del multihilado son:

1. Ralentización durante la espera por recursos compartidos.

2. Sobrecarga adicional de la UCP necesaria para gestionar los hilos.
3. Complejidad sin recompensa, como la tonta idea de tener un hilo separado para actualizar cada elemento de un array.
4. Patologías que incluyen la inanición, la competición y el interbloqueo.

Una ventaja adicional de los hilos es que sustituyen a las conmutaciones de contexto de ejecución "ligera" (del orden de 100 instrucciones) por conmutaciones de contexto de ejecución "pesada" (del orden de miles de instrucciones). Puesto que todos los hilos de un determinado proceso comparten el mismo espacio de memoria, una conmutación de proceso ligera sólo cambia la ejecución del programa y las variables locales. Por otro lado, un cambio de proceso -la conmutación de contexto pesada- debe intercambiar todo el espacio de memoria.

El multihilado es como irrumpir paso a paso en un mundo completamente nuevo y aprender un nuevo lenguaje de programación o al menos un conjunto de conceptos de lenguaje nuevos. Con la apariencia de soporte a hilos, en la mayoría de sistemas operativos de microcomputador han ido apareciendo extensiones para hilos en lenguajes de programación y bibliotecas. En todos los casos, la programación de hilos (1) parece misteriosa y requiere un cambio en la forma de pensar al programar; y (2) parece similar al soporte de hilos en otros lenguajes, por lo que al entender los hilos se entiende una lengua común. Y aunque el soporte de hilos puede hacer que Java parezca un lenguaje más complicado, no hay que echar la culpa a Java. Los hilos son un truco.

Una de las mayores dificultades con los hilos se debe a que, dado que un recurso -como la memoria de un objeto- podría estar siendo compartido por más de un hilo, hay que asegurarse de que múltiples hilos no intenten leer y cambiar ese recurso simultáneamente. Esto requiere de un uso juicioso de la palabra clave **synchronized,** que es una herramienta útil pero que debe ser totalmente comprendida puesto que puede presentar silenciosamente situaciones de interbloqueos.

Además, hay determinado arte en la aplicación de los hilos. Java está diseñado para permitir la creación de tantos objetos como se necesite para solucionar un problema -al menos en teoría. (Crear millones de objetos para un análisis de elementos finitos de ingeniería, por ejemplo, podría no ser práctico en Java.) Sin embargo, parece que hay un límite superior al número de hilos a crear, puesto que en algún momento, un número de hilos más elevado da muestras de colapso. Este punto crítico no se alcanza con varios miles, como en el caso de los objetos, sino en unos pocos cientos, o incluso a veces menos de 1.200. Como a menudo sólo se crea un puñado de hilos para solucionar un problema, este límite no suele ser tal, aunque puede parecer una limitación en diseños generales.

Un aspecto significativo y no intuitivo de los hilos es que, debido a la planificación de los hilos, se puede hacer que una aplicación se ejecute generalmente *más*

rápido insertando llamadas a **sleep**( ) dentro del bucle principal de **run**( ). Esto hace, sin duda, que su uso parezca un arte, especialmente cuando los retrasos más largos parecen incrementar el rendimiento. Por supuesto, la razón por la que ocurre esto es que retrasos más breves pueden hacer que la interrupción del planificador del final del **sleep**( ) se dé antes de que el hilo en ejecución esté listo para ir a dormir, forzando al planificador a detenerlo y volver a arrancarlo más tarde para que pueda acabar con lo que estaba haciendo, para ir después a dormir. Hay que pensar bastante para darse cuenta en lo complicadas que pueden ser las cosas.

Algo que alguien podría echar de menos en este capítulo es un ejemplo de animación, que es una de las cosas más populares que se hacen con los *applets.* Sin embargo, con el Java JDK (disponible en *http:/7java.sun.com)* viene la solución completa (con sonido) a este problema, dentro de la sección de demostración. Además, se puede esperar que en las versiones futuras de Java se incluya un mejor soporte para animaciones, a la vez que están apareciendo distintas soluciones de animación para la Web, no Java, y que no sean de programación, que pueden ser superiores a los enfoques tradicionales. Si se desean explicaciones de cómo funcionan las animaciones en Java, puede verse *Core Java* 2, de Horstmann & Cornell, Prentice-Hall, 1997. Para acceder a discusiones más avanzadas en el multihilado, puede verse *Concurrent Programming in Java,* de Doug Lea, Addison-Wesley, 1997, o *Java Threads* de Oaks & Wong, O'Reilly, 1997.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide,* disponible a bajo coste en *http://www.BruceEckel.com.*

1. Heredar una clase de **Thread** y superponer el método **run(**). Dentro de **run(**), imprimir un mensaje y llamar después a **sleep(**). Repetir esto tres veces, después de volver de **run(**) finalice. Poner un mensaje de empiece en el constructor y superponer **finalize(**) para imprimir un mensaje de apagado. Hacer una clase hilo separada que llame a **System.gc(**) y **System.runFinalization(**) dentro de **run(**), imprimiendo un mensaje a medida que lo hace. Hacer varios objetos hilo de ambos tipos y ejecutarlos para ver qué ocurre.

2. Modificar **Compartiendo2.java** para añadir un bloque **synchronized** dentro del método **run** ( ) de **Doscontadores** en vez de sincronizar todo el método **run(**).

3. Crear dos subclases **Thread,** una con un **run(**) que empiece, capture la referencia al segundo objeto **Thread** y llame después a **wait(**). El método **run(**) de la otra clase debería llamar a **notify.All(**) para el primer hilo, tras haber pasado cierto número de segundos, de forma que el primer hilo pueda imprimir un mensaje.

4. En **Contador5.java** dentro de **Teletipo2,** retirar el **yield(**) y explicar los resultados. Reemplazar el **yield(**) con un **sleep(**) y explicar los resultados.

5. En **grupoHilos.java,** reemplazar la llamada a **sis.suspend(**) con una llamada a **wait(**) para el grupo de hilos, haciendo que espere durante dos segundos. Para que esto funcione correctamente hay que adquirir el bloqueo de **sis** dentro de un bloque **synchronized.**

6. Cambiar **Demonios.java,** de forma que **main(**) tenga un **sleep(**) en vez de un **readllne**( ). Experimentar con distintos tiempos en la **sleep(**) para ver qué ocurre.

7. En el Capítulo 8, localizar el ejemplo **ControlesInvernadero.java,** que consiste en tres archivos. En **Evento.java,** la clase **Evento** se basa en vigilar el tiempo. Cambiar **Evento** de forma que sea un **Thread,** y cambiar el resto del diseño de forma que funcione con este nuevo **Evento** basado en **Thread.**

8. Modificar el Ejercicio 7, de forma que se use la clase **java.util.Timer** del JDK 1.3 para ejecutar el sistema.

9. A partir de **OndaSeno.java** del Capítulo 14, crear un programa (un applet/aplicación usando la clase **Console)** que dibuje una onda seno animada que parezca desplazarse por la ventana como si fuera un osciloscopio, dirigiendo la animación con un **Thread.** La velocidad de la animación debería controlarse con un control **java.swingJSlider.**

10. Modificar el Ejercicio 9, de forma que se creen varios paneles onda seno dentro de la aplicación. El número de paneles debería controlarse con etiquetas HTML o parámetros de línea de comando.

11. Modificar el Ejercicio 9, de forma que se use la clase **java.swing.Timer** para dirigir la animación. Nótese la diferencia entre éste y **java.util.Timer.**

14: Crear ventanas y applets

Una guía de diseño fundamental es "haz las cosas simples de forma sencilla, y las cosas difíciles hazlas posibles." [1]

[1] Hay una variación de este dicho que se llama "el principio de asombrarse al mínimo", que dice en su esencia: "No sorprenda al usuario".

La meta original de diseño de la biblioteca de interfaz gráfico de usuario (IGU) en Java 1.0 era permitir al programador construir un IGU que tuviera buen aspecto en todas las plataformas. Esa meta no se logró. En su lugar, el Abstract Window Toolkit (AWT) de Java 1.0 produce un IGU que tiene una apariencia igualmente mediocre en todos los sistemas. Además, es restrictivo: sólo se pueden usar cuatro fuentes y no se puede acceder a ninguno de los elementos de IGU más sofisticados que existen en el sistema operativo. El modelo de programación de AWT de Java 1.0 también es siniestro y no es orientado a objetos. En uno de mis seminarios, un estudiante (que había estado en Sun durante la creación de Java) explicó el porqué de esto: el AWT original había sido conceptualizado, diseñado e implementado en un mes. Ciertamente es una maravilla de la productividad, además de una lección de por qué el diseño es importante.
La situación mejoró con el modelo de eventos del AWT de Java 1.1, que toma un enfoque orientado a objetos mucho más claro, junto con la adición de JavaBeans, un modelo de programación basado en componentes orientado hacia la creación sencilla de entornos de programación visuales. Java 2 acaba esta transformación alejándose del AWT de Java 1.0 esencialmente, reemplazando todo con las Java Foundation Classes (JFC), cuya parte IGU se denomina "Swing". Se trata de un conjunto de JavaBeans fáciles de usar, y fáciles de entender que pueden ser arrastrados y depositados (además de programados a mano) para crear un IGU con el que uno se encuentre finalmente satisfecho. La regla de la "revisión 3" de la industria del software (un producto no es bueno hasta su tercera revisión) parece cumplirse también para los lenguajes de programación.

Este capítulo no cubre toda la moderna biblioteca Swing de Java 2, y asume razonablemente que Swing es la biblioteca IGU destino final de Java. Si por alguna razón fuera necesario hacer uso del "viejo" AWT (porque se está intentando dar soporte a código antiguo o se tienen limitaciones impuestas por el navegador), es posible hacer uso de la introducción que había en la primera edición de este libro, descargable de http://www.BruceEcke1.com (incluida también en el CD ROM que se adjunta a este libro).

Al principio de este capítulo veremos cómo las cosas son distintas si se trata de crear un applet o si se trata de crear una aplicación ordinaria haciendo uso de Swing, y cómo crear programas que son tanto applets como aplicaciones, de forma que se pueden ejecutar bien dentro de un browser, o bien desde línea de comandos. Casi todos los ejemplos IGU de este libro serán ejecutables bien como applet, o como aplicaciones.

Hay que ser conscientes de que este capítulo no es un vocabulario exhaustivo de los componentes Swing o de los métodos de las clases descritas. Todo lo que aquí se presenta es simple a propósito.

La biblioteca Swing es vasta y la meta de este capítulo es sólo introducirnos con la parte esencial y más agradable de los conceptos. Si se desea hacer más, Swing puede proporcionar lo que uno desee siempre que uno se enfrente a investigarlo.

Asumimos que se ha descargado e instalado la documentación HTML de las bibliotecas de Java (que es gratis) de *http://jaua.sun.com* y que se navegará a lo largo de las clases **javax.swing** de esa documentación para ver los detalles completos y los métodos de la biblioteca Swing. Debido a la simplicidad del diseño Swing, generalmente esto será suficiente información para solucionar todos los problemas. Hay numerosos libros (y bastante voluminosos) dedicados exclusivamente a Swing y son altamente recomendables si se desea mayor nivel de profundidad, o cuando se desee cambiar el comportamiento por defecto de la biblioteca Swing.

A medida que se aprendan más cosas sobre Swing se descubrirá que: Swing es un modelo de programación mucho mejor que lo que se haya visto probablemente en otros lenguajes y entornos de desarrollo. Los JavaBeans (que no se presentarán hasta el final de este capítulo) constituyen el marco de esa biblioteca.

Los "constructores IGU (entornos de programación visual) son un aspecto *de rigueur* de un entorno de desarrollo Java completo. Los JavaBeans y Swing permiten al constructor IGU escribir código por nosotros a medida que se ubican componentes dentro de formularios utilizando herramientas gráficas. Esto no sólo acelera rápidamente el desarrollo utilizando construcción IGU, sino que permite un nivel de experimentación mayor, y por consiguiente la habilidad de probar más diseños y acabar generalmente con uno mejor.

La simplicidad y la tan bien diseñada naturaleza de Swing significan que incluso si se usa un constructor IGU en vez de codificar a mano, el código resultante será más completo -esto soluciona un gran problema con los constructores IGU del pasado, que podían generar de forma sencilla código ilegible.

Swing contiene todos los componentes que uno espera ver en un IU moderno, desde botones con dibujos hasta árboles y tablas. Es una gran biblioteca, pero

está diseñada para tener la complejidad apropiada para la tarea a realizar -es algo simple, no hay que escribir mucho código, pero a medida que se intentan hacer cosas más complejas, el código se vuelve proporcionalmente más complejo. Esto significa que nos encontramos ante un punto de entrada sencillo, pero se tiene a mano toda la potencia necesaria.

A mucho de lo que a uno le gustaría de Swing se le podría denominar "ortogonalidad de uso". Es decir, una vez que se captan las ideas generales de la biblioteca, se pueden aplicar en todas partes.

En primer lugar, gracias a las convenciones estándar de denominación, muchas de las veces, mientras se escriben estos ejemplos, se pueden adivinar los nombres de los métodos y hacerlo bien a la primera, sin investigar nada más. Ciertamente éste es el sello de un buen diseño de biblioteca.

Además, generalmente se pueden insertar componentes a los componentes ya existentes de forma que todo funcione correctamente.

En lo que a velocidad se refiere, todos los componentes son "ligeros", y Swing se ha escrito completamente en Java con el propósito de lograr portabilidad.

La navegación mediante teclado es automática -se puede ejecutar una aplicación Java sin usar el ratón, y no es necesaria programación extra. También se soporta desplazamiento sin esfuerzo -simplemente se envuelven los componentes en un **JScrollPane** a medida que se añaden al formulario. Aspectos como etiquetas de aviso simplemente requieren de una línea de código.

Swing también soporta una faceta bastante más radical denominada *"apariencia* conectable" que significa que se puede cambiar dinámicamente la apariencia del IU para que se adapte a las expectativas de los usuarios que trabajen en plataformas y sistemas operativos distintos. Incluso es posible (aunque difícil) inventar una apariencia propia.

El applet básico

Una de las metas de diseño de Java es la creación de *applets,* que son pequeños programas que se ejecutan dentro del navegador web. Dado que tienen que ser seguros, se limitan a lo que pueden lograr. Sin embargo, los *applets* son una herramienta potente que soporta programación en el lado cliente, uno de los aspectos fundamentales de la Web.

Restricciones de applets

La programación dentro de un *applet* es tan restrictiva que a menudo se dice que se está "dentro de una caja de arena" puesto que siempre se tiene a alguien -es decir, el sistema de seguridad de tiempo de ejecución de Java- vigilando.

Sin embargo, uno también se puede salir de la caja de arena y escribir aplicaciones normales en vez de *applets,* en cuyo caso se puede acceder a otras facetas del S.O. Hemos estado escribiendo aplicaciones normales a lo largo de todo este libro, pero han sido *aplicaciones de consola* sin componentes gráficos. También se puede usar Swing para construir interfaces IGU para aplicaciones normales.

Generalmente se puede responder a la pregunta de qué es lo que un *applet* puede hacer mirando a lo que *se supone* que hace: extender la funcionalidad de una página web dentro de un navegador. Puesto que, como navegador de la Red, nunca se sabe si una página web proviene de un sitio amigo o no, se desea que todo código que se ejecute sea seguro. Por tanto, las mayores restricciones que hay que tener en cuenta son probablemente:

1. *Un applet no puede tocar el disco local.* Esto significa escribir *o* leer, puesto que no se desearía que un *applet* pudiera leer y transmitir información privada a través de Internet sin permiso. Se evita la escritura, por supuesto, dado que supondría una invitación abierta a los virus. Java ofrece *firmas digitales* para *applets.* Muchas restricciones de *applets* se suavizan cuando se elige la ejecución de *applets de confianza* (los firmados por una fuente de confianza) para tener acceso a la máquina.

2. *Puede llevar más tiempo mostrar los applets,* puesto que hay que descargarlos por completo cada vez, incluyendo una solicitud distinta al servidor por cada clase diferente. El navegador puede introducir el *applet* en la caché, pero no hay ninguna garantía de que esto ocurra. Debido a esto, probablemente habría que empaquetar los *applets* en un JAR (Archivo Java) que combina todos los componentes del *applet* (incluso otros archivos **.class** además de imágenes y sonidos) dentro de un único archivo comprimido que puede descargarse en una única transacción servidora. El "firmado digital" está disponible para cada entrada digital de un archivo JAR.

Ventajas de los applets

Si se puede vivir con las restricciones, los *applets* tienen también ventajas significativas cuando se construyen aplicaciones cliente/servidor u otras aplicaciones en red:

1. *No hay aspectos de instalación.* Un *applet* tiene independencia completa de la plataforma (incluyendo la habilidad de reproducir sin problemas archivos de sonido, etc.) por lo que no hay que hacer ningún cambio en el código en función de la plataforma ni hay que llevar a cabo ninguna "adaptación" en el momento de la instalación. De hecho, la instalación es automática cada vez que el usuario carga la página web que contiene *applets,* de forma que las actualizaciones se darán de forma inadvertida y

automáticamente. En los sistemas cliente/servidor tradicionales, construir e instalar nuevas versiones del software cliente es siempre una auténtica pesadilla.

2. *No hay que preocuparse de que el código erróneo cause ningún mal a los sistemas de alguien,* gracias a la propia seguridad implícita en la esencia de Java y en la estructura de los *applets*. Esto, junto con el punto anterior, convierte a Java en un lenguaje popular para las denominadas aplicaciones cliente/servidor de *intranet* que sólo residen dentro de un campo de operación restringido dentro de una compañía donde se puede especificar y/o controlar el entorno del usuario (el navegador web y sus añadidos).

Debido a que los ***applets*** se integran automáticamente con el **HTML,** hay que incluir un sistema de documentación embebido independiente de la plataforma para dar soporte al ***applet.*** Se trata de un problema interesante, puesto que uno suele estar acostumbrado a que la documentación sea parte del programa en vez de al revés.

Marcos de trabajo de aplicación

Las bibliotecas se agrupan generalmente dependiendo de su funcionalidad. Algunas bibliotecas, por ejemplo, se usan como tales, independientemente. Las clases **String** y **ArrayList** de la biblioteca estándar de Java son ejemplos de esto. Otras bibliotecas están diseñadas específicamente como bloques que permiten construir otras clases. Cierta categoría de biblioteca es el ***marco de trabajo de aplicación,*** cuya meta es ayudar en la construcción de aplicaciones proporcionando una clase o un conjunto de clases que producen el comportamiento básico deseado para toda aplicación de un tipo particular. Posteriormente, para adaptar el comportamiento a nuestras propias necesidades, hay que heredar de la clase aplicación y superponer los métodos que interesen. El marco de trabajo de aplicación es un buen ejemplo de "separar las cosas que cambian de las que permanecen invariables", puesto que intenta localizar todas las partes únicas de un programa en los métodos superpuestos **[2]**.

[2] Éste es un ejemplo del patrón de diseño denominado *método* plantilla.

Los ***applets*** se construyen utilizando un marco de trabajo de aplicación. Se hereda de **JApplet** y se superponen los métodos apropiados. Hay unos pocos métodos que controlan la creación y ejecución de un ***applet*** en una página web:

Método	Operación
init( )	Se invoca automáticamente para lograr la primera inicialización del *applet*, incluyendo la disposición de los componentes. Este método siempre se superpone.
start( )	Se invoca cada vez que se visualiza un *applet* en el

Método	Operación
	navegador para permitirle empezar sus operaciones normales (especialmente las que se apagan con **stop**()).También se invoca tras **init**().
stop( )	Se invoca cada vez que un *applet* se aparta de la vista de un navegador web para permitir al *applet* apagar operaciones caras. Se invoca también inmediatamente antes de **destroy**().
destroy( )	Se invoca cada vez que se está descargando un *applet* de una página para llevar a cabo la liberación final de recursos cuando se deja de usar el *applet*.

Con esta información ya se puede crear un ***applet*** simple:

```
//: c13:Applet1.java
// Very simple applet.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
} ///:~
```

Nótese que no se exige a los ***applets*** tener un método **main**( ). Todo se incluye en el marco de trabajo de aplicación; el código de arranque se pone en el **init**( ).

En este programa, la única actividad que se hace es poner una etiqueta de texto en el ***applet,*** vía la clase **JLabel** (la vieja **AWT** se apropió del nombre **Label** además de otros nombres de componentes, por lo que es habitual ver que los componentes Swing empiezan por una **"J").** El constructor de esta clase toma un **Stnng** y lo usa para crear la etiqueta. En el programa de arriba se coloca esta etiqueta en el formulario.

El método **init**( ) es el responsable de poner todos los componentes en el formulario haciendo uso del método **add**( ). Se podría pensar que debería ser posible invocar simplemente a **add**( ) por sí mismo, y de hecho así solía ser en el antiguo AWT. Sin embargo, Swing requiere que se añadan todos los componentes al "panel de contenido" de un formulario, y por tanto hay que invocar a **getContentPane**( ) como parte del proceso **add**( ).

Ejecutar applets dentro de un navegador web

Para ejecutar este programa, hay que ubicarlo dentro de una página web y ver esa página dentro de un navegador web con Java habilitado. Para ubicar un ***applet*** dentro de una página web se pone una etiqueta especial dentro de la fuente HTML de esa página web **[3]** para indicar a la misma cómo cargar y ejecutar el ***applet.***

[3] Se asume que el lector está familiarizado con lo básico de HTML. No es demasiado difícil, y hay cientos de libros y otros recursos.

Este proceso era muy simple cuando Java en sí era simple y todo el mundo estaba en la misma 1ínea e incorporaba el mismo soporte Java en sus navegadores web. Se podría haber continuado simplemente con un fragmento muy pequeño de código HTML dentro de la página web, así:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Después vinieron las guerras de navegadores y lenguajes y perdimos nosotros (los programadores y los usuarios finales). Tras cierto periodo de tiempo, JavaSoft se dio cuenta de que no podía esperar a que los navegadores siguieran soportando el buen gusto de Java, y la única solución era proporcionar algún tipo de añadido que se relacionara con el mecanismo de extensión del navegador. Utilizando el mecanismo de extensión (que los vendedores de navegadores no pueden deshabilitar -en un intento de ganar ventaja competitiva- sin romper con todas las extensiones de terceros), JavaSoft garantiza que un vendedor antagonista no pueda arrancar Java de su navegador web.

Con Internet Explorer, el mecanismo de extensión es el control ActiveX, y con Netscape, los *plugins*. He aquí el aspecto que tiene la página HTML más sencilla para **Applet [4]:**

[4] Esta página -en particular la porción "clsid"- parecía funcionar bien tanto con JDK 1.2.2 como JDK 1.3 rc-1. Sin embargo, puede ser que en el futuro haya que cambiar algo la etiqueta. Pueden encontrar detalles al respecto en *java.sun.com.*

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="100"             height="50"             align="baseline"
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-
1_2_2-win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM           NAME="type"           VALUE="application/x-java-
applet;version=1.2.2">
```

```
<COMMENT>
  <EMBED type=
    "application/x-java-applet;version=1.2.2"
    width="200" height="200" align="baseline"
    code="Applet1.class" codebase="."
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html">
  <NOEMBED>
</COMMENT>
   No Java 2 support for APPLET!!
  </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Algunas de estas líneas eran demasiado largas por lo que fue necesario envolverlas para que encajaran en la página. El código del código fuente de este libro (que se encuentra en el CD ROM de este libro, y se puede descargar de http://www.BruceEcke1.com) funcionará sin que haya que preocuparse de corregir estos envoltorios de líneas.

El valor **code** da el nombre del archivo **.class** en el que reside el *applet.* Los valores **width** y **height** especifican el tamaño inicial del *applet* (en píxeles, como antiguamente). Hay otros elementos que se pueden ubicar dentro de la etiqueta *applet:* un lugar en el que encontrar otros archivos **.class** en Internet (**codebase**), información de alineación (**align**), un identificador especial que posibilita la intercomunicación de los *applets* (**name**), y parámetros de *applet* que proporcionan información sobre lo que ese *applet* puede recuperar. Los parámetros son de la forma:

```
<param name="identifier" value = "information">
```

y puede haber tantos como uno desee.

El paquete de código fuente de este libro proporciona una página HTML por cada *applet* de este **li**bro, y por consiguiente muchos ejemplos de la etiqueta *applet.* Se pueden encontrar descripciones completas de los detalles de ubicación de los *applets* en las páginas web de *http://java.sun.com.*

Utilizar *Appletviewer*

El **JDK** de Sun (descargable gratuitamente de http://www.sun.com) contiene una herramienta denominada el *Appletviewer* que extrae las etiquetas <appleb del archivo HTML y ejecuta los *applets* sin mostrar el texto HTML que les rodea.

Debido a que el ***Appletviewer*** ignora todo menos las etiquetas APPLET, se puede poner esas etiquetas en el archivo fuente Java como comentarios:

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

De esta forma, se puede ejecutar **"appletviewer MiApplet.java"** y no hay que crear los pequeños archivos HTML para ejecutar pruebas. Por ejemplo, se pueden añadir las etiquetas HTML comentadas a **Applet1.java:**

```
//: c13:Applet1b.java
// Embedding the applet tag for Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
} ///:~
```

Ahora, se puede invocar al ***applet*** con el comando:

```
appletviewer Applet1b.java
```

En este libro, se usará esta forma para probar los ***applets*** de manera sencilla. En breve, se verá otro enfoque de codificación que permite ejecutar ***applets*** desde la línea de comandos sin el ***Appletviewer:***

Probar applets

Se puede llevar a cabo una prueba sencilla sin tener ninguna conexión de red, arrancando el navegador web y abriendo el archivo HTML que contiene la etiqueta ***applet.*** **Al** cargar el archivo HTML, el navegador descubrirá la etiqueta ***applet*** y tratará de acceder al archivo **.class** especificado por el valor **code.** Por supuesto, miremos primero la variable CLASSPATH para saber dónde buscar, y si el archivo **.class** no está en el CLASSPATH aparecerá un mensaje de error en la línea de estado del navegador indicando que no se puede encontrar ese archivo **.class.**

Cuando se desea probar esto en un sitio web, las cosas son algo más complicadas. En primer lugar, hay que ***tener*** un sitio web, lo que para la gran mayoría de la gente es un Proveedor de Servicios de Internet **[5]**, un tercero, en una ubicación remota. Puesto que el ***applet*** es simplemente un archivo o un

conjunto de archivos, el ISP no tiene que proporcionar ningún soporte especial para Java. También

[5] N. del traductor: En inglés. *Internet Service Provider* o *ISP.*

hay que tener alguna forma de mover los archivos HTML y **.class** desde un sistema al directorio correcto de la máquina ISP Esto se suele hacer con un programa de FTP (File Transfer Protocol), de los que existen muchísimos disponibles gratuitamente o ***shareware.*** Por tanto, podría parecer que simplemente hay que mover los archivos a la máquina ISP con FTP, después conectarse al sitio y solicitar el archivo HTML utilizando el navegador; si aparece el ***applet*** y funciona, entonces todo probado, ¿verdad?

He aquí donde uno puede equivocarse. Si el navegador de la máquina cliente no puede localizar el archivo **.class** en el servidor, buscará en el *CLASSPATH* de la máquina ***local.*** Por consiguiente, podría ocurrir que no se esté cargando adecuadamente el ***applet*** desde el servidor, pero todo parece ir bien durante la prueba porque el navegador lo encuentra en la máquina local. Sin embargo, al conectarse otro, puede que su navegador no lo encuentre. Por tanto, al probar hay que asegurarse de borrar los archivos **.class** relevantes (o el archivo **.jar**) de la máquina local para poder verificar que existan en la ubicación adecuada dentro del servidor.

Uno de los lugares más insidiosos en los que me ocurrió esto es cuando inocentemente ubiqué un ***applet*** dentro de un **package**. Tras ubicar el archivo HTML y el ***applet*** en el servidor, resultó que el servidor no encontraba la trayectoria al ***applet*** debido al nombre del paquete. Sin embargo, mi navegador lo encontró en el ***CLASSPATH*** local. Por tanto, yo era el único que podía cargar el ***applet*** adecuadamente. Me llevó bastante tiempo descubrir que el problema era la sentencia **package**. En general, es recomendable no incorporar sentencias **package** con los ***applets.***

Ejecutar ***applets*** desde la línea de comandos

Hay veces en las que se desearía hacer un programa con ventanas para algo más que para que éste resida en una página web. Quizás también se desearía hacer alguna de estas cosas en una aplicación "normal" a la vez que seguir disponiendo de la portabilidad instantánea de Java. En los capítulos anteriores de este libro hemos construido aplicaciones de línea de comandos, pero en algunos entornos operativos (como Macintosh, por ejemplo) no hay línea de comandos. Por tanto, por muchas razones se suele desear construir programas con ventanas pero sin ***applets*** haciendo uso de Java. Éste es verdaderamente un deseo muy codiciado.

La biblioteca Swing permite construir aplicaciones que preserven la apariencia del entorno operativo subyacente. Si se desea construir aplicaciones con ventanas, tiene sentido hacerlo **[5]** sólo si se puede hacer uso de la última versión de Java y herramientas asociadas de forma que se puedan entregar las aplicaciones sin necesidad de liar a los usuarios. Si por alguna razón uno se ve forzado a usar versiones antiguas de Java, que se lo piense bien antes de acometer la construcción de una aplicación con ventanas, especialmente, si tiene un tamaño mediano o grande.

[5] En mi opinión, y después de aprender Swing, no se deseará perder el tiempo con versiones anteriores.

A menudo se deseará ser capaz de crear clases que puedan invocarse bien como ventanas o bien como applets. Esto es especialmente apropiado cuando se prueban los applets, pues es generalmente mucho más rápido y sencillo ejecutar la aplicación-applet resultante desde la línea de comandos que arrancar el navegador web o el **Appletviewer**.

Para crear un applet que pueda ejecutarse desde la línea de comandos de la consola, simplemente hay que añadir un método **main**( ) al applet que construya una instancia del applet dentro de un **JFrame [6].** Como ejemplo sencillo, he aquí **Appletlb.java** modificado para que funcione como applet y como aplicación:

[6] Como se describió anteriormente, el AWT ya tenia el término "**Frame**" por lo que Swing usa el nombre **JFrame**.

```
//: c13:Applet1c.java
// An application and an applet.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1c extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
  // A main() for the application:
  public static void main(String[] args) {
    JApplet applet = new Applet1c();
    JFrame frame = new JFrame("Applet1c");
    // To close the application:
    Console.setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(100,50);
    applet.init();
```

```
    applet.start();
    frame.setVisible(true);
  }
} ///:~
```

Sólo se ha añadido al applet el método **main**( ), y el resto permanece igual. Se crea el applet y se añade a un **JFrame** para que pueda ser mostrado.

La línea:

```
Console.setupClosing(frame);
```

hace que se cierre convenientemente la ventana. **Console** proviene de **com.bruceekcel.swing** y será explicada algo más tarde.

Se puede ver que en el método **main**( ) se minimiza explícitamente el *applet* y se arranca, puesto que en este caso no tenemos un navegador que lo haga automáticamente. Por supuesto, así no se logra totalmente el comportamiento del navegador, que también llama a **stop**( ) y **destroy**( ), pero es aceptable en la mayoría de situaciones. Si esto constituye un problema es posible forzar uno mismo estas llamadas **[7]**.

[7] Esto tendrá sentido una vez que se haya avanzado en este capitulo. En primer lugar, se hace la referencia **JApplet** miembro **static** de la clase (en vez de una variable local del **main**()), y después se invoca a **applet.stop**() y **applet.destroy**() dentro de **WindowsAdapter.windowClosing**() antes de invocar a **System.exit**().

Nótese la última línea:

```
frame.setVisible(true);
```

Sin ella, no se vería nada en pantalla.

Un marco de trabajo de visualización

Aunque el código que convierte programas en *applets* y aplicaciones a la vez produce resultados de gran valor, si se usa en todas partes, se convierte en una distracción y un derroche de papel. En vez de esto, se usará el siguiente marco de trabajo de visualización para los ejemplos Swing de todo el resto del libro:

```
//: com:bruceeckel:swing:Console.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
package com.bruceeckel.swing;
```

```
import javax.swing.*;
import java.awt.event.*;

public class Console {
  // Create a title string from the class name:
  public static String title(Object o) {
    String t = o.getClass().toString();
    // Remove the word "class":
    if(t.indexOf("class") != -1)
      t = t.substring(6);
    return t;
  }
  public static void setupClosing(JFrame frame) {
    // The JDK 1.2 Solution as an
    // anonymous inner class:
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    // The improved solution in JDK 1.3:
    // frame.setDefaultCloseOperation(
    //     EXIT_ON_CLOSE);
  }
  public static void
  run(JFrame frame, int width, int height) {
    setupClosing(frame);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
  public static void
  run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));
    setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(width, height);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
  public static void
  run(JPanel panel, int width, int height) {
    JFrame frame = new JFrame(title(panel));
    setupClosing(frame);
    frame.getContentPane().add(panel);
    frame.setSize(width, height);
    frame.setVisible(true);
  }
} ///:~
```

Esta herramienta puede usarse cuando se desee, por ello está en **com.bruceeckel.swing.** La clase **Console** consiste únicamente en métodos **static.** El primero se usa para extraer el nombre de la clase (usando **RTTI)** del objeto y para eliminar la palabra "class", que suele incorporarse en **getClass**( ). Éste usa los métodos **String indexOf**( ) para determinar si está o no la palabra "class", y **substring**( ) para producir la nueva cadena de caracteres sin "class" o el espacio del final. Este nombre se usa para etiquetar la ventana que mostrarán los métodos **run**( ).

El método **setupClosing**( ) se usa para esconder el código que hace que un **JFrame** salga del programa al cerrarlo. El comportamiento por defecto es no hacer nada, por lo que si no se llama a **setupClosing**( ) o se escribe un código equivalente para el **JFrame,** la aplicación no se cerrará. La razón por la que este código está oculto va más allá de ubicarlo directamente en los métodos **run**( ) subsecuentes se debe en parte a que nos permite usar el método por sí mismo cuando lo que se desea hacer es más complicado que lo proporcionado por **run**( ). Sin embargo, también aísla un factor de cambio: Java 2 tiene dos formas de hacer que se cierren ciertos tipos de ventanas. En el JDK 1.2, la solución es crear una nueva clase **WindowAdapter** e implementar **windowClosing**( ) como se vio anteriormente (se explicará el significado completo de hacer esto algo más adelante en este capítulo).

Sin embargo, durante la creación de JDK 1.3 los diseñadores de la biblioteca observaron que generalmente sería necesario cerrar las ventanas siempre que se cree algo que no sea un *applet,* por lo que añadieron el método **setDefaultCloseOperation**( ) tanto a **JFrame** como a **JDialog**. Desde el punto de vista de la escritura de código, el método nuevo es mucho más sencillo de usar pero este libro se escribió mientras seguía sin implementarse JDK 1.3 en Linux y en otras plataformas, por lo que en pos de la portabilidad entre versiones, se aisló el cambio dentro de **setupClosing**( ).

El método **run**( ) está sobrecargado para que funcione con **JApplets**, **JPanels** y **JFrames**. Nótese que sólo se invoca a **init**( ) y **start**( ) en el caso de que se trate de un **JApplet**.

Ahora, es posible ejecutar cualquier *applet* desde la consola creando un método **main**( ) que contenga un método como:

```
Console.run(new MyClass(), 500, 300);
```

en la que los dos últimos argumentos son la anchura y altura de la visualización. **He** aquí
AppletIc.java modificado para usar **Console**:

```
//: c13:Applet1d.java
```

```
// Console runs applets from the command line.
// <applet code=Applet1d width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1d extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
  public static void main(String[] args) {
    Console.run(new Applet1d(), 100, 50);
  }
} ///:~
```

Esto permite la eliminación de código repetido a la vez que proporciona la mayor flexibilidad posible a la hora de ejecutar los ejemplos.

Usar el Explorador de Windows

Si se usa Windows, se puede simplificar el proceso de ejecutar un programa Java de línea de comandos configurando el Explorador de Windows -el navegador de archivos de Windows, no el Internet Explorer- de forma que se pueda simplemente pulsar dos veces en un **.class** para ejecutarlo. Este proceso conlleva varios pasos.

Primero, descargar e instalar el lenguaje de programación Perl de *http://www.Perl.org*. En esta misma ubicación hay documentación e instrucciones relativas al lenguaje.

A continuación, hay que crear el siguiente script sin la primera y última líneas (este script es parte del paquete de código fuente de este libro):

```
//:! c13:RunJava.bat
@rem = '--*-Perl-*--
@echo off
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem ';
#!perl
$file = $ARGV[0];
$file =~ s/(.*)\..*/\1/;
$file =~ s/(.*\\)*(.*)/$+/;
`java $file`;
__END__
:endofperl
```

```
///:~
```

Ahora, se abre el Explorador de Windows, se selecciona "Ver", "Opciones de Carpeta", y después se pulsa en "Tipos de Archivo". Se presiona el botón "Nuevo Tipo". En "Descripción del tipo" introducir "Fichero de clase Java". En el caso de "Extensiones Asociadas", introducir "class". Bajo "Acciones", presionar el botón "Nuevo". En "Acción" introducir "Open" y en "Aplicación a utilizar para realizar la acción" introducir una línea como:

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

Hay que personalizar la trayectoria ubicada antes de "EjecutarJava.bat" para que contenga la localización en la que cada uno ubique el archivo **.bat.**

Una vez hecha esta instalación, se puede ejecutar cualquier programa Java simplemente pulsando dos veces en el archivo **.class** que contenga un método **main**( ).

Hacer un botón

Hacer un botón es bastante simple: simplemente se invoca al constructor **JButton** con la etiqueta que se desee para el botón. Se verá más adelante que se pueden hacer cosas más elegantes, como poner imágenes gráficas en los botones.

Generalmente se deseará crear un campo para el botón dentro de la clase, de forma que podamos referirnos al mismo más adelante.

E1 **JButton** es un componente -su propia pequeña ventana- que se repintará automáticamente como parte de la actualización. Esto significa que no se pinta explícitamente ningún botón, ni ningún otro tipo de control; simplemente se ubican en el formulario y se deja que se encarguen ellos mismos de pintarse. Por tanto, para ubicar un botón en un formulario, se hace dentro de **init**( ):

```
//: c13:Button1.java
// Putting buttons on an applet.
// <applet code=Button1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button1 extends JApplet {
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
```

```
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
  }
  public static void main(String[] args) {
    Console.run(new Button1(), 200, 50);
  }
} ///:~
```

En este caso, se ha añadido algo nuevo: antes de ubicar los elementos en el panel de contenidos, se ha asignado a éste un "gestor de disposición", de tipo **FiowLayout.** Un gestor de disposiciones permite organizar el control dentro de un formulario. El comportamiento normal para un *applet* es usar el **BorderLayout,** pero éste no funcionaría aquí pues (se aprenderá algo más adelante cuando se explique en detalle cómo controlar la disposición de un formulario) por defecto cubre cada control completamente con cada uno que se añada. Sin embargo, **FlowLayout** hace que los controles fluyan por el formulario, de izquierda a derecha y de arriba hacia abajo.

Capturar un evento

Uno se dará cuenta de que si se compila y ejecuta el *applet* de arriba, no ocurre nada cuando se presionan los botones. Es aquí donde hay que escribir algún tipo de código para ver qué ocurrirá. La base de la programación conducida por eventos, que incluye mucho de lo relacionado con IGU, es atar los eventos al código que responda a los mismos.
La forma de hacer esto en Swing es separando limpiamente la interfaz (los componentes gráficos) y la implementación (el código que se desea ejecutar cuando se da cierto evento en un componente). Cada componente Swing puede reportar todos los eventos que le pudieran ocurrir, y puede reportar también cada tipo de evento individualmente. Por tanto, si uno no está interesado, por ejemplo, en ver si se está moviendo el ratón por encima del botón, no hay por qué registrar interés en ese evento. Se trata de una forma muy directa y elegante de manejar la programación dirigida por eventos, y una vez que se entienden los conceptos básicos se puede usar componentes Swing de forma tan sencilla que nunca antes se podría imaginar -de hecho, el modelo se extiende a todo lo que pueda clasificarse como JavaBean (sobre los que se aprenderá más adelante en este Capítulo).

En primer lugar, nos centraremos simplemente en el evento de interés principal para los componentes que se usen. En el caso de un **JButton**, este "evento de interés" es que se presione el botón.

Para registrar interés en que se presione un botón, se invoca al método **addActionListener**( ) del **JButton**. Este método espera un parámetro que e s un objeto que implemente la interfaz **ActionListener**, que contiene un único método denominado **actionPerformed**( ). Por tanto, todo lo que hay que hacer para adjuntar código a un **JButton** es implementar la interfaz **ActionListener** en una clase y registrar un evento de esa clase con el **JButton** vía **addActionListener**( ). El método será invocado al presionar el botón (a esto se le suele denominar retrollamada).

Pero, ¿cuál debería ser el resultado de presionar ese botón? Nos gustaría ver que algo cambia en la pantalla, por tanto, se introducirá un nuevo componente Swing: el **JTextField**. Se trata de un lugar en el que se puede escribir texto, o en este caso, ser modificado por el programa. Aunque hay varias formas de crear un **JTextField**, la más sencilla es decir al constructor lo ancho que se desea que sea éste. Una vez ubicado el **JTextField** en el formulario, se puede modificar su contenido utilizando el método **setTed**( ) (hay muchos otros métodos en **JTextField**, que pueden encontrarse en la documentación **HTML** del **JDK** disponible en *http://java.sun.com*). Ésta es la apariencia que tiene:

```
//: c13:Button2.java
// Responding to button presses.
// <applet code=Button2 width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2 extends JApplet {
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  JTextField txt = new JTextField(10);
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e){
      String name =
        ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  }
  BL al = new BL();
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
```

```
    cp.add(txt);
  }
  public static void main(String[] args) {
    Console.run(new Button2(), 200, 75);
  }
} ///:~
```

Crear un **JTextFiled** y ubicarlo en el lienzo conlleva los mismos pasos que en los **JButtons,** o cualquier componente Swing. **La** diferencia en el programa de arriba radica en la creación de **BL** de la clase **ActionLitener** ya mencionada. El argumento al **actionPerformed**( ) es de tipo **ActionEvent,** que contiene toda la información sobre el evento y su procedencia. En este caso, quería describir el botón que se presionaba: **getSource**( ) produce el objeto en el que se originó el evento, y asumí que **JButton.getText**( ) devuelve el texto que está en el botón, y que éste está en el **JTextField** para probar que sí que se estaba invocando al código al presionar el botón.

En **hit**( ) se usa **addActionListener**( ) para registrar el objeto **BL** con ambos botones. Suele ser más conveniente codificar el **ActionListener** como una clase interna anónima, especialmente desde que se tiende sólo a usar una instancia de cada clase oyente. Puede modificarse **Boton2.java** para usar clases internas anónimas como sigue:

```
//: c13:Button2b.java
// Using anonymous inner classes.
// <applet code=Button2b width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button2b extends JApplet {
  JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");
  JTextField txt = new JTextField(10);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String name =
        ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  };
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
```

```
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(txt);
  }
  public static void main(String[] args) {
    Console.run(new Button2b(), 200, 75);
  }
} ///:~
```

Se preferirá el enfoque de usar clases anónimas internas (siempre que sea posible) para los ejemplos de este libro.

Áreas de texto

Un **JTextArea** es como un **JTextField** excepto en que puede tener múltiples líneas y tiene más funcionalidad. Un método particularmente útil es **append**( ); con él se puede incorporar alguna salida de manera sencilla a la **JTextArea,** logrando así una mejora de Swing (dado que se puede hacer desplazamiento hacia delante y hacia atrás) sobre lo que se había logrado hasta la fecha haciendo uso de programas de línea de comandos que imprimían a la salida estándar. Como ejemplo, el siguiente programa rellena una **JTextArea** con la salida del generador **geografía** del Capítulo 10**:**

```
//: c13:TextArea.java
// Using the JTextArea control.
// <applet code=TextArea width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class TextArea extends JApplet {
  JButton
    b = new JButton("Add Data"),
    c = new JButton("Clear Data");
  JTextArea t = new JTextArea(20, 40);
  Map m = new HashMap();
  public void init() {
    // Use up all the data:
    Collections2.fill(m,
      Collections2.geography,
      CountryCapitals.pairs.length);
    b.addActionListener(new ActionListener() {
```

```
      public void actionPerformed(ActionEvent e){
        for(Iterator it= m.entrySet().iterator();
            it.hasNext();){
          Map.Entry me = (Map.Entry)(it.next());
          t.append(me.getKey() + ": "
            + me.getValue() + "\n");
        }
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("");
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
  }
  public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
  }
} ///:~
```

En el método **init**( ), se rellena **Map** con todos los países y sus capitales. Nótese que para ambos botones se crea y añade el **ActionListener** sin definir una variable intermedia, puesto que nunca es necesario hacer referencia a ese oyente en todo el programa. El botón "Añadir Datos" da formato e incorpora todos los datos, mientras que el botón "Borrar Datos" hace uso de **setText**( ) para eliminar todo el texto del **JTextArea.**

Al añadir el **JTextArea** al applet, se envuelve en un **JScrollPane** para controlar el desplazamiento cuando se coloca demasiado texto en la pantalla. Esto es todo lo que hay que hacer para lograr capacidades de desplazamiento al completo. Habiendo intentado averiguar cómo hacer el equivalente en otros entornos de programación, estoy muy impresionado con la simplicidad y el buen diseño de componentes como **JScrollPane.**

Controlar la disposición

La forma de colocar los componentes en un formulario en Java probablemente sea distinta de cualquier otro sistema **IGU** que se haya usado. En primer lugar, es todo código; no hay "recursos" que controlen la ubicación de los componentes. Segundo, la forma de colocar los componentes en un formulario no está controlada por ningún tipo de posicionado absoluto sino por un "gestor de disposición" que decide cómo disponer los componentes basándose en el orden en

que se invoca a **add**( ) para ellos. El tamaño, forma y ubicación de los componentes será notoriamente diferente de un gestor de disposición a otro. Además, los gestores de disposición se adaptan a las dimensiones del *applet* o a la ventana de la aplicación, por lo que si se cambia la dimensión de la ventana, cambiarán como respuesta el tamaño, la forma y la ubicación de los componentes.

JApplet, **JFrame**, **JWindow** y **JDialog** pueden todos producir un Container con **getContentPane**() que puede contener y mostrar Componentes. En Container, hay un método denominado **setLayout**( ) que permite elegir entre varios gestores de disposición distintos. Otras clases, como **JPanel**, contienen y muestran los componentes directamente, de forma que también se establece el gestor de disposición directamente, sin usar el cuadro de contenidos.

En esta sección se explorarán los distintos gestores de disposición ubicando botones en los mismos (puesto que es lo más sencillo que se puede hacer con ellos). No habrá ninguna captura de eventos de los botones, puesto que simplemente se pretende que estos ejemplos muestren cómo se disponen los botones.

BorderLayout

El *applet* usa un esquema de disposición por defecto: el **BorderLayout** (varios de los ejemplos anteriores variaban éste al **FlowLayout**). Sin otra instrucción, éste toma lo que se **add**( ) (añade) al mismo y lo coloca en el centro, extendiendo el objeto hasta los bordes. Sin embargo, hay más. Este gestor de disposición incorpora los conceptos de cuatro regiones limítrofes en torno a un área central. Cuando se añade algo a un panel que está haciendo uso de un **BorderLayout** se puede usar el método sobrecargado **add**( ) que toma como primer parámetro un valor constante. Este valor puede ser cualquiera de los siguientes:

BorderLayout.**NORTH** (superior)
BorderLayout.**SOUTH** (inferior)
BorderLayout.**EAST** (derecho)
BorderLayout.**WEST** (izquierdo)
BorderLayout.**CENTER** (rellenar el centro hasta los bordes de todos los demás componentes)

Si no se especifica un área al ubicar un objeto, éste ira por defecto a **CENTER.** He aquí un simple ejemplo. Se usa la disposición por defecto, puesto que **JApplet** se pone por defecto a **BorderLayout:**

```
//: c13:BorderLayout1.java
// Demonstrates BorderLayout.
// <applet code=BorderLayout1
// width=300 height=250> </applet>
```

```
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH,
      new JButton("North"));
    cp.add(BorderLayout.SOUTH,
      new JButton("South"));
    cp.add(BorderLayout.EAST,
      new JButton("East"));
    cp.add(BorderLayout.WEST,
      new JButton("West"));
    cp.add(BorderLayout.CENTER,
      new JButton("Center"));
  }
  public static void main(String[] args) {
    Console.run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

Para todas las ubicaciones excepto **CEN'IER,** el elemento que se añada se comprime hasta caber en la menor cantidad de espacio posible en una de las dimensiones, extendiéndose al máximo en la otra dimensión. Sin embargo, **CENTER** se extiende en ambas dimensiones para ocupar todo el centro.

FlowLayout

Simplemente "hace fluir" los componentes del formulario de izquierda a derecha hasta que se llena el espacio de arriba, después se mueve una fila hacia abajo y continúa fluyendo.

He aquí un ejemplo que establece el gestor de disposición a **FlowLayout** y después ubica botones en el formulario. Se verá que con **FlowLayout** los componentes tomarán su tamaño "natural". Por ejemplo un **JButton,** será del tamaño de su cadena de caracteres:

```
//: c13:FlowLayout1.java
// Demonstrates FlowLayout.
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class FlowLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Button " + i));
  }
  public static void main(String[] args) {
    Console.run(new FlowLayout1(), 300, 250);
  }
} ///:~
```

Se compactarán todos los componentes a su menor tamaño posible en un **FlowLayout,** de forma que se podría obtener un comportamiento ligeramente sorprendente. Por ejemplo, dado que un **JLabel** será del tamaño de su cadena de caracteres, intentar justificar el texto a la derecha conduce a que lo mostrado utilizando **FiowLayout** no varíe.

GridLayout

Un **GridLayout** permite construir una tabla de componentes, y al añadirlos se ubican de izquierda a derecha y de arriba abajo en la rejilla. En el constructor se especifica el número de filas y columnas que se necesiten y éstas se distribuyen en proporciones iguales:

```
//: c13:GridLayout1.java
// Demonstrates GridLayout.
// <applet code=GridLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Button " + i));
  }
  public static void main(String[] args) {
    Console.run(new GridLayout1(), 300, 250);
  }
} ///:~
```

En este caso hay 21 casillas y sólo 20 botones. La última casilla se deja vacía porque en un **GridLayout** no se produce ningún "balanceo".

GridBag Layout

El **GridBagLayout** proporciona un control tremendo a la hora de decidir exactamente cómo distribuir en sí las regiones de la ventana además de cómo reformatear cada región cuando se redimensionan las ventanas. Sin embargo, también es el gestor de disposiciones más complicado, y bastante difícil de entender. Inicialmente está pensado para generación automática de código por parte de un constructor de IGU (los buenos constructores de IGU utilizarán **GridBagLayout** en vez de posicionamiento absoluto). Si uno tiene un diseño tan complicado que cree que debe usar **GridBagLayout**, habría que usar una buena herramienta de construcción de IGU para ese diseño. Si siente la necesidad de saber detalles intrínsecos, le recomiendo la lectura de ***Core*** Java 2, de Hostmann & Cornell (Prentice-Hall, 1999), o un libro dedicado a la Swing, como punto de partida.

Posicionamiento absoluto

También es posible establecer la posición absoluta de los componentes gráficos así:

1. Poner a **null** el gestor de disposiciones del Container: **setlayout**(**nul1**).

2. Invocar a **setBounds**( ) o **reshape**( ) (en función de la versión del lenguaje) por cada componente, pasando un rectángulo circundante con sus coordenadas en puntos. Esto se puede hacer en el constructor o en **paint**( ), en función de lo que se desee lograr.

Algunos constructores de IGU usan este enfoque de forma extensiva, pero ésta no suele ser la mejor forma de generar código. Los constructores de IGU más útiles usan en vez de éste el **GridBagLayout**.

BoxLayout

Debido a que la gente tiene tantas dificultades a la hora de entender y trabajar con **GridBagLayout**, Swing también incluye el **BoxLayout**, que proporciona muchos de los beneficios de **GridBagLayout** sin la complejidad, de forma que a menudo se puede usar cuando se precisen disposiciones codificadas a mano (de nuevo, si el diseño se vuelve demasiado complicado, es mejor usar un constructor de IGU que genere **GridBagLayouts** automáticamente). **BoxLayout** permite controlar la ubicación de los componentes vertical u horizontalmente, y controlar el espacio entre componentes utilizando algo que denomina "puntales y pegamento". En primer lugar, veremos cómo usar directamente **BoxLayout** de la misma forma que se ha demostrado el uso de los demás gestores de disposiciones:

```
//: c13:BoxLayout1.java
// Vertical and horizontal BoxLayouts.
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BoxLayout1 extends JApplet {
  public void init() {
    JPanel jpv = new JPanel();
    jpv.setLayout(
      new BoxLayout(jpv, BoxLayout.Y_AXIS));
    for(int i = 0; i < 5; i++)
      jpv.add(new JButton("" + i));
    JPanel jph = new JPanel();
    jph.setLayout(
      new BoxLayout(jph, BoxLayout.X_AXIS));
    for(int i = 0; i < 5; i++)
      jph.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, jpv);
    cp.add(BorderLayout.SOUTH, jph);
  }
  public static void main(String[] args) {
    Console.run(new BoxLayout1(), 450, 200);
  }
} ///:~
```

El constructor de **BoxLayout** es un poco diferente del de los otros gestores de disposición -se proporciona el **Container** que va **a** ser controlado por el **BoxLayout** como primer argumento, y la dirección de la disposición como segundo argumento.

Para simplificar las cosas, hay un contenedor especial denominado **Box** que usa **BoxLayout** como gestor nativo. El ejemplo siguiente distribuye los componentes horizontal y verticalmente usando **Box,** que tiene dos métodos **static** para crear cajas con alineación vertical y horizontal:

```
//: c13:Box1.java
// Vertical and horizontal BoxLayouts.
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box1 extends JApplet {
```

```
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++)
      bv.add(new JButton("" + i));
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++)
      bh.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box1(), 450, 200);
  }
} ///:~
```

Una vez que se tiene una **Box,** se pasa como segundo parámetro al añadir componentes al panel contenedor.

Los puntales añaden espacio entre componentes, midiéndose este espacio en puntos. Para usar un puntal, simplemente se añade éste entre la adición de los componentes que se desea separar:

```
//: c13:Box2.java
// Adding struts.
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++) {
      bv.add(new JButton("" + i));
      bv.add(Box.createVerticalStrut(i*10));
    }
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++) {
      bh.add(new JButton("" + i));
      bh.add(Box.createHorizontalStrut(i*10));
    }
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box2(), 450, 300);
```

```
  }
} ///:~
```

Los puntales separan los componentes en una cantidad fija, mientras que la cola actúa al contrario: separa los componentes tanto como sea posible. Por consiguiente, es más un elástico que "pegamento" (y el diseño en el que se basó se denominaba "elásticos y puntales" por lo que la elección del término es algo misteriosa).

```
//: c13:Box3.java
// Using Glue.
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JLabel("Hello"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("Applet"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("World"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JLabel("Hello"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("Applet"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("World"));
    bv.add(Box.createVerticalGlue());
    bv.add(bh);
    bv.add(Box.createVerticalGlue());
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box3(), 450, 300);
  }
} ///:~
```

Un puntal funciona de forma inversa, pero un área rígida fija los espacios entre componentes en ambas direcciones:

```
//: c13:Box4.java
// Rigid Areas are like pairs of struts.
// <applet code=Box4
// width=450 height=300> </applet>
```

```
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JButton("Top"));
    bv.add(Box.createRigidArea(
      new Dimension(120, 90)));
    bv.add(new JButton("Bottom"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JButton("Left"));
    bh.add(Box.createRigidArea(
      new Dimension(160, 80)));
    bh.add(new JButton("Right"));
    bv.add(bh);
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box4(), 450, 300);
  }
} ///:~
```

Habría que ser consciente de que las áreas rígidas son algo controvertidas. Puesto que usan valores absolutos, algunos piensan que pueden causar demasiados problemas como para merecer la pena.

¿El mejor enfoque?

Swing es potente; puede lograr mucho con unas pocas líneas de código. Los ejemplos mostrados en este libro son razonablemente simples, y por propósitos de aprendizaje tiene sentido escribirlos a mano. De hecho se puede lograr, simplemente, combinando disposiciones. En algún momento, sin embargo, deja de tener sentido codificar a mano formularios IGU -se vuelve demasiado complicado y no constituye un buen uso del tiempo de programación. Los diseñadores de Java y Swing orientaron el lenguaje y las bibliotecas de forma que soportaran herramientas de construcción de IGU, creadas con el propósito específico de facilitar la experiencia de programación. **A** medida que se entiende todo lo relacionado con disposiciones y con cómo tratar con esos eventos (como se describe a continuación), no es particularmente importante que se sepan los detalles de cómo distribuir componentes a mano -deje que la herramienta apropiada haga el trabajo por usted (Java, después de todo, está diseñado para incrementar la productividad del programador).

El modelo de eventos de Swing

En el modelo de eventos de Swing, un componente puede iniciar ("disparar") un evento. Cada tipo de evento está representado por una clase distinta. Cuando se dispara un evento, éste es recibido por uno o más "oyentes", que actúan sobre ese evento. Por consiguiente, la fuente de un evento y el lugar en el que éste es gestionado podrían ser diferentes. Por tanto, generalmente se usan los componentes Swing tal y como son, aunque es necesario escribir el código al que se invoca cuando los componentes reciben un evento, de forma que nos encontramos ante un excelente ejemplo de la separación entre la interfaz y la implementación.

Cada oyente de eventos es un objeto de una clase que implementa un tipo particular de **interfaz** oyente. Por tanto, como programador, todo lo que hay que hacer es crear un objeto oyente y registrarlo junto con el componente que dispara el evento. El registro se lleva a cabo invocando a un método **addXXXIistener**( ) en el componente que dispara el evento, donde 'XXX" representa el tipo de evento por el que se escucha. Se pueden conocer fácilmente los tipos de eventos que pueden gestionarse fijándose en los nombres de los métodos "**addListener**", y si se intenta escuchar por eventos erróneos se descubrirá el error en tiempo de compilación. Más adelante en este capítulo se verá que los JavaBeans también usan los nombres de los métodos "**addListener**" para determinar qué eventos puede manejar un Bean.

Después, toda la lógica de eventos irá dentro de una clase oyente. Cuando se crea una clase oyente, su única restricción es que debe implementar la interfaz apropiada. Se puede crear una clase oyente global, pero ésta es una situación en la que tienden a ser bastante útiles las clases internas, no sólo por proporcionar una agrupación lógica de las clases oyentes dentro del IU o de las clases de la lógica de negocio a las que sirven, sino (como se verá más adelante) por el hecho de que una clase interna proporciona una forma elegante más allá de una clase y los límites del subsistema.

Todos los ejemplos mostrados en este capítulo hasta el momento han usado el modelo de eventos de Swing, pero el resto de la sección acabará de describir los detalles de ese modelo.

Tipos de eventos y oyentes

Todos los componentes Swing incluyen **addXXXListener**( ) y **removeXXXListener**( ) de forma que se pueden añadir y eliminar los tipos de oyentes apropiados de cada componente. Se verá que en cada caso el **"XXX"** representa el parámetro del método, por ejemplo: **addMiOyente** (**MiOyente m)**. **La** tabla siguiente incluye los eventos básicos asociados, los oyentes y los métodos, junto con los componentes básicos que soportan esos eventos particulares proporcionando los métodos **addXXXListener**( ) y

removeXXXListener( ). Debería tenerse en mente que el modelo de eventos fue diseñado para ser extensible, de forma que se podrían encontrar otros tipos de eventos y oyentes que no estén en esta tabla.

Evento, interfaz oyente y métodos add y remove	**Componentes que soportan este evento**
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 se cree que implemente la **interfaz Adjustable.**
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	***Component** y sus derivados incluyendo **JButton, JCanvas, 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,** and **JFrame.**
FocusEvent **FocusListener** **addFocusListener( )** **removeFocusListener( )**	**Component** y **derivados*.**
KeyEvent **KeyListener** **addKeyListener( )** **removeKeyListener( )**	**Component** y **derivados*.**
MouseEvent (para ambos clic y movimiento) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** y **derivados*.**
MouseEvent [8] (para ambos clic y movimiento)	**Component** y **derivados*.**

Evento, interfaz oyente y métodos add y remove	Componentes que soportan este evento
MouseMotionListener addMouseMotionListener( ) removeMouseMotionListener( )	
WindowEvent WindowListener addWindowListener( ) removeWindowListener( )	**Window** y sus derivados incluyendo **JDialog, JFileDialog, and JFrame.**
ItemEvent ItemListener addItemListener( ) removeItemListener( )	**JCheckBox, JCheckBoxMenuItem, JComboBox, JList,** y cualquier cosa que implemente la **interfase ItemSelectable.**
TextEvent TextListener addTextListener( ) removeTextListener( )	Cualquier cosa derivado de **JTextComponent,** incluyendo **JTextArea** y **JTextField.**

[8] No existe evento **MouseMotionEvent** incluso aunque parece que debería haberlo. Hacer clic y el movimiento se combinan en **MouseEvent,** por lo que esta segunda ocurrencia de **MouseEvent** en la tabla no es ningún error.

Se puede ver que cada tipo de componente sólo soporta ciertos tipos de eventos. Se vuelve bastante difícil mirar todos los eventos que soporta cada componente. Un enfoque más sencillo es modificar el programa **MostrarLimpiarMetodos.java** del Capítulo 12, de forma que muestre todos los oyentes de eventos soportados por cualquier componente Swing que se introduzca.

El Capítulo 12 introdujo la ***reflectividad*** y usaba esa faceta para buscar los métodos de una clase particular -bien la lista de métodos al completo, o bien un subconjunto de aquéllos cuyos nombres coincidan con la palabra dave que se proporcione. La magia de esto es que puede mostrar automáticamente ***todos*** los métodos de una clase sin forzarla a recorrer la jerarquía de herencias hacia arriba examinando las clases base en cada nivel. Por consiguiente, proporciona una valiosa herramienta para ahorrar tiempo a la hora de programar: dado que la mayoría de los nombres de los métodos de Java son bastante descriptivos, se pueden buscar los nombres de métodos que contengan una palabra de interés en particular. Cuando se encuentre lo que uno cree estar buscando, compruebe la documentación en línea.

Sin embargo, cuando llegamos al Capítulo 12 no se había visto Swing, por lo que la herramienta de ese capítulo se desarrolló como aplicación de línea de comandos. Aquí está la versión más útil de **IGU,** especializada en buscar los métodos "**addListener**" de los componentes Swing:

```
//: c13:ShowAddListeners.java
// Display the "addXXXListener" methods of any
// Swing class.
// <applet code = ShowAddListeners
// width=500 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.io.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class ShowAddListeners extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField name = new JTextField(25);
  JTextArea results = new JTextArea(40, 65);
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      // Convert to an array of Strings:
      n = new String[m.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      reDisplay();
    }
  }
  void reDisplay() {
```

```
    // Create the result set:
    String[] rs = new String[n.length];
    int j = 0;
    for (int i = 0; i < n.length; i++)
      if(n[i].indexOf("add") != -1 &&
        n[i].indexOf("Listener") != -1)
          rs[j++] =
            n[i].substring(n[i].indexOf("add"));
    results.setText("");
    for (int i = 0; i < j; i++)
      results.append(
        StripQualifiers.strip(rs[i]) + "\n");
  }
  public void init() {
    name.addActionListener(new NameL());
    JPanel top = new JPanel();
    top.add(new JLabel(
      "Swing class name (press ENTER):"));
    top.add(name);
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH, top);
    cp.add(new JScrollPane(results));
  }
  public static void main(String[] args) {
    Console.run(new ShowAddListeners(), 500,400);
  }
} ///:~
```

Aquí se vuelve a usar la clase **EliminarCalificadores** definida en el Capítulo 12 importando la biblioteca *com.bruceeckel.uti1*.

La IGU contiene un **JTextField** nombre en el que se puede introducir el nombre de la clase Swing que se desea buscar. Los resultados se muestran en una **JTextArea**.

Se verá que no hay botones en los demás componentes mediante los que indicar que se desea comenzar la búsqueda. Esto se debe a que **JTextField** está monitorizada por un **ActionListener**. Siempre que se haga un cambio y se presione **ENTER** se actualiza la lista inmediatamente. Si el texto no está vacío, se usa dentro de **Class.forName**( ) para intentar buscar la clase. Si el nombre es incorrecto, **Class.forName**( ) fallará, lo que significa que lanzará una excepción. Ésta se atrapa y se pone la **JTextArea** a "No coinciden". Pero si se teclea un nombre correcto (teniendo en cuenta las mayúsculas y minúsculas), **Class.forName**( ) tiene éxito y **getMethods**( ) devolverá un array de objetos **Method**. Cada uno de los objetos del array se convierte en un **String** vía **toString**( ) (esto produce la signatura completa del método) y se añade a n, un array de **Strings**. El array **n** es un miembro de **class ShowAddListeners** y se usa en la actualización de la pantalla siempre que se invoca a **representar**( ).

El método **representar**( ) crea un array de **Strings** llamado **rs** (de "result set" - "conjunto resultado" en inglés). De este conjunto se copian en n aquellos elementos que contienen "add" y "Listener". Después se usan **indexOf**( ) y **substring**( ) para eliminar los calificadores como **public**, **static**, etc. Finalmente, **EliminarCalificadores.strip**( ) remueve los calificadores de nombre extra.

Este programa constituye una forma conveniente de investigar las capacidades de un componente Swing. Una vez que se conocen los eventos soportados por un componente en particular, no es necesario buscar nada para que reaccione ante el evento. Simplemente:

1. Se toma el nombre de la clase evento y se retira la palabra "**Event**". Se añade la palabra "**Listener**" a lo que queda. Ésta es la interfaz oyente a implementar en la clase interna.

2. Implementar la interfaz de arriba y escribir los métodos de los eventos a capturar. Por ejemplo, se podría estar buscando movimientos de ratón, por lo que se escribe código para el método **mouseMoved**( ) de la interfaz **MouseMotionListener**. (Hay que implementar los otros métodos, por supuesto, pero a menudo hay un atajo que veremos pronto.)

3. Crear un objeto de la clase oyente del paso **2.** Registrarlo con el componente con el método producido al prefijar **"add** al nombre del Oyente. Por ejemplo, **addMouseMotionListener(**).

He aquí algunos de las interfaces **Oyente**:

Interface Oyente y 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	**keyPressed(KeyEvent)**

Interface Oyente y Adaptador	Métodos de la interfaz
KeyAdapter	keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Éste no es un listado exhaustivo, en parte porque el modelo de eventos permite crear tipos de eventos y oyentes personalizados. Por consiguiente, frecuentemente se tendrá acceso a bibliotecas que han inventado sus propios eventos, de forma que el conocimiento que se adquiera en este capítulo te permitirá adivinar como utilizar esos eventos.

Utilizar adaptadores de oyentes por simplicidad

En la tabla de arriba, se puede ver que algunas interfaces oyentes sólo tienen un método. Éstas son triviales de implementar puesto que se implementarán sólo cuando se desee escribir ese método en particular. Sin embargo, las interfaces oyentes que tienen múltiples métodos pueden ser menos agradables de usar. Por ejemplo, algo que hay que hacer siempre que se cree una aplicación es proporcionar un **WindowListener** al **JFrame** de forma que cuando se logre el evento **windowClosing**( ) se llame a **System.exit**( ) para salir de la aplicación. Pero dado que **WindowListener** es una interfaz, hay que implementar todos los demás métodos incluso aunque no hagan nada. Esto puede resultar molesto.

Para solucionar el problema, algunas (aunque no todas) de las interfaces oyentes que tienen más de un método se suministran con adaptadores, cuyos nombres pueden verse en la tabla de arriba. Cada adaptador proporciona métodos por defecto vacíos para cada uno de los métodos de la interfaz.

Después, todo lo que hay que hacer es heredar del adaptador y superponer sólo los métodos que se necesiten cambiar. Por ejemplo, el **WindowListener** típico a usar tendrá la siguiente apariencia (recuérdese que se ha envuelto en la clase **Console** de *com.bruceeckel.swing*:

```
class MyWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

La única razón para la existencia de los adaptadores es facilitar la creación de las clases Oyente.

Sin embargo, los adaptadores también tienen un inconveniente. Imagínese que se escribe un **WindowAdapter** como el de arriba:

```
class MyWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

No funciona, pero uno se volverá loco tratando de averiguar porqué, pues compila **y** se ejecuta correctamente -excepto por el hecho de que cerrar la ventana no saldrá del programa**.** ¿Se **ve** el problema? Está en el nombre del método: **WindowClosing**( ) **en** vez de **windowClosing**( ). Un simple error con una mayúscula produce la acción de un método completamente nuevo. Sin embargo, no es el método que se invoca cuando se cierra la ventana, por- lo que no se obtendrán los resultados deseados. Con excepción de este inconveniente, una **interfaz** garantizará que los métodos estén correctamente implementados.

Seguimiento de múltiples eventos

Para probar que estos eventos se están disparando verdaderamente, y como experimento interesante, merece la pena crear un *applet* que haga un seguimiento de comportamiento extra en un **JButton** (que no sea si está o no presionado). Este ejemplo también muestra cómo heredar tu propio objeto botón puesto que se usa como destino de todos los eventos de interés. Para lograrlo, simplemente se puede heredar de **JButton [9]**.

[9] En Java 1.0/1.1 no se podía heredar de manera útil del objeto botón. Este no era sino uno de los muchos fallos de diseño de que adolecía.

La clase **MiBoton** es una clase interna de **RastrearEvento**, por lo que **MiBoton** puede llegar a la ventana padre y manipular sus campos de texto, que es lo necesario para poder escribir la información de estado en los campos de texto, que es lo necesario para poder escribir la información de estado en los campos del padre. Por supuesto ésta es una solución limitada, puesto que **MiBoton** puede usarlo sólo en conjunción con **RastrearEvento**. A este tipo de código se le suele denominar “altamente acoplado”:

```
//: c13:TrackEvent.java
// Show events as they happen.
// <applet code=TrackEvent
//  width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class TrackEvent extends JApplet {
  HashMap h = new HashMap();
  String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  MyButton
    b1 = new MyButton(Color.blue, "test1"),
    b2 = new MyButton(Color.red, "test2");
  class MyButton extends JButton {
    void report(String field, String msg) {
      ((JTextField)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());
      }
      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 mml =
      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(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  }
  public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
      JTextField t = new JTextField();
      t.setEditable(false);
      c.add(new JLabel(event[i], JLabel.RIGHT));
      c.add(t);
      h.put(event[i], t);
    }
    c.add(b1);
    c.add(b2);
```

```
  }
  public static void main(String[] args) {
    Console.run(new TrackEvent(), 700, 500);
  }
} ///:~
```

En el constructor de **MiBoton,** se establece el color del botón con una llamada a **setBackground**( ). Los oyentes están todos instalados con simples llamadas a métodos.

La clase **RastrearEvento** contiene un **HashMap** para guardar las cadenas que representan el tipo de evento y **JTexiFields** donde se guarda la información sobre el evento. Por supuesto, éstos podrían haberse creado de forma estática en vez de ponerlos en un **HashMap,** pero creo que estará de acuerdo en que es mucho más fácil de usar y cambiar. En particular, si se necesita añadir o retirar un nuevo tipo de evento en **RastrearEvento,** simplemente se añadirá o retirará un **String** en el array **evento** -todo lo demás sucede automáticamente.

Cuando se invoca a **informar**( ) se le da el nombre del evento y el parámetro **String** del evento. Usa el **HashMap h** en la clase externa para buscar el **JTextField** asociado con ese nombre de evento y después coloca el parámetro **String** en ese campo.

Es divertido ejecutar este ejemplo puesto que realmente se puede ver lo que está ocurriendo con los eventos del programa.

Un catálogo de componentes Swing

Ahora que se entienden los gestores de disposición y el modelo de eventos, ya podemos ver cómo se pueden usar los componentes Swing. Esta sección es un viaje no exhaustivo por los componentes de Swing y las facetas que probablemente se usarán la mayoría de veces. Cada ejemplo pretende ser razonablemente pequeño, de forma que se pueda tomar el código y usarlo en los programas que cada uno desarrolle.

Se puede ver fácilmente qué aspecto tiene cada uno de los ejemplos al ejecutarlo viendo las páginas HTML en el código fuente descargable para este capítulo.

Mantenga en mente:

1. La documentación HTML de *java.sun.com* contiene todas las clases y métodos Swing (aquí sólo se muestran unos pocos).

2. Debido a la convención de nombres usada en los eventos Swing, es bastante fácil adivinar cómo escribir e instalar un manipulador para un tipo particular de evento. Se

puede usar el programa de búsqueda **MostrarAddListeners.java** visto antes en este capítulo para ayudar a investigar un componente particular,

3. Cuando las cosas se complican habría que pasar a un constructor de **IGU.**

Botones

Swing incluye varios tipos de botones. Todos los botones, casillas de verificación, botones de opción e incluso los elementos de menú se heredan de **AbstractButton** (que, dado que incluye elementos de menú, se habría llamado probablemente "**AbstractChooser**" o algo igualmente general). En breve veremos el uso de elementos de menú, pero el ejemplo siguiente muestra los distintos tipos de botones disponibles:

```
//: c13:Buttons.java
// Various Swing buttons.
// <applet code=Buttons
//  width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Buttons extends JApplet {
  JButton jb = new JButton("JButton");
  BasicArrowButton
    up = new BasicArrowButton(
      BasicArrowButton.NORTH),
    down = new BasicArrowButton(
      BasicArrowButton.SOUTH),
    right = new BasicArrowButton(
      BasicArrowButton.EAST),
    left = new BasicArrowButton(
      BasicArrowButton.WEST);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(jb);
    cp.add(new JToggleButton("JToggleButton"));
    cp.add(new JCheckBox("JCheckBox"));
    cp.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);
    cp.add(jp);
  }
  public static void main(String[] args) {
    Console.run(new Buttons(), 350, 100);
  }
} ///:~
```

Éste comienza con el **BasicArrowButton** de *javax.swing.plaf.basic*, después continúa con los diversos tipos específicos de botones. **Al** ejecutar el ejemplo, se verá que el botón de conmutación guarda su última posición, dentro o fuera. Pero las casillas de verificación y los botones de opción se comportan exactamente igual simplemente pulsando para activarlos o desactivarlos (ambos se heredan de **JToggleButton**).

Grupos de botones

Si se desea que varios botones de opción se comporten en forma de "or exclusivo o **XOR,** hay que añadirlos a un "grupo de botones". Pero, como demuestra el ejemplo de debajo, se puede añadir cualquier **AbstractButton** a un **ButtonGroup**.

Para evitar repetir mucho código, este ejemplo usa la *reflectividad* para generar los grupos de distintos tipos de botones. Esto se ve en **hacerBPanel**( ), que crea un grupo de botones y un **JPanel**.

El segundo parámetro a **hacerBPanel**( ) es un array de **String**. Por cada **String**, se añade un bo**tón de la clase indicada por el primer argumento al JPanel:**

```
//: c13:ButtonGroups.java
// Uses reflection to create groups
// of different types of AbstractButton.
// <applet code=ButtonGroups
//  width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
import com.bruceeckel.swing.*;

public class ButtonGroups extends JApplet {
  static String[] ids = {
    "June", "Ward", "Beaver",
    "Wally", "Eddie", "Lumpy",
  };
  static JPanel
  makeBPanel(Class bClass, String[] ids) {
```

```
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String title = bClass.getName();
    title = title.substring(
      title.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(title));
    for(int i = 0; i < ids.length; i++) {
      AbstractButton ab = new JButton("failed");
      try {
        // Get the dynamic constructor method
        // that takes a String argument:
        Constructor ctor = bClass.getConstructor(
          new Class[] { String.class });
        // Create a new object:
        ab = (AbstractButton)ctor.newInstance(
          new Object[]{ids[i]});
      } catch(Exception ex) {
        System.err.println("can't create " +
          bClass);
      }
      bg.add(ab);
      jp.add(ab);
    }
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(makeBPanel(JButton.class, ids));
    cp.add(makeBPanel(JToggleButton.class, ids));
    cp.add(makeBPanel(JCheckBox.class, ids));
    cp.add(makeBPanel(JRadioButton.class, ids));
  }
  public static void main(String[] args) {
    Console.run(new ButtonGroups(), 500, 300);
  }
} ///:~
```

El título del borde se toma del nombre de la clase, eliminando la información de trayectoria. El **AbstractButton** se inicializa a **JButton**, que tiene la etiqueta "Fallo" por lo que si se ignora el mensaje de excepción, se seguirá viendo el problema en pantalla. El método **getConstructor**( ) produce un objeto Constructor que toma el array de argumentos de los tipos del array **Class** pasado a **getConstructor**( ). Después todo lo que se hace es llamar a **newInstance**( ), pasándole **un** array de **Object** que contiene los parámetros actuales -en este caso, simplemente el **String** del array **ids.**

Esto añade un poco de complejidad a lo que es un proceso simple. Para lograr comportamiento **XOR** con botones, se crea un grupo de botones y se añade cada

botón para el que se desea ese comportamiento en el grupo. Cuando se ejecuta el programa, se verá que todos los botones excepto **JButton** exhiben este comportamiento "**or exclusivo**".

Iconos

Se puede usar un **Icon** dentro de un **JLabel** o cualquier cosa heredada de **AbstractButton** (incluyendo **JButton**, **JCheckBox**, **JRadioButton**, y los distintos tipos de **JMenuItem**). Utilizar **Icons** con **JLabels** es bastante directo (se verá un ejemplo más adelante). El ejemplo siguiente explora todas las formas adicionales de usar **Icons** con botones y sus descendientes.

Se puede usar cualquier archivo ***gif*** que se desee, pero los que se usan en este ejemplo son parte de la distribución de código de este libro, disponible en *http://www.BruceEckel.com*. Para abrir un archivo e incorporar la imagen, simplemente se crea un **ImageIcon** y se le pasa el nombre del archivo. **A** partir de ese momento, se puede usar el **Icon** resultante en el programa. Nótese que en este ejemplo, la información de trayectoria está codificada **a** mano: **se** necesitará cambiar la trayectoria para hacerla corresponder con la ubicación de los archivos de imágenes:

```
//: c13:Faces.java
// Icon behavior in Jbuttons.
// <applet code=Faces
//  width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Faces extends JApplet {
  // The following path information is necessary
  // to run via an applet directly from the disk:
  static String path =
    "C:/aaa-TIJ2-distribution/code/c13/";
  static Icon[] faces = {
    new ImageIcon(path + "face0.gif"),
    new ImageIcon(path + "face1.gif"),
    new ImageIcon(path + "face2.gif"),
    new ImageIcon(path + "face3.gif"),
    new ImageIcon(path + "face4.gif"),
  };
  JButton
    jb = new JButton("JButton", faces[3]),
    jb2 = new JButton("Disable");
  boolean mad = false;
  public void init() {
```

```
    Container cp = getContentPane();
    cp.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!");
    cp.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");
        }
      }
    });
    cp.add(jb2);
  }
  public static void main(String[] args) {
    Console.run(new Faces(), 400, 200);
  }
} ///:~
```

Se puede usar un **Icon** en muchos constructores, pero también se puede usar **setIcon**( ) para añadir o cambiar un **Icon**. Este ejemplo también muestra cómo un J B u t t o n (o cualquier **AbstractButton**) puede establecer los distintos tipos de iconos que aparecen cuando le ocurren cosas a ese botón: cuando se presiona, se deshabilita o se "pasa sobre él" (el ratón se mueve sobre él sin hacer clic). Se verá que esto da a un botón una imagen animada genial.

Etiquetas de aviso

El ejemplo anterior añadía una "etiqueta de aviso" al botón. Casi todas las clases que se usen para crear interfaces de usuario se derivan de **JComponent**, que contiene un método denominado **setToolTipText**(**String**). Por tanto, para casi todo lo que se coloque en un formulario, todo lo que se necesita hacer es decir (siendo **jc** el objeto de cualquier clase derivada **JComponent**):

```
jc.setToolTipText("My tip");
```

y cuando el ratón permanece sobre ese **JComponent**, durante un periodo de tiempo predeterminado, aparecerá una pequeña caja con el texto junto al puntero del ratón.

Campos de texto

Este ejemplo muestra el comportamiento extra del que son capaces los **JTextFields**:

```
//: c13:TextFields.java
// Text fields and Java events.
// <applet code=TextFields width=375
// height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TextFields extends JApplet {
  JButton
    b1 = new JButton("Get Text"),
    b2 = new JButton("Set Text");
  JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  String s = new String();
  UpperCaseDocument
    ucd = new UpperCaseDocument();
  public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(new T1A());
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
```

```
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.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) {
    Console.run(new TextFields(), 375, 125);
  }
}

class UpperCaseDocument extends PlainDocument {
  boolean upperCase = true;
  public void setUpperCase(boolean flag) {
    upperCase = flag;
  }
  public void insertString(int offset,
```

```
    String string, AttributeSet attributeSet)
    throws BadLocationException {
      if(upperCase)
        string = string.toUpperCase();
      super.insertString(offset,
        string, attributeSet);
  }
} ///:~
```

El **JTextField t3** se incluye como un lugar a reportar cuando se dispare el oyente del **JTextField t1.** Se verá que el oyente de la acción de un **JTextField** se dispara sólo al presionar la tecla "enter".

E1 **JTextField t1** tiene varios oyentes asignados. E1 **t1** es un **DocumentListener** que responde a cualquier cambio en el "documento" (en este caso los contenidos de **JTextField).** Copia automáticamente todo el texto de **t1** a **t2.** Además, el documento de **t1** se pone a una clase derivada de **PlainDocument,** llamada **DocumentoMayusculas,** que fuerza a que todos sus caracteres sean mayúsculas. Detecta automáticamente los espacios en blanco y lleva a cabo los borrados, ajustando los intercalados y gestionando todo como cabría esperar.

Bordes

JComponent contiene un método denominado **setBorder**( ), que permite ubicar varios bordes interesantes en cualquier componente visible. El ejemplo siguiente demuestra varios de los distintos bordes disponibles, utilizando un método denominado **showBorder**( ) que crea un **JPanel** y pone el borde en cada caso. También usa RTTI para averiguar el nombre del borde que se está usando (eliminando toda información de trayectoria), y pone después ese nombre en un **JLabel** en el medio del panel:

```
//: c13:Borders.java
// Different Swing borders.
// <applet code=Borders
//  width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Borders extends JApplet {
  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 void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.setLayout(new GridLayout(2,4));
    cp.add(showBorder(new TitledBorder("Title")));
    cp.add(showBorder(new EtchedBorder()));
    cp.add(showBorder(new LineBorder(Color.blue)));
    cp.add(showBorder(
      new MatteBorder(5,5,30,30,Color.green)));
    cp.add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    cp.add(showBorder(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    cp.add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
  }
  public static void main(String[] args) {
    Console.run(new Borders(), 500, 300);
  }
} ///:~
```

También se pueden crear bordes personalizados y ponerlos dentro de botones, etiquetas, etc. -cualquier cosa derivada de **JComponent.**

JScrollPanes

La mayoría de las veces simplemente se deseará dejar que un **JScrollPane** haga su trabajo, pero también se pueda controlar qué barras de desplazamiento están permitidas -la vertical, la horizontal, ambas o ninguna:

```
//: c13:JScrollPanes.java
// Controlling the scrollbars in a JScrollPane.
// <applet code=JScrollPanes width=300 height=725>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class JScrollPanes extends JApplet {
  JButton
```

```
    b1 = new JButton("Text Area 1"),
    b2 = new JButton("Text Area 2"),
    b3 = new JButton("Replace Text"),
    b4 = new JButton("Insert Text");
  JTextArea
    t1 = new JTextArea("t1", 1, 20),
    t2 = new JTextArea("t2", 4, 20),
    t3 = new JTextArea("t3", 1, 20),
    t4 = new JTextArea("t4", 10, 10),
    t5 = new JTextArea("t5", 4, 20),
    t6 = new JTextArea("t6", 10, 10);
  JScrollPane
    sp3 = new JScrollPane(t3,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp4 = new JScrollPane(t4,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp5 = new JScrollPane(t5,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
    sp6 = new JScrollPane(t6,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t5.append(t1.getText() + "\n");
    }
  }
  class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.setText("Inserted by Button 2");
      t2.append(": " + t1.getText());
      t5.append(t2.getText() + "\n");
    }
  }
  class B3L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String s = " Replacement ";
      t2.replaceRange(s, 3, 3 + s.length());
    }
  }
  class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.insert(" Inserted ", 10);
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
```

```
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);
    // Initialize listeners and add components:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
    cp.add(b4);
    cp.add(sp3);
    cp.add(sp4);
    cp.add(sp5);
    cp.add(sp6);
  }
  public static void main(String[] args) {
    Console.run(new JScrollPanes(), 300, 725);
  }
} ///:~
```

Utilizar argumentos distintos en el constructor **JScrollPane** permite controlar las barras de desplazamiento disponibles. Este ejemplo también adorna un poco los distintos elementos usando bordes.

Un minieditor

El control **JTextPane** proporciona un gran soporte a la edición sin mucho esfuerzo. El ejemplo siguiente hace un uso muy sencillo de esto, ignorando el grueso de la funcionalidad de la clase:

```
//: c13:TextPane.java
// The JTextPane control is a little editor.
// <applet code=TextPane width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
```

```
import com.bruceeckel.util.*;

public class TextPane extends JApplet {
  JButton b = new JButton("Add Text");
  JTextPane tp = new JTextPane();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() +
            sg.next() + "\n");
      }
    });
    Container cp = getContentPane();
    cp.add(new JScrollPane(tp));
    cp.add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    Console.run(new TextPane(), 475, 425);
  }
} ///:~
```

El botón simplemente añade texto generado al azar. La intención del **JTextPane** es permitir editar texto *in situ*, de forma que se verá que no hay método **append**( ). En este caso (hay que admitir que se trata de un uso pobre de las capacidades de **JTextPane**), debemos capturar el texto, modificarlo y volverlo a ubicar en su sitio utilizando **setText**( ).

Como se mencionó anteriormente, el comportamiento de la disposición por defecto de un ***applet*** es usar el **BorderLayout**. Si se añade algo al panel sin especificar más detalles, simplemente se rellena el centro del panel hasta los bordes. Sin embargo, si se especifica una de las regiones que le rodean (**NORTH**, **SOUTH**, **EAST** o **WEST**) como se hace aquí, el componente se encajará en esa región -en este caso, el botón se anidará abajo, en la parte inferior de la pantalla.

Fíjese en las facetas incluidas en **JTextPane**, como la envoltura automática de líneas. Usando la documentación **JDK** es posible buscar otras muchas facetas.

Casillas de verificación

Una casilla de verificación proporciona una forma sencilla de hacer una elección de tipo activado/desactivado; consiste en una pequeña caja y una etiqueta. La caja suele guardar una pequeña "x" (o alguna otra indicación que se establezca), o está vacía, en función de si el elemento está o no seleccionado.

Normalmente se creará un **JCheckBox** utilizando un constructor que tome la etiqueta como parámetro. Se puede conseguir y establecer su estado, y también la etiqueta si se desea leerla o cambiarla una vez creado el **JCheckBox**.

Siempre que se asigne valor o se limpie un **JCheckBox**, se da un evento que se puede capturar de manera análoga a como se hace con un botón, utilizando un **ActionListener**. El ejemplo siguiente usa un **JTextArea** para enumerar todas las casillas de verificación seleccionadas:

```
//: c13:CheckBoxes.java
// Using JCheckBoxes.
// <applet code=CheckBoxes width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CheckBoxes extends JApplet {
  JTextArea t = new JTextArea(6, 15);
  JCheckBox
    cb1 = new JCheckBox("Check Box 1"),
    cb2 = new JCheckBox("Check Box 2"),
    cb3 = new JCheckBox("Check Box 3");
  public void init() {
    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);
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(cb1);
    cp.add(cb2);
    cp.add(cb3);
  }
  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) {
    Console.run(new CheckBoxes(), 200, 200);
  }
} ///:~
```

El método **Rastrear**( ) envía el nombre del **JCheckBox** seleccionado y su estado actual al **JTextArea** utilizando **append**( ), por lo que se verá una lista acumulativa de casillas de verificación seleccionadas y cuál es su estado.

Botones de opción

El concepto de botón de opción en programación de IGU proviene de las radios de coche pre-electrónicas con botones mecánicos: cuando se oprimía un botón cualquier otro botón que estuviera pulsado, saltaba. Por consiguiente, permite forzar una elección única entre varias.

Todo lo que se necesita para establecer un grupo asociado de **JRadioButtons** es añadirlos a un **ButtonGroup** (se puede tener cualquier número de **ButtonGroups** en un formulario). Uno de los botones puede tener su valor inicial por defecto a **true** (utilizando el segundo argumento del constructor). Si se intenta establecer más de un botón de opción a **true** sólo quedará con este valor el último al que se le asigne.

He aquí un ejemplo simple del uso de botones de opción. Nótese que se pueden capturar eventos de botones de opción, al igual que con los otros:

```
//: c13:RadioButtons.java
// Using JRadioButtons.
// <applet code=RadioButtons
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class RadioButtons extends JApplet {
  JTextField t = new JTextField(15);
  ButtonGroup g = new ButtonGroup();
  JRadioButton
    rb1 = new JRadioButton("one", false),
    rb2 = new JRadioButton("two", false),
    rb3 = new JRadioButton("three", false);
  ActionListener al = new ActionListener() {
```

```
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio button " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public void init() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    cp.add(rb1);
    cp.add(rb2);
    cp.add(rb3);
  }
  public static void main(String[] args) {
    Console.run(new RadioButtons(), 200, 100);
  }
} ///:~
```

Para mostrar el estado, se usa un campo de texto. Este campo se pone a no editable pues se usa para mostrar datos, no para recogerlos. Por consiguiente, es una alternativa al uso de una **JLabel.**

Combo boxes (listas desplega bles)

Al igual que un grupo de botones de opción, una lista desplegable es una forma de obligar al usuario a seleccionar sólo un elemento a partir de un grupo de posibles elementos. Sin embargo, es una forma más compacta de lograrlo, y es más fácil cambiar los elementos de la lista sin sorprender **al** usuario. (Se pueden cambiar botones de opción dinámicamente, pero esto tiende a ser demasiado visible.)

El JComboBox de Java no es como el cuadro combinado de Windows, que permite seleccionar a partir de una lista o tipo de la propia selección. Con un **JComboBox** se puede elegir uno y sólo un elemento de la lista. En el ejemplo siguiente, la **JComboBox** comienza con cierto número de entradas, añadiéndosele otras cuando se presiona un botón:

```
//: c13:ComboBoxes.java
// Using drop-down lists.
// <applet code=ComboBoxes
// width=200 height=100> </applet>
import javax.swing.*;
```

```
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class ComboBoxes extends JApplet {
  String[] description = { "Ebullient", "Obtuse",
    "Recalcitrant", "Brilliant", "Somnescent",
    "Timorous", "Florid", "Putrescent" };
  JTextField t = new JTextField(15);
  JComboBox c = new JComboBox();
  JButton b = new JButton("Add items");
  int count = 0;
  public void init() {
    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());
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    cp.add(c);
    cp.add(b);
  }
  public static void main(String[] args) {
    Console.run(new ComboBoxes(), 200, 100);
  }
} ///:~
```

La **JTextField** muestra los "índices seleccionados", que es la secuencia del elemento actualmente seleccionado, así como la etiqueta del botón de opción.

Listas

Estas cajas son significativamente diferentes de las **JComboBox**, y no sólo en apariencia. Mientras que una **JComboBox** se despliega al activarla, una **JList** ocupa un número fijo de líneas en la pantalla todo el tiempo y no cambia. Si se

desea ver los elementos de la lista, simplemente se invoca a **getSelectedValues**( ), que produce un array de **String** de los elementos seleccionados.

Una **JList** permite selección múltiple: si se hace control-clic en más de un elemento (manteniendo pulsada la tecla "control" mientras que se llevan a cabo varios clic de ratón) el elemento original sigue resaltado y se puede seleccionar tantos como se desee. Si se selecciona un elemento, y después se pulsa mayúsculas-clic en otro elemento, se seleccionan todos los elementos comprendidos entre ambos. Para eliminar un elemento de un grupo se puede hacer control-clic sobre él.

```
//: c13:List.java
// <applet code=List width=250
// height=375> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class List extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  DefaultListModel lItems=new DefaultListModel();
  JList lst = new JList(lItems);
  JTextArea t = new JTextArea(flavors.length,20);
  JButton b = new JButton("Add Item");
  ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(count < flavors.length) {
        lItems.add(0, flavors[count++]);
      } else {
        // Disable, since there are no more
        // flavors left to be added to the List
        b.setEnabled(false);
      }
    }
  };
  ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] items=lst.getSelectedValues();
          for(int i = 0; i < items.length; i++)
```

```
            t.append(items[i] + "\n");
        }
    };
  int count = 0;
  public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Create Borders for components:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.black);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Add the first four items to the List
    for(int i = 0; i < 4; i++)
      lItems.addElement(flavors[count++]);
    // Add items to the Content Pane for Display
    cp.add(t);
    cp.add(lst);
    cp.add(b);
    // Register event listeners
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
  }
  public static void main(String[] args) {
    Console.run(new List(), 250, 375);
  }
} ///:~
```

Cuando se presiona el botón, añade elementos a la parte **superior** de la lista (porque el segundo argumento de **addItem**( ) es 0).

Podemos ver que también se han añadido bordes a las listas.
Si se desea poner un array **de Strings** en una **JList,** hay una solución mucho más simple: se pasa el array al constructor **JList**, y construye la lista automáticamente. **La** única razón para usar el "modelo lista" en el ejemplo de arriba es **que** la lista puede ser manipulada durante la ejecución del programa.

Las **JLists** no proporcionan soporte directo para el desplazamiento. Por supuesto, todo lo que hay que hacer es envolver la **JList** en un **JScrollPane** y se logrará la gestión automática de todos los detalles.

Paneles Tabulados

El **JTabbedPane** permite crear un "diálogo con lengüetas", que tiene lengüetas de carpetas de archivos ejecutándose a través de un borde, y todo lo que hay que hacer es presionar una lengüeta para presentar un diálogo diferente:

```
//: c13:TabbedPane1.java
// Demonstrates the Tabbed Pane.
// <applet code=TabbedPane1
// width=350 height=200> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class TabbedPane1 extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  JTabbedPane tabs = new JTabbedPane();
  JTextField txt = new JTextField(20);
  public void init() {
    for(int i = 0; i < flavors.length; i++)
      tabs.addTab(flavors[i],
        new JButton("Tabbed pane " + i));
    tabs.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        txt.setText("Tab selected: " +
          tabs.getSelectedIndex());
      }
    });
    Container cp = getContentPane();
    cp.add(BorderLayout.SOUTH, txt);
    cp.add(tabs);
  }
  public static void main(String[] args) {
    Console.run(new TabbedPane1(), 350, 200);
  }
} ///:~
```

En Java, el uso de algún tipo de mecanismo de "paneles tabulados" es bastante importante pues en programación de ***applets*** se suele intentar desmotivar el uso de diálogos emergentes añadiendo automáticamente un pequeño aviso a cualquier diálogo emergente de un ***applet.***
Cuando se ejecute el programa se verá que el **JTabbedPane** comprime las lengüetas si hay demasiadas, de forma que quepan en una fila. Podemos ver esto redimensionando la ventana al ejecutar el programa desde la línea de comandos de la consola.

Cajas de mensajes

Los entornos de ventanas suelen contener un conjunto estándar de cajas de mensajes que permiten enviar información al usuario rápidamente, o capturar información proporcionada por éste. En Swing, estas cajas de mensajes se encuentran en **JOptionPane.** Hay muchas posibilidades diferentes (algunas bastante sofisticadas), pero las que más habitualmente se usan son probablemente el diálogo mensaje y el diálogo confirmación, invocados usando el **static JOptionPanel.showMessageDialog**( ) **y JOptionPane.showConfirmDialog**( ). El ejemplo siguiente muestra un subconjunto de las cajas de mensajes disponibles con **JOptionPane:**

```
//: c13:MessageBoxes.java
// Demonstrates JoptionPane.
// <applet code=MessageBoxes
// width=200 height=150> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class MessageBoxes extends JApplet {
  JButton[] b = { new JButton("Alert"),
    new JButton("Yes/No"), new JButton("Color"),
    new JButton("Input"), new JButton("3 Vals")
  };
  JTextField txt = new JTextField(15);
  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 void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        cp.add(b[i]);
      }
      cp.add(txt);
    }
    public static void main(String[] args) {
      Console.run(new MessageBoxes(), 200, 200);
    }
} ///:~
```

Para poder escribir un único **ActionListener**, he usado un enfoque con riesgo, comprobando las etiquetas **String** de los botones. El problema de todo esto es que es fácil provocar algún fallo con las etiquetas en el uso de mayúsculas, y este fallo podría ser difícil de localizar.

Nótese que **showOptionDialog**( ) y **showInputDialog**( ) proporcionan objetos de retorno que contienen el valor introducido por el usuario.

Menús

Cada componente capaz de guardar un menú, incluyendo **JApplet**, **JFrame**, **JDialog** y sus descendientes, tiene un método **setMenuBar**( ) que acepta un **JMenuBar** (sólo se puede tener un **JMenuBar** en un componente particular). Se añaden **JMenus** al **JMenuBar**, y **JMenuItems** a los **JMenus**. Cada **JMenuItem** puede tener un **ActionListener** asociado, que se dispara al seleccionar ese elemento del menú.

A diferencia de un sistema basado en el uso de recursos, con Java y Swing hay que ensamblar a mano todos los menús en código fuente. He aquí un ejemplo de menú sencillo:

```
//: c13:SimpleMenus.java
// <applet code=SimpleMenus
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class SimpleMenus extends JApplet {
  JTextField t = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
  };
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  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 void init() {
    for(int i = 0; i < items.length; i++) {
      items[i].addActionListener(al);
      menus[i%3].add(items[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
      mb.add(menus[i]);
    setJMenuBar(mb);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
  }
  public static void main(String[] args) {
    Console.run(new SimpleMenus(), 200, 75);
  }
} ///:~
```

El uso del operador módulo en "**i%3**" distribuye los elementos de menú entre los tres **JMenus**. Cada **JMenuItem** debe tener un **ActionListener** adjunto; aquí se

usa el mismo **ActionListener** en todas partes pero generalmente será necesario uno individual para cada **JMenuItem**.

JMenuItem hereda de **AbstractButton**, por lo que tiene algunos comportamientos propios de los botones. Por sí mismo, proporciona un elemento que se puede ubicar en un menú desplegable. También hay tres tipos heredados de **JMenuItem**: **JMenu** para albergar otros **JMenuItems** (de forma que se pueden tener menús en cascada), **JCheckBoxMenuItem**, que produce una marca que indica si se ha seleccionado o no ese elemento de menú, y **JRadioButtonMenuItem**, que contiene un botón de opción.

Como ejemplo más sofisticado, he aquí el de los sabores de helado de nuevo, utilizado para crear menús. Este ejemplo también muestra menús en cascada, mnemónicos de teclado, **JCheckBoxMenuItems**, y la forma de cambiar menús dinámicamente:

```
//: c13:Menus.java
// Submenus, checkbox menu items, swapping menus,
// mnemonics (shortcuts) and action commands.
// <applet code=Menus width=300
// height=100> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Menus extends JApplet {
  String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };
  JTextField t = new JTextField("No flavor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("File"),
    m = new JMenu("Flavors"),
    s = new JMenu("Safety");
  // Alternative approach:
  JCheckBoxMenuItem[] safety = {
    new JCheckBoxMenuItem("Guard"),
    new JCheckBoxMenuItem("Hide")
  };
  JMenuItem[] file = {
    new JMenuItem("Open"),
  };
  // A second menu bar to swap to:
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
```

```
    JMenuItem[] other = {
      // Adding a menu shortcut (mnemonic) is very
      // simple, but only JMenuItems can have them
      // in their constructors:
      new JMenuItem("Foo", KeyEvent.VK_F),
      new JMenuItem("Bar", KeyEvent.VK_A),
      // No shortcut:
      new JMenuItem("Baz"),
    };
    JButton b = new JButton("Swap Menus");
    class BL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Refresh the frame
      }
    }
    class ML implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        String actionCommand =
          target.getActionCommand();
        if(actionCommand.equals("Open")) {
          String s = t.getText();
          boolean chosen = false;
          for(int i = 0; i < flavors.length; i++)
            if(s.equals(flavors[i])) chosen = true;
          if(!chosen)
            t.setText("Choose a flavor first!");
          else
            t.setText("Opening "+ s +". Mmm, mm!");
        }
      }
    }
    class FL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
      }
    }
    // Alternatively, you can create a different
    // class for each different MenuItem. Then you
    // Don't have to figure out which one it is:
    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 cold? " + target.getState());
    }
  }
  public void init() {
    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[0].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
      JMenuItem mi = new JMenuItem(flavors[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Add separators at intervals:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
      s.add(safety[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
      file[i].addActionListener(fl);
```

```
      f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);
    setJMenuBar(mb1);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);
    // Set up the system for swapping menus:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    cp.add(b, BorderLayout.NORTH);
    for(int i = 0; i < other.length; i++)
      fooBar.add(other[i]);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
  public static void main(String[] args) {
    Console.run(new Menus(), 300, 100);
  }
} ///:~
```

En este programa hemos ubicado los elementos del menú en arrays y después se recorre cada array invocando a **add**( ) por cada **JMenuItem**. De esta forma se logra que la adición o substracción de un elemento de menú sea bastante menos tediosa.

Este programa crea no uno, sino dos **JMenuBar** para demostrar que se pueden intercambiar barras de menú activamente mientras se ejecuta el programa. Se puede ver cómo un **JMenuBar** está hecho de **JMenus**, y cada **JMenu** está hecho de **JMenuItems**, **JCheckBoxMenuItems**, o incluso otros **JMenus** (logrando así submenús). Cuando se ensambla un **JMenuBar**, éste puede instalarse en el programa ac**tual** con el método **setTMenuBar**( ). Nótese que **al** presionar el botón, se comprueba qué menú está actualmente instalado invocando a **getJMenuBar(**), **y** pone la otra barra de menú en su sitio.

Al probar "Abrir" nótese que el deletreo y el uso de mayúsculas es crítico, pero Java no señala ningún error si no hay coincidencia exacta con "Abrir". Este tipo de comparación de cadenas de caracteres es una fuente de errores de programación.

La comprobación y la no comprobación de los elementos de menú se llevan a cabo automáticamente. El código que gestiona los **JCheckBoxMenuItems** muestran dos formas de determinar lo que se comprobó: la comprobación de cadenas de caracteres (que, como se mencionó arriba, no es un enfoque muy seguro, aunque se verá a menudo) y la coincidencia de todos los objetos destino de eventos. Como se ha mostrado, el método **getState**( ) puede usarse para revelar el estado. También se puede cambiar el estado de un

JCheckBoxMenuItem con **setState**( ). Los eventos de los menús son un poco inconsistentes y pueden conducir a confusión: los **JMenuItems** usan **ActionListeners**, pero los **JCheckboxMenuItems** usan **ItemListeners**. Los objetos **JMenu** también pueden soportar **ActionListeners**, pero eso no suele ser de ayuda. En general, se adjuntarán oyentes a cada **JMenuItem**, **JCheckBoxMenuItem** o **JRadioButtonMenuItem**, pero el ejemplo muestra **ItemListeners** y **ActionListeners** adjuntados a los distintos componentes menú.

Swing soporta mnemónicos o "atajos de teclado", de forma que se puede seleccionar cualquier cosa derivada de **AbstractButton** (botón, elemento de menú, etc.) utilizando el teclado en vez del ratón. Éstos son bastante simples: para **JMenuItem** se puede usar el constructor sobrecargado que toma como segundo argumento el identificador de la clave. Sin embargo, la mayoría de **AbstractButtons** no tiene constructores como éste, por lo que la forma más general de solucionar el problema es usar el método **setMnemonic**( ). El ejemplo de arriba añade mnemónicos al botón y a algunos de los elementos de menús; los indicadores de atajos aparecen automáticamente en los componentes.

También se puede ver el uso de **setActionCommand**( ). Éste parece un poco extraño porque en cada caso el "comando de acción" es exactamente el mismo que la etiqueta en el componente del menú. (¿Por qué no usar simplemente la etiqueta en vez de esta cadena de caracteres alternativa? El problema es la internacionalización. Si se vuelve a direccionar el programa a otro idioma, sólo se deseará cambiar la etiqueta del menú, y no cambiar el código (lo cual podría sin duda introducir nuevos errores). Por ello, para que esto sea sencillo para el código que comprueba la cadena de texto asociada a un componente de menú, se puede mantener inmutable el "comando de acción" a la vez que se puede cambiar la etiqueta de menú. Todo el código funciona con el "comando de acción", pero no se ve afectada por los cambios en las etiquetas de menú. Nótese que en este programa, no se examinan los comandos de acción de todos los componentes del menú, por lo que los no examinados no tienen su comando de acción.

La mayoría del trabajo se da en los oyentes. **BL** lleva a cabo el intercambio de **JMenuBar**. En **ML**, se toma el enfoque de "averigua quién llama" logrando la fuente del **ActionEvent** y convirtiéndola en un **JMenuItem**, consiguiendo después la cadena de caracteres del comando de acción para pasarlo a través de una sentencia **if** en cascada.

El oyente **FL** es simple, incluso aunque esté gestionando todos los sabores distintos del menú de sabores. Este enfoque es útil para tomar el enfoque usado con **FooL**, **BarL** y **BazL**, en los que sólo se les ajunta un componente menú de forma que no es necesaria ninguna lógica extra de detección y se sabe exactamente quién invocó al oyente. Incluso con la profusión de clases generadas de esta manera, el código interno tiende a ser menor y el proceso es más a prueba de torpes.

Se puede ver que el código de menú se vuelve largo y complicado rápidamente. Éste es otro caso en el que la solución apropiada es usar un constructor de IGU. Una buena herramienta también gestionará el mantenimiento de menús.

Menús emergentes

La forma más directa de implementar un **JPopupMenu** es crear una clase interna que extienda
MouseAdapter, después añadir un objeto de esa clase interna a cada componente que se desea para producir un comportamiento emergente:

```
//: c13:Popup.java
// Creating popup menus with Swing.
// <applet code=Popup
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Popup extends JApplet {
  JPopupMenu popup = new JPopupMenu();
  JTextField t = new JTextField(10);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.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) {
    Console.run(new Popup(), 300, 200);
  }
} ///:~
```

Se añade el mismo **ActionListener** a cada **JMenuItem**, de forma que tome el texto de la etiqueta de menú y lo inserte en el **JTextField**.

Generación de dibujos

En un buen marco de trabajo de **IGU,** la generación de dibujos debería ser razonablemente sencilla -y lo es, en la biblioteca Swing. El problema del ejemplo de dibujo es que los cálculos que determinan dónde van las cosas son bastante más complicados que las rutinas a las llamadas de generación de dibujos, y estos cálculos suelen mezclarse junto con las llamadas a dibujos de forma que puede parecer que la interfaz es más complicada que lo que es en realidad.

Por simplicidad, considérese el problema de representar datos en la pantalla -aquí, los datos los proporcionará el método **Math.sin**( ) incluido que es la función matemática seno. Para hacer las cosas un poco más interesantes, y para demostrar más allá lo fácil que es utilizar componentes Swing, se puede colocar un deslizador en la parte de abajo del formulario para controlar dinámicamente el número de ondas cíclicas sinoidales que se muestran. Además, si se redimensiona la ventana, se verá que la onda seno se reajusta al nuevo tamaño de la ventana.
Aunque se puede pintar cualquier **JComponent**, y usarlo, por consiguiente, como lienzo, si simplemente se desea una superficie de dibujo, generalmente se heredará de un **JPanel** (es el enfoque más directo). Sólo hay que superponer el método **paintComponent**( ), que se invoca siempre que hay que repintar ese componente (generalmente no hay que preocuparse por esto pues se encarga

Swing). Cuando es invocado, Swing le pasa un objeto **Graphics**, que puede usarse para dibujar o pintar en la superficie.

En el ejemplo siguiente, toda la inteligencia de pintado está en la clase **DibujarSeno**; la clase **OndaSeno** simplemente configura el programa y el control de deslizamiento. Dentro de **DibujarSeno**, el método **establecerCiclos**( ) proporciona la posibilidad de permitir a otro objeto -en este caso el control de deslizamientos- controlar el número de ciclos.

```
//: c13:SineWave.java
// Drawing with Swing, using a JSlider.
// <applet code=SineWave
//  width=700 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class SineDraw extends JPanel {
  static final int SCALEFACTOR = 200;
  int cycles;
  int points;
  double[] sines;
  int[] pts;
  SineDraw() { setCycles(5); }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    pts = new int[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI/SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maxWidth = getWidth();
    double hstep = (double)maxWidth/(double)points;
    int maxHeight = getHeight();
    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 class SineWave extends JApplet {
  SineDraw sines = new SineDraw();
  JSlider cycles = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(sines);
    cycles.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        sines.setCycles(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, cycles);
  }
  public static void main(String[] args) {
    Console.run(new SineWave(), 700, 400);
  }
} ///:~
```

Todos los miembros de datos y arrays se usan en el cálculo de los puntos de la onda senoidal: **ciclos** indica el número de ondas senoidales completas deseadas; **puntos** contiene el número total de puntos a dibujar, **senos** contiene los valores de la función seno, y **pts** contiene las coordenadas y de los puntos a dibujar en el **JPanel.** El método **establecerCiclos**( ) crea los arrays de acuerdo con el número de puntos deseados y rellena el array **senos** con números. Llamar a **repaint**( ), fuerza a **establecerCiclos**( ) a que se invoque a **paintComponent**( ), de forma que se lleven a cabo el resto de cálculos y redibujado.

Lo primero que hay que hacer al superponer **paintComponent**( ) es invocar a la versión de la clase base del método. Después se puede hacer lo que se desee; normalmente, esto implica usar los métodos **Graphics** que se pueden encontrar en la documentación de **java.awt.Graphics** (en la documentación HTML de *http://java.sun.com*) para dibujar y pintar píxeles en el **JPanel.** Aquí, se puede ver que casi todo el código está involucrado en llevar a cabo los cálculos; de hecho, las dos únicas llamadas a métodos que manipulan la pantalla son **setColor**( ) y **drawline**( ). Probablemente se tendrá una experiencia similar al crear programas que muestren datos gráficos -se invertirá la mayor parte del tiempo en averiguar qué es lo que se desea dibujar, mientras que el proceso de dibujado en sí será bastante simple.

Cuando creamos este programa, la mayoría del tiempo se invirtió en mostrar la onda seno en la pantalla. Una vez que lo logramos, pensamos que sería bonito

poder cambiar dinámicamente el número de ciclos. Mis experiencias programando al intentar hacer esas cosas en otros lenguajes, me hicieron dudar un poco antes de acometerlo, pero resultó ser la parte más fácil del proyecto. Creamos un **JSlider** (los argumentos son valores más a la izquierda del **JSlider,** los de más a la derecha y el valor de comienzo, respectivamente, pero también hay otros constructores) y lo volcamos en el **JApplet.** Después miramos en la documentación HTML y descubrimos que el único oyente era **addchangelistener,** que fue disparado siempre que se cambiase el deslizador lo suficiente como para producir un valor diferente. El único método para esto era el **stateChanged**( ), de nombre obvio, que proporcionó un objeto **ChangeEvent,** de forma que podía mirar hacia atrás a la fuente del cambio y encontrar el nuevo valor. Llamando a **establecerCiclos**( ) del objeto **senos,** se incorporó el nuevo valor y se redibujó el **JPanel.**

En general, se verá que la mayoría de los problemas Swing se pueden solucionar siguiendo un proceso similar, y se averiguará que suele ser bastante simple, incluso si nunca antes se ha usado un componente particular.

Si el problema es más complejo, hay otras alternativas de dibujo más sofisticadas, incluyendo componentes JavaBeans de terceras partes y el **API** Java 2D. Estas soluciones se escapan del alcance de este libro, pero habría que investigarlo cuando el código de dibujo se vuelve demasiado oneroso.

Cajas de diálogo

Una caja de diálogo es una ventana que saca otra ventana. Su propósito es tratar con algún aspecto específico sin desordenar la ventana original con esos detalles. Las cajas de diálogo se usan muy intensivamente en entornos de programación con ventanas, pero se usan menos frecuentemente en los ***applets.***

Para crear una caja de diálogo, se hereda de **JDialog**, que es simplemente otra clase de **Window**, como **JFrarne**. Un **JDialog** tiene un gestor de disposiciones (por defecto **BorderLayout**) y se le añaden oyentes para que manipulen eventos. Una diferencia significativa **al** llamar a **windowClosing**( ) es que no se desea apagar la aplicación. En vez de esto, se liberan los recursos usados por la ventana de diálogos llamando a **dispose**( ). He aquí un ejemplo muy sencillo:

```
//: c13:Dialogs.java
// Creating and using Dialog Boxes.
// <applet code=Dialogs width=125 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
class MyDialog extends JDialog {
  public MyDialog(JFrame parent) {
    super(parent, "My dialog", true);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("Here is my dialog"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dispose(); // Closes the dialog
      }
    });
    cp.add(ok);
    setSize(150,125);
  }
}

public class Dialogs extends JApplet {
  JButton b1 = new JButton("Dialog Box");
  MyDialog dlg = new MyDialog(null);
  public void init() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dlg.show();
      }
    });
    getContentPane().add(b1);
  }
  public static void main(String[] args) {
    Console.run(new Dialogs(), 125, 75);
  }
} ///:~
```

Una vez que se crea el **JDialog**, se debe llamar al método **show**( ) para mostrarlo y activarlo. Para que se cierre el diálogo hay que llamar a **dispose**( ).

Veremos que cualquier cosa que surja de un objeto, incluyendo las cajas de diálogo, "no es de confianza". Es decir, se obtienen advertencias al usarlas. Esto es porque, en teoría, sería posible confundir al usuario y hacerle pensar que está tratando con una aplicación nativa regular y hacer que tecleen el número de su tarjeta de crédito que viajará después por la Web. Un ***applet*** siempre viaja adjunto a una página web, y será visible desde un navegador, mientas que las cajas de diálogo se desasocian -por lo que sería posible, en teoría. Como resultado no suele ser frecuente ver ***applets*** que hagan uso de cajas de diálogo.

El ejemplo siguiente es más complejo; la caja de diálogo consta de una rejilla (usando **GridLayout**) **de** un tipo especial de botón definido como clase **BotonToe**. Este botón dibuja un marco en torno a sí mismo y, dependiendo de su

estado, un espacio en blanco, una **"x"** o una "o" en el medio. Comienza en blanco y después, dependiendo de a quién le toque, cambia a **"x"** o a "o". Sin embargo, también cambiará entre **"x"** y "o" al hacer clic en el botón. (Esto convierte el concepto tic-tac-toe en sólo un poco más asombroso de lo que ya es.) Además, se puede establecer que la caja de diálogo tenga un número cualquiera de filas y columnas, cambiando los números de la ventana de aplicación principal.

```
//: c13:TicTacToe.java
// Demonstration of dialog boxes
// and creating your own components.
// <applet code=TicTacToe
//  width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TicTacToe extends JApplet {
  JTextField
    rows = new JTextField("3"),
    cols = new JTextField("3");
  static final int BLANK = 0, XX = 1, OO = 2;
  class ToeDialog extends JDialog {
    int turn = XX; // Start with x's turn
    // w = number of cells wide
    // h = number of cells high
    public ToeDialog(int w, int h) {
      setTitle("The game itself");
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        cp.add(new ToeButton());
      setSize(w * 50, h * 50);
      // JDK 1.3 close dialog:
      //#setDefaultCloseOperation(
      //#  DISPOSE_ON_CLOSE);
      // JDK 1.2 close dialog:
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
          dispose();
        }
      });
    }
    class ToeButton extends JPanel {
      int state = BLANK;
      public ToeButton() {
        addMouseListener(new ML());
      }
      public void paintComponent(Graphics g) {
```

```
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
        if(state == XX) {
          g.drawLine(x1, y1,
            x1 + wide, y1 + high);
          g.drawLine(x1, y1 + high,
            x1 + wide, y1);
        }
        if(state == OO) {
          g.drawOval(x1, y1,
            x1 + wide/2, y1 + high/2);
        }
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(state == BLANK) {
            state = turn;
            turn = (turn == XX ? OO : XX);
          }
          else
            state = (state == XX ? OO : XX);
          repaint();
        }
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JDialog d = new ToeDialog(
        Integer.parseInt(rows.getText()),
        Integer.parseInt(cols.getText()));
      d.setVisible(true);
    }
  }
  public void init() {
    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);
    Container cp = getContentPane();
```

```
    cp.add(p, BorderLayout.NORTH);
    JButton b = new JButton("go");
    b.addActionListener(new BL());
    cp.add(b, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    Console.run(new TicTacToe(), 200, 100);
  }
} ///:~
```

Dado que los **statics** sólo pueden estar en el nivel externo de la clase, las clases internas no pueden tener datos **static** o clases internas **static**.

El método **paintComponent**( ) dibuja el cuadrado alrededor del panel y la "x" o la "o". Esto implica muchos cálculos tediosos pero es directo.

El **MouseListener** captura los eventos de ratón, **y** comprueba en primer lugar si el panel ha escrito algo. Si no, se invoca a la ventana padre para averiguar a quién le toca, **y** qué se usa para establecer el estado del **BotonToe**. Vía el mecanismo de clases internas, posteriormente el **BotonToe** vuelve al padre y cambia el turno. Si el botón ya está mostrando una "x" o una **"o",** se cambia. En estos cálculos se puede ver el uso del "**if**-**else**" ternario descrito en el Capítulo **3.** Después de un cambio de estado, se repinta el **BotonToe**.

El constructor de **DialogoToe** es bastante simple: añade en un **GridLayout** tantos botones como se solicite, y después lo redimensiona a **50** píxeles de lado para cada botón.

TicTacToe da entrada a toda la aplicación creando los **JTextFields** (para la introducción de las filas y columnas de la rejilla de botones) y el botón "comenzar" con su **ActionListener**. Cuando se presiona este botón se recogen todos los datos de los **JTextFields**, y puesto que se encuentran en forma de **Strings**, se convierten en **ints** usando el método **static Integer.parseInt**( ).

Diálogos de archivo

Algunos sistemas operativos tienen varias cajas de diálogo pre-construidas para manejar la selección de ciertos elementos como fuentes, colores, impresoras, etc. Generalmente, todos los sistemas operativos gráficos soportan la apertura y salvado de archivos, sin embargo, el **JFileChooser** de Java encapsula estas operaciones para facilitar su uso.

La aplicación siguiente ejercita dos formas de diálogos **JFileChooser**, uno para abrir y otro para guardar. **La** mayoría del código debería parecer ya familiar al lector, y es en los oyentes de acciones de ambos clic sobre los botones donde se dan las operaciones más interesantes:

```
//: c13:FileChooserTest.java
// Demonstration of File dialog boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class FileChooserTest extends JFrame {
  JTextField
    filename = new JTextField(),
    dir = new JTextField();
  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);
    Container cp = getContentPane();
    cp.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);
    cp.add(p, BorderLayout.NORTH);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demonstrate "Open" dialog:
      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();
      // Demonstrate "Save" dialog:
      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) {
    Console.run(new FileChooserTest(), 250, 110);
  }
} ///:~
```

Nótese que se pueden aplicar muchas variaciones a **JFileChooser,** incluyendo filtros para limitar los nombres de archivo permitidos.

Para un diálogo abrir archivo se puede invocar a **showOpenDialog**( ), y en el caso de salvar archivo el diálogo al que se invoca es **showSaveDialog**( ). Estos comandos no devuelven nada hasta cerrar el diálogo. El objeto **JFileChooser** sigue existiendo, pero se pueden leer datos del mismo. Los métodos **getSelectedFile**( ) y **getCurrentDirectory**( ) son dos formas de interrogar por los resultados de la operación. Si devuelven **null** significa que el usuario canceló el diálogo.

HTML en componentes Swing

Cualquier componente que pueda tomar texto, también puede tomar texto **HTML,** al que dará formato de acuerdo a reglas **HTML.** Esto significa que se puede añadir texto elegante a los componentes Swing fácilmente. Por ejemplo:

```
//: c13:HTMLButton.java
// Putting HTML text on Swing components.
// <applet code=HTMLButton width=200 height=500>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class HTMLButton extends JApplet {
  JButton b = new JButton("<html><b><font size=+2>" +
    "<center>Hello!<br><i>Press me now!");
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        getContentPane().add(new JLabel("<html>"+
          "<i><font size=+4>Kapow!"));
        // Force a re-layout to
        // include the new label:
        validate();
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b);
  }
  public static void main(String[] args) {
    Console.run(new HTMLButton(), 200, 500);
  }
} ///:~
```

Hay que empezar el texto con '<**htrnl**>' y después se pueden usar etiquetas HTML normales. Nótese que no hay obligación de incluir las etiquetas de cierre habituales.

El **ActionListener** añade una etiqueta **JLabel** nueva al formulario, que también contiene texto **HTML.** Sin embargo, no se añade esta etiqueta durante **hit**( ) por lo que hay que llamar al método **validate**( ) del contenedor para forzar una redistribución de los componentes (y por consiguiente mostrar la nueva etiqueta).

También se puede usar texto **HTML** para **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** y **JCheckBox**.

Deslizadores y barras de progreso

Un deslizador (que ya se ha usado en el ejemplo de la onda senoidal) permite al usuario introducir datos moviéndose hacia delante y hacia atrás, lo que es intuitivo en algunas situaciones (por ejemplo, controles de volumen). Una barra de progreso muestra datos en forma relativa, desde "lleno" hasta "vacío" de forma que el usuario obtiene una perspectiva. Nuestro ejemplo favorito para éstos es simplemente vincular el deslizador a la barra de progreso de forma que al mover uno el otro varíe en consecuencia:

```
//: c13:Progress.java
// Using progress bars and sliders.
```

```
// <applet code=Progress
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Progress extends JApplet {
  JProgressBar pb = new JProgressBar();
  JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2,1));
    cp.add(pb);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Slide Me"));
    pb.setModel(sb.getModel()); // Share model
    cp.add(sb);
  }
  public static void main(String[] args) {
    Console.run(new Progress(), 300, 200);
  }
} ///:~
```

La clave para vincular juntos ambos elementos es compartir su modelo en la línea:

```
pb.setModel(sb.getModel());
```

Por supuesto, también se podría controlar ambos usando un oyente, pero esto es más directo en situaciones simples.

El **JProgressBar** es bastante directo, pero el **JSlider** tiene un montón de opciones, como la orientación y las marcas de mayor y menor. Nótese lo directo que es añadir un borde.

Árboles

Utilizar un **JTree** puede ser tan simple como decir:

```
add(new JTree(
```

```
    new Object[] {"this", "that", "other"}));
```

Esto muestra un árbol primitivo. Sin embargo, el **API** de los árboles es vasto - ciertamente uno de los mayores de Swing. Parece que casi se puede hacer cualquier cosa con los árboles, pero tareas más sofisticadas podrían requerir de bastante información y experimentación.

Afortunadamente, hay un terreno neutral en la biblioteca: los componentes árbol "por defecto", que generalmente hacen lo que se necesita. Por tanto, la mayoría de las veces se pueden usar estos componentes, y sólo hay que profundizar en los árboles en casos especiales.

El ejemplo siguiente usa los componentes árbol "por defecto" para mostrar un árbol en un ***applet.*** **Al** presionar el botón, se añade un nuevo subárbol bajo el nodo actualmente seleccionado (si no se selecciona ninguno se usa el nodo raíz):

```
//: c13:Trees.java
// Simple Swing tree example. Trees can
// be made vastly more complex than this.
// <applet code=Trees
//  width=250 height=250></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.bruceeckel.swing.*;

// Takes an array of Strings and makes the first
// element a node and the rest leaves:
class Branch {
  DefaultMutableTreeNode r;
  public Branch(String[] data) {
    r = new DefaultMutableTreeNode(data[0]);
    for(int i = 1; i < data.length; i++)
      r.add(new DefaultMutableTreeNode(data[i]));
  }
  public DefaultMutableTreeNode node() {
    return r;
  }
}

public class Trees extends JApplet {
  String[][] data = {
    { "Colors", "Red", "Blue", "Green" },
    { "Flavors", "Tart", "Sweet", "Bland" },
    { "Length", "Short", "Medium", "Long" },
    { "Volume", "High", "Medium", "Low" },
    { "Temperature", "High", "Medium", "Low" },
```

```
    { "Intensity", "High", "Medium", "Low" },
  };
  static int i = 0;
  DefaultMutableTreeNode root, child, chosen;
  JTree tree;
  DefaultTreeModel model;
  public void init() {
    Container cp = getContentPane();
    root = new DefaultMutableTreeNode("root");
    tree = new JTree(root);
    // Add it and make it take care of scrolling:
    cp.add(new JScrollPane(tree),
      BorderLayout.CENTER);
    // Capture the tree's model:
    model =(DefaultTreeModel)tree.getModel();
    JButton test = new JButton("Press me");
    test.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(i < data.length) {
          child = new Branch(data[i++]).node();
          // What's the last one you clicked?
          chosen = (DefaultMutableTreeNode)
            tree.getLastSelectedPathComponent();
          if(chosen == null) chosen = root;
          // The model will create the
          // appropriate event. In response, the
          // tree will update itself:
          model.insertNodeInto(child, chosen, 0);
          // This puts the new node on the
          // currently chosen node.
        }
      }
    });
    // Change the button's colors:
    test.setBackground(Color.blue);
    test.setForeground(Color.white);
    JPanel p = new JPanel();
    p.add(test);
    cp.add(p, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    Console.run(new Trees(), 250, 250);
  }
} ///:~
```

La primera clase, **Rama**, es una herramienta para tomar un array de **String** y construir un **DefaultMutableTreeNode** con el primer **String** como raíz y el resto de los **Strings** del array como hojas. Después se puede llamar a **nodo**( ) para producir la raíz de esta "rama".

La clase **Arboles** contiene un array bi-dimensional de **Strings** a partir del cual se pueden construir **Ramas**, y una **static int i** para recorrer este array. Los objetos **DefaultMutableTreeNode** guardan los nodos, pero la representación física en la pantalla la controla el **JTree** y su modelo asociado, el **DefaultTreeModel**. Nótese que cuando se añade el **JTree** al ***applet,*** se envuelve en un **JScrollPane** -esto es todo lo necesario para el desplazamiento automático.

E1 **JTree** lo controla su ***modelo.*** Cuando se hace un cambio al modelo, éste genera un evento que hace que el **JTree** desempeñe cualquier actualización necesaria a la representación visible del árbol.

En **bit**( ), se captura el modelo llamando a **getModel**( ). Cuando se presiona el botón, se crea una nueva "rama". Después, se encuentra el componente actualmente seleccionado (o se usa la raíz si es que no hay ninguno) y el método **insertNodeInto**( ) del modelo hace todo el trabajo de cambiar el árbol y actualizarlo.

Un ejemplo como el de arriba puede darnos lo que necesitamos de un árbol. Sin embargo, los árboles tienen la potencia de hacer casi todo lo que se pueda imaginar -en todas partes donde se ve la expresión "por defecto" dentro del ejemplo, podría sustituirse por otra clase para obtener un comportamiento diferente. Pero hay que ser conscientes: casi todas estas clases tienen grandes interfaces, por lo que se podría invertir mucho tiempo devanándonos los sesos para entender los aspectos intrínsecos de los árboles. Independientemente de esto, constituyen un buen diseño y cualquier alternativa es generalmente mucho peor.

Tablas

Al igual que los árboles, las tablas de Swing son vastas y potentes. Su intención principal era ser la interfaz "rejilla" popular para bases de datos vía Conectividad de Bases de Datos Java (JDBC, ***Java DataBase Connectivity,*** que se estudiará en el Capítulo 15), y por consiguiente, tienen una gran flexibilidad, a cambio de una carga de complejidad. Aquí hay materia suficiente como para un gran rompecabezas, e incluso para justificar la escritura de un libro entero. Sin embargo, también es posible crear una **JTable** relativamente simple si se entienden las bases.

La JTable controla la forma de mostrar los datos, pero el **TableModel** controla los datos en sí. Por ello, para crear una tabla, se suele crear primero el **TableModel.** Se puede implementar la interfaz **TableModel** al completo pero suele ser más simple heredar de la clase ayudante **AbstractTableModel:**

```
//: c13:Table.java
// Simple demonstration of JTable.
```

```
// <applet code=Table
//  width=350 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.*;
import com.bruceeckel.swing.*;

public class Table extends JApplet {
  JTextArea txt = new JTextArea(4, 20);
  // The TableModel controls all the data:
  class DataModel extends AbstractTableModel {
    Object[][] data = {
      {"one", "two", "three", "four"},
      {"five", "six", "seven", "eight"},
      {"nine", "ten", "eleven", "twelve"},
    };
    // Prints data when table changes:
    class TML implements TableModelListener {
      public void tableChanged(TableModelEvent e){
        txt.setText(""); // Clear it
        for(int i = 0; i < data.length; i++) {
          for(int j = 0; j < data[0].length; j++)
            txt.append(data[i][j] + " ");
          txt.append("\n");
        }
      }
    }
    public DataModel() {
      addTableModelListener(new TML());
    }
    public int getColumnCount() {
      return data[0].length;
    }
    public int getRowCount() {
      return data.length;
    }
    public Object getValueAt(int row, int col) {
      return data[row][col];
    }
    public void
    setValueAt(Object val, int row, int col) {
      data[row][col] = val;
      // Indicate the change has happened:
      fireTableDataChanged();
    }
    public boolean
    isCellEditable(int row, int col) {
      return true;
```

```
    }
  }
  public void init() {
    Container cp = getContentPane();
    JTable table = new JTable(new DataModel());
    cp.add(new JScrollPane(table));
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    Console.run(new Table(), 350, 200);
  }
} ///:~
```

ModeloDatos contiene un array de datos, pero también se podrían obtener los datos a partir de otra fuente, como una base de datos. El constructor añade un **TableModelListener** que imprime el array cada vez que se cambia la tabla. El resto de métodos siguen la convención de nombres de los Beans, y **JTable** los usa cuando quiere presentar la información en **ModeloDatos**. **AbstractTableModel** proporciona métodos por defecto para **setValueAt**( ) e **isCellEditable**( ) que evitan cambios a los datos, por lo que si se desea poder editar los datos, hay que superponer estos métodos.

Una vez que se tiene un **TableModel**, simplemente hay que pasárselo al constructor **JTable**. Se encargará de todos los detalles de presentación, edición y actualización. Este ejemplo también pone la **JTable** en un **JScrollPane**.

Seleccionar la Apariencia

Uno de los aspectos más interesantes de Swing es la "Apariencia œnectable". Ésta permite al programa emular la apariencia y comportamiento de varios entornos operativos. Incluso se puede hacer todo tipo de cosas elegantes como cambiar dinámicamente la apariencia y el comportamiento mientras se está ejecutando el programa. Sin embargo, por lo general simplemente se desea una de las dos cosas, o seleccionar un aspecto y comportamiento "multiplataformas" (el "metal" del Swing), o seleccionar el aspecto y comportamiento del sistema en el que se está, de forma que el programa

Java parece haber sido creado específicamente para ese sistema. El código para seleccionar entre estos comportamientos es bastante simple -pero hay que ejecutarlo ***antes*** de crear cualquier componente visual, puesto que los componentes se construirán basados en la apariencia actual y no se cambiarán simplemente porque se cambie la apariencia y el comportamiento a mitad de camino durante el programa (ese proceso es más complicado y fuera de lo común, y está relegado **a** libros específicos de Swing).

De hecho, si se desea usar el aspecto y comportamiento multiplataformas ("metal") característico de los programas Swing, no hay que hacer nada -es la opción por defecto. Pero si en vez de ello se desea usar el aspecto y comportamiento del entorno operativo actual, simplemente se inserta el código siguiente, generalmente el principio del **main**( ) pero de cualquier forma antes de añadir componentes:

```
try {
  UIManager.setLookAndFeel(UIManager.
    getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

No es necesario poner nada en la cláusula **catch** porque el **UIManager** redireccionará por defecto al aspecto y comportamiento multiplataforma si fallan los intentos de optar por otra alternativa. Sin embargo, durante la depuración, puede ser útil la excepción, pues al menos se puede desear poner una sentencia de impresión en la cláusula ***catch***.

He aquí un programa que toma un parámetro de línea de comandos para seleccionar un comportamiento y una imagen, y que muestra la apariencia de varios de estos comportamientos:

```
//: c13:LookAndFeel.java
// Selecting different looks & feels.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class LookAndFeel extends JFrame {
  String[] choices = {
    "eeny", "meeny", "minie", "moe", "toe", "you"
  };
  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");
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < samples.length; i++)
```

```
      cp.add(samples[i]);
  }
  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(System.err);
      }
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else usageError();
    // Note the look & feel must be set before
    // any components are created.
    Console.run(new LookAndFeel(), 300, 200);
  }
} ///:~
```

Se puede ver que una de las opciones es especificar explícitamente una cadena de caracteres para un aspecto y un comportamiento, como se ha visto con **MotifLookAndFeel**. Sin embargo, ése y el "metal" por defecto son los únicos que se pueden usar legalmente en cualquier plataforma; incluso aunque hay cadenas de caracteres para los aspectos y comportamientos de Windows y Macintosh, éstos sólo pueden usarse en sus plataformas respectivas (se producen al invocar a **getSystemLookAndFeelClassName**( ) estando en esa plataforma particular).

También es posible crear un paquete de "apariencia" personalizado, por ejemplo, si se está construyendo un marco de trabajo para una compañía que quiere una apariencia distinta. Éste es un gran trabajo y se escapa con mucho del alcance de

este libro (¡de hecho, se descubrirá que se escapa del alcance de casi todos los libros dedicados a Swing!).

El Portapapeles

Las JFC soportan algunas operaciones con el portapapeles del sistema (gracias al paquete *java.awt.datatransfer*). Se pueden copiar objetos **String** al portapapeles como si de texto se tratara, y se puede pegar texto de un portapapeles dentro de objetos **String**. Por supuesto, el portapapeles está diseñado para guardar cualquier tipo de datos, pero cómo represente el portapapeles la información que en él se deposite depende de cómo hagan el programa las copias y los pegados. El **API** de portapapeles de Java proporciona extensibilidad mediante el concepto de versión. Todos los datos provenientes del portapapeles tienen asociadas varias versiones en los que se puede convertir (por ejemplo, un gráfico podría representarse como un **string** de números o como una imagen) y es posible consultar si ese portapapeles en particular soporta la versión en la que uno está interesado.

El programa siguiente es una mera demostración de cortar, copiar y pegar con datos **String** dentro de un **JTextArea.** Fíjese que las secuencias de teclado que suelen usarse para cortar, copiar y pegar funcionan igualmente. Incluso si se echa un vistazo a cualquier **JTextArea** o **JTextField** de cualquier otro programa, se descubre que también soportan las secuencias de teclas correspondientes al portapapeles. Este ejemplo simplemente añade control programático del portapapeles, ofreciendo una serie de técnicas que pueden usarse siempre que se desee capturar texto en cualquier cosa que no sea un **JTextComponent.**

```
//: c13:CutAndPaste.java
// Using the clipboard.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceeckel.swing.*;

public class CutAndPaste extends JFrame  {
  JMenuBar mb = new JMenuBar();
  JMenu edit = new JMenu("Edit");
  JMenuItem
    cut = new JMenuItem("Cut"),
    copy = new JMenuItem("Copy"),
    paste = new JMenuItem("Paste");
  JTextArea text = new JTextArea(20, 20);
  Clipboard clipbd =
    getToolkit().getSystemClipboard();
  public CutAndPaste()  {
    cut.addActionListener(new CutL());
```

```
    copy.addActionListener(new CopyL());
    paste.addActionListener(new PasteL());
    edit.add(cut);
    edit.add(copy);
    edit.add(paste);
    mb.add(edit);
    setJMenuBar(mb);
    getContentPane().add(text);
  }
  class CopyL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String selection = text.getSelectedText();
      if (selection == null)
        return;
      StringSelection clipString =
        new StringSelection(selection);
      clipbd.setContents(clipString,clipString);
    }
  }
  class CutL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String selection = text.getSelectedText();
      if (selection == null)
        return;
      StringSelection clipString =
        new StringSelection(selection);
      clipbd.setContents(clipString, clipString);
      text.replaceRange("",
        text.getSelectionStart(),
        text.getSelectionEnd());
    }
  }
  class PasteL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Transferable clipData =
        clipbd.getContents(CutAndPaste.this);
      try {
        String clipString =
          (String)clipData.
            getTransferData(
              DataFlavor.stringFlavor);
        text.replaceRange(clipString,
          text.getSelectionStart(),
          text.getSelectionEnd());
      } catch(Exception ex) {
        System.err.println("Not String flavor");
      }
    }
  }
  public static void main(String[] args) {
```

```
    Console.run(new CutAndPaste(), 300, 200);
  }
} ///:~
```

La creación y adición del menú y del **JTextArea** deberían parecer ya una actividad de andar por casa. Lo diferente es la creación del campo **clipbd** de **Clipboard** que se lleva a cabo a través del **Toolkit.**

Toda la acción se da en los oyentes. Tanto el oyente **Copiar** como el **CortarL** son iguales excepto por la última línea de **CortarL,** que borra la línea que se está copiando. **Las** dos líneas especiales son la creación de un objeto **StringSelection** a partir del **String** y la llamada a **setContents**( ) con esta **StringSelection.** El propósito del resto del código es poner un **String** en el portapapeles.

En **PegarL,** se extraen datos del portapapeles utilizando **getContents**( ). Lo que se obtiene es un objeto **Transferable**, bastante anónimo, y se desconoce lo que contiene. Una forma de averiguarlo es llamar a **getTransferDataFlavours**( ), que devuelve un array de objetos **DataFlavor** que indica las versiones soportadas por ese objeto en particular. También se le puede preguntar directamente con **isDataFlavorSupported**( ) al que se le pasa la versión en la que uno está interesado. Aquí, sin embargo, se sigue el enfoque de invocar a **getTransferData**( ) asumiendo que el contenido soporta la versión **String**, y si no lo hace se soluciona el problema en el gestor de excepciones. Más adelante es posible que cada vez se de soporte a más versiones.

Empaquetando un applet en un archivo JAR

Un uso importante de la utilidad JAR es optimizar la carga de *applets.* En Java *1.0,* la gente tendía a intentar concentrar todo su código en una única clase *applet* de forma que el cliente sólo necesitaría un acceso al servidor para descargar el código del *applet.* Esto no sólo conducía a programas complicados y difíciles de leer, sino que el archivo **.class** seguía sin estar comprimido, por lo que su descarga no era tan rápida como podría haberlo sido.

Los archivos JAR solucionan el problema comprimiendo todos los archivos **.class** en un único archivo que descarga el navegador. Ahora se puede crear el diseño correcto sin preocuparse de cuántos archivos **.class** conllevará, y el usuario obtendrá un tiempo de descarga mucho menor.

Considérese **TicTacToe.java**. Parece contener una única clase, pero, de hecho, contiene cinco clases internas, con lo que el total son seis. Una vez compilado el programa, se empaqueta en un archivo JAR con la línea:

```
jar cf TicTacToe.jar *.class
```

Ahora se puede crear una página HTML con la nueva etiqueta **archive** para indicar el nombre del archivo JAR. He aquí la etiqueta usando la forma vieja de la etiqueta HTML, a modo ilustrativo:

```
<head><title>TicTacToe Example Applet
</title></head>
<body>
<applet code=TicTacToe.class
        archive=TicTacToe.jar
        width=200 height=100>
</applet>
</body>
```

Habrá que ponerlo en la forma nueva (más complicada y menos clara) vista anteriormente en este capítulo si queremos que funcione.

Técnicas de programación

Dado que la programación IGU en Java ha sido una tecnología en evolución con algunos cambios muy importantes entre Java 1.0/1.1 y la biblioteca Swing de Java 2, ha habido algunas estructuras y formas de obrar de programación antiguas que se han convertido precisamente en los ejemplos que vienen con Swing. Además, Swing permite programar de más y mejores formas que con los modelos viejos. En esta sección se demostrarán algunos de estos aspectos presentando y examinando algunos casos de programación.

Correspondencia dinámica de objetos

Uno de los beneficios del modelo de eventos de Swing es la flexibilidad. Se puede añadir o quitar comportamiento de eventos simplemente con llamadas a métodos. Esto se demuestra en el ejemplo siguiente:

```
//: c13:DynamicEvents.java
// You can change event behavior dynamically.
// Also shows multiple actions for an event.
// <applet code=DynamicEvents
//  width=250 height=400></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class DynamicEvents extends JApplet {
```

```
  ArrayList v = new ArrayList();
  int i = 0;
  JButton
    b1 = new JButton("Button1"),
    b2 = new JButton("Button2");
  JTextArea txt = new JTextArea();
  class B implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("A button was pressed\n");
    }
  }
  class CountListener implements ActionListener {
    int index;
    public CountListener(int i) { index = i; }
    public void actionPerformed(ActionEvent e) {
      txt.append("Counted Listener "+index+"\n");
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Button 1 pressed\n");
      ActionListener a = new CountListener(i++);
      v.add(a);
      b2.addActionListener(a);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Button2 pressed\n");
      int end = v.size() - 1;
      if(end >= 0) {
        b2.removeActionListener(
          (ActionListener)v.get(end));
        v.remove(end);
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());
    JPanel p = new JPanel();
    p.add(b1);
    p.add(b2);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(txt));
  }
  public static void main(String[] args) {
```

```
    Console.run(new DynamicEvents(), 250, 400);
  }
} ///:~
```

El nuevo enfoque de este ejemplo radica en:

1. Hay más de un oyente para cada **Button.** Generalmente, los componentes manejan eventos de forma *multidifusión,* lo que quiere decir que se pueden registrar muchos oyentes para un único evento. En los componentes especiales en los que un evento se maneje de forma *unidifusion,* se obtiene una **TooManyListenersException.**

2. Durante la ejecución del programa, se añaden y eliminan dinámicamente los oyentes del **Button b2.** La adición se lleva a cabo de la forma vista anteriormente, pero cada componente también tiene un método **removeXXXListener(**) para eliminar cualquier tipo de oyente.

Este tipo de flexibilidad permite una potencia de programación muchísimo mayor.

El lector se habrá dado cuenta de que no se garantiza que se invoque a los oyentes en el orden en el que se añaden (aunque de hecho, muchas implementaciones sí que funcionan así).

Separar la lógica de negocio de la lógica IU

En general, se deseará diseñar clases de forma que cada una "sólo haga una cosa". Esto es especialmente importante cuando se ve involucrado el código de interfaz de usuario, puesto que es fácil vincular "lo que se está haciendo" con "cómo mostrarlo". Este tipo de emparejamiento evita la reutilización de código. Es mucho más deseable separar la "lógica de negocio" del IGU. De esta forma, no sólo se puede volver a usar más fácilmente la lógica de negocio, sino que también se facilita la reutilización del IGU.

Otro aspecto son los sistemas ***multihilo*** en los que los "objetos de negocio" suelen residir en una máquina completamente separada. Esta localización central de las reglas de negocio permite que los cambios sean efectivos al instante para toda nueva transacción, y es, por tanto, un método adecuado para actualizar un sistema. Sin embargo, se pueden usar estos objetos de negocio en muchas aplicaciones de forma que no estén vinculados a una única forma de presentación en pantalla. Es decir, su propósito debería restringirse a llevar a cabo operaciones de negocio y nada más.

El ejemplo siguiente muestra lo sencillo que resulta separar la lógica de negocio del código IGU:

```
//: c13:Separation.java
// Separating GUI logic and business objects.
```

```
// <applet code=Separation
// width=250 height=150> </applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.applet.*;
import com.bruceeckel.swing.*;

class BusinessLogic {
  private int modifier;
  public BusinessLogic(int mod) {
    modifier = mod;
  }
  public void setModifier(int mod) {
    modifier = mod;
  }
  public int getModifier() {
    return modifier;
  }
  // Some business operations:
  public int calculation1(int arg) {
    return arg * modifier;
  }
  public int calculation2(int arg) {
    return arg + modifier;
  }
}

public class Separation extends JApplet {
  JTextField
    t = new JTextField(15),
    mod = new JTextField(15);
  BusinessLogic bl = new BusinessLogic(2);
  JButton
    calc1 = new JButton("Calculation 1"),
    calc2 = new JButton("Calculation 2");
  static int getValue(JTextField tf) {
    try {
      return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
      return 0;
    }
  }
  class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculation1(getValue(t))));
    }
  }
```

```
  class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculation2(getValue(t))));
    }
  }
  // If you want something to happen whenever
  // a JTextField changes, add this listener:
  class ModL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      bl.setModifier(getValue(mod));
    }
    public void removeUpdate(DocumentEvent e) {
      bl.setModifier(getValue(mod));
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    calc1.addActionListener(new Calc1L());
    calc2.addActionListener(new Calc2L());
    JPanel p1 = new JPanel();
    p1.add(calc1);
    p1.add(calc2);
    cp.add(p1);
    mod.getDocument().
      addDocumentListener(new ModL());
    JPanel p2 = new JPanel();
    p2.add(new JLabel("Modifier:"));
    p2.add(mod);
    cp.add(p2);
  }
  public static void main(String[] args) {
    Console.run(new Separation(), 250, 100);
  }
} ///:~
```

Se puede ver que **LogicaNegocio** es una clase directa que lleva a cabo sus operaciones sin ningún tipo de línea de código relacionada con un entorno IGU. Simplemente hace su trabajo. Separación mantiene un seguimiento de todos los detalles de IU, y se comunica con **LogicaNegocio** sólo a través de su interfaz **public**. Todas las operaciones se centran en conseguir información de ida y vuelta a través del IU y el objeto **LogicaNegocio**. Así, a cambio Separación simplemente hace su trabajo. Dado que Separación sólo sabe que está hablando a un objeto **LogicaNegocio** (es decir, no está altamente acoplado), se podría hacer que se comunique con otros tipos de objetos sin grandes problemas.

Pensar en términos de separar IU de la lógica de negocio también facilita las cosas cuando se está adaptando código antiguo para que funcione con Java.

Una forma canónica

Las clases internas, el modelo de eventos de Swing, y el hecho de que el viejo modelo de eventos siga siendo soportado junto con las nuevas facetas de biblioteca basadas en estilos de programación antiguos, vienen a añadir elementos de confusión al proceso de diseño de código. Ahora hay incluso más medios para que la gente escriba código desagradable.

Excepto en circunstancias de extenuación, siempre se puede usar el enfoque más simple y claro: clases oyente (escritas generalmente como clases internas) para solucionar necesidades de manejo de eventos. Así se ha venido haciendo en la mayoría de ejemplos de este capítulo.
Siguiendo este modelo uno debería ser capaz de reducir las sentencias de su programa que digan: "Me pregunto qué causó este evento". Cada fragmento de código está involucrado con hacer algo, y no con la comprobación de tipos. Ésta es la mejor forma de escribir código; no sólo es fácil de conceptuar, sino que es aún más fácil de leer y mantener.

Programación visual y Beans

Hasta este momento, en este libro hemos visto lo valioso que es Java para crear fragmentos reusables de código. La unidad "más reusable" de código ha sido la clase, puesto que engloba una unidad cohesiva de características (campos) y comportamientos (métodos) que pueden reutilizarse directamente vía composición o mediante la herencia.

La herencia y el polimorfismo resultan partes esenciales de la programación orientada a objetos, pero en la mayoría de ocasiones, cuando se juntan en una aplicación, lo que se desea son componentes que hagan exactamente lo que uno necesita. Se desearía juntar estos fragmentos en un diseño exactamente igual que un ingeniero electrónico junta chips en una placa de circuitos. Parece también que debería existir alguna forma de acelerar este estilo de programación basado en el "ensamblaje modular".

La "programación visual" tuvo éxito por primera vez -tuvo mucho éxito- con el Visual Basic (VB) de Microsoft, seguido de un diseño de segunda generación en el Delphi de Borland (la inspiración primaria del diseño de los JavaBeans). Con estas herramientas de programación los componentes se representan visualmente, lo que tiene sentido puesto que suelen mostrar algún tipo de componente visual como botones o campos de texto. La representación visual, de hecho, suele ser la apariencia exacta del componente dentro del programa en ejecución. Por tanto,

parte del proceso de programación visual implica arrastrar un componente desde una paleta para depositarlo en un formulario. La herramienta constructora de aplicaciones escribe el código correspondiente, y ese código será el encargado de crear todos los componentes en el programa en ejecución.

Para completar un programa, sin embargo, no basta con simplemente depositar componentes en un formulario. A menudo hay que cambiar las características de un componente, como su color, el texto que tiene, la base de datos a la que se conecta, etc. A las características que pueden modificarse en tiempo de ejecución se les denomina propiedades. Las propiedades de un componente pueden manipularse dentro de la herramienta constructora de aplicaciones, y al crear el programa se almacena esta información de configuración, de forma que pueda ser regenerada al comenzar el programa.

Hasta este momento, uno ya se ha acostumbrado a la idea de que un objeto es más que un conjunto de características; también es un conjunto de comportamientos. En tiempo de diseño, los comportamientos de un componente visual se representan mediante eventos, que indican que "aquí hay algo que le puede ocurrir al componente". Habitualmente, se decide qué es lo que se desea que ocurra cuando se da el evento, vinculando código a ese evento.

Aquí está la parte crítica: la herramienta constructora de aplicaciones utiliza la reflectividad para interrogar dinámicamente al componente y averiguar qué propiedades y eventos soporta. Una vez que sabe cuáles son, puede mostrar las propiedades y permitir cambiarlas (salvando el estado al construir el programa), y también mostrar los eventos. En general, se hace algo como doble clic en el evento y la herramienta constructora de aplicaciones crea un cuerpo de código que vincula a ese evento en particular. Todo lo que hay que hacer en ese momento es escribir el código a ejecutar cuando se dé el evento.

Todo esto conlleva a que mucho código lo haga la herramienta generadora de aplicaciones. Como resultado, uno puede centrarse en qué apariencia tiene el programa y en qué es lo que se supone que debe hacer, y confiar en la herramienta constructora de aplicaciones para que gestione los detalles de conexión. La razón del gran éxito de las herramientas constructoras de aplicaciones es que aceleran dramáticamente el proceso de escritura de una aplicación -sobretodo la interfaz de usuario, pero a menudo también otras porciones de la aplicación.

Qué es un Bean?

Como puede desprenderse, un Bean es simplemente un bloque de código generalmente embebido en una clase. El aspecto clave es la habilidad de la herramienta constructora de aplicaciones para descubrir las propiedades y eventos de ese componente. Para crear un componente **VB,** el programador tenía

que escribir un fragmento de código bastante complicado siguiendo determinadas convenciones para exponer las propiedades y eventos. Delphi era una herramienta de programación visual de segunda generación y el lenguaje se diseñó activamente en torno a la programación visual, por lo que es mucho más fácil crear componentes visuales. Sin embargo, Java ha desarrollado la creación de componentes visuales hasta conducirla a su estado más avanzado con los JavaBeans, porque un Bean no es más que una clase. No hay que escribir ningún código extra o usar extensiones especiales del lenguaje para convertir algo en un Bean. Todo lo que hay que hacer, de hecho, es modificar ligeramente la forma de nombrar los métodos. Es el nombre del método el que dice a la herramienta constructora de aplicaciones si algo es una propiedad, un evento o simplemente un método ordinario.

En la documentación de Java, a esta convención de nombres se le denomina erróneamente "un patrón de diseño". Este nombre es desafortunado, puesto que los patrones de diseño (ver ***Thinking in Patterns with Java***, descargable de ***http://www.BruceEckel.com***) ya conllevan bastantes retos sin necesidad de este tipo de confusiones. No es un patrón de diseño, es simplemente una convención de nombres, y es bastante simple:

1. En el caso de una propiedad de nombre **xxx,** se suelen crear dos métodos: **getXxx**( ) y **setXxx**( ). Nótese que la primera letra tras "**get**" y "**set**" se pone automáticamente en minúscula para generar el nombre de la propiedad. El tipo producido por el método "**get**" es el mismo que constituye el argumento para el método "**set**". El nombre de la propiedad y el tipo del "**set**" y "**get**" no tienen por qué guardar relación.

2. Para una propiedad **boolean**, se puede usar el enfoque "**get**" y "**set**" de arriba, pero también se puede usar "**is**" en vez de "**get**".

3. Los métodos generales del Bean no siguen la convención de nombres de arriba, siendo **public.**

4. En el caso de eventos, se usa el enfoque del "oyente" de Swing. Es exactamente lo que se ha estado viendo: **addFooBarListener**(**FooBarlistener)** y **removeFooBarListerner**(**FooBarListener)** para manejar un **FooBarEvent. La** mayoría de las veces los eventos y oyentes ya definidos serán suficientes para satisfacer todas las necesidades, pero también se pueden crear interfaces oyentes y eventos propios.

El Punto 1 de los citados responde a la pregunta sobre algo que el lector podría haber visto al mi**rar** al código viejo frente al código nuevo: muchos nombres de método han sufrido pequeños cambios de nombre, aparentemente no significativos. Ahora puede verse que la mayoría de esos cambios están

relacionados con adaptar las convenciones de nombrado del "get" y "set" para convertir ese componente particular en un Bean.

Para crear un Bean simple pueden seguirse estas directrices:

```
//: frogbean:Frog.java
// A trivial JavaBean.
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) {
    // ...
  }
  // An "ordinary" public method:
  public void croak() {
    System.out.println("Ribbet!");
```

```
    }
} ///:~
```

En primer lugar, puede verse que es simplemente una clase. Generalmente todos los campos serán **private** y accesibles sólo a través de métodos. Siguiendo esta convención de nombres, las propiedades son saltos, color, lugares y saltador (nótese el cambio de mayúscula a minúscula en el caso de la primera letra). Aunque el nombre del identificador interno es el mismo que el nombre de la propiedad en los tres primeros casos, en saltador se puede ver que el nombre de la propiedad no obliga a usar ningún identificador particular para variables internas (ni de hecho, a ***tener*** ninguna variable interna para esa propiedad).

Los eventos que maneja este Bean son **ActionEvent** y **KeyEvent**, basados en el nombrado de los métodos "**añadir**" y "**remover**" del oyente asociado. Finalmente, podemos ver que el método ordinario **croar**( ) sigue siendo parte del Bean simplemente por ser un método **public**, y no porque se someta a ningún esquema de nombres.

Extraer **BeanInfo** con el **Introspector**

Una de las partes más críticas del esquema de los Beans se dan al sacar un Bean de una paleta y colocarlo en un formulario. La herramienta constructora de aplicaciones debe ser capaz de crear el Bean (lo que puede hacer siempre que haya algún constructor por defecto) y después, sin acceder al código fuente del Bean, extraer toda la información necesaria para crear la hoja de propiedades y los manejadores de eventos.

Parte de la solución ya es evidente desde la parte final del Capítulo 12: la ***reflectividad*** de Java permite descubrir todos los métodos de una clase anónima. Esto es perfecto para solucionar el problema de los Beans sin que sea necesario usar ninguna palabra clave extra del lenguaje, como las que hay que usar en otros lenguajes de programación visual. De hecho, una de las razones primarias por las que se añadió reflectividad a Java era dar soporte a los Beans (aunque la reflectividad también soporta la serialización de objetos y la invocación remota de métodos). Por tanto, podría esperarse que el creador de la herramienta constructora de aplicaciones hubiera aplicado reflectividad a cada Bean para recorrer sus métodos y localizar las propiedades y eventos del Bean.

Esto es verdaderamente posible, pero los diseñadores de Java querían proporcionar una herramienta estándar, no sólo para que los Beans fueran más fáciles de usar, sino también para proporcionar una pasarela estándar de cara a la creación de Beans más complejos. Esta herramienta es la clase **Introspector**, y su método más importante es el **static getBeanInfo**( ). **A** este método se le pasa una referencia a una **Class**, e interroga concienzudamente a la clase,

devolviendo un objeto **BeanInfo** que puede ser después diseccionado para buscar en él propiedades, métodos y eventos.

Generalmente no hay que preocuparse por esto -probablemente se irán obteniendo los Beans ya diseñados por un tercero, y no habrá que conocer toda la magia subyacente en ellos. Simplemente se arrastrarán los Beans al formulario, se configurarán sus propiedades y se escribirán manipuladores para los eventos en los que se esté interesado. Sin embargo, es interesante y educativo ejercitar el uso del **Instrospector** para mostrar la información relativa a un Bean, así que he aquí una herramienta que lo hace:

```
//: c13:BeanDumper.java
// Introspecting a Bean.
// <applet code=BeanDumper width=600 height=500>
// </applet>
import java.beans.*;
import java.lang.reflect.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class BeanDumper extends JApplet {
  JTextField query =
    new JTextField(20);
  JTextArea results = new JTextArea();
  public void prt(String s) {
    results.append(s + "\n");
  }
  public void dump(Class bean){
    results.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
        bean, java.lang.Object.class);
    } catch(IntrospectionException e) {
      prt("Couldn't introspect " +
        bean.getName());
      return;
    }
    PropertyDescriptor[] properties =
      bi.getPropertyDescriptors();
    for(int i = 0; i < properties.length; i++) {
      Class p = properties[i].getPropertyType();
      prt("Property type:\n  " + p.getName() +
        "Property name:\n  " +
        properties[i].getName());
      Method readMethod =
        properties[i].getReadMethod();
```

```
      if(readMethod != null)
        prt("Read method:\n  " + readMethod);
      Method writeMethod =
        properties[i].getWriteMethod();
      if(writeMethod != null)
        prt("Write method:\n  " + writeMethod);
      prt("====================");
    }
    prt("Public methods:");
    MethodDescriptor[] methods =
      bi.getMethodDescriptors();
    for(int i = 0; i < methods.length; i++)
      prt(methods[i].getMethod().toString());
    prt("=====================");
    prt("Event support:");
    EventSetDescriptor[] events =
      bi.getEventSetDescriptors();
    for(int i = 0; i < events.length; i++) {
      prt("Listener type:\n  " +
        events[i].getListenerType().getName());
      Method[] lm =
        events[i].getListenerMethods();
      for(int j = 0; j < lm.length; j++)
        prt("Listener method:\n  " +
          lm[j].getName());
      MethodDescriptor[] lmd =
        events[i].getListenerMethodDescriptors();
      for(int j = 0; j < lmd.length; j++)
        prt("Method descriptor:\n  " +
          lmd[j].getMethod());
      Method addListener =
        events[i].getAddListenerMethod();
      prt("Add Listener Method:\n  " +
          addListener);
      Method removeListener =
        events[i].getRemoveListenerMethod();
      prt("Remove Listener Method:\n  " +
        removeListener);
      prt("====================");
    }
  }
  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 void init() {
      Container cp = getContentPane();
      JPanel p = new JPanel();
      p.setLayout(new FlowLayout());
      p.add(new JLabel("Qualified bean name:"));
      p.add(query);
      cp.add(BorderLayout.NORTH, p);
      cp.add(new JScrollPane(results));
      Dumper dmpr = new Dumper();
      query.addActionListener(dmpr);
      query.setText("frogbean.Frog");
      // Force evaluation
      dmpr.actionPerformed(
        new ActionEvent(dmpr, 0, ""));
    }
    public static void main(String[] args) {
      Console.run(new BeanDumper(), 600, 500);
    }
} ///:~
```

VolcadorBean.volcar( ) es el método que hace todo el trabajo. Primero intenta crear un objeto **BeanInfo**, y si tiene éxito llama a los métodos de **BeanInfo** que producen información sobre las propiedades, métodos y eventos. En **Instrospector.getBeanInfo**( ), se verá que hay un segundo parámetro. Éste dice a **Introspector** dónde parar en la jerarquía de herencia. Aquí se para antes de analizar todos los métodos de **Object**, puesto que no nos interesan.
En el caso de las propiedades, **getPropertyDescriptors**( ) devuelve un array de **PropertyDescriptors**. Por cada **PropertyDescriptor** se puede invocar a **getPropertyType**( ) para averiguar la clase del objeto pasado hacia dentro y hacia fuera vía los métodos de propiedad. Después, por cada propiedad se puede acceder a sus pseudónimos (extraídos de los métodos de nombre) con **getName**( ), el método para leer con **getFteadMethod**( ), y el método para escribir con **getWriteMethod**( ). Estos dos últimos métodos devuelven un objeto **Method,** que puede, de hecho, ser usado para invocar al método correspondiente en el objeto (esto es parte de la reflectividad).

En el caso de los métodos **public** (incluyendo los métodos de propiedad), **getMethodDescriptors**( ) devuelve un array de **MethodDescriptors.** Por cada uno se puede obtener el objeto **Method** asociado e imprimir su nombre.

En el caso de los eventos, **getEventSetDescriptors**( ) devuelve un array de (**¿qué** otra cosa podría ser?) **EventSetDescriptors.** Cada uno de éstos puede ser interrogado para averiguar la clase del oyente, los métodos de esa clase

oyentes, y los métodos de adición y eliminación de oyentes. El programa **VolcadorBean** imprime toda esta información.

Al empezar, el programa fuerza la evaluación de **beanrana.Rana.** La salida, una vez eliminados los detalles extra innecesarios por ahora, es:

```
class name: Frog
Property type:
  Color
Property name:
  color
Read method:
  public Color getColor()
Write method:
  public void setColor(Color)
====================
Property type:
  Spots
Property name:
  spots
Read method:
  public Spots getSpots()
Write method:
  public void setSpots(Spots)
====================
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)
====================
Public methods:
public void setJumps(int)
public void croak()
public void removeActionListener(ActionListener)
public void addActionListener(ActionListener)
public int getJumps()
```

```
public void setColor(Color)
public void setSpots(Spots)
public void setJumper(boolean)
public boolean isJumper()
public void addKeyListener(KeyListener)
public Color getColor()
public void removeKeyListener(KeyListener)
public Spots getSpots()
=====================
Event support:
Listener type:
  KeyListener
Listener method:
  keyTyped
Listener method:
  keyPressed
Listener method:
  keyReleased
Method descriptor:
  public void keyTyped(KeyEvent)
Method descriptor:
  public void keyPressed(KeyEvent)
Method descriptor:
  public void keyReleased(KeyEvent)
Add Listener Method:
  public void addKeyListener(KeyListener)
Remove Listener Method:
  public void removeKeyListener(KeyListener)
====================
Listener type:
  ActionListener
Listener method:
  actionPerformed
Method descriptor:
  public void actionPerformed(ActionEvent)
Add Listener Method:
  public void addActionListener(ActionListener)
Remove Listener Method:
  public void removeActionListener(ActionListener)
====================
```

Esto revela la mayoría de lo que **Introspector** ve a medida que produce un objeto **BeanInfo** a partir del Bean. Se puede ver que el tipo de propiedad es independiente de su nombre. Nótese que los nombres de propiedad van en minúsculas. (La única ocasión en que esto no ocurre es cuando el nombre de propiedad empieza con más de una letra mayúscula en una fila.) **Y** recuerde que los nombres de método que se ven en este caso (como los métodos de lectura y escritura) son de hecho producidos a partir de un objeto **Method** que puede usarse para invocar al método asociado del objeto.

La lista de métodos **public** incluye la lista de métodos no asociados a una propiedad o evento, como **crear**( ), además de los que sí lo están. Éstos son todos los métodos a los que programando se puede invocar en un Bean, y la herramienta constructora de aplicaciones puede elegir listados todos al hacer llamadas a métodos, para facilitarte la tarea.

Finalmente, se puede ver que se analizan completamente los eventos en el oyente, sus métodos y los métodos de adición y eliminación de oyentes. Básicamente, una vez que se dispone del
BeanInfo, se puede averiguar todo lo que sea relevante para el Bean. También se puede invocar a los métodos para ese Bean, incluso aunque no se tenga otra información excepto el objeto (otra faceta, de nuevo, de la reflectividad).

Un Bean más sofisticado

El siguiente ejemplo es ligeramente más sofisticado, a la vez que frívolo. Es un **JPanel** que dibuja un pequeño círculo en torno al ratón cada vez que se mueve el ratón. Cuando se presiona el ratón, aparece la palabra "¡Bang!" en medio de la pantalla, y se dispara un oyente de acción.

Las propiedades que pueden cambiarse son el tamaño del círculo, además del color, tamaño, y texto de la palabra que se muestra al presionar el ratón. Un **BeanExplosion** también tiene su propio **addActionlistener**( ) y **removeActionListener**( ), de forma que uno puede adjuntar su propio oyente a disparar cuando el usuario haga clic en el **BeanExplosion**. Habría que ser capaz de reconocer la propiedad y el soporte al evento:

```
//: bangbean:BangBean.java
// A graphical Bean.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBean extends JPanel
     implements Serializable {
  protected int xm, ym;
  protected int cSize = 20; // Circle size
  protected String text = "Bang!";
  protected int fontSize = 48;
  protected Color tColor = Color.red;
  protected 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);
  }
  // This is a unicast listener, which is
  // the simplest form of listener management:
  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();
      // Call the listener's method:
      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 se verá es que **BeanExplosion** implementa la interfaz **Serializable.** Esto significa que la herramienta constructora de aplicaciones puede "acceder" a toda la información del **BeanExplosion** usando la serialización una vez que el diseñador del programa haya ajustado los valores de las propiedades. Cuando se crea el Bean como parte de la aplicación en ejecución se restauran estas propiedades y se realmacenan, obteniendo así exactamente lo que se diseñó.

Se puede ver que todos los campos son **private**, que es lo que generalmente se hará con un Bean -permitir el acceso sólo a través de métodos, generalmente usando el esquema "de propiedades".

Cuando se echa un vistazo a la signatura de **addActionListener**( ), se ve que puede lanzar una
TooManyListenersException. Esto indica que es unidifusión, lo que significa que sólo notifica el evento a un oyente. Generalmente, se usarán eventos multidifusión de forma que puedan notificarse a varios oyentes. Sin embargo, eso conlleva aspectos para los que uno no estará preparado hasta acceder al siguiente capítulo, por lo que se volverá a retomar este tema en él (bajo el título de "Revisar los JavaBeans"). Un evento unidifusión "circunvala" este problema.

Cuando se hace clic con el ratón, se pone el texto en el medio del **BeanExplosion** y si el campo
actionListener no es **null**, se invoca su **actionPerformed**( ), creando un nuevo objeto **ActionEvent** en el proceso. Cada vez que se mueva el ratón, se capturan sus nuevas coordenadas y se vuelve a dibujar el lienzo (borrando cualquier texto que esté en él, como se verá).

He aquí la clase **PruebaBeanExplosion** que permite probar el Bean como otro ***applet*** o aplicación:

```
//: c13:BangBeanTest.java
// <applet code=BangBeanTest
// width=400 height=500></applet>
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBeanTest extends JApplet {
  JTextField txt = new JTextField(20);
  // During testing, report actions:
  class BBL implements ActionListener {
    int count = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("BangBean action "+ count++);
    }
  }
  public void init() {
    BangBean bb = new BangBean();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Too many listeners");
    }
    Container cp = getContentPane();
    cp.add(bb);
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    Console.run(new BangBeanTest(), 400, 500);
  }
} ///:~
```

Cuando un Bean está en un entorno de desarrollo, no se usará su clase, pero es útil proporcionar un método de prueba rápida para cada uno de los Beans. **PruebaBeanExplosion** coloca un **BeanExplosion** dentro del applet, adjuntando un **ActionListener** simple al **BeanExplosion** para imprimir una cuenta de eventos al **JTextField** cada vez que se da un **ActionEvent**. Generalmente, por supuesto, la herramienta constructora de aplicaciones crearía la mayoría de código que usa el Bean.

Cuando se ejecuta el **BeanExplosion** a través de **VolcadoBean** o se pone **BeanExplosion** dentro de un entorno de desarrollo que soporta Beans, se verá que hay muchas más propiedades y acciones de lo que parece en el código de arriba. Esto es porque **BeanExplosion** se hereda de **JPanel**, y **JPanel** es también un Bean, por lo que se están viendo también sus propiedades y eventos.

Empaquetar un Bean

Antes de poder incorporar un Bean en una herramienta de construcción visual habilitada para Beans, hay que ponerlo en un contenedor estándar de Beans, que es un archivo **JAR** que incluye todas las clases Bean además de un archivo de manifiesto que dice: "Esto es un Bean".

Un archivo manifiesto no es más que un archivo de texto que sigue un formato particular. En el caso del **BeanExplosion**, el archivo manifiesto tiene la apariencia siguiente (sin la primera y última líneas):

```
//:! :BangBean.mf
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
///:~
```

La primera línea indica la versión del esquema de manifiesto, que mientras Sun no indique lo contrario es la 1.0. La segunda línea (las líneas en blanco se ignoran) nombra el archivo **BeanExplosion.class**, y la tercera dice, "Es un Bean". Sin la tercera línea, la herramienta constructora de programas no reconocería la clase como un Bean.

El único punto del que hay que asegurarse es que se pone el nombre correcto en el campo "Name". Si se vuelve atrás a **BeanExplosion.java**, se verá que está en el **package beanexplosion** (y por consiguiente en un subdirectorio denominado "**beanexplosion**" que está fuera del *classpath*), y el nombre del archivo de manifiesto debe incluir esta información de paquete. Además, hay que colocar el archivo de manifiesto en el directorio superior a la raíz de la trayectoria de los paquetes, lo que en este caso significa ubicar el archivo en el directorio padre al subdirectorio "**beanexplosion**".

Posteriormente hay que invocar a **jar** desde el directorio que contiene al archivo de manifiesto, así:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Esta línea asume que se desea que el archivo JAR resultante se denomine **BeanExplosion.jar**, y que se ha puesto el ***manifiesto*** en un archivo denominado **BeanExplosion.mf**.

Uno podría preguntarse "¿Qué ocurre con todas las demás clases que se generaron al compilar **BeanExplosion.java**?" Bien, todas acabaron dentro del subdirectorio **beanexplosion**, y se verá que el último parámetro para la línea de comandos de **jar** de arriba es el subdirectorio **beanexplosión**.

Cuando se proporciona a **jar** el nombre de un subdirectorio, empaqueta ese subdirectorio al completo en el archivo **jar** (incluyendo, en este caso, el archivo de código fuente **BeanExplosion.java** original -uno podría elegir no incluir el código fuente con sus Beans). Además, si se deshace esta operación y se desempaqueta el archivo JAR creado, se descubrirá que el archivo de ***manifiesto*** no está dentro, pero quejar ha creado su propio archivo de ***manifiesto*** (basado parcialmente en el nuestro) de nombre **MANIFEST.MF** y lo ha ubicado en el subdirectorio META-INF (es decir, "metainformación"). Si se abre este archivo de ***manifiesto*** también se verá que **jar** ha añadido información de firma digital por cada archivo, de la forma:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

En general, no hay que preocuparse por nada de esto, y si se hacen cambios se puede modificar simplemente el archivo de ***manifiesto*** original y volver a invocar a **jar** para que cree un archivo JAR nuevo para el Bean. También se pueden añadir otros Beans al archivo JAR simplemente añadiendo su información al de ***manifiesto.***

Una cosa en la que hay que fijarse es que probablemente se querrá poner cada Bean en su propio subdirectorio, puesto que cuando se crea un archivo **JAR** se pasa a la utilidad **jar** el nombre de un subdirectorio y éste introduce todo lo que hay en ese subdirectorio en el archivo **JAR.** Se puede ver que tanto **Rana** como **BeanExplosion** están en sus propios subdirectorios.
Una vez que se tiene el Bean adecuadamente insertado en el archivo JAR, se puede incorporar a un entorno constructor de programas habilitado para Beans. **La** manera de hacer eso cambia de una herramienta a otra, pero Sun proporciona un banco de pruebas para JavaBeans disponible gratuitamente en su ***Beam Development Kit*** (BDK) denominado el "**beanbox**". (El BDK puede descargarse de http://java.sun.com/beans.) Para ubicar un Bean en la **beanbox**, se copia el archivo JAR en el directorio "jars" del BDK antes de iniciar el **beanbox**.

Soporte a Beans más complejo

Se puede ver lo remarcadamente simple que resulta construir un Bean. Pero uno no está limitado a lo que ha visto aquí. La arquitectura de JavaBeans proporciona un punto de entrada simple, pero también se puede escalar a situaciones más complejas. Estas situaciones van más allá del alcance de este libro, pero se presentarán de forma breve en este momento. Hay más detalles en *http://java.sun.com/beans*.

Un lugar en el que se puede añadir sofisticación es con las propiedades. Los ejemplos de arriba sólo mostraban propiedades simples, pero también es posible representar múltiples propiedades en un array. **A** esto se le llama una ***propiedad indexada.*** Simplemente se proporcionan los métodos adecuados (siguiendo de nuevo una convención para los nombres de los métodos) y el **Introspector** reconoce la propiedad indexada de forma que la herramienta constructora de aplicaciones pueda responder de forma apropiada.

Las propiedades pueden vincularse, lo que significa que se notificarán a otros objetos vía un **PropertyChangeEvent**. Los otros objetos pueden después elegir cambiarse a sí mismos basándose en el cambio sufrido por el Bean.

Las propiedades pueden limitarse, lo que significa que los otros objetos pueden vetar cambios en esa propiedad si no los aceptan. A los otros objetos se les avisa usando un **PropertyChangeEvent**, y puede lanzar una **PropertyVetoException** para evitar que ocurra el cambio y restaurar los valores viejos.

También se puede cambiar la forma de representar el Bean en tiempo de diseño:

1. Se puede proporcionar una hoja general de propiedades del Bean en particular. Esta hoja de propiedades se usará para todos los otros Beans, invocándose el nuestro automáticamente al seleccionar el Bean.

2. Se puede crear un editor personalizado para una propiedad particular, de forma que se use la hoja de propiedades ordinaria, pero cuando se edite la propiedad especial, se invocará automáticamente a este editor.

3. Se puede proporcionar una clase **BeanInfo** personalizada para el Bean, que produzca información diferente de la información por defecto que crea en **Introspector**.

4. También es posible pasar a modo "experto" o no por cada **FeatureDescriptors** para distinguir entre facetas básicas y aquéllas más complicadas.

Más sobre Beans

Hay otro aspecto que no se pudo describir aquí. Siempre que se cree un Bean, cabría esperar que se ejecute en un entorno multihilo. Esto significa que hay que entender los conceptos relacionados con los hilos, y que se presentarán en el

Capítulo 14. En éste se verá una sección denominada "Volver a visitar los Beans" que describirá este problema y su solución.

Hay muchos libros sobre JavaBeans; por ejemplo, JavaBeans de Elliotte Rusty Harold (IDG, 1998).

Resumen

De todas las bibliotecas de Java, la biblioteca IGU ha visto los cambios más dramáticos de Java 1.0 a Java 2. La **AWT** de Java 1.0 se criticó como uno de los peores diseños jamás vistos, y aunque permitía crear programas portables, el IGU resultante era "igualmente mediocre en todas las plataformas". También imponía limitaciones y era extraño y desagradable de usar en comparación con las herramientas de desarrollo de aplicaciones nativas disponibles en cada plataforma particular.

Cuando Java **1.1** presentó el nuevo modelo de eventos y los JavaBeans, se dispuso el escenario -de forma que era posible crear componentes IGU que podían ser arrastrados y depositados fácilmente dentro de herramientas de construcción de aplicaciones visuales. Además, el diseño del modelo de eventos y de los Beans muestra claramente un alto grado de consideración por la facilidad de programación y la mantenibilidad del código (algo que no era evidente en la **AWT** de Java 1.0).

Pero hasta que no aparecieron las clases JFC/Swing, el trabajo no se dio por concluido. Con los componentes Swing, la programación de IGU multiplataforma se convirtió en una experiencia civilizada.

De hecho, lo único que se echa de menos es la herramienta constructora de aplicaciones, que es donde radica la verdadera revolución. El Visual Basic y Visual C++ de Microsoft requieren de herramientas de construcción de aplicaciones propias de Microsoft, al igual que ocurre con Borland en el caso de Delphi y C++. Si se desea que la herramienta de construcción de aplicaciones sea mejor, hay que cruzar los dedos y esperar a que el productor haga lo que uno espera. Pero Java es un entorno abierto, y por tanto, no sólo permite entornos de construcción de aplicaciones de otros fabricantes, sino que trata de hacer énfasis en que se construyan. Y para que estas herramientas sean tomadas en serio, deben soportar JavaBeans. Esto significa un campo de juego por niveles: si sale una herramienta constructora de aplicaciones mejor, uno no está obligado a usar la que venía usando -puede decidir cambiarse a la nueva e incrementar su productividad. Este tipo de entorno competitivo para herramientas constructoras de aplicaciones IGU no se había visto nunca anteriormente, y el mercado resultante no puede sino ser beneficioso para la productividad del programador.

Este capítulo solamente pretendía dar una introducción a la potencia de Swing al lector de forma que pudiera ver lo relativamente simple que es introducirse en las bibliotecas. Lo visto hasta la fecha será probablemente suficiente para una porción significativa de necesidades de diseño de IU.

Sin embargo, Swing tiene muchas más cosas -se pretende que sea un conjunto de herramientas de extremada potencia para el diseño de IU. Probablemente proporciona alguna forma de lograr absolutamente todo lo que se nos pudiera ocurrir.

Si uno no ve aquí todo lo que necesita, siempre puede optar por la documentación en línea de Sun y buscar en la Web, y si sigue siendo poco, buscar un libro dedicado a Swing -un buen punto de partida es The **JFC** Swing Tutorial, de Walrath & Campione (Addison Welsey, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide,* disponible a bajo coste en *http://www.BruceEckel.corn.*

1. Crear un applet/aplicación usando la clase **Console** mostrada en este capítulo. Incluir un campo de texto y tres botones. Al presionar cada botón, hacer que aparezca algún texto distinto en el campo de texto.

2. Añadir una casilla de verificación al applet creado en el Ejercicio 1, capturar el evento e insertar un texto distinto en el campo de texto.

3. Crear un applet/aplicación utilizando **Console**. En la documentación HTML de java.sun.com, encontrar el **JPasswordField** y añadirlo al programa. Si el usuario teclea la contraseña correcta, usar **JOptionPane** para proporcionar un mensaje de éxito al usuario.

4. Crear un applet/aplicación usando **Console**, y añadir todos los componentes que tengan un método **addActionListener**( ). (Consultarlos todos en la documentación HTML de *http://java.sun.com*. Truco: usar el índice.) Capturar sus eventos y mostrar un mensaje apropiado para cada uno dentro de un campo de texto.

5. Crear un applet/aplicación usando **Console**, con un **JButton**, y un **JTextField**. Escribir y adjuntar el oyente apropiado de forma que si el foco reside en el botón, los caracteres del mismo aparezcan en el **JTextField**.

6. Crear un applet/aplicación usando **Console**. Añadir al marco principal todos los componentes descritos en este capítulo, incluyendo menús y una caja de diálogo.

7. Modificar **CamposTexto.java** de forma que los caracteres de **t2** retengan el uso de mayúsculas y minúsculas original, en vez de forzarlas automáticamente a mayúsculas.

8. Localizar y descargar uno o más entornos de desarrollo constructores de IGU gratuitos disponibles en Internet, o comprar un producto comercial. Descubrir qué hay que hacer para añadir **BeanExplosion** al entorno, y hacerlo.

9. Añadir **Rana.class** al archivo de manifiesto como se muestra en este capítulo y ejecutar **jar** para crear un archivo JAR que contenga tanto a **Rana** como a **BeanExplosion**. Ahora descargar e instalar el BDK de Sun, o bien usar otra herramienta constructora de programas habilitada para Beans, y añadir el archivo JAR al entorno, de forma que se puedan probar estos dos Beans.

10. Crear su propio JavaBean denominado **Valvula**, que contenga dos propiedades: una **bolean** denominada "**on**" y otra **int** denominada "**nivel**". Crear un archivo de manifiesto, usar **jar** para empaquetar el Bean, y después cargarlo en la **beanbox** o en una herramienta constructora de programas habilitada para Beans, y así poder probarlo.

11. Modificar **CajasMensajes.java** de forma que tenga un **ActionListener** individual por cada botón (en vez de casar el texto del botón).

12. Monitorizar un nuevo tipo de evento en **RastrearEvento.java**, añadiendo el nuevo código de manejo del evento. Primero habrá que decidir cuál será el nuevo tipo de evento a monitorizar.

13. Heredar un nuevo tipo de botón de **JButton**. Cada vez que se presione este botón, debería cambiar de color a un valor seleccionado al azar. Ver **CajasColores.java** del Capítulo 13 para tener un ejemplo de cómo generar un valor de código al azar.

14. Modificar **PanelTexto.java** para que use un **JTextArea** en vez de un **JTextPane**.

15. Modificar **Menus.java** de forma que use botones de opción en vez de casillas de verificación en los menús.

16. Simplificar **Lista.java** pasándole el array al constructor y eliminando la adición dinámica de elementos a la lisa.

17. Modificar **OndaSeno.java** para que **Dibujarseno** sea un JavaBean añadiendo métodos "**getter**" y "**setter**".

18. ¿Recuerdas el juguete "**Telesketch**", con dos controles, uno encargado del movimiento vertical del punto del dibujo, y otro que controla el movimiento horizontal? Crear uno de ésos usando **OndaSeno.java** como comienzo. En vez de controles rotatorios, usar barras de desplazamiento. Añadir un botón que borre toda la pantalla.

19. Crear un "indicador de progresos asintótico" que vaya cada vez más despacio a medida que se acerca a su meta. Añadir comportamiento errático al azar de forma que periódicamente parezca que comienza a acelerar.

20. Modificar **Progreso.java** de forma que no comparta modelos sino que use un oyente para conectar el deslizador y la barra de progreso.

21. Seguir las instrucciones de la sección "Empaquetando un applet en un archivo JAR" para ubicar **TicTacToe.java** en un archivo JAR. Crear una página HTML con la versión (complicada y difícil) de la etiqueta applet, y modificarla para que use la etiqueta de archivo para usar el archivo JAR. (Truco: empezar con la página HTML de **TicTacToe.java** que viene con la distribución de código fuente de este libro.)

22. Crear un applet/aplicación usando **Console**. Este debería tener tres deslizadores, para los valores del rojo, verde y azul de **java.awt.Color**. El resto del formulario debería ser un **JPanel** que muestre el color determinado por los tres deslizadores. Incluir también campos de texto no editables que muestren los valores RGB actuales.

23. En la documentación HTML de *javax.swing*, buscar **JColorChooser**. Escribir un programa con un botón que presente este selector de color como un diálogo.

24. Casi todo componente Swing se deriva de **Component**, que tiene un método **setCursor**( ). Buscarlo en la documentación HTML. Crear un applet y cambiar el cursor a uno de los almacenados en la clase **Cursor**.

25. A partir de **MostrarAddListeners.java,** crear un programa con la funcionalidad completa de **LimpiarMostrarMetodos.java** del Capítulo 10.

15: Descubriendo Problemas

Antes de que C fuese domesticada en ANSI C, tuvimos un pequeño chiste: "¡Mi código compila, así es que debería correr!" (Ha ha!)

Esto fue gracioso sólo si entendiste C, porque en aquel entonces el compilador de C aceptaría solamente acerca de cualquier cosa; C fue verdaderamente un lenguaje ensamblador portable creado para ver si se logró desarrollar un sistema operativo portátil (Unix) que podría ser movido de una arquitectura de la máquina a otra sin reescribirlo desde el principio en el lenguaje ensamblador de la máquina nueva. Así es que C fue de hecho creada como un efecto secundario de fortalecer a Unix y no como un lenguaje de programación multiuso.

Porque C fue enfocada en programadores que escribieron sistemas operativos en el lenguaje ensamblador, eso estaba implícitamente asumida que esos programadores supieron lo que fueron haciendo y no necesitó redes de seguridad. Por ejemplo, los programadores de lenguajes ensambladores no necesitaron que el compilador compruebe tipos de argumento y uso, y si se decidieron usar un tipo de datos de otro modo que fue originalmente pretendido, que ciertamente deban tener buena razón para hacer eso, y el compilador no se puso en medio del camino. Así, obtener tu programa ANSI C para compilar fue sólo el primer paso en el largo proceso de desarrollar un programa libre de error de programación.

El desarrollo de ANSI C junto con las reglas más fuertes acerca de lo que el compilador aceptaría vino después de que los montones de gente usasen C para los proyectos aparte de escribir sistemas operativos, y después de la apariencia de C++, lo cual mejoró grandemente tus oportunidades de correr un programa decentemente una vez que es compilado. Muchas de estas mejoras vino a través de la comprobación de tipo fuertemente estática: *fuertemente* porque el compilador te impidió abusar el tipo, *estática* porque ANSI C y C++ realizan la comprobación de tipo en la fase de compilación.

Para muchas personas (me incluyo), la mejora fue tan dramática que pareció que la comprobación de tipo fuertemente estática fue la respuesta para una gran porción de nuestros problemas. Ciertamente, una de las motivaciones para Java fue que la comprobación de tipo C++ no fue lo suficientemente fuerte (primordialmente porque C++ tuvo que estar bajo la compatibilidad de C, y también estaba encadenada a sus limitaciones). Así Java ha ido a la par más allá para aprovecharse de los beneficios de comprobación de tipo, y desde que Java tiene mecanismos de comprobación de lenguaje que existen en el tiempo de ejecución (C++ no lo hace; Lo que queda en tiempo de ejecución es básicamente el lenguaje de ensamblado – muy rápido, pero sin autoconocimiento), no está restringido para la comprobación de tipo sólo

estático. **[1]**

[1] No obstante, lo es primordialmente orientado a la comprobación estática. Hay un sistema alternativo, una llamada tipografía latente o tipografía dinámica o tipografía débil, en el cual el tipo de un objeto es todavía forzado, pero es implementado en el tiempo de ejecución, cuando el tipo es usado, más bien en la fase de compilación. Escribir código en tal lenguaje – Python (*http://www.python.org*) es un excelente ejemplo – da al programador mucho más flexibilidad y requiere mucho menos cantidad de verbosidad para satisfacer al compilador, y aún todavía garantiza que los objetos son usados correctamente. Sin embargo, para un programador convencido la comprobación fuerte, estática de tipo es la única solución apreciable, la tipografía latente es excomulgada y la flama seria de las guerras han resultado de comparaciones entre los dos acercamientos. Como alguien que está todo el tiempo en seguimiento de la mayor productividad, he encontrado el valor de la tipografía latente para ser muy convincente. Además, la habilidad para pensar acerca de los asuntos de tipografía latente te ayuda, creo, a solucionar problemas que son difíciles de pensar acerca de lenguajes fuertes, estáticamentes tipificados.

Parece que, sin embargo, esos mecanismos de comprobación de lenguaje nos pueden tomar sólo en lo que va de nuestra búsqueda para desarrollar un programa que trabaja correctamente. C++ nos dio programas que trabajaron bastante antes que los programas de C, pero a menudo todavía tuvo problemas como fugas de memoria y problemas delicados, enterrados. Java llegó muy lejos por mucho tiempo para solucionar esos problemas, pero está todavía dentro de lo posible escribir un programa Java conteniendo a insectos sucios. Además (a pesar de las demandas de desempeño asombrosas siempre importunadas por las críticas excesivas en *Sun*), toda la seguridad produce en los costos operativos adicionales añadidos *Java*, así algunas veces nos topamos con el reto de obtener nuestros programas *Java* para correr lo suficientemente rápido para una necesidad particular (aunque es usualmente más importante tener un programa de trabajo que uno que corre a una velocidad particular).

Este capítulo presenta herramientas para solucionar los problemas que el compilador no soluciona. En cierto sentido, admitimos que el compilador nos puede tomar sólo en lo que va de la creación de programas robustos, pero también nos movemos más allá del compilador y crearemos un sistema de construcción y código que conoce más sobre lo que es un programa y de lo que no está supuesto a hacer.

Una de los pasos más grandes hacia adelante es la incorporación de prueba de unidades automatizada. Esto significa escribir pruebas e incorporar esas pruebas en un sistema de la constitución que compila tu código y corre las pruebas cada tiempo único, como si las pruebas fueron parte del proceso de la compilación (pronto comenzarás a depender de ellas como si son). Para este libro, un sistema personalizado de prueba fue desarrollado para asegurar la exactitud de la salida del programa (y para desplegar la salida directamente en el listado de código), pero el sistema de prueba del **JUnit** del estándar del defacto también será usado cuando es apropiado. Para estar seguro de que la prueba es automática, las pruebas son ejecutadas como parte del proceso de

construcción usando **Ant**, una herramienta de fuente abierta que también se ha llegado a ser un defacto estándar en el desarrollo *Java*, y **CVS**, otra herramienta de fuente abierta que mantiene un depositario conteniendo todo tu código fuente para un proyecto particular.

JDK 1.4 introdujo un mecanismo de aserción para beneficiar en la verificación de código en el tiempo de ejecución. Uno de los usos más apremiantes de aserciones es el *Diseño por contrato* (DBC), una manera formalizada para describir la exactitud de una clase. En conjunción con la prueba automatizada, DBC puede ser una herramienta poderosa.

Algunas veces el probar unidades no es suficiente, y necesitas seguirle la pista a los problemas en un programa que corre, sino no corre bien. En JDK 1.4, la API de registro de actividades fue introducida para permitirte fácilmente reportar información acerca de tu programa. Ésta es una mejora significativa sobre agregar y quitar declaraciones **println**() para seguirle la pista a un problema, y esta sección entrará en bastante detalle para darte un curso básico minucioso en este API. Este capítulo también le provee una introducción eliminando fallos de un programa, mostrando la información que un depurador típico le puede proveer a la ayuda en el descubrimiento de problemas sutiles. Finalmente, aprenderás acerca de trazado de perfil y cómo descubrir los cuellos de botella que causan que tu programa corra muy lentamente.

Prueba de Unidades

Una realización reciente en la práctica de programación es el valor dramático de prueba de unidades. Éste es el proceso de construir pruebas integradas en todo el código que creas y ejecutando esas pruebas cada vez que haces una construcción. De ese modo, el proceso de construcción puede revisar en busca de más que solamente errores de sintaxis, también le enseñas a revisar en busca de errores semánticos también. Los lenguajes de programación de estilo C, y C++ en particular, típicamente han apreciado el desempeño sobre programar de forma segura. La razón de que desarrollar programas en Java es por si mucho más rápido (casi al doble de lo rápido, por la mayoría de cuentas) que en C++ es por la red de seguridad de Java: características como recolección de basura y comprobación mejorada de tipo. Integrando prueba de unidades en tu proceso de la construcción, puede extender esta red de seguridad, dando como resultado un desarrollo más rápido. También puedes ser más atrevido en los cambios que le haces, más fácilmente rediseña tu código cuando descubres desperfectos del diseño o de implementación, y en general produces un mejor producto, más rápidamente.

El efecto de prueba de unidades en el desarrollo es tan significativo que es usado a lo largo de este libro, no sólo para validar el código en el libro, sino que también desplegar la salida esperada. Mi experiencia con prueba de unidades comenzó cuando me percaté eso, garantizar la exactitud de código en un libro, cada programa en ese libro debe ser automáticamente extraído y organizado en un árbol de origen, junto con un sistema apropiado de construcción. El sistema de la constitución usado en este libro es Ant (descrito más adelante en este capítulo), y después de que lo instales, solamente

puedes escribir ant para construir todo el código para el libro. El efecto de la extracción automática y el proceso de la compilación en la calidad de código del libro fueron tan inmediatos y dramáticos que eso pronto se convirtió (en mi mente) en un requisito para cualquier libro de programación – ¿cómo puedes confiar en código que no compilaste? También descubrí que si quise hacer cambios radicales, podría hacer eso usando búsqueda y reemplazo a todo lo largo del libro o simplemente golpeando duramente el código alrededor. Supe que si introduje un desperfecto, el extractor de código y el sistema de construcción lo depurarían afuera.

Como los programas se pusieron más complejos, sin embargo, también me encontré con que hubo un hueco serio en mi sistema. Poder compilar exitosamente programas es claramente un primer paso importante, y para un libro publicado que parece uno bastante revolucionario; Usualmente por las presiones de publicación, es muy típico al azar abrir un libro de programación y descubrir un desperfecto de codificación. Sin embargo, me mantuve obteniendo mensajes de lectores reportando problemas semánticos en mi código. Estos problemas pudieron ser descubiertos sólo corriendo el código. Naturalmente, entendí esto y tomé algunos anteriores pasos débiles hacia implementar un sistema que realizaría pruebas automáticas de ejecución, pero había sucumbido para publicar horarios, todo el rato sabiendo que hubo definitivamente algo malo con mi proceso y que regresaría para morderme en forma de informes penosos de error de programación (en el mundo de fuentes abiertas, **[2]** la vergüenza es uno de los primeros factores motivadores para aumentar la calidad de un código).

[2] Aunque la versión electrónica de este libro está libremente disponible, no es fuente abierta.

El otro problema fue que carecí de una estructura para el sistema de prueba. Eventualmente, comencé a saber de prueba de unidades y **JUnit**, lo cual proveyó una base para una estructura de prueba. Encontré las versiones iniciales de **JUnit** para estar intolerable porque requirieron que el programador escriba demasiado código para crear aun la suite de prueba más simple. Versiones más recientes han reducido significativamente este código requerido usando reflexión, así es que están mucho más satisfactorios.

Necesité solucionar otro problema, sin embargo, y eso debió validar la salida de un programa y demostrar la salida validada en el libro. Había recibido quejas regulares de que no mostré bastante salida de programa en el libro. Mi objetivo era que el lector debería estar corriendo los programas mientras lee el libro, y muchos lectores hicieron justamente eso y se beneficiaron de él. Una razón oculta para esa actitud, sin embargo, fue que no tuve una forma para probar que la salida mostrada en el libro fuera correcta. De experiencia, supe eso con el paso del tiempo, algo ocurriría a fin de que la salida no fuera correcta (o, no la entendería bien en primer lugar). El cuadro de trabajo simple de prueba mostrado aquí no sólo capta la salida de la consola del programa – y la mayoría de los programas en este libro producen salida de consola – pero eso también la compara a la salida esperada que se imprimió en el libro como parte del listado del código fuente, así es que los lectores pueden ver lo que será la salida y también conocerá que esta salida ha sido verificada por el

proceso de construcción, y que se pueden verificar por si mismos.

Quise ver si el sistema de prueba podría ser aun más fácil y más simple de usar, aplicando el principio de la *Programación Extrema* de *"hacer las cosas más simple que posiblemente podría emplearse como un punto de partida, y luego desarrollar el sistema como los exige el uso"*. (Además, quise tratar de reducir la cantidad de código de prueba en un intento por equipar más funcionabilidad en menos código para presentaciones de pantalla.) El resultado **[3]** es el cuadro de trabajo de prueba simple descrito a continuación.

[3] El primer intento, de cualquier manera. Encuentro que el proceso de construir algo por primera vez eventualmente produce compenetraciones y nuevas ideas.

Una Prueba de Cuadro de trabajo Sencillo

La meta principal de este cuadro de trabajo **[4]** es verificar la salida de los ejemplos en el libro. Ya has visto líneas como

```
private static Test monitor = new Test();
```

[4] Inspirado por el módulo doctest de Python.

Al principio de la mayoría de clases que contienen un método **main**(). La tarea del objeto monitor es interceptar y salvar una copia de la salida estándar y el error estándar en un archivo del texto. Este archivo se usa luego para verificar la salida de un programa de ejemplo comparando el contenido del archivo con la salida esperada.

Comenzamos por definir las excepciones que serán lanzadas por este sistema de prueba. La excepción multiuso para la librería es la clase base para los demás. Note que extiende a **RuntimeException** a fin de que las excepciones comprobadas no sean complejas:

```
//: com:bruceeckel:simpletest:SimpleTestException.java
package com.bruceeckel.simpletest;

public class SimpleTestException extends RuntimeException {
  public SimpleTestException(String msg) {
    super(msg);
  }
} ///:~
```

Una prueba básica es comprobar que el número de líneas enviados la consola por el programa equivale al número esperado de líneas:

```
//: com:bruceeckel:simpletest:NumOfLinesException.java
package com.bruceeckel.simpletest;

public class NumOfLinesException
```

```
extends SimpleTestException {
  public NumOfLinesException(int exp, int out) {
    super("Number of lines of output and "
      + "expected output did not match.\n" +
      "expected: <" + exp + ">\n" +
      "output:   <" + out + "> lines)");
  }
} ///:~
```

O, el número de líneas podría ser correcto, pero uno o más líneas no podrían concordar:

```
//: com:bruceeckel:simpletest:LineMismatchException.java
package com.bruceeckel.simpletest;
import java.io.PrintStream;

public class LineMismatchException
  extends SimpleTestException {
  public LineMismatchException(
    int lineNum, String expected, String output) {
    super("line " + lineNum +
      " of output did not match expected output\n" +
      "expected: <" + expected + ">\n" +
      "output:   <" + output + ">");
  }
} ///:~
```

Este sistema de prueba surte efecto interceptando la salida de la consola usando la clase **TestStream** para reemplazar la salida estándar de la consola y el error de la consola:

```
//: com:bruceeckel:simpletest:TestStream.java
// Simple utility for testing program output. Intercepts
// System.out to print both to the console and a buffer.
package com.bruceeckel.simpletest;
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class TestStream extends PrintStream {
  protected int numOfLines;
  private PrintStream
    console = System.out,
    err = System.err,
    fout;
  // To store lines sent to System.out or err
  private InputStream stdin;
  private String className;
  public TestStream(String className) {
    super(System.out, true); // Autoflush
    System.setOut(this);
```

```
    System.setErr(this);
    stdin = System.in; // Save to restore in dispose()
    // Replace the default version with one that
    // automatically produces input on demand:
    System.setIn(new BufferedInputStream(new InputStream(){
      char[] input = ("test\n").toCharArray();
      int index = 0;
      public int read() {
        return
          (int)input[index = (index + 1) % input.length];
      }
    }));
    this.className = className;
    openOutputFile();
  }
  // public PrintStream getConsole() { return console; }
  public void dispose() {
    System.setOut(console);
    System.setErr(err);
    System.setIn(stdin);
  }
  // This will write over an old Output.txt file:
  public void openOutputFile() {
    try {
      fout = new PrintStream(new FileOutputStream(
        new File(className + "Output.txt")));
    } catch (FileNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
  // Override all possible print/println methods to send
  // intercepted console output to both the console and
  // the Output.txt file:
  public void print(boolean x) {
    console.print(x);
    fout.print(x);
  }
  public void println(boolean x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(char x) {
    console.print(x);
    fout.print(x);
  }
  public void println(char x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(int x) {
    console.print(x);
```

```
    fout.print(x);
  }
  public void println(int x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(long x) {
    console.print(x);
    fout.print(x);
  }
  public void println(long x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(float x) {
    console.print(x);
    fout.print(x);
  }
  public void println(float x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(double x) {
    console.print(x);
    fout.print(x);
  }
  public void println(double x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(char[] x) {
    console.print(x);
    fout.print(x);
  }
  public void println(char[] x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(String x) {
    console.print(x);
    fout.print(x);
  }
  public void println(String x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void print(Object x) {
```

```
    console.print(x);
    fout.print(x);
  }
  public void println(Object x) {
    numOfLines++;
    console.println(x);
    fout.println(x);
  }
  public void println() {
    if(false) console.print("println");
    numOfLines++;
    console.println();
    fout.println();
  }
  public void
  write(byte[] buffer, int offset, int length) {
    console.write(buffer, offset, length);
    fout.write(buffer, offset, length);
  }
  public void write(int b) {
    console.write(b);
    fout.write(b);
  }
} ///:~
```

El constructor para **TestStream**, después de llamar el constructor para la clase base, primero salva referencias para la salida estándar y el error estándar, y luego **redirecciona** ambos flujos al objeto **TestStream**. Los métodos estático **setOut**() y **setErr**() ambos toman un argumento **PrintStream**. Las referencias **System.out** y **System.err** están desconectados de su objeto normal y en lugar de eso son conectados dentro del objeto **TestStream**, así **TestStream** también debe ser un **PrintStream** (o equivalentemente, algo heredado de **PrintStream**). La referencia estándar original de salida **PrintStream** es captada en la referencia de la consola dentro de **TestStream**, y cada vez que la salida de la consola es interceptada, es enviada a la consola original también como para un archivo de salida. El método **dispose**() se usa para establecer referencias estándar de la E/S de regreso a sus objetos originales cuando **TestStream** queda listo con ellos.

Para la prueba automática de ejemplos que requieren entrada de usuario desde la consola, el constructor redirecciona llamadas a la entrada estándar. La entrada estándar actual es almacenado en una referencia a fin de que **dispose**() lo pueda restaurar a su estado original. Usando a **System.setIn**(), una clase interna anónima es determinada para manejar varias peticiones para la entrada por el programa bajo prueba. El método **read**() de esta clase interna produce las letras "prueba" seguida por una nueva línea.

TestStream sobrescribe una colección variada de métodos **PrintStream** **print**() y **println**() para cada tipo. Cada uno de estos métodos escribe ambos a la salida "*estándar*" y a un archivo de salida. El método **expect**() luego puede usarse para experimentar si la salida producida por un programa equivale a la salida esperada provista como el argumento para **expect**().

Estas herramientas son usadas en la clase **Test**:

```
//: com:bruceeckel:simpletest:Test.java
// Simple utility for testing program output. Intercepts
// System.out to print both to the console and a buffer.
package com.bruceeckel.simpletest;
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class Test {
  // Bit-shifted so they can be added together:
  public static final int
    EXACT = 1 << 0, // Lines must match exactly
    AT_LEAST = 1 << 1, // Must be at least these lines
    IGNORE_ORDER = 1 << 2, // Ignore line order
    WAIT = 1 << 3; // Delay until all lines are output
  private String className;
  private TestStream testStream;
  public Test() {
    // Discover the name of the class this
    // object was created within:
    className =
      new Throwable().getStackTrace()[1].getClassName();
    testStream = new TestStream(className);
  }
  public static List fileToList(String fname) {
    ArrayList list = new ArrayList();
    try {
      BufferedReader in =
        new BufferedReader(new FileReader(fname));
      try {
        String line;
        while((line = in.readLine()) != null) {
          if(fname.endsWith(".txt"))
            list.add(line);
          else
            list.add(new TestExpression(line));
        }
      } finally {
        in.close();
      }
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    return list;
  }
  public static List arrayToList(Object[] array) {
    List l = new ArrayList();
    for(int i = 0; i < array.length; i++) {
      if(array[i] instanceof TestExpression) {
        TestExpression re = (TestExpression)array[i];
        for(int j = 0; j < re.getNumber(); j++)
```

```
            l.add(re);
        } else {
          l.add(new TestExpression(array[i].toString()));
        }
      }
      return l;
    }
    public void expect(Object[] exp, int flags) {
      if((flags & WAIT) != 0)
        while(testStream.numOfLines < exp.length) {
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            throw new RuntimeException(e);
          }
        }
        List output = fileToList(className + "Output.txt");
        if((flags & IGNORE_ORDER) == IGNORE_ORDER)
          OutputVerifier.verifyIgnoreOrder(output, exp);
        else if((flags & AT_LEAST) == AT_LEAST)
          OutputVerifier.verifyAtLeast(output,
            arrayToList(exp));
        else
          OutputVerifier.verify(output, arrayToList(exp));
      // Clean up the output file - see c06:Detergent.java
      testStream.openOutputFile();
    }
    public void expect(Object[] expected) {
      expect(expected, EXACT);
    }
    public void expect(Object[] expectFirst,
      String fname, int flags) {
      List expected = fileToList(fname);
      for(int i = 0; i < expectFirst.length; i++)
        expected.add(i, expectFirst[i]);
      expect(expected.toArray(), flags);
    }
    public void expect(Object[] expectFirst, String fname) {
      expect(expectFirst, fname, EXACT);
    }
    public void expect(String fname) {
      expect(new Object[] {}, fname, EXACT);
    }
} ///:~
```

Hay varias versiones sobrecargadas de **expect**() provista por conveniencia (así es que el programador cliente puede, por ejemplo, proveer el nombre del archivo conteniendo la salida esperada en lugar de un montón de líneas esperadas de salida). Estos métodos sobrecargados todos llaman al método principal **expect**(), lo cual toma como argumentos un arreglo de Objetos conteniendo líneas esperadas de salida y un **int** conteniendo varias banderas. **flags** son implementados usando alternación de bit, con cada bit

correspondiente a una bandera particular como se definió al principio de **Test.java**.

La primera parte del método **expect**() inspecciona el argumento **flags** para ver si debería atrasar el procesamiento para permitirle un programa lento capturarlo. Luego llama un método estático **fileToList**(), lo cual convierte el contenido del archivo de salida producido por un programa en una Lista. El método **fileToList**() también envuelve cada objeto **String** en un objeto **OutputLine**; La razón para esto se aclarará. Finalmente, el método **expect**() llama el método **verify**() apropiado basado en el argumento de banderas.

Hay tres verificadores: **Verify**(), **verifyIgnoreOrder**(), y **verifyAtLeast**(), correspondiente a los modos **EXACT**, **IGNORE_ORDER**, y **AT_LEAST**, respectivamente:

```
//: com:bruceeckel:simpletest:OutputVerifier.java
package com.bruceeckel.simpletest;
import java.util.*;
import java.io.PrintStream;

public class OutputVerifier {
  private static void verifyLength(
    int output, int expected, int compare) {
    if((compare == Test.EXACT && expected != output)
      || (compare == Test.AT_LEAST && output < expected))
      throw new NumOfLinesException(expected, output);
  }
  public static void verify(List output, List expected) {
    verifyLength(output.size(),expected.size(),Test.EXACT);
    if(!expected.equals(output)) {
      //find the line of mismatch
      ListIterator it1 = expected.listIterator();
      ListIterator it2 = output.listIterator();
      while(it1.hasNext()
        && it2.hasNext()
        && it1.next().equals(it2.next()));
      throw new LineMismatchException(
        it1.nextIndex(), it1.previous().toString(),
        it2.previous().toString());
    }
  }
  public static void
  verifyIgnoreOrder(List output, Object[] expected) {
    verifyLength(expected.length,output.size(),Test.EXACT);
    if(!(expected instanceof String[]))
      throw new RuntimeException(
        "IGNORE_ORDER only works with String objects");
    String[] out = new String[output.size()];
    Iterator it = output.iterator();
    for(int i = 0; i < out.length; i++)
      out[i] = it.next().toString();
    Arrays.sort(out);
    Arrays.sort(expected);
```

```
    int i =0;
    if(!Arrays.equals(expected, out)) {
      while(expected[i].equals(out[i])) {i++;}
      throw new SimpleTestException(
        ((String) out[i]).compareTo(expected[i]) < 0
          ? "output: <" + out[i] + ">"
          : "expected: <" + expected[i] + ">");
    }
  }
  public static void
  verifyAtLeast(List output, List expected) {
    verifyLength(output.size(), expected.size(),
      Test.AT_LEAST);
    if(!output.containsAll(expected)) {
      ListIterator it = expected.listIterator();
      while(output.contains(it.next())) {}
      throw new SimpleTestException(
        "expected: <" + it.previous().toString() + ">");
    }
  }
} ///:~
```

Los métodos **verify** prueban ya sea la salida producida por un programa que corresponde a la salida esperada como se especificó por el modo particular. Si esto no es el caso, los métodos **verify** levantan una excepción que aborta el proceso de construcción.

Cada uno de los métodos **verify** usan a **verifyLength**() para probar el número de líneas de salida. El modo EXACT pide que la salida y los arreglos esperados de salida sean el mismo tamaño, y que cada línea de salida sea igual a la línea correspondiente en el arreglo esperado de salida. **IGNORE_ORDER** todavía requiere que ambos arreglos sean el mismo tamaño, pero la orden real de apariencia de las líneas es ignorada (los dos arreglos de salida deben ser permutaciones del uno al otro). El modo **IGNORE_ORDER** se usa para probar ejemplos de hilado donde, debido para la planificación poco determinista de hilos por el JVM, es posible que la secuencia de líneas de salida producidas por un programa no pueda ser predicha. El modo **AT_LEAST** no requiere que los dos arreglos sean el mismo tamaño, pero cada línea de salida esperada debe ser contenida en la salida real producida por un programa, a pesar de la orden. Esta característica es en particular útil para probar ejemplos de programa que contienen líneas de salida que o pueden ser impresos, como es el caso con la mayor parte de los ejemplos de suministro con colección de basura. Note que los tres modos son canónicos; Es decir, si una prueba pasa en el modo **IGNORE_ORDER**, luego también pasará en el modo **AT_LEAST**, y si pasa en el modo **EXACT**, también pasará en los otros dos modos.

Nota qué tan simple es la implementación de loa métodos de verificación es **verify**(), por ejemplo, simplemente llama el método **equals**() provisto por la clase **List**, y **verifyAtLeast**() llama a **List.containsAll**(). Recuerde que las dos salidas **Lists** pueden contener a ambos **OutputLine** o los objetos **RegularExpression**. La razón para envolver el objeto simple **String** en

OutputLines ahora debería ponerse claro; Este acercamiento nos permite sobrescribir el método **equals**(), lo cual es necesario para tomar ventaja del API de Java **Collections**

Los objetos en el arreglo **expect**() puede ser ya sea **Strings** o **TestExpressions**, que puede encapsular una expresión normal (descrita en el Capítulo 12), que es útil para probar ejemplos que producen salida aleatoria. La clase **TestExpression** encapsula a un **String** representando una expresión normal particular.

```
//: com:bruceeckel:simpletest:TestExpression.java
// Regular expression for testing program output lines
package com.bruceeckel.simpletest;
import java.util.regex.*;

public class TestExpression implements Comparable {
  private Pattern p;
  private String expression;
  private boolean isRegEx;
  // Default to only one instance of this expression:
  private int duplicates = 1;
  public TestExpression(String s) {
    this.expression = s;
    if(expression.startsWith("%% ")) {
      this.isRegEx = true;
      expression = expression.substring(3);
      this.p = Pattern.compile(expression);
    }
  }
  // For duplicate instances:
  public TestExpression(String s, int duplicates) {
    this(s);
    this.duplicates = duplicates;
  }
  public String toString() {
    if(isRegEx) return p.pattern();
    return expression;
  }
  public boolean equals(Object obj) {
    if(this == obj) return true;
    if(isRegEx) return (compareTo(obj) == 0);
    return expression.equals(obj.toString());
  }
  public int compareTo(Object obj) {
    if((isRegEx) && (p.matcher(obj.toString()).matches()))
      return 0;
    return
      expression.compareTo(obj.toString());
  }
  public int getNumber() {  return duplicates; }
  public String getExpression() { return expression;}
  public boolean isRegEx() { return isRegEx; }
} ///:~
```

TestExpression puede distinguir patrones normales de expresión de literales **String**. El segundo constructor le permite líneas idénticas múltiples de expresión que están envueltos en un único objeto por conveniencia.

Este sistema experimental ha sido razonablemente útil, y el ejercicio de crearlo y empezarlo a utilizar ha sido invaluable. Sin embargo, en el final no estoy complacido con esto y tengo ideas que probablemente serán implementadas en la siguiente edición del libro (o posiblemente antes).

JUnit

Aunque el cuadro de trabajo de prueba descrito anteriormente te permite verificar salida de programa simple y fácilmente, en algunos casos puedes querer realizar más funcionabilidad extensiva de pruebas en un programa. **JUnit**, disponible en *www.junit.org*, es un estándar rápidamente emergente para escribir pruebas repetibles para los programas Java, y provee ambas pruebas simples y complicadas.

El **JUnit** original se basó probablemente en JDK 1.0 y así no podría hacer uso de las facilidades de reflexión de Java. Como consecuencia, escribir pruebas de la unidad con el **JUnit** viejo fue una actividad más bien ocupada y poco concisa, y encontré el diseño para ser ingrato. Por esto, le escribí a mi cuadro de trabajo de prueba de unidades para Java, **[5]** yendo al otro extremo y "haciendo la cosa lo más simple posible podría trabajar." **[6]** Desde luego, **JUnit** ha sido modificado y usa reflexión para simplificar enormemente el proceso de escribir código de prueba de la unidad. Aunque todavía tienes la opción de escribir código del viejo modo con suites experimentales y todos los otros detalles complicados, creo que en la gran mayoría de los casos puedes seguir el acercamiento simple mostrado aquí (y hace tu vida más agradable).

[5] Originalmente colocado en *Piensa en Patrones* en *www.BruceEckel.com* (con Java).

[6] Una frase clave de *Programación Extrema* (XP). Irónicamente, uno de los autores del JUnit (*Kent Beck*) es también el autor de *Programación Extrema Explicada* (*Addison-Wesley 2000*) y un proponedor principal de XP.

En el acercamiento más simple para usar a **JUnit**, pones todas tus pruebas en una subclase de **TestCase**. Cada prueba debe ser pública, no debe tomar argumentos, retorna **void**, y debe tener un nombre de método a partir de la palabra "**test**". La reflexión de **Junit** identificará estos métodos como las pruebas individuales y establécelos y córrelos uno a la vez, tomando medidas para evitar efectos secundarios entre las pruebas.

Tradicionalmente, el método **setUp**() crea e inicializa un conjunto común de objetos que serán usados en todas las pruebas; Sin embargo, también simplemente puedes poner toda semejante inicialización en el constructor para la clase de prueba. El **JUnit** crea un objeto para cada prueba para asegurar que no habrá efectos secundarios entre operaciones de prueba. Sin embargo, todos los objetos para todas las pruebas son creados de inmediato (en vez de

crear el objeto correctamente antes de la prueba), así la única diferencia entre usar a **setUp**() y el constructor es que **setUp**() es llamado directamente antes de la prueba. En la mayoría de situaciones éste no será un asunto, y puedes destinar al acercamiento del constructo r para simplicidad.

Si necesitas realizar cualquier limpieza total después de cada prueba (si modificas varias estáticas que necesitan ser restauradas, archivos abiertos que necesitan estar cerrados, conexiones abiertas de la red, etc.), Escribes un método **tearDown**(). Esto es también opcional.

El siguiente ejemplo usa este acercamiento simple para crear pruebas **JUnit** que ejercita la clase estándar Java **ArrayList**. Para rastrear cómo **JUnit** crea y limpia sus objetos de prueba, **CountedList** es heredado de **ArrayList** y la información rastreada es añadida:

```
//: c15:JUnitDemo.java
// Simple use of JUnit to test ArrayList
// {Depends: junit.jar}
import java.util.*;
import junit.framework.*;

// So we can see the list objects being created,
// and keep track of when they are cleaned up:
class CountedList extends ArrayList {
  private static int counter = 0;
  private int id = counter++;
  public CountedList() {
    System.out.println("CountedList #" + id);
  }
  public int getId() { return id; }
}

public class JUnitDemo extends TestCase {
  private static com.bruceeckel.simpletest.Test monitor =
    new com.bruceeckel.simpletest.Test();
  private CountedList list = new CountedList();
  // You can use the constructor instead of setUp():
  public JUnitDemo(String name) {
    super(name);
    for(int i = 0; i < 3; i++)
      list.add("" + i);
  }
  // Thus, setUp() is optional, but is run right
  // before the test:
  protected void setUp() {
    System.out.println("Set up for " + list.getId());
  }
  // tearDown() is also optional, and is called after
  // each test. setUp() and tearDown() can be either
  // protected or public:
  public void tearDown() {
    System.out.println("Tearing down " + list.getId());
  }
```

```
// All tests have method names beginning with "test":
public void testInsert() {
  System.out.println("Running testInsert()");
  assertEquals(list.size(), 3);
  list.add(1, "Insert");
  assertEquals(list.size(), 4);
  assertEquals(list.get(1), "Insert");
}
public void testReplace() {
  System.out.println("Running testReplace()");
  assertEquals(list.size(), 3);
  list.set(1, "Replace");
  assertEquals(list.size(), 3);
  assertEquals(list.get(1), "Replace");
}
// A "helper" method to reduce code duplication. As long
// as the name doesn't start with "test," it will not
// be automatically executed by JUnit.
private void compare(ArrayList lst, String[] strs) {
  Object[] array = lst.toArray();
  assertTrue("Arrays not the same length",
    array.length == strs.length);
  for(int i = 0; i < array.length; i++)
    assertEquals(strs[i], (String)array[i]);
}
public void testOrder() {
  System.out.println("Running testOrder()");
  compare(list, new String[] { "0", "1", "2" });
}
public void testRemove() {
  System.out.println("Running testRemove()");
  assertEquals(list.size(), 3);
  list.remove(1);
  assertEquals(list.size(), 2);
  compare(list, new String[] { "0", "2" });
}
public void testAddAll() {
  System.out.println("Running testAddAll()");
  list.addAll(Arrays.asList(new Object[] {
    "An", "African", "Swallow"}));
  assertEquals(list.size(), 6);
  compare(list, new String[] { "0", "1", "2",
     "An", "African", "Swallow" });
}
public static void main(String[] args) {
  // Invoke JUnit on the class:
  junit.textui.TestRunner.run(JUnitDemo.class);
  monitor.expect(new String[] {
    "CountedList #0",
    "CountedList #1",
    "CountedList #2",
    "CountedList #3",
    "CountedList #4",
```

```
      // '.' indicates the beginning of each test:
      ".Set up for 0",
      "Running testInsert()",
      "Tearing down 0",
      ".Set up for 1",
      "Running testReplace()",
      "Tearing down 1",
      ".Set up for 2",
      "Running testOrder()",
      "Tearing down 2",
      ".Set up for 3",
      "Running testRemove()",
      "Tearing down 3",
      ".Set up for 4",
      "Running testAddAll()",
      "Tearing down 4",
      "",
      "%% Time: .*",
      "",
      "OK (5 tests)",
      "",
    });
  }
} ///:~
```

Para establecer prueba de unidades, sólo debes importar **junit.framework.*** y extender a **TestCase**, como lo hace **JUnitDemo**. Además, debes crear a un constructor que toma un argumento **String** y lo pasa a su constructor **super**.

Para cada prueba, un objeto nuevo del **JUnitDemo** será creado, y así de todos los miembros poco estáticos también serán creados. Esto quiere decir que un objeto nuevo (la lista) **CountedList** será creado e inicializado para cada prueba, ya que es un campo de **JUnitDemo**. Además, el constructor será llamado por cada prueba, así es que la lista será inicializada con los **strings** “0”, “1”, y “2” antes que cada prueba sea ejecutada.

Para comentar el comportamiento de **setUp**() y **tearDown**(), estos métodos son creados para desplegar información acerca de la prueba que será inicializada o limpia. Note que los métodos de la clase base son **protected**, así es que los métodos sobrescritos pueden ser ya sea **protected** o **public**.

testInsert() y **testReplace**() demuestran métodos de prueba típicos, ya que siguen la convención requerida de firma y de nombramiento. El **JUnit** descubre estos métodos usando reflexión y corre cada uno como una prueba. Dentro de los métodos, realizas varias operaciones deseadas y usa métodos de aserción del **JUnit** (el cual todo comienza con el nombre **assert**) para verificar la exactitud de tus pruebas (el rango completo de declaraciones **assert** puede ser encontrado en los **JUnit javadocs** para **junit.framework.Assert**). Si la aserción fracasa, la expresión y valores que causó el fracaso será desplegado. Esto es usualmente suficientemente, pero también puedes usar la versión sobrecargada de cada declaración de aserción del **JUnit** y puedes incluir a un **String** que será impreso si la aserción fracasa.

Las declaraciones de aserción no son requeridas; También simplemente puedes correr la prueba sin aserciones y le puedes considerar a ella un éxito si ninguna de las excepciones es lanzada.

El método **compare**() es un ejemplo de un método ayudante que no es ejecutado por **JUnit** pero en lugar de eso es usado por otras pruebas en la clase. Con tal de que el nombre del método no empiece con **test**, **JUnit** no lo corre o espera que tenga una firma particular. Aquí, **compare**() es privado para hacer énfasis en que es usado dentro de la clase de prueba, pero también podría ser público. Los métodos de prueba restantes eliminan código duplicado refactorizándolo en el método **compare**().

Para ejecutar las pruebas del **JUnit**, el método estático **TestRunner.run**() es invocado en **main**(). Este método recibe la clase que contiene la colección de pruebas, y automáticamente configura y corre todas las pruebas. De la salida **expect**(), puedes ver que todos los objetos necesarios para correr todo las pruebas son creados primero, en un lote – esto es donde la construcción ocurre. **[7]** Antes de cada prueba, el método **setUp**() es llamado. Luego la prueba es corrida, seguida por el método **tearDown**(). El **JUnit** demarca cada prueba con un **'.'**.

[7] Bill Venners y yo hemos discutido esto durante un tiempo, y no hemos podido figurar el por qué se hace así en vez de crear cada objeto correctamente antes de que la prueba sea corrida. Es probable que sea simplemente un artefacto del JUnit de manera que fue originalmente implementado.

Aunque probablemente puedes sobrevivir fácilmente por sólo usar el acercamiento más simple para **JUnit** como se muestra en el ejemplo precedente, **JUnit** fue originalmente diseñado con una abundancia de estructuras complicadas. Si eres curioso, fácilmente puedes aprender más acerca de ellos, porque la descarga del **JUnit** de *www.JUnit.org* viene con documentación y manuales de instrucción.

Mejorando la fiabilidad con aserciones

Las aserciones, la cual has visto anteriormente en los ejemplos usados en este libro, fueron añadidos a la versión de JDK 1.4 para auxiliar a programadores en mejorar la fiabilidad de sus programas. Las aseveraciones correctamente usadas, pueden acrecentar robustez de programa comprobando que ciertas condiciones son satisfechas durante la ejecución de tu programa. Por ejemplo, supón que tienes un campo numérico en un objeto que representa el mes en el Calendario Juliano. Sabes que este valor siempre debe estar en el rango 1-12, y una aserción puede usarse para inspeccionar esto y reportar un error si en cierta forma cae fuera de ese rango. Si estás dentro de un método, puedes comprobar la validez de un argumento con una aserción. Éstas son pruebas importantes para asegurarse de que tu programa es correcto, pero no pueden ser realizadas por la comprobación de fase de compilación, y no caen en el alcance de prueba de unidades. En esta sección, consideraremos la mecánica del mecanismo de aserción, y la forma que puedes usar aserciones para a

medias implementar el diseño por el concepto del contrato.

Sintaxis de Aserción

Dado que puedes simular el efecto de aserciones usando otros modelos estructurados de programación, puede alegarse que el punto integral de añadir aserciones para Java es que son fáciles de escribir. Las declaraciones de aserción vienen en dos formas:

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

Ambos de estas declaraciones dicen “afirmo que la expresión de **boolean** producirá un valor **true**.” Si esto no es el caso, la aserción producirá una excepción **AssertionError**. Ésta es una subclase **Throwable**, y como algo semejante no requiere una especificación de excepción.

Desafortunadamente, la primera forma de aseveración no produce cualquier información conteniendo la expresión de **boolean** en la excepción producida por una aserción fallida (al contrario de la mayoría de los mecanismos de aserción de otros lenguajes). Aquí hay un ejemplo demostrando el uso de la primera forma:

```
//: c15:Assert1.java
// Non-informative style of assert
// Compile with: javac -source 1.4 Assert1.java
// {JVMArgs: -ea} // Must run with -ea
// {ThrowsException}

public class Assert1 {
  public static void main(String[] args) {
    assert false;
  }
} ///:~
```

Las aserciones son puestas en JDK 1.4 por defecto (esto es molesto, pero los diseñadores lograron convencerse ellos mismos de que fue una buena idea). Para impedir errores de fases de compilación, debes compilar con la bandera:

```
-source 1.4
```

Si no usas esta bandera, pondrás a un mensaje charlador a decir que **assert** es una palabra clave en JDK 1.4 y no podrá ser utilizado como un identificador más.

Si precisamente corres el programa de la forma que normalmente haces, sin varias banderas especiales de aserción, nada ocurrirá. Debes permitir aserciones cuando corres el programa. La forma más fácil para hacer esto es con la bandera **-ea**, pero también lo puedes deletrear: **-enableassertions**.

Esto correrá el programa y ejecutará varias declaraciones de aserción, así es que conseguirás:

```
Exception in thread "main" java.lang.AssertionError
        at Assert1.main(Assert1.java:8)
```

Puedes ver que la salida no contiene mucho en la forma de información útil. Por otra parte, si usas la expresión de información, producirás un mensaje útil cuando la aserción fracasa.

Para usar la segunda forma, provees una expresión de información que se desplegó como parte del rastro de la pila de excepción. Esta expresión de información puede producir cualquier tipo de datos en absoluto. Sin embargo, la expresión de información más útil típicamente será un **string** con texto que es útil para el programador. Aquí hay un ejemplo:

```
//: c15:Assert2.java
// Assert with an informative message
// {JVMArgs: -ea}
// {ThrowsException}

public class Assert2 {
  public static void main(String[] args) {
    assert false: "Here's a message saying what happened";
  }
} ///:~
```

Ahora la salida es:

```
Exception in thread "main" java.lang.AssertionError: Here's a
message saying what happened
        at Assert2.main(Assert2.java:6)
```

Aunque lo que ves aquí es simplemente un objeto simple **String**, la expresión de información puede producir cualquier clase de objeto, así es que típicamente construirás a un **string** más complicado conteniendo, por ejemplo, el/los valor/es de objetos que se involucró con la aserción fallida.

Porque la única forma para ver información interesante de una aserción fallida es usar la expresión de información, esa es la forma que está todo el tiempo usada en este libro, y la primera forma es considerada a ser una elección pobre.

También puedes decidir poner aserciones encendidas y apagadas basado en el nombre de clase o el nombre del paquete (es decir, puedes habilitar o puedes inhabilitar aserciones en un paquete entero). Puedes encontrar los detalles en la documentación de JDK 1.4 en aserciones. Esto puede ser útil si tienes un proyecto grande instrumentado con aserciones y quieres cerrar una cierta cantidad de ellas. Sin embargo, registrar o depurar (ambos descrito más adelante en este capítulo) son probablemente mejores herramientas para

capturar esa clase de información. Este libro precisamente pondrá en todas las aserciones cuando es necesario, así es que ignoraremos el control bien granulado de aserciones.

Hay otra forma que puedes controlar aserciones: Programáticamente, enganchando en el objeto **ClassLoader**. JDK 1.4 le añadió varios métodos nuevos a **ClassLoader** que permiten la habilitación y deshabilitación dinámica de aserciones, incluyendo **setDefaultAssertionStatus**(), que establece el estatus de aserción para todas las clases cargadas después. Así es que podrías pensar casi silenciosamente de que podrías poner en todas las aserciones como éste:

```
//: c15:LoaderAssertions.java
// Using the class loader to enable assertions
// Compile with: javac -source 1.4 LoaderAssertions.java
// {ThrowsException}

public class LoaderAssertions {
  public static void main(String[] args) {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
    new Loaded().go();
  }
}

class Loaded {
  public void go() {
    assert false: "Loaded.go()";
  }
} ///:~
```

Aunque esto elimina la necesidad para usar la bandera **-ea** en la línea de comando cuando el programa Java es corrido, no es una solución completa porque todavía debes compilar todo con la bandera **-source 1.4**. Eso puede ser tan franco permitir aserciones usando argumentos de líneas de comando; Al entregar un producto autónomo, probablemente tienes que establecer un escrito de ejecución para que el usuario inicie el programa de cualquier manera, para configurar otros parámetros de arranque.

Eso tiene sentido, sin embargo, decidir que quieres requerir aserciones a estar habilitado cuando el programa es corrido. Puedes lograr con la siguiente cláusula **static**, colocada en la clase principal de tu sistema:

```
static {
  boolean assertionsEnabled = false;
  // Note intentional side effect of assignment:
  assert assertionsEnabled = true;
  if (!assertionsEnabled)
    throw new RuntimeException("Assertions disabled");
}
```

Si las aserciones están habilitadas, entonces la declaración **assert** será ejecutada y **assertionsEnabled** será puesto a **true**. La aserción nunca fracasará, porque el valor de retorno de la asignación es el valor asignado. Si las aserciones no están habilitadas, la declaración **assert** no será ejecutada y **assertionsEnabled** permanecerá **false**, dando como resultado la excepción.

Usando Aserciones por Diseño por Contrato

El *Diseño Por Contrato* (DBC) es un concepto desarrollado por Bertrand Meyer, creador del lenguaje de programación *Eiffel*, para ayudar en la creación de programas robustos garantizando que los objetos siguen ciertas reglas que no pueden ser verificadas por la fase de compilación en la verificación de tipo. **[8]** Estas reglas son determinado por la naturaleza del problema que está siendo solucionado, el cual está fuera del alcance de lo que el compilador puede conocer y probar.

[8] Los Diseños por contrato están descritos en detalle en el Capítulo 11 de Ingeniería de Software Orientado a Objetos, 2da Edición, por Bertrand Meyer, Prentice Hall 1997.

Aunque las aserciones directamente no implementan *DBC* (como lo hace el lenguaje *Eiffel*), pueden estar acostumbrados a crear un estilo informal de programación DBC.

La idea fundamental de DBC es que un contrato claramente especificado existe entre el proveedor de un servicio y el consumidor o el cliente de ese servicio. En la programación orientada a objetos, los servicios están usualmente abastecidos por objetos, y el límite del objeto – la división entre el proveedor y el consumidor – es la interfaz del objeto de la clase. Cuando los clientes llaman un método público particular, esperan cierto comportamiento de esa llamada: Un cambio estatal en el objeto, y un valor previsible de retorno. La tesis de Meyer está que:

1. Este comportamiento puede ser claramente especificado, como si fuera un contrato.
2. Este comportamiento puede ser garantizado implementando ciertas comprobaciones de tiempos de ejecución, el cual él llama condiciones previas, postcondiciones e invariantes.

De todos modos estás de acuerdo que el punto 1 es siempre verdadero, eso parece ser verdadero para bastantes situaciones para hacer DBC un acercamiento interesante. (Creo que, como cualquier solución, hay límites para su utilidad. Pero si conoces estos límites, sabes cuando tratar de aplicarlo) En Particular, una parte muy valiosa del proceso del diseño es la expresión de las restricciones DBC para una clase particular; Si eres incapaz de especificar las restricciones, probablemente no sabes bastante acerca de lo que estás tratando de construir.

Verificar instrucciones

Antes de entrar de fondo en las facilidades DBC, considera el uso más simple para aserciones, el cual Meyer llama *instrucción de comprobación*. Una

instrucción de comprobación expresa tu convicción que una propiedad particular estará satisfecha en este punto en tu código. La idea de la instrucción de comprobación es expresar conclusiones poco obvias en el código, no sólo para verificar la prueba, sino que también como documentación para los lectores futuros del código.

Por ejemplo, en un proceso de química, puedes titular un líquido claro en otro, y cuándo alcanzas un cierto punto, todo se pone azul. Esto no es obvio del color de los dos líquidos; Está en parte de una reacción compleja. Una instrucción útil de comprobación en la terminación del proceso de titulación afirmaría que el líquido resultante es azul.

Otro ejemplo es el método **Thread.holdsLock**() introducido en JDK 1.4. Esto sirve para situaciones complicadas de hilos (como iterar a través de una colección en una forma segura por hilos) donde debes confiar en el programador del cliente u otra clase en tu sistema usando la biblioteca correctamente, en vez de la palabra clave **synchronized** a solas. Asegurar que el código propiamente siga los dictámenes de tu diseño de la biblioteca, puede afirmar que el hilo actual ciertamente sustenta el bloqueo:

```
assert Thread.holdsLock(this); // lock-status assertion
```

Las instrucciones de comprobación son una adición valiosa a tu código. Desde que las aserciones pueden ser deshabilitadas, las instrucciones de comprobación deberían ser usadas cada vez que tienes conocimiento poco obvio acerca del estado de tu objeto o programa.

Precondiciones

Una precondición es una prueba para asegurarse de que el cliente (el código llamando este método) ha cumplido con su parte del contrato. Esto casi siempre significa comprobar los argumentos en el mismo comienzo de una llamada de método (antes de que hagas cualquier otra cosa en ese método) para asegurarse de que esos argumentos son apropiados para uso en el método. Ya que nunca sabes lo que un cliente va a darte, las comprobaciones de precondición son siempre una buena idea.

Postcondiciones

Una prueba de **postcondición** comprueba los resultados de lo que hiciste en el método. Este código se sitúa al final de la llamada de método, antes de la declaración **return**, si hay uno. Por mucho tiempo, los métodos complicados donde el resultado de los cálculos debería verificarse antes de devolverlos (es decir, en situaciones donde por alguna razón no siempre puedes confiar en los resultados), comprobaciones de **postcondición** son esenciales, pero a cualquier hora que puedes describir restricciones en el resultado del método, es sabio expresar esas restricciones en código como una **postcondición**. En Java estos son codificados como aserciones, pero las declaraciones de aserción se diferenciarán de un método a otro.

Invariantes

Un invariante da afianzamientos acerca del estado del objeto que será mantenido entre llamadas de método. Sin embargo, no restringe un método de por ahora divergiendo de esos afianzamientos durante la ejecución del método. Precisamente dice que la información del estado del objeto siempre obedecerá estas reglas:

1. En la entrada para el método.
2. Antes de salir el método.

Además, el invariante es un afianzamiento acerca del estado del objeto después de la construcción.

Según la esta descripción, un invariante efectivo sería definido como un método, probablemente nombrado **invariant**(), lo cual sería invocado después de construcción, y al comienzo y final de cada método. El método podría ser invocado como:

```
assert invariant();
```

Así, si elegiste desactivar aserciones por las razones de desempeño, no habría costos operativos en absoluto.

Remitiendo DBC

Aunque enfatiza la importancia de poder expresar **precondiciones**, **postcondiciones**, e invariantes, y el valor de usar estos durante el desarrollo, Meyer admite que no está del todo práctico incluir todo código DBC en un producto que remite. Puedes remitir la comprobación DBC basada en la cantidad de confianza que puedes colocar en el código en un punto particular. Aquí está la orden de relajación, de más seguro a menos seguro:

1. La comprobación del invariante al principio de cada método puede ser deshabilitado primero, ya que la comprobación del invariante al final de cada método garantizará que la condición del objeto será válida al principio de cada llamada de método. Es decir, generalmente puedes confiar que el estado del objeto no cambiará entre las llamadas de método. Este es una suposición tan segura que podrías escoger para escribir código con comprobaciones del invariante sólo al final.

2. La comprobación de **postcondición** puede estar deshabilitada después, si tienes prueba de unidades razonable que comprueba que tus métodos devuelven valores apropiados. Ya que la comprobación del invariante observa el estado del objeto, la comprobación de **postcondición** sólo valida los resultados del cálculo durante el método, y por eso pueden ser descartados a favor de la prueba de unidades. La prueba de unidades no será tan segura como una comprobación de **postcondición** de tiempo de ejecución, pero puede ser basta, especialmente si te basta la confianza en el código.

3. La comprobación del invariante al final de una llamada de método puede estar deshabilitada si te basta la certeza de que el cuerpo de método no pone el objeto en un estado inválido. Puede ser posible asegurarse esto con prueba de unidades de la caja blanca (es decir, las pruebas de la unidad que tienen acceso a los campos privados, así es que pueden validar el estado del objeto). Así, aunque no puede ser

considerablemente tan robusto como las llamadas para **invariant**(), cabe emigrar la comprobación del invariante de pruebas de tiempos de ejecución para las pruebas de tiempo en la construcción (por la prueba de unidades), lo mismo que con **postcondiciones**.

4. Finalmente, como último recurso puedes desactivar comprobaciones de precondición. Éste es la cosa menos segura y menos aconsejable para hacer, porque aunque sabes y tienes control sobre tu código, no tienes el control sobre qué argumentos el cliente puede pasar a un método. Sin embargo, en una situación donde (a) el desempeño es necesario y perfilando señala las comprobaciones de precondición como un cuello de botella y (b) tenéis alguna clase de seguridad razonable que el cliente no violará precondiciones (como en el caso donde has escrito el código del cliente por ti mismo) puede ser aceptable desactivar comprobaciones de precondición.

No deberías quitar el código que realiza las comprobaciones descritas aquí como desactivas las comprobaciones. Si un problema es descubierto, querrás fácilmente volverte contra las comprobaciones a fin de que rápidamente puedas descubrir el problema.

Ejemplo: DBC + prueba de unidades de la caja blanca

El siguiente ejemplo demuestra la potencia de combinar conceptos de Diseño por contrato con prueba de unidades. Demuestra una pequeña parte de la clase de cola primero que entra, primero que sale que es implementada como un arreglo circular hacia afuera – es decir, un arreglo usado en una moda circular primero - (FIFO). Cuando el final del arreglo es alcanzado, la clase envuelve de regreso más o menos al comienzo.

Podemos hacer un número de definiciones contractuales para esta cola:

1. Precondición (para un **put**()): Los elementos nulos no son admitidos al ser añadidos a la cola.
2. Precondición (para un **put**()): Es ilegal meter elementos en una cola llena.
3. Precondición (para un **get**()): Es ilegal tratar de obtener elementos de una cola vacía.
4. Postcondition (para un **get**()): Los elementos nulos no pueden ser producidos desde el arreglo.
5. Invariante: La región en el arreglo que contiene objetos no puede contener varios elementos nulos.
6. Invariante: La región en el arreglo que no contiene objetos debe tener sólo valores nulos.

Aquí hay una forma que podrías implementar estas reglas, pude usar las llamadas explícitas de método para cada tipo de elemento DBC:

```
//: c15:Queue.java
```

```
// Demonstration of Design by Contract (DBC) combined
// with white-box unit testing.
// {Depends: junit.jar}
import junit.framework.*;
import java.util.*;

public class Queue {
  private Object[] data;
  private int
    in = 0, // Next available storage space
    out = 0; // Next gettable object
  // Has it wrapped around the circular queue?
  private boolean wrapped = false;
  public static class
  QueueException extends RuntimeException {
    public QueueException(String why) { super(why); }
  }
  public Queue(int size) {
    data = new Object[size];
    assert invariant(); // Must be true after construction
  }
  public boolean empty() {
    return !wrapped && in == out;
  }
  public boolean full() {
    return wrapped && in == out;
  }
  public void put(Object item) {
    precondition(item != null, "put() null item");
    precondition(!full(), "put() into full Queue");
    assert invariant();
    data[in++] = item;
    if(in >= data.length) {
      in = 0;
      wrapped = true;
    }
    assert invariant();
  }
  public Object get() {
    precondition(!empty(), "get() from empty Queue");
    assert invariant();
    Object returnVal = data[out];
    data[out] = null;
    out++;
    if(out >= data.length) {
      out = 0;
      wrapped = false;
    }
    assert postcondition(
      returnVal != null, "Null item in Queue");
    assert invariant();
    return returnVal;
  }
```

```
// Design-by-contract support methods:
private static void
precondition(boolean cond, String msg) {
  if(!cond) throw new QueueException(msg);
}
private static boolean
postcondition(boolean cond, String msg) {
  if(!cond) throw new QueueException(msg);
  return true;
}
private boolean invariant() {
  // Guarantee that no null values are in the
  // region of 'data' that holds objects:
  for(int i = out; i != in; i = (i + 1) % data.length)
    if(data[i] == null)
      throw new QueueException("null in queue");
  // Guarantee that only null values are outside the
  // region of 'data' that holds objects:
  if(full()) return true;
  for(int i = in; i != out; i = (i + 1) % data.length)
    if(data[i] != null)
      throw new QueueException(
        "non-null outside of queue range: " + dump());
  return true;
}
private String dump() {
  return "in = " + in +
    ", out = " + out +
    ", full() = " + full() +
    ", empty() = " + empty() +
    ", queue = " + Arrays.asList(data);
}
// JUnit testing.
// As an inner class, this has access to privates:
public static class WhiteBoxTest extends TestCase {
  private Queue queue = new Queue(10);
  private int i = 0;
  public WhiteBoxTest(String name) {
    super(name);
    while(i < 5) // Preload with some data
      queue.put("" + i++);
  }
  // Support methods:
  private void showFullness() {
    assertTrue(queue.full());
    assertFalse(queue.empty());
    // Dump is private, white-box testing allows access:
    System.out.println(queue.dump());
  }
  private void showEmptiness() {
    assertFalse(queue.full());
    assertTrue(queue.empty());
    System.out.println(queue.dump());
```

```
}
public void testFull() {
  System.out.println("testFull");
  System.out.println(queue.dump());
  System.out.println(queue.get());
  System.out.println(queue.get());
  while(!queue.full())
    queue.put("" + i++);
  String msg = "";
  try {
    queue.put("");
  } catch(QueueException e) {
    msg = e.getMessage();
    System.out.println(msg);
  }
  assertEquals(msg, "put() into full Queue");
  showFullness();
}
public void testEmpty() {
  System.out.println("testEmpty");
  while(!queue.empty())
    System.out.println(queue.get());
  String msg = "";
  try {
    queue.get();
  } catch(QueueException e) {
    msg = e.getMessage();
    System.out.println(msg);
  }
  assertEquals(msg, "get() from empty Queue");
  showEmptiness();
}
public void testNullPut() {
  System.out.println("testNullPut");
  String msg = "";
  try {
    queue.put(null);
  } catch(QueueException e) {
    msg = e.getMessage();
    System.out.println(msg);
  }
  assertEquals(msg, "put() null item");
}
public void testCircularity() {
  System.out.println("testCircularity");
  while(!queue.full())
    queue.put("" + i++);
  showFullness();
  // White-box testing accesses private field:
  assertTrue(queue.wrapped);
  while(!queue.empty())
    System.out.println(queue.get());
  showEmptiness();
```

```
      while(!queue.full())
        queue.put("" + i++);
      showFullness();
      while(!queue.empty())
        System.out.println(queue.get());
      showEmptiness();
    }
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(Queue.WhiteBoxTest.class);
  }
} ///:~
```

El contador **in** indica la posición social en el arreglo donde el siguiente objeto irá, y el contador **out** indica dónde el siguiente objeto vendrá. La bandera **wrapped** muestra que adentro ha pasado *alrededor del círculo* y ahora aparece detrás de **out**. Cuando **in** y **out** coinciden, la cola está vacía (si **wrapped** es **false**) o llena (si **wrapped** es **true**).

Puedes ver que los métodos **put**() y **get**() llaman a los métodos **precondition**(), **postcondition**(), e **invariant**(), los métodos privados el cual son definidos más abajo en la clase. **precondition**() y **postcondition**() son los métodos ayudantes diseñados para aclarar el código. Noto que **precondition**() devuelve **void**, porque no es usada con **assert**. Como previamente notó, generalmente querrás guardarte precondiciones en tu código; Sin embargo, envolviéndolos en una llamada de método de **precondition**(), tienes mejores opciones si te reduces al movimiento horrendo de desactivarlas.

postcondición() e **invariant**() retornan un valor **Boolean** a fin de que puedan ser usados en declaraciones **assert**. Luego, si las aserciones están deshabilitadas por razones de desempeño, no habrá llamadas de método en absoluto.

invariant() realiza verificaciones de validez internas en el objeto. Puedes ver que ésta es una operación costosa para hacer en ambos el comienzo y final de cada llamada de método, como Meyer sugiere. Sin embargo, es muy valioso tener así de claramente representado en el código, y me ayudó a obtener la implementación para ser correcto. Además, si haces varios cambios a la implementación, el **invariant**() asegurará que no hayas descifrado el código. Pero puedes ver que sería medianamente trivial activar las pruebas del **invariant** de las llamadas de método en el código de prueba de la unidad. Si tus pruebas de la unidad son razonablemente cabales, puedes tener un nivel razonable de confianza que los **invariantes** serán respetados.

Note que el método ayudante **dump**() devuelve a un **string** conteniendo todos los datos en vez de imprimir los datos directamente. Este acercamiento permite muchas más opciones en lo que se refiere a cómo puede estar la información usada.

El **WhiteBoxTest**, subclase de **TestCase** es creada como una clase interna a fin de que tenga acceso a los elementos privados de **Queue** y puede así

validar la implementación subyacente, no simplemente el comportamiento de la clase como en una prueba de caja blanca. El constructor agrega algunos datos a fin de que el **Queue** esté parcialmente lleno para cada prueba. Se quiere decir que los métodos de soporte **showFullness**() y **showEmptiness**() son llamados para comprobar que el **Queue** está lleno o vacío, respectivamente. Cada uno de los cuatro métodos de prueba asegura que un aspecto diferente de la operación **Queue** funciona correctamente.

Nota que combinando a DBC con prueba de unidades, no sólo sacas lo mejor de ambos mundos, pero también tienes un camino de migración – puedes activar pruebas DBC para las pruebas de la unidad en vez de simplemente desactivándolas, así es que todavía tienes algo nivelado de experimentación.

Construyendo con Ant

Me percaté de que para un sistema estar construido en una moda robusta y fidedigna, necesité automatizar todo lo que entre en el proceso de la construcción. El tiempo pasó, y dos acontecimientos ocurrieron. Primero, que yo comencé a crear proyectos más complicados comprendiendo muchos archivos más. El camino de mantenimiento del cual los archivos necesitaron compilación llegó a ser más de lo que pude (o quise) pensar. En segundo lugar, por esta complejidad a la que comencé a darme cuenta de que no importa cuán simple el proceso de la constitución podría ser, si haces algo más que un par de vece, comienzas a ponerte descuidado, y las partes del proceso comienzan a caer a través de los cracks.

Automatiza todo

La utilidad **make** apareció junto con C como una herramienta para crear a la función primaria del sistema operativo. Unix **make** es comparar la fecha de dos archivos y realizar alguna operación que traerá esos dos archivos al día con cada otro. Las relaciones entre todos los archivos en tus proyectos y las reglas necesarias para ponerlos al día (la regla usualmente es ejecutando el compilador C/C++ en un archivo fuente) están contenidas en un **makefile**. El programador crea un **makefile** conteniendo la descripción de cómo construir el sistema. Cuando quieres traer el sistema al día, simplemente escribes **make** en la línea de comando. Hasta el día de hoy, instalar programas de Unix/Linux consiste de desempacarlos y escribiendo comandos **make**.

Problemas con make

El concepto de **make** es claramente una buena idea, y esta idea proliferada para producir muchas versiones de **make**. Los vendedores del compiladores C y C++ típicamente incluyeron su variación de **make** junto con su compilador – estas variaciones a menudo tomaron libertades con lo que las personas consideradas para ser las reglas estándar del **makefile**, así los **makefiles** resultantes no correrían con cada otro. El problema fue finalmente solucionado (como a menudo ha sido el caso) por un **make** que fue, y todavía es, también superior a todos los demás **makes**, y es gratis, así no hay resistencia a usarla:

GNU **make**. **[9]** Esta herramienta tiene una característica significativamente mejor determinada que las otras versiones de **make** y está disponible en todas las plataformas.

[9] Excepto por la compañía ocasional que, por razones más allá de la comprensión, está todavía convencido que las herramientas de fuente cerrada son en cierta forma mejores o tienen soporte técnico superior. Las únicas situaciones donde he visto así de cierto son cuando las herramientas le tienen una base muy pequeña al usuario, pero aun así sería más seguro contratar a asesores para modificar herramientas de fuente abierta, y así apalanca el proyecto previo y garantiza que el proyecto por el que pagas no se volverá indisponible para ti (y también sería más probablemente que encontrarás a otros asesores ya listos para acelerar en el programa).

En las dos ediciones previas de Piensa en Java, usé **makefiles** para construir todo el código en el árbol de código fuente del libro. Automáticamente generé a estos **makefiles** – uno en cada directorio, y un **makefile** maestro en el directorio raíz que llamaría el resto – usando una herramienta que originalmente escribí en C++ (en cuestión de 2 semanas) para Piensa en C++, y más tarde reescrito en *Python* (en cuestión de la mitad de día) llamado *MakeBuilder.py* **[10]** que trabajó para ambos Windows y Linux/Unix, pero tuve que escribir código adicional para hacer que esto ocurra Y nunca lo probé en la Macintosh. Allí dentro yace el primer problema con **make**: Lo puedes obtener para trabajar en plataformas múltiples, pero no es esencialmente de interplataforma. Así para un lenguaje que es supuesto para ser "escribe una vez, corre dondequiera" (es decir, Java), puedes gastar una parte de esfuerzo metiendo el mismo comportamiento en el sistema de la construcción si usa **make**.

[10] Esto no está disponible en el sitio Web porque es demasiado hecho a la medida para ser generalmente útil.

El resto de problemas con **make** probablemente pueden estar resumidos diciendo que es como una parte de herramientas desarrolladas para Unix; La persona creando la herramienta no podría resistir la tentación por crear su sintaxis de lenguaje, y como consecuencia, Unix se llena de herramientas que son todos notablemente diferentes, e igualmente incomprensible. Esto es, la sintaxis **make** es realmente difícil de entender en su totalidad – la he estado aprendiendo por años – y tiene montones de cosas molestas como su insistencia en etiquetas en lugar de espacios. **[11]**

[11] Otras herramientas están bajo desarrollo, que tratan de reparar los problemas con make sin hacer compromisos de *Ant*. Vaya, por ejemplo, *www.a-a-p.org* o busca en la Web "*bjam*".

Todo lo que se dice, note que todavía encuentro **GNU make** indispensable para muchos de los proyectos que creo.

Ant: El estándar del defacto

Todos estos asuntos con **make** le irritaron a un programador Java llamado

James Duncan Davidson lo suficiente como para causar que él creara **Ant** como una herramienta de fuente abierta tan emigrado para el proyecto Apache en *http://jakarta.apache.org/ant*. Este sitio contiene la descarga completa incluyendo al ejecutable **Ant** y documentación. **Ant** ha crecido y ha mejorado hasta que es ahora generalmente aceptado como la herramienta de construcción del estándar del defacto para proyectos Java.

Para la interplataforma **make Ant**, el formato para los archivos de descripción de proyecto es XML (cubierto en Piensa en Java Empresarial). En lugar de un **makefile**, creas a un **buildfile**, lo cual es nombrado por defecto **build.xml** (éste te permite precisamente decir **ant** en la línea de comando. Si nombras tu otra cosa del **buildfile**, tienes que especificar ese nombre con una bandera de línea de comando).

El requisito sólo rígido para tu **buildfile** es que sea un archivo válido XML. **Ant** compensa los asuntos específicos en la plataforma como el fin de la línea de caracteres y separadores de la ruta del directorio. Puedes usar etiquetas o espacios en el **buildfile** como escojas. Además, la sintaxis y los nombres de la etiqueta usados en **buildfiles** dan como resultado código legible, comprensible (y así, mantenible).

Encima de todo esto, **Ant** es diseñado para ser extensible, con una interfase estándar que te permite escribir tus tareas si los que originó con **Ant** no es suficiente (sin embargo, usualmente son, y el arsenal regularmente se expande).

A diferencia de **make**, la curva de aprendizaje para Ant es razonablemente suave. No necesitas saber mucho para crear un **buildfile** que compila código Java en un directorio. Aquí está un archivo **build.xml** muy básico, por ejemplo, del Capítulo 2 de este libro:

```
<?xml version="1.0"?>

<project name="Thinking in Java (c02)"
  default="c02.run" basedir=".">
  <!-- build all classes in this directory -->
  <target name="c02.build">
    <javac
      srcdir="${basedir}"
      classpath="${basedir}/.."
      source="1.4"
    />
  </target>

  <!-- run all classes in this directory -->
  <target name="c02.run" depends="c02.build">
    <antcall target="HelloDate.run"/>
  </target>

  <target name="HelloDate.run">
    <java
      taskname="HelloDate"
```

```
      classname="HelloDate"
      classpath="${basedir};${basedir}/.."
      fork="true"
      failonerror="true"
    />
  </target>

  <!-- delete all class files -->
  <target name="clean">
    <delete>
      <fileset dir="${basedir}" includes="**/*.class"/>
      <fileset dir="${basedir}" includes="**/*Output.txt"/>
    </delete>
    <echo message="clean successful"/>
  </target>

</project>
```

La primera línea manifiesta que este archivo se conforma a la versión 1.0 de XML. XML se parece mucho al HTML (note que la sintaxis del comentario es idéntica), excepto que puedes hacer tus nombres de la etiqueta y el formato estrictamente debe conformarse a las reglas XML. Por ejemplo, una etiqueta abridora como **project** o debe acabar dentro de la etiqueta en su cuadral de cierre con un slash (**/>**) o debe tener una etiqueta que hace juego de cierre como ve al final del archivo (**</project>**). Dentro de una etiqueta puedes tener atributos, pero los valores de atributo deben estar rodeados en citas. XML permite formateo libre, pero la sangría como ves aquí es típica.

Cada **buildfile** puede manejar un proyecto solo descrito por su etiqueta <**project**>. El proyecto tiene un atributo opcional **name** que es usado cuando se despliega información acerca de la construcción. El atributo por omisión es requerido y se refiere al destino que se construye cuando precisamente escribes **ant** en la línea de comando sin dar un nombre específico de destino. El directorio de referencia **basedir** puede ser usado en otros lugares en el **buildfile**.

Un destino tiene dependencias y tareas. Las dependencias dicen “¿cuáles otros destinos deben construirse antes de que este destino pueda construirse?” Notarás que el destino predeterminado a construir es **c02.run**, y el destino del **c02.run** dice que a su vez depende de **c02.build**. Así, el destino del **c02.build** debe ser ejecutado antes de que **c02.run** pueda ser ejecutado. Dividir en partes el **buildfile** de este modo no sólo da facilidades para entender, pero también te permite escoger lo que quieres hacer por la línea de comando **Ant**; Si dices **ant c02.build**, entonces sólo compilará el código, pero si dices **ant co2.run** (o, por el destino predeterminado, simplemente **ant**), entonces primero hará cosas seguras han sido construidas, y luego ejecuta los ejemplos.

Entonces, el proyecto para ser exitoso, los destinos **c02.build** y **c02.run** primero deben tener éxito, en ese orden. El destino de **c02.build** contiene una tarea única, el cual es una orden que realmente hace el trabajo de traer

cosas al día. Esta tarea le corre el compilador del **javac** en todos los archivos Java en este directorio base actual; Note la sintaxis **${}** usada para producir el valor de una variable previamente definido, y que la orientación de slashes en rutas del directorio no es importante, ya que **Ant** compensa a merced del sistema operativo en el que lo corres. El atributo del **classpath** da una lista del directorio para agregar al **classpath** de **Ant**, y la fuente especifica al compilador a usar (éste es de hecho sólo notada por JDK 1.4 y más allá). Noto que el compilador Java es responsable de ordenar las dependencias entre las clases mismas, así es que no tienes que explícitamente indicar las dependencias del interarchivo como debes con **make** y C/C++ (esto ahorra muchísimo esfuerzo).

Para correr los programas en el directorio (el cual, en este caso, es sólo el sencillo programa **HelloDate**), este **buildfile** usa una tarea nombrada **antcall**. Esta tarea hace una invocación recursiva de **Ant** en otro destino, el cual en este caso solamente usa java para ejecutar el programa. Note que la tarea del java tiene un atributo del **taskname**; Este atributo está realmente disponible para todas las tareas, y es usado cuando **Ant** devuelve información de registro de actividades.

Como podría esperar, la etiqueta del java también tiene opciones para establecer el nombre de clase para ser ejecutada, y el **classpath**. Además, el

```
fork="true"
failonerror="true"
```

Los atributos dicen a **Ant** que se bifurque fuera de un nuevo proceso para correr este programa, y fallar la construcción **Ant** si el programa fracasa. Puedes buscar todas las tareas diferentes y sus atributos en la documentación que viene con la descarga de **Ant**.

El último destino es uno que es típicamente encontrado en cada **buildfile** ; Te permite decir **ant clean** y suprimir todos los archivos que han sido creados para realizar esta construcción. Cada vez que creas a un **buildfile**, deberías tener el cuidado de incluir un destino **clean**, porque eres la persona que quien típicamente conoce más que nada acerca de cuál puede ser suprimido y qué debería ser conservado.

El destino **clean** introduce alguna sintaxis nueva. Puedes suprimir artículos solos con la versión de una línea de esta tarea, como éste:

```
<delete file="${basedir}/HelloDate.class"/>
```

La versión de multilínea de la tarea te permite especificar a un **fileset**, lo cual es una descripción más complicada de un conjunto de archivos y puede especificar archivos para incluir y excluir usando comodines. En este ejemplo, los **filesets** a suprimir incluyen todos los archivos en este directorio y todos los subdirectorios que tienen una extensión **.class**, y todos los archivos en el subdirectorio actual que acaba con **Output.txt**.

El **buildfile** mostrado aquí es medianamente simple; Dentro del árbol de código fuente de este libro (que es bajable desde disco de *www.BruceEckel.com*) encontrarás **buildfiles** más complicados. También, **Ant** es capaz de hacer bastante más que lo que destinamos para este libro. Para los detalles completos de sus capacidades, vea la documentación que viene con la instalación de **Ant**.

Extensiones Ant

Ant viene con una extensión API a fin de que puedas crear tus tareas escribiéndolas en Java. Puedes encontrar detalles completos en la documentación oficial de **Ant** y en los libros publicados en **Ant**.

Como alternativa, simplemente puedes escribir un programa Java y lo puedes llamar desde **Ant**; Así, no tienes que aprender la extensión del API. Por Ejemplo, para compilar el código en este libro, necesitamos asegurarnos que la versión de Java que el usuario corre es JDK 1.4 o mayor, así es que creamos el siguiente programa:

```
//: com:bruceeckel:tools:CheckVersion.java
// {RunByHand}
package com.bruceeckel.tools;

public class CheckVersion {
  public static void main(String[] args) {
    String version = System.getProperty("java.version");
    char minor = version.charAt(2);
    char point = version.charAt(4);
    if(minor < '4' || point < '1')
      throw new RuntimeException("JDK 1.4.1 or higher " +
        "is required to run the examples in this book.");
    System.out.println("JDK version "+ version + " found");
  }
} ///:~
```

Esto simplemente usa **System.getProperty**() para descubrir la versión Java, y lanza una excepción si no es por lo menos 1.4. Cuando **Ant** ve la excepción, hará un alto. Ahora puedes incluir lo siguiente en cualquier **buildfile** donde quieres comprobar el número de versión:

```
<java
      taskname="CheckVersion"
      classname="com.bruceeckel.tools.CheckVersion"
      classpath="${basedir}"
      fork="true"
      failonerror="true"
/>
```

Si usas este acercamiento para agregar las herramientas, los puedes escribir y probar rápidamente, y si es justificado, puedes invertir el esfuerzo adicional y

puedes escribir una extensión **Ant**.

Control de versión con CVS

Un sistema de control de revisión es una clase de herramienta que ha sido desarrollada durante muchos años para ayudar a manejar proyectos grandes de programación del equipo. También ha resultado ser fundamental para el éxito de virtualmente todos los proyectos de la fuente abierta, porque los equipos de la fuente abierta son casi siempre distribuidos globalmente por la Internet. Entonces aun si hay funcionamiento de sólo dos personas de un proyecto, se aprovechan de usar un *sistema de control de revisión*.

El *sistema de control de revisión* del estándar del defacto para los proyectos de la fuente abierta es llamado *Sistema Concurrente de Versiones* (*CVS*), disponible en www.cvshome.org. Porque es fuente abierta y así muchas personas sabrán cómo usarlo, CVS es también una elección común para proyectos terminados. Algunos proyectos aun usan a CVS como una forma para distribuir el sistema. CVS tiene los beneficios usuales del popular proyecto de fuente abierta: El código ha sido revisado a fondo, está disponible para su revisión y modificación, y los desperfectos se corrigen rápidamente.

CVS guarda tu código en un depositario en un servidor. Este servidor puede estar en una red de área local, pero está típicamente disponible en la Internet a fin de que las personas en el equipo puedan obtener actualizaciones sin estar en una posición particular. Para conectarse a CVS, debes tener una contraseña y nombre de usuario asignado, así hay un nivel razonable de seguridad; Para más seguridad, puedes usar el protocolo del **ssh** (aunque éstas son herramientas Linux, están fácilmente disponibles en Windows usando **Cygwin** – vea *www.cygwin.com*). Algunos ambientes gráficos de desarrollo (como el editor libre *Eclipse*; vea *www.eclipse.org*) provee excelente integración de CVS.

Una vez que el depositario es inicializado por tu administrador de sistema, los miembros del equipo pueden pasar una copia del árbol de código a través de una comprobación. Por ejemplo, una vez que tu máquina es registrada en el servidor CVS correcto (los detalles de los cuales se omiten aquí), puedes realizar la comprobación de resultados de salida inicial con una orden como esto:

```
cvs –z5 co TIJ3
```

Esto se conectará con el servidor CVS y negociará la verificación de resultados de salida (**'co'**) del depositario de código llamado **TIJ3**. El argumento **'- z5'** dice al CVS que los programas en ambos extremos a comunica usan un nivel de compresión del **gzip** de **5** para acelerar la transferencia sobre la red.

Una vez esta orden es completada, tendrás una copia del depositario de código en tu máquina local. Además, verás que cada directorio en el depositario tiene un subdirectorio adicional denominado CVS que es donde está toda la información CVS de los archivos que en ese directorio son almacenados.

Ahora que tienes tu copia del depositario CVS, puedes hacer cambios a los archivos para desarrollar el proyecto. Típicamente, estos cambios incluyen correcciones y adiciones de característica junto con código experimental y los **buildfiles** modificados necesarios para compilar y correr las pruebas. Te encontrarás con que es muy insociable registrarse en código que exitosamente no corre todas sus pruebas, porque entonces todos los demás en el equipo obtendrán el código inservible (y así falla sus constituciones).

Cuando has hecho tus mejoras y estáis listos a registrar la entrada de ellos, debes experimentar un proceso de dos pasos que lo estar el quid de sincronización de código CVS. Primero, actualizas tu depositario local para sincronizarlo con el depositario principal CVS mudándose a la raíz de tu depositario local de código y corriendo este comando:

```
cvs update –dP
```

En este punto, no estás obligado a entrar al sistema porque el subdirectorio CVS guarda la información de entrada en el sistema para el depositario remoto, y el depositario remoto conserva información distintiva acerca de tu máquina como una verificación para verificar tu identidad.

La **'- dP'** bandera es opcional; **'- d'** dice a CVS que cree más directorios nuevos en tu máquina local que podría haber sido añadida a la depositaria principal, y **'- P'** dice a CVS que recorte directorios en tu máquina local que ha sido vaciada en el confidente principal. Ninguna de estas cosas ocurre por defecto.

La actividad principal de actualización, sin embargo, es realmente interesante. Realmente deberías correr actualización de forma regular, no sólo antes de que hagas una verificación, porque sincroniza a tu depositario local con el depositario principal. Si encuentra más archivos en el confidente principal que son más nuevos que archivos en tu depositario local, trae los cambios encima de tu máquina local. Sin embargo, no solo copia los archivos, sino que en lugar de eso hace una comparación línea por línea de los archivos y parcha los cambios del depositario principal en tu versión local. Si has hecho algunos cambios a un archivo y alguien más ha hecho cambios al mismo archivo, CVS parchará los cambios conjuntamente con tal de que los cambios no ocurran a las mismas líneas de código (CVS corresponde al contenido de las líneas, y no simplemente los números de la línea, así es que aun si los números de la línea cambian, podrá sincronizar correctamente). Así, puedes estar trabajando en el mismo archivo como alguien más, y cuándo haces una actualización, cualquier cambio que la otra persona ha cometido al depositario principal serán anexados con tus cambios.

Por supuesto, es posible que dos personas pudieran hacer cambios a las mismas líneas del mismo archivo. Éste es un accidente debido a la falta de comunicación; Normalmente dirás cada quien esté trabajando en su parte para no pisar el código de cada quien (también, si los archivos son tan grandes esto tiene sentido para dos personas diferentes para dedicarse a las partes diferentes del mismo archivo, podrías considerar separar los archivos grandes

en archivos más pequeños para la administración del proyecto más fácil). Si esto ocurre, CVS simplemente nota la colisión y te obliga a resolverla arreglando las líneas de código que colisiona.

Note que ninguno de los archivos de tu máquina son movidos en el depositario principal durante una actualización. La actualización trae sólo archivos cambiados del depositario principal en tu máquina y los parches en varias modificaciones que has hecho. ¿Entonces cómo meten tus modificaciones en el depositario principal? Éste es el segundo paso: Lo comete.

Cuando escribes

```
cvs commit
```

CVS pondrá en marcha tu editor predeterminado y te preguntará a ti que escribas una descripción de tu modificación. Esta descripción será introducida en el depositario a fin de que otros sepan cual ha sido cambiado. Después de eso, tus archivos modificados serán colocados en el depositario principal así ellos estarán disponibles para los demás la próxima vez que hacen una actualiz ación.

CVS tiene otras capacidades, solamente comprobación, actualización, y comisión son lo que estarás desempeñándote la mayoría de las veces. Para información detallada acerca de CVS, los libros están disponibles, y el sitio Web principal CVS tiene documentación completa: *www.cvshome.org*. Además, puedes buscar en la Internet usando a *Google* u otros motores de búsqueda; Hay varias introducciones muy condensadas para CVS que te puede iniciar sin abarrancarte con demasiados detalles (el *Gentoo Linux CVS Tutorial* por *Daniel Robbins* (*www.gentoo.org/doc/cvs-tutorial.html*) es en particular franco).

Construcciones diarias

Incorporando compilación y probando en tus **buildfiles**, puedes seguir la práctica de realizar construcciones diarias, apoyadas por las personas De Programación Extremas y otros. A pesar del número de características que actualmente has implementado, siempre guardas tu sistema en un estado en el cual puede construirse exitosamente, de tal manera que si alguien realiza una verificación de resultados de salida y ejecuta **Ant**, el **buildfile** realizará todas las compilaciones y correrá todas las pruebas sin fallar.

Ésta es una técnica poderosa. Quiere decir que siempre tienes, como una línea de fondo, un sistema que compila y pasa todas sus pruebas. En cualquier momento, siempre puedes ver que el verdadero estado del proceso de desarrollo es examinando los rasgos que son de hecho implementados dentro del sistema ejecutable. Uno de los beneficios de este acercamiento es que nadie tiene que perder el tiempo sacando de entre manos un informe explicando lo que sigue con el sistema; Todos pueden ver por ellos mismos revisando una construcción actual y ejecutando el programa.

Corriendo construcciones diariamente, o a menudo, también aseguran que si alguien (accidentalmente, suponemos) comprueba que en los cambios causan

que las pruebas fallen, estarás al tanto en brevemente, antes de que esos problemas tengan posibilidad de propagar más problemas en el sistema. **Ant** aun tiene una tarea que se enviará por correo electrónico, porque muchos equipos colocan su **buildfile** como un trabajo **cron [12]** para automáticamente correrlo diariamente, o mejor varias veces al día, y envían un email si fracasa. Hay también una herramienta de fuente abierta que automáticamente realiza construcciones y provee una página de Web para demostrar el estado de proyecto; vea *http://cruisecontrol.sourceforge.net*

[12] Cron es un programa que fue desarrollado bajo *Unix* para correr programa en tiempos especificados. Sin embargo, está también disponible en versiones libres bajo Windows, y como un servicio del Windows NT/2000: *http://www.kalab.com/freeware/cron/cron.htm*.

Registro De Actividades

El registro de actividades es el proceso de reportar información acerca de un programa en funcionamiento. En un programa depurado, esta información puede ser información común de estado que describe el progreso del programa (por ejemplo, si tienes un programa de instalación, puedes registrar los pasos tomados durante la instalación, los directorios donde almacenaste archivos, los valores de arranque para el programa, etc.).

El registro de actividades es también muy útil durante la corrección de errores. Sin registro de actividades, podrías tratar de descifrar el comportamiento de un programa insertando declaraciones **println**(). Muchos ejemplos en este libro usan esa misma técnica, y a falta de un depurador (un tema que será introducido en poco tiempo), se trata de todo lo que tienes. Sin embargo, una vez te decides que el programa trabaja correctamente, probablemente sacarás las declaraciones **println**(). Entonces si te topas con más problemas, puedes necesitar reponerlos adentro. Es mucho más agradable si puedes echar alguna clase de declaraciones de salida, lo cual sólo será usado cuando sea necesario.

Antes de la disponibilidad de la API de registro de actividades en JDK 1.4, los programadores a menudo usarían una técnica que confía en el hecho que el compilador Java optimizará código que nunca será llamado. Si **debug** es un **boolean static final** y dices:

```
if(debug) {
  System.out.println("Debug info");
}
```

Usando esta técnica, puedes colocar código del rastro a lo largo de tu programa y fácilmente lo puedes revolver de vez en cuando. Entonces cuando **debug** es **false**, el compilador completamente quitará el código dentro de los refuerzos (así el código no causa cualquier costos operativos de tiempos de ejecución en absoluto cuando no es usado).

La API de registro de actividades en JDK 1.4 provee una facilidad más sofisticada para reportar información acerca de tu programa con casi la misma eficiencia de la técnica en el ejemplo anterior.

Una desventaja a la técnica, sin embargo, es que debes recompilar tu código para activar tus declaraciones de rastro de vez en cuando, mientras que es generalmente más conveniente poder activar el rastro sin recompilar el programa usando un archivo de configuración que puedes cambiar para modificar las propiedades de registro de actividades.

La API de registro de actividades en JDK 1.4 provee una facilidad más sofisticada para reportar información acerca de tu programa con casi la misma eficiencia de la técnica en el ejemplo anterior. Para un registro de actividades informativo muy simple, puedes hacer algo como esto:

```
//: c15:InfoLogging.java
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.io.*;

public class InfoLogging {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("InfoLogging");
  public static void main(String[] args) {
    logger.info("Logging an INFO-level message");
    monitor.expect(new String[] {
      "%% .* InfoLogging main",
      "INFO: Logging an INFO-level message"
    });
  }
} ///:~
```

La salida durante una ejecución es:

```
Jul 7, 2002 6:59:46 PM InfoLogging main
INFO: Logging an INFO-level message
```

Note que el sistema de registro de actividades ha detectado el nombre de la clase y método del cual el mensaje de registro se originó. No es garantizado que estos nombres sean correctos, así es que no debes confiar en su exactitud. Si quieres asegurar que el nombre correcto de la clase y del método sean impresos, puedes usar un método más complicado para registrar el mensaje, como éste:

```
//: c15:InfoLogging2.java
// Guaranteeing proper class and method names
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.io.*;
```

```
public class InfoLogging2 {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("InfoLogging2");
  public static void main(String[] args) {
    logger.logp(Level.INFO, "InfoLogging2", "main",
      "Logging an INFO-level message");
    monitor.expect(new String[] {
      "%% .* InfoLogging2 main",
      "INFO: Logging an INFO-level message"
    });
  }
} ///:~
```

El método **logp**() toma argumentos del nivel de registro de actividades (te enterarás de esto después), el nombre de clase y método, y el **string** de registro de actividades. Puedes ver que es mucho más simple si solo confía en el acercamiento automático si la clase y los nombres de método reportados durante el registro de actividades no sean críticos.

Registrando Niveles

La API de registro de actividades provee niveles múltiples de información y la habilidad para convertirse en un nivel diferente durante la ejecución de programa. Así, dinámicamente puedes colocar el nivel de registro de actividades para cualquiera de los siguientes estados:

Nivel	**Efecto**	**Valor Numérico**
OFF	Ninguno de los mensajes de registro de actividades son reportados.	**Integer.MAX_VALUE**
SEVERE	Los únicos mensajes de registro de actividades con el nivel **SEVERE** son reportados.	1000
WARNING	Registrando mensajes con niveles de **WARNING** y **SEVERE** son reportados.	900
INFO	Registrando mensajes con niveles de **INFO** y arriba son reportados.	800

CONFIG	Registrando mensajes con niveles de **CONFIG** y arriba son reportados.	700
FINE	Registrando mensajes con niveles de **FINE** y arriba son reportados.	500
FINER	Registrando mensajes con niveles de **FINER** y arriba son reportados.	400
FINEST	Registrando mensajes con niveles de **FINEST** y arriba son reportados.	300
ALL	Todos los mensajes de registro de actividades son reportados.	**Integer.MIN_VALUE**

Aun puedes heredar de **java.util.Logging.Level** (que ha protegido a los constructores) y puede definir tu nivel. Esto podría, por ejemplo, tener un valor de menos de 300, así es que el nivel está menos de **FINEST**. Luego registrando mensajes en tu nivel nuevo no aparecería cuándo el nivel es **FINEST**.

Puedes ver el efecto de probar los niveles diferentes de registro de actividades en el siguiente ejemplo:

```
//: c15:LoggingLevels.java
import com.bruceeckel.simpletest.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.LogManager;

public class LoggingLevels {
  private static Test monitor = new Test();
  private static Logger
    lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
  private static void logMessages() {
    lgr.info("com : info");
```

```
    lgr2.info("com.bruceeckel : info");
    util.info("util : info");
    test.severe("test : severe");
    rand.info("random : info");
  }
  public static void main(String[] args) {
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
    util.setLevel(Level.FINEST);
    test.setLevel(Level.FINEST);
    rand.setLevel(Level.FINEST);
    System.out.println("individual loggers set to FINEST");
    logMessages();
    lgr.setLevel(Level.SEVERE);
    System.out.println("com level: SEVERE");
    logMessages();
    monitor.expect("LoggingLevels.out");
  }
} ///:~
```

Las primeras pocas líneas de **main**() son necesarias porque el nivel predeterminado de registrar mensajes que serán reportados es **INFO** y mayor (más severos). Si no cambias esto, entonces los mensajes de nivel **CONFIG** y debajo no serán reportados (prueba remover las líneas para ver lo que ocurre).

Puedes tener objetos múltiples del registrador en tu programa, y estos registradores son organizados en un árbol jerárquico, lo cual puede ser programáticamente asociado con el **namespace** del paquete. Los registradores hijos le siguen la pista a su padre de inmediato y por defecto pasan los registros del registro de actividades al límite del padre.

El objeto "**raíz**" del registrador está todo el tiempo creado por defecto, y es la base del árbol de objetos del registrador. Llevas una referencia al registrador de la raíz llamando el método estático **Logger.getLogger**(**""**). Note que toma una cadena vacía en vez de ningunos argumentos.

Cada objeto **Logger** puede tener uno o más objetos **Handler** asociados con él. Cada objeto **Handler** le provee a un estrategia **[13]** para publicar la información de registro de actividades, lo cual está contenido en objetos **LogRecord**. Para crear un tipo nuevo de **Handler**, simplemente heredas de la clase **Handler** y sobrescribes el método **publish**() (junto con **flush**() y **close**(), para negociar con varios flujos que puedes usar en el **Handler**).

[13] Un algoritmo conectable. Las estrategias te permiten fácilmente cambiar una parte de una solución mientras dejas el resto inalterados. Son a menudo usados (como en este caso) como las formas para permitir al programador cliente proveer una porción del código necesario para solucionar un problema particular. Para más detalles, vea *Piensa en Patrones* (con Java) en *www.BruceEckel.com*.

El registrador raíz siempre tiene a un manipulador asociado por defecto, lo cual

envía la salida a la consola. Para acceder a los manipuladores, llamas a **getHandlers**() en el objeto **Logger**. En el ejemplo anterior, sabemos que hay sólo un manipulador así es que técnicamente no necesitamos iterar a través de la lista, pero es más seguro hacer eso en general porque alguien más pudo haber agregado a otros manipuladores para el registrador raíz. El nivel predeterminado de cada manipulador es **INFO**, así para ver todos los mensajes, colocamos el nivel a **ALL** (que equivale a **FINEST**).

El arreglo de niveles permite experimentación fácil de todos los valores **Level**. El registrador es colocado para cada valor y todos los niveles diferentes de registro de actividades son intentados. En la salida puedes ver sólo mensajes en el nivel de registro de actividades actualmente seleccionado, y esos mensajes que son más severos, son reportados.

LogRecords

Un **LogRecord** es un ejemplo de un objeto Mensajero, **[14]** cuyo trabajo es simplemente llevar información de un sitio a otro. Todos los métodos en el **LogRecord** son **getters** y **setters**. Aquí está un ejemplo que descarga toda la información almacenada en un **LogRecord** usando los métodos **getter**:

[14] Un término acuñado por Bill Venners. Éste puede o no puede ser un patrón del diseño.

```
//: c15:PrintableLogRecord.java
// Override LogRecord toString()
import com.bruceeckel.simpletest.*;
import java.util.ResourceBundle;
import java.util.logging.*;

public class PrintableLogRecord extends LogRecord {
  private static Test monitor = new Test();
  public PrintableLogRecord(Level level, String str) {
    super(level, str);
  }
  public String toString() {
    String result = "Level<" + getLevel() + ">\n"
      + "LoggerName<" + getLoggerName() + ">\n"
      + "Message<" + getMessage() + ">\n"
      + "CurrentMillis<" + getMillis() + ">\n"
      + "Params";
    Object[] objParams = getParameters();
    if(objParams == null)
      result += "<null>\n";
    else
      for(int i = 0; i < objParams.length; i++)
        result += "  Param # <" + i + " value " +
          objParams[i].toString() + ">\n";
    result += "ResourceBundle<" + getResourceBundle()
      + ">\nResourceBundleName<" + getResourceBundleName()
      + ">\nSequenceNumber<" + getSequenceNumber()
      + ">\nSourceClassName<" + getSourceClassName()
```

```
      + ">\nSourceMethodName<" + getSourceMethodName()
      + ">\nThread Id<" + getThreadID()
      + ">\nThrown<" + getThrown() + ">";
    return result;
  }
  public static void main(String[] args) {
    PrintableLogRecord logRecord = new PrintableLogRecord(
      Level.FINEST, "Simple Log Record");
    System.out.println(logRecord);
    monitor.expect(new String[] {
      "Level<FINEST>",
      "LoggerName<null>",
      "Message<Simple Log Record>",
      "%% CurrentMillis<.+>",
      "Params<null>",
      "ResourceBundle<null>",
      "ResourceBundleName<null>",
      "SequenceNumber<0>",
      "SourceClassName<null>",
      "SourceMethodName<null>",
      "Thread Id<10>",
      "Thrown<null>"
    });
  }
} ///:~
```

PrintableLogRecord es una extensión simple de **LogRecord** que sobrescribe a **toString**() para llamar todos los métodos **getter** disponibles en **LogRecord**.

Manipuladores

Como notó previamente, fácilmente puedes crear a tu propio manipulador heredando de **Handler** y definiendo a **publish**() para realizar tus operaciones deseadas. Sin embargo, hay manipuladores predefinidos que probablemente complacerán tus necesidades sin hacer cualquier trabajo adicional:

StreamHandler	Escribe registros formateados a **OutputStream**
ConsoleHandler	Escribe registros formateados a **System.err**
FileHandler	Escribe registros formateados de registro ya sea para un archivo único, o para un conjunto de archivos alternables de registro
SocketHandler	Escribe registros formateados de registro para puertos remotos TCP
MemoryHandler	Los búferes registran registros en la memoria

Por ejemplo, a menudo quieres almacenar salida de registro de actividades a un archivo. El **FileHandler** facilita esto:

```
//: c15:LogToFile.java
// {Clean: LogToFile.xml,LogToFile.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class LogToFile {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("LogToFile");
  public static void main(String[] args) throws Exception {
    logger.addHandler(new FileHandler("LogToFile.xml"));
    logger.info("A message logged to the file");
    monitor.expect(new String[] {
      "%% .* LogToFile main",
      "INFO: A message logged to the file"
    });
  }
} ///:~
```

Cuando corres éste programa, notarás dos cosas. Primero, si bien enviamos la salida a un archivo, todavía verás salida de la consola. Eso es porque cada mensaje es convertido a un **LogRecord**, lo cual es primero usado por el objeto local del registrador, el cual lo pasa a sus propios manipuladores. En este punto el **LogRecord** es pasado al objeto padre, lo cual tiene a sus propios manipuladores. Este proceso continúa hasta que el registrador de la raíz sea alcanzado. El registrador de la raíz viene con un **ConsoleHandler** predeterminado, así es que el mensaje aparece en la pantalla también como aparece en el archivo de registro (puedes desactivar este comportamiento llamando a **setUseParentHandlers**(**false**)).

La segunda cosa que notarás es que el contenido del archivo de registro está en formato XML, lo cual verá algo así como esto:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2002-07-08T12:18:17</date>
  <millis>1026152297750</millis>
  <sequence>0</sequence>
  <logger>LogToFile</logger>
  <level>INFO</level>
  <class>LogToFile</class>
  <method>main</method>
  <thread>10</thread>
  <message>A message logged to the file</message>
</record>
</log>
```

El formato predeterminado de salida para un **FileHandler** es XML. Si quieres cambiar el formato, debes adjuntar a un objeto diferente **Formatter** al manipulador. Aquí, un **SimpleFormatter** sirve para que el archivo devuelva como formato simple de texto:

```
//: c15:LogToFile2.java
// {Clean: LogToFile2.txt,LogToFile2.txt.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class LogToFile2 {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("LogToFile2");
  public static void main(String[] args) throws Exception {
    FileHandler logFile= new FileHandler("LogToFile2.txt");
    logFile.setFormatter(new SimpleFormatter());
    logger.addHandler(logFile);
    logger.info("A message logged to the file");
    monitor.expect(new String[] {
      "%% .* LogToFile2 main",
      "INFO: A message logged to the file"
    });
  }
} ///:~
```

El archivo **LogToFile2.txt** se parecerá a esto:

```
Jul 8, 2002 12:35:17 PM LogToFile2 main
INFO: A message logged to the file
```

Manipuladores Múltiples

Puedes registrar a los manipuladores múltiples con cada objeto **Logger**. Cuando la petición de un registro de actividades llega al **Logger**, notifica a todos los manipuladores que han sido registrados con eso **[15]** , como el nivel de registro de actividades para el **Logger** es mayor o igual a eso de la petición de registro de actividades. Cada manipulador, a su vez, tiene su propio nivel de registro de actividades; Si el nivel del **LogRecord** es mayor o igual al nivel del manipulador, entonces ese manipulador publica el registro.

[15] Éste es el patrón del diseño del *Observador* (ibid).

Aquí está un ejemplo que agrega a un **FileHandler** y un **ConsoleHandler** al objeto **Logger**:

```
//: c15:MultipleHandlers.java
// {Clean: MultipleHandlers.xml,MultipleHandlers.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
```

```
public class MultipleHandlers {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("MultipleHandlers");
  public static void main(String[] args) throws Exception {
    FileHandler logFile =
      new FileHandler("MultipleHandlers.xml");
    logger.addHandler(logFile);
    logger.addHandler(new ConsoleHandler());
    logger.warning("Output to multiple handlers");
    monitor.expect(new String[] {
      "%% .* MultipleHandlers main",
      "WARNING: Output to multiple handlers",
      "%% .* MultipleHandlers main",
      "WARNING: Output to multiple handlers"
    });
  }
} ///:~
```

Cuando corres el programa, notarás que la salida de la consola ocurre dos veces; Eso es porque el comportamiento predeterminado del registrador raíz está todavía habilitado. Si quieres cerrar esto, haz una llamada a **setUseParentHandlers**(**false**):

```
//: c15:MultipleHandlers2.java
// {Clean: MultipleHandlers2.xml,MultipleHandlers2.xml.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class MultipleHandlers2 {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("MultipleHandlers2");
  public static void main(String[] args) throws Exception {
    FileHandler logFile =
      new FileHandler("MultipleHandlers2.xml");
    logger.addHandler(logFile);
    logger.addHandler(new ConsoleHandler());
    logger.setUseParentHandlers(false);
    logger.warning("Output to multiple handlers");
    monitor.expect(new String[] {
      "%% .* MultipleHandlers2 main",
      "WARNING: Output to multiple handlers"
    });
  }
} ///:~
```

Ahora verás sólo un mensaje de la consola.

Escribiendo tus Manipuladores

Fácilmente puedes escribir a manipuladores personalizados heredando de la

clase **Handler**. Para hacer esto, sólo no debes implementar al el método **publish**() (que realice la información real), sino que también **flush**() y **close**(), el cual asegura que el flujo usado para reportar es correctamente limpiado. Aquí está un ejemplo que almacena información del **LogRecord** en otro objeto (un **List** de **String**). Al final del programa, el objeto es impreso para la consola:

```
//: c15:CustomHandler.java
// How to write custom handler
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.util.*;

public class CustomHandler {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("CustomHandler");
  private static List strHolder = new ArrayList();
  public static void main(String[] args) {
    logger.addHandler(new Handler() {
      public void publish(LogRecord logRecord) {
        strHolder.add(logRecord.getLevel() + ":");
        strHolder.add(logRecord.getSourceClassName()+":");
        strHolder.add(logRecord.getSourceMethodName()+":");
        strHolder.add("<" + logRecord.getMessage() + ">");
        strHolder.add("\n");
      }
      public void flush() {}
      public void close() {}
    });
    logger.warning("Logging Warning");
    logger.info("Logging Info");
    System.out.print(strHolder);
    monitor.expect(new String[] {
      "%% .* CustomHandler main",
      "WARNING: Logging Warning",
      "%% .* CustomHandler main",
      "INFO: Logging Info",
      "[WARNING:, CustomHandler:, main:, " +
      "<Logging Warning>, ",
      ", INFO:, CustomHandler:, main:, <Logging Info>, ",
      "]"
    });
  }
} ///:~
```

La salida de la consola viene del registrador raíz. Cuando el **ArrayList** es impreso, puedes ver que sólo la información seleccionada ha sido capturada en el objeto.

Filtros

Cuando escribes el código para enviar un mensaje de registro de actividades a un objeto **Logger**, a menudo te decides, a la hora que escribes el código, qué nivel el mensaje de registro de actividades debería ser (la API de registro de actividades ciertamente te permite idear más sistemas complejos en donde el nivel del mensaje puede ser determinado dinámicamente, pero esto es menos común en la práctica). El objeto **Logger** tiene un nivel que puede estar colocado a fin de que puede decidir qué el nivel de mensaje aceptar; Todos los otros serán ignorados. Esto puede ser considerado como una funcionabilidad básica de filtrado, y son a menudo todo lo que necesitas.

Algunas veces, sin embargo, necesitas más filtrado sofisticado a fin de que puedas decidirte ya sea por aceptar o denegar un mensaje basado en algo más que simplemente el nivel actual. Para lograr esto puedes escribir los objetos personalizados **Filter**. **Filter** es una interfaz que tiene un método único, **boolean isLoggable**(**LogRecord record**), lo cual se decide si este **LogRecord** particular es lo suficientemente interesante para reportar o no.

Una vez que creas un Filtro, le registras con ya sea un **Logger** o un **Handler** usando el método **setFilter**(). Por ejemplo, supón que a ti te gustaría sólo registrar informes acerca de **Ducks**:

```
//: c15:SimpleFilter.java
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class SimpleFilter {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("SimpleFilter");
  static class Duck {};
  static class Wombat {};
  static void sendLogMessages() {
    logger.log(Level.WARNING,
      "A duck in the house!", new Duck());
    logger.log(Level.WARNING,
      "A Wombat at large!", new Wombat());
  }
  public static void main(String[] args) {
    sendLogMessages();
    logger.setFilter(new Filter() {
      public boolean isLoggable(LogRecord record) {
        Object[] params = record.getParameters();
        if(params == null)
          return true; // No parameters
        if(record.getParameters()[0] instanceof Duck)
          return true;  // Only log Ducks
        return false;
      }
    });
    logger.info("After setting filter..");
    sendLogMessages();
    monitor.expect(new String[] {
```

```
        "%% .* SimpleFilter sendLogMessages",
        "WARNING: A duck in the house!",
        "%% .* SimpleFilter sendLogMessages",
        "WARNING: A Wombat at large!",
        "%% .* SimpleFilter main",
        "INFO: After setting filter..",
        "%% .* SimpleFilter sendLogMessages",
        "WARNING: A duck in the house!"
      });
  }
} ///:~
```

Antes de colocar al **Filter**, los mensajes acerca de **Ducks** y **Wombats** son reportados. El **Filter** es creado como una clase interna anónima que considera el parámetro **LogRecord** para ver si un **Duck** fue pasado como un argumento adicional para el método **log**(). Si es así, retorna **true** para señalar que el mensaje debería ser procesado.

Note que la firma de **getParameters** dice que devolverá a un **Object**. Sin embargo, si ninguno de los argumentos adicionales han sido pasados al método **log**(), **getParameters**() retornarán nulos (en la violación de su firma – ésta es una mala práctica de programación). Así en vez de dado que un arreglo es devuelto (como prometido) y verificado para ver si es de longitud cero, debemos inspeccionar para nulo. Si no haces esto correctamente, entonces la llamada para **logger.info**() causará que una excepción sea lanzado.

Formateadores

Un **Formatter** es una forma de insertar una operación de formateo en las fases de elaboración de un Manipulador. Si registras un objeto **Formatter** con un **Handler**, entonces antes de que el **LogRecord** sea publicado por el **Handler**, es enviado primero al **Formatter**. Después de formatear, el **LogRecord** es devuelto al **Handler**, lo cual luego lo publica.

Para escribir un **Formatter** personalizado, extiendes la clase **Formatter** y sobrescribe a **format**(**LogRecord record**). Luego, registra al **Formatter** con el **Handler** usando la llamada **setFormatter**(), como lo verás aquí:

```
//: c15:SimpleFormatterExample.java
import com.bruceeckel.simpletest.*;
import java.util.logging.*;
import java.util.*;

public class SimpleFormatterExample {
  private static Test monitor = new Test();
  private static Logger logger =
    Logger.getLogger("SimpleFormatterExample");
  private static void logMessages() {
    logger.info("Line One");
    logger.info("Line Two");
```

```
  }
  public static void main(String[] args) {
    logger.setUseParentHandlers(false);
    Handler conHdlr = new ConsoleHandler();
    conHdlr.setFormatter(new Formatter() {
      public String format(LogRecord record) {
        return record.getLevel()  + "  :  "
          + record.getSourceClassName()  + " -:- "
          + record.getSourceMethodName()  + " -:- "
          + record.getMessage() + "\n";
      }
    });
    logger.addHandler(conHdlr);
    logMessages();
    monitor.expect(new String[] {
      "INFO  :  SimpleFormatterExample -:- logMessages "
        + "-:- Line One",
      "INFO  :  SimpleFormatterExample -:- logMessages "
        + "-:- Line Two"
    });
  }
} ///:~
```

Recuerdo que un registrador como **myLogger** tiene a un manipulador predeterminado que pone del registrador del padre (el registrador raíz, en este caso). Aquí, desactivamos al manipulador predeterminado llamando a **setUseParentHandlers**(**false**), y luego incluimos un manipulador de la consola para usarlo a cambio. El nuevo **Formatter** es creado como una clase interna anónima en la declaración **setFormatter**(). La declaración sobrescrita **format**() simplemente extrae una cierta cantidad de la información del **LogRecord** y la formatea en un **string**.

Ejemplo: Enviando e-mail para reportar mensajes de registro

Realmente puedes tener uno de tus manipuladores de registro de actividades para enviar un email a fin de que puedas estar automáticamente notificado de problemas importantes. El siguiente ejemplo usa la *API JavaMail* para desarrollar un agente de usuario del correo para enviar un correo electrónico.

El *API JavaMail* es un conjunto de clases que interactúa para el protocolo subyacente de envío por correo (*IMAP*, *POP*, *SMTP*). Puedes idear un mecanismo de notificación en alguna condición excepcional en el código ejecutable registrando a un **Handler** adicional para enviar un email.

```
//: c15:EmailLogger.java
// {RunByHand} Must be connected to the Internet
// {Depends: mail.jar,activation.jar}
import java.util.logging.*;
import java.io.*;
import java.util.Properties;
```

```
import javax.mail.*;
import javax.mail.internet.*;

public class EmailLogger {
  private static Logger logger =
    Logger.getLogger("EmailLogger");
  public static void main(String[] args) throws Exception {
    logger.setUseParentHandlers(false);
    Handler conHdlr = new ConsoleHandler();
    conHdlr.setFormatter(new Formatter() {
      public String format(LogRecord record) {
        return record.getLevel() + "  :  "
          + record.getSourceClassName() + ":"
          + record.getSourceMethodName() + ":"
          + record.getMessage() + "\n";
      }
    });
    logger.addHandler(conHdlr);
    logger.addHandler(
      new FileHandler("EmailLoggerOutput.xml"));
    logger.addHandler(new MailingHandler());
    logger.log(Level.INFO,
      "Testing Multiple Handlers", "SendMailTrue");
  }
}

// A handler that sends mail messages
class MailingHandler extends Handler {
  public void publish(LogRecord record) {
    Object[] params = record.getParameters();
    if(params == null) return;
    // Send mail only if the parameter is true
    if(params[0].equals("SendMailTrue")) {
      new MailInfo("bruce@theunixman.com",
        new String[] { "bruce@theunixman.com" },
        "smtp.theunixman.com", "Test Subject",
        "Test Content").sendMail();
    }
  }
  public void close() {}
  public void flush() {}
}

class MailInfo {
  private String fromAddr;
  private String[] toAddr;
  private String serverAddr;
  private String subject;
  private String message;
  public MailInfo(String from, String[] to,
    String server, String subject, String message) {
    fromAddr = from;
    toAddr = to;
```

```
    serverAddr = server;
    this.subject = subject;
    this.message = message;
  }
  public void sendMail() {
    try {
      Properties prop = new Properties();
      prop.put("mail.smtp.host", serverAddr);
      Session session =
        Session.getDefaultInstance(prop, null);
      session.setDebug(true);
      // Create a message
      Message mimeMsg = new MimeMessage(session);
      // Set the from and to address
      Address addressFrom = new InternetAddress(fromAddr);
      mimeMsg.setFrom(addressFrom);
      Address[] to = new InternetAddress[toAddr.length];
      for(int i = 0; i < toAddr.length; i++)
        to[i] = new InternetAddress(toAddr[i]);
      mimeMsg.setRecipients(Message.RecipientType.TO,to);
      mimeMsg.setSubject(subject);
      mimeMsg.setText(message);
      Transport.send(mimeMsg);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

MailingHandler es uno de los **Handlers** registrados con el registrador. Para enviar un email, el **MailingHandler** usa el objeto **MailInfo**. Cuando un mensaje de registro de actividades es enviado con un parámetro adicional de "**SendMailTrue**", el **MailingHandler** envía un email.

El objeto **MailInfo** contiene la información necesaria del estado, tal como dirección para, dirección de, y la información de asunto requerida para enviar un correo electrónico. Esta información de estado es provista al objeto **MailInfo** a través del constructor cuando es instanciado.

Para enviar un email primero debes establecer un **Session** con el servidor *Simple Mail Transfer Protocol* (*SMTP*). Esto se hace pasando la dirección del servidor dentro de un objeto **Properties**, en una propiedad llamado **mail.smtp.host**. Estableces una sesión llamando a **Session.getDefaultInstance**(), pasándole el objeto **Properties** como el primer argumento. El segundo argumento es una instancia de **Authenticator** que puede servir para autenticar el usuario. Pasar un valor nulo al argumento **Authenticator** no especifica autenticación. Si la bandera de depuración en el objeto **Properties** está colocada, la información estimando la comunicación entre el servidor SMTP y el programa será impresa.

MimeMessage es una abstracción de un mensaje de e-mail de la Internet que extiende la clase **Message**. Construye un mensaje que cumple con el formato

MIME (*Extensiones de Correo de Internet Multiuso*). Un **MimeMessage** se construye pasándole una instancia de **Session**. Puedes establecer las direcciones de y para creando una instancia de clase **InternetAddress** (una subclase de **Address**). Envías el mensaje usando la llamada estática **Transport.send**() de la clase abstracta **Transport**. Una implementación de **Transport** usa un protocolo específico (generalmente *SMTP*) para comunicar con el servidor para enviar el mensaje.

Controlando Niveles de Registro de Actividades a través de Namespaces

Aunque no es obligatorio, se aconseja darle a un registrador el nombre de la clase en la cual es usado. Esto te permite manipular el nivel de registro de actividades de grupos de registradores que radican en la misma jerarquía del paquete, en la granularidad de la estructura del paquete del directorio. Por ejemplo, puedes modificar todos los niveles de registro de actividades de todos los paquetes en **com**, o simplemente los que están en **com.bruceeckel**, o simplemente los que están en **com.bruceeckel.util**, como se muestra en el siguiente ejemplo:

```
//: c15:LoggingLevelManipulation.java
import com.bruceeckel.simpletest.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.Handler;
import java.util.logging.LogManager;

public class LoggingLevelManipulation {
  private static Test monitor = new Test();
  private static Logger
    lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
  static void printLogMessages(Logger logger) {
    logger.finest(logger.getName() + " Finest");
    logger.finer(logger.getName() + " Finer");
    logger.fine(logger.getName() + " Fine");
    logger.config(logger.getName() + " Config");
    logger.info(logger.getName() + " Info");
    logger.warning(logger.getName() + " Warning");
    logger.severe(logger.getName() + " Severe");
  }
  static void logMessages() {
    printLogMessages(lgr);
    printLogMessages(lgr2);
    printLogMessages(util);
    printLogMessages(test);
    printLogMessages(rand);
  }
```

```
  static void printLevels() {
    System.out.println(" -- printing levels -- "
      + lgr.getName() + " : " + lgr.getLevel()
      + " " + lgr2.getName() + " : " + lgr2.getLevel()
      + " " + util.getName() + " : " + util.getLevel()
      + " " + test.getName() + " : " + test.getLevel()
      + " " + rand.getName() + " : " + rand.getLevel());
  }
  public static void main(String[] args) {
    printLevels();
    lgr.setLevel(Level.SEVERE);
    printLevels();
    System.out.println("com level: SEVERE");
    logMessages();
    util.setLevel(Level.FINEST);
    test.setLevel(Level.FINEST);
    rand.setLevel(Level.FINEST);
    printLevels();
    System.out.println(
      "individual loggers set to FINEST");
    logMessages();
    lgr.setLevel(Level.FINEST);
    printLevels();
    System.out.println("com level: FINEST");
    logMessages();
    monitor.expect("LoggingLevelManipulation.out");
  }
} ///:~
```

Como puedes ver en este código, si le pasas a **getLogger**() un **string** representando a un **namespace**, el **Logger** resultante controlará los niveles de severidad de ese **namespace**; Es decir, todo los paquetes dentro de ese **namespace** serán afectados por cambios para el nivel de severidad del registrador.

Cada **Logger** mantiene una pista de su predecesor existente **Logger**. Si un registrador hijo ya tiene un nivel de registro de actividades determinado, entonces ese nivel es usado en lugar del nivel de registro de actividades del padre. Cambiar el nivel de registro de actividades del padre no afecta el nivel de registro de actividades hijos una vez que el hijo tiene su propio nivel de registro de actividades.

Aunque el nivel de registradores individuales está colocado para **FINEST**, sólo los mensajes con un nivel de registro de actividades igual o más severos que **INFO** son impresos porque usamos al **ConsoleHandler** del registrador de la raíz, lo cual está en **INFO**.

Porque no está en el mismo **namespace**, el nivel de registro de actividades al azar permanece no afectado cuando el nivel de registro de actividades del registrador **com** o **com.bruceeckel**se varía.

Registrando Prácticas para Proyectos Grandes

A primera vista, la API de registro de actividades Java puede parecer bastante sobre-diseñada para la mayoría de problemas de programación. Las características adicionales y las habilidades no vienen bien hasta que empieces a construir proyectos más grandes. En esta sección veremos estas características y formas recomendadas para usarlos. Si sólo estás usando registro de actividades en proyectos más pequeños, probablemente no necesitarás usar estas características.

Archivos de configuración

El siguiente archivo muestra cómo puedes configurar registradores en un proyecto usando un archivo de propiedades:

```
//:! c15:log.prop
#### Configuration File ####
# Global Params
# Handlers installed for the root logger
handlers= java.util.logging.ConsoleHandler
java.util.logging.FileHandler
# Level for root logger—is used by any logger
# that does not have its level set
.level= FINEST
# Initialization class—the public default constructor
# of this class is called by the Logging framework
config = ConfigureLogging

# Configure FileHandler
# Logging file name  - %u specifies unique
java.util.logging.FileHandler.pattern = java%g.log
# Write 100000 bytes before rotating this file
java.util.logging.FileHandler.limit = 100000
# Number of rotating files to be used
java.util.logging.FileHandler.count = 3
# Formatter to be used with this FileHandler
java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter

# Configure ConsoleHandler
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter

# Set Logger Levels #
com.level=SEVERE
com.bruceeckel.level = FINEST
com.bruceeckel.util.level = INFO
com.bruceeckel.test.level = FINER
random.level= SEVERE
///:~
```

El archivo de configuración te permite asociar a los manipuladores con el registrador raíz. Los manipuladores de la propiedad especifican la lista

separada en coma de manipuladores que tienes el deseo de registrar con el registrador raíz. Aquí, registramos al **FileHandler** y el **ConsoleHandler** con el registrador raíz. La propiedad **.level** especifica el nivel predeterminado para el registrador. Este nivel es usado por todos los registradores que son hijos del registrador raíz y no tienen su propio nivel especificado. Note que, sin un archivo de propiedades, el nivel predeterminado de registro de actividades del registrador de la raíz es **INFO**. Esto es porque, en la ausencia de un archivo de configuración personalizado, la máquina virtual usa la configuración del archivo **JAVA_HOME\jre\lib\logging.properties**.

Rotando archivos de registro

El archivo de configuración anterior genera archivos rotativos de registro, que se usan para impedir cualquier archivo de registro de volverse demasiado grande. Estableciendo el valor **FileHandler.limit**, das el máximo número de bytes permitidos en un archivo de registro antes de que el siguiente comience a llenarse. **FileHandler.count** determina el número de archivos rotativos de registro a usar; el archivo de configuración mostrado aquí especifica tres archivos. Si los tres archivos están llenos a su máximo, entonces el primer archivo comienza a llenarse otra vez, sobrescribiendo el viejo contenido.

Alternativamente, toda la salida puede ser metida en un único archivo dando un valor **FileHandler.count** de uno. (Los parámetros **FileHandler** están explicados en detalle en la documentación JDK).

Para que el siguiente programa use el archivo de configuración anterior, debes especificar el parámetro **java.util.logging.config.file** en la línea de comando:

```
java -Djava.util.logging.config.file=log.prop ConfigureLogging
```

El archivo de configuración sólo puede modificar el registrador raíz. Si quieres agregar los filtros y manipuladores a otros registradores, debes escribir el código para hacerlo dentro de un archivo Java, como se notó en el constructor:

```
//: c15:ConfigureLogging.java
// {JVMArgs: -Djava.util.logging.config.file=log.prop}
// {Clean: java0.log,java0.log.lck}
import com.bruceeckel.simpletest.*;
import java.util.logging.*;

public class ConfigureLogging {
  private static Test monitor = new Test();
  static Logger lgr = Logger.getLogger("com"),
    lgr2 = Logger.getLogger("com.bruceeckel"),
    util = Logger.getLogger("com.bruceeckel.util"),
    test = Logger.getLogger("com.bruceeckel.test"),
    rand = Logger.getLogger("random");
  public ConfigureLogging() {
    /* Set Additional formatters, Filters and Handlers for
       the loggers here. You cannot specify the Handlers
       for loggers except the root logger from the
```

```
      configuration file. */
  }
  public static void main(String[] args) {
    sendLogMessages(lgr);
    sendLogMessages(lgr2);
    sendLogMessages(util);
    sendLogMessages(test);
    sendLogMessages(rand);
    monitor.expect("ConfigureLogging.out");
  }
  private static void sendLogMessages(Logger logger) {
    System.out.println(" Logger Name : "
      + logger.getName() + " Level: " + logger.getLevel());
    logger.finest("Finest");
    logger.finer("Finer");
    logger.fine("Fine");
    logger.config("Config");
    logger.info("Info");
    logger.warning("Warning");
    logger.severe("Severe");
  }
} ///:~
```

La configuración dará como resultado la salida siendo enviada a los archivos llamado **java0.log**, **java1.log**, y **java2.log** en el directorio del cual este programa es ejecutado.

Prácticas sugeridas

Aunque no está obligatorio, generalmente deberías considerar destinar un registrador para cada clase, siguiendo el estándar de establecer el nombre del registrador para ser igual que el nombre completamente calificado de la clase. Como se mostró anteriormente, esto tiene en cuenta el control más fino granulado de registro de actividades por la habilidad de habilitar y deshabilitar registros basado en **namespaces**.

Si no incrustas el nivel de registro de actividades para las clases individuales en ese paquete, entonces las clases individuales predeterminadas para el nivel de registro de actividades determinado para el paquete (dado que nombras los registradores según su paquete y la clase).

Si controlas el nivel de registro de actividades en un archivo de configuración en lugar de cambiarlo dinámicamente en tu código, entonces puedes modificar niveles de registro de actividades sin recompilar tu código. La recompilación no es siempre una opción cuando el sistema es desplegado; A menudo, sólo los archivos **class** son enviados al ambiente de destino.

Algunas veces hay un requisito para ejecutar algún código para realizar actividades de inicialización como agregar **Handlers**, **Filters**, y **Formatters** para registradores. Esto puede ser logrado incrustando la propiedad **config** en el archivo de propiedades. Puedes tener clases múltiples cuya inicialización puede hacerse usando la propiedad **config**. Estas clases deberían ser

especificadas usando valores delimitados en espacio como éste:

```
config = ConfigureLogging1 ConfigureLogging2 Bar Baz
```

Las clases especificadas en esta moda invocarán a sus constructores predeterminados.

Resumen

Aunque ésta ha sido una introducción medianamente minuciosa para la API de registro de actividades, no incluye todo. Por ejemplo, no hemos hablado del **LogManager** o detalles de los manipuladores incorporados diversos, algo semejante como **MemoryHandler**, **FileHandler**, **ConsoleHandler**, etc. Deberías ir a la documentación JDK para más detalles.

Depuración

Aunque el uso juicioso de declaraciones **System.out** o información de registro de actividades puede producir un entendimiento valioso en el comportamiento de un programa, **[16]** para los problemas difíciles este acercamiento se pone difícil y consume tiempo. Además, puedes necesitar asomarte más profundamente en el programa que las declaraciones **print** permitirán. Para esto, necesitas un depurador.

[16] Aprendí C++ primordialmente imprimiendo información, ya que en el momento aprendí que no había depuradores disponibles.

Además de desplegar información más rápidamente y fácilmente que este podría producir con declaraciones **print**, un depurador también establecerá puntos de ruptura y luego detendrá el programa cuando alcance esos puntos de ruptura. Un depurador también puede desplegar la condición del programa en cualquier instante, puedes ver los valores de variables en los que estás interesado, dan un paso a través del programa línea por línea, se conectan a un programa remotamente ejecutable, y más. Especialmente cuando empiezas a construir sistemas más grandes (donde los problemas fácilmente pueden volverse sepultados), conviene familiarizarse con depuradores.

Depuración con JDB

El *Java Debugger* (*JDB*) es un depurador de línea de comando que se embarca con el JDK. JDB es por lo menos conceptualmente un descendiente del *Depurador Gnu* (*GDB*, el cual estaba inspirado por el original *Unix DB*), en términos de las instrucciones para depurar y su interfaz de línea de comando. JDB es muy apropiado para aprender acerca de depurar y realizar tareas más simples de corrección de errores, y es de ayuda para saber que está todo el tiempo disponible dondequiera que el JDK esté instalado. Sin embargo, para proyectos más grandes probablemente querrás usar un depurador gráfico, descrito más adelante.

Supón que has escrito el siguiente programa:

```
//: c15:SimpleDebugging.java
// {ThrowsException}
public class SimpleDebugging {
  private static void foo1() {
    System.out.println("In foo1");
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    int j = 1;
    j--;
    int i = 5 / j;
  }
  public static void main(String[] args) {
    foo1();
  }
} ///:~
```

Si miras a **foo3**(), el problema es obvio; divides por cero. Pero supón que este código es enterrado en un programa extenso (como se sobreentiende aquí por la secuencia de llamadas) y no sabes donde ponerte a buscar el problema. Como resulta, la excepción que será lanzada dará bastante información para que halles el problema (éste es simplemente una de las grandes cosas acerca de excepciones). Pero déjanos solamente suponer que el problema es más difícil que eso, y que necesitas ejercitar en él más profundamente y obtener más información de lo que una excepción provee.

Para correr JDB, debes decir al compilador que genere información de depuración compilando a **SimpleDebugging.java** con la bandera **-g**. Luego comienzas a depurar el programa con la línea de comando:

```
jdb SimpleDebugging
```

Esto levanta JDB y te da un indicador de comando. Puedes mirar la lista de órdenes disponibles JDB escribiendo **'?'** en el indicador.

Aquí hay un rastro interactivo de depuración que demuestra cómo encontrar un problema:

```
Initializing jdb ...
> catch Exception
```

Le indica que *JDB* está esperando una orden, y las órdenes introducidas por el usuario son demostradas en negrita. La orden **catch Exception** causa que un punto de ruptura sea determinado en cualquier punto donde una excepción es lanzada (sin embargo, el depurador se detendrá de cualquier manera, aun si

explícitamente no das este comentario – las excepciones parecen ser puntos de ruptura predeterminados en *JDB*).

```
Deferring exception catch Exception.
It will be set after the class is loaded.
> run
```

Ahora el programa correrá hasta el siguiente punto de ruptura, lo cual en este caso está donde la excepción ocurre. Aquí está el resultado de la orden **run**:

```
run SimpleDebugging
>
VM Started: In foo1
In foo2
In foo3
Exception occurred: java.lang.ArithmeticException
(uncaught)"thread=main", SimpleDebugging.foo3(), line=18 bci=15
18          int i = 5 / j;
```

El programa corre hasta la línea 18, donde generó la excepción, pero *JDB* no sale cuando teclea la excepción. El depurador también despliega la línea de código que causó la excepción. Puedes listar el punto donde la ejecución se detuvo en el programa fuente por la orden de lista como se mostró aquí:

```
main[1] list
14      private static void foo3() {
15        System.out.println("In foo3");
16        int j = 1;
17        j--;
18 =>     int i = 5 / j;
19      }
20
21      public static void main(String[] args) {
22        foo1();
23      }
```

El puntero ("=>") en este listado muestra el punto actual de donde la ejecución reanudará. Podrías reanudar la ejecución por la orden **cont** (continua). Pero hacer eso hará a JDB salir en la excepción, imprimiendo el rastro de la pila.

La orden **locals** descarga el valor de todas las variables locales:

```
main[1] locals
Method arguments:
Local variables:
j = 0
```

Puedes ver que el valor de j=0 es lo que causó la excepción.

La orden **wherei** imprime los almacenamientos de pila empujados en el método de pila del hilo actual:

```
main[1] wherei
[1] SimpleDebugging.foo3 (SimpleDebugging.java:18), pc = 15
[2] SimpleDebugging.foo2 (SimpleDebugging.java:11), pc = 8
[3] SimpleDebugging.foo1 (SimpleDebugging.java:6), pc = 8
[4] SimpleDebugging.main (SimpleDebugging.java:22), pc = 0
```

Cada línea después de la orden del **wherei** representa una llamada de método y el punto donde la llamada regresará (el cual es mostrado por el valor del contador de programa **pc**). Aquí la secuencia de llamada fue **main**(), **foo1**(), **foo2**(), y **foo3**(). Puedes hacer estallar el almacenamiento de la pila empujada cuando la llamada estaba hecha para **foo3**() con la orden **pop**:

```
main[1] pop
main[1] wherei
[1] SimpleDebugging.foo2 (SimpleDebugging.java:11), pc = 8
[2] SimpleDebugging.foo1 (SimpleDebugging.java:6), pc = 8
[3] SimpleDebugging.main (SimpleDebugging.java:22), pc = 0
```

Puedes hacer al JDB dar un paso directo la llamada para **foo3**() otra vez con la orden **reenter**:

```
main[1] reenter
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=15
bci=0
System.out.println("In foo3");
```

La orden **list** nos muestra que la ejecución comienza en el principio de **foo3**():

```
main[1] list
11          foo3();
12        }
13
14        private static void foo3() {
15 =>       System.out.println("In foo3");
16          int j = 1;
17          j--;
18          int i = 5 / j;
19        }
20
```

JDB también te permite modificar el valor de las variables locales. La división por cero que se produjo ejecutando este pedazo de código la última vez puede ser evitado cambiando el valor de **j**. Puedes hacer esto directamente en el depurador, así es que puedes continuar depurando el programa sin vuelta atrás y cambiando el archivo fuente. Antes de que establezcas el valor de **j**,

tendrás que ejecutar a través de la línea 25 ya que es donde **j** está declarada.

```
main[1] step
> In foo3

Step completed: "thread=main", SimpleDebugging.foo3(), line=16 bci=8
16        int j = 1;

main[1] step
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=17 bci=10
17        j--;

main[1] list
13
14      private static void foo3() {
15        System.out.println("In foo3");
16        int j = 1;
17 =>     j--;
18        int i = 5 / j;
19      }
20
21      public static void main(String[] args) {
22        foo1();
```

En este punto, **j** es definida y puedes establecer su valor a fin de que la excepción pueda ser evitada.

```
main[1] set j=6
 j=6 = 6
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=18
bci=13
18         int i = 5 / j;
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo3(), line=19
bci=17
19      }
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo2(), line=12
bci=11
12      }
main[1] list
8
9       private static void foo2() {
10        System.out.println("In foo2");
11        foo3();
12 =>   }
13
14      private static void foo3() {
15        System.out.println("In foo3");
```

```
16        int j = 1;
17        j--;
main[1] next
>
Step completed: "thread=main", SimpleDebugging.foo1(), line=7
bci=11
7      }
main[1] list
3    public class SimpleDebugging {
4      private static void foo1() {
5        System.out.println("In foo1");
6        foo2();
7 =>   }
8
9      private static void foo2() {
10       System.out.println("In foo2");
11       foo3();
12     }
main[1] next
>
Step completed: "thread=main", SimpleDebugging.main(), line=23
bci=3
23     }

main[1] list
19     }
20
21     public static void main(String[] args) {
22       foo1();
23 =>  }
24   } ///:~
main[1] next
>
The application exited
```

next ejecuta una línea a la vez. Puedes ver que la excepción es evitada y podemos continuar dando un paso a través del programa. **list** está acostumbrado a mostrar la posición en el programa donde la ejecución procederá.

Depuradores gráficos

Usar un depurador de línea de comando como JDB puede ser inadecuado. Debes usar órdenes explícitas para hacer cosas como mirar al estado de las variables (locales, descargados), listando el punto de ejecución en el código de la fuente (la lista), encontrando los hilos en el sistema (los hilos), estableciendo puntos de ruptura (detente en, detente hasta), etc. Un depurador gráfico te permite hacer todas estas cosas con algunos clics y también ver los últimos detalles de programa siendo depurado sin usar órdenes explícitas.

Así, aunque puedes querer comenzar experimentando con JDB, probablemente

lo encontrarás bastante más productivo aprender a usar un depurador gráfico para rápidamente seguirles la pista a tus bichos. Durante el desarrollo de esta edición de este libro, empezamos a usar el entorno de edición y de desarrollo IBM Eclipse, lo cual contiene un muy buen depurador gráfico para Java. Eclipse está bien diseñado e implementado, y puedes hacer un download de eso gratis de www.Eclipse.org (ésta es una herramienta gratis, no un demo o software de libre evaluación – gracias a IBM para invertir el dinero, el tiempo, y el esfuerzo para hacer esto disponible para todo el mundo).

Otras herramientas gratis de desarrollo tienen depuradores gráficos igualmente, como Netbeans de Sun y la versión libre del JBuilder de Borland.

Perfilando y optimizando

"Deberíamos olvidarnos de eficiencias pequeñas, deberíamos decir acerca del 97 % del tiempo: La optimización prematura es la raíz de todo mal."–Donald Knuth

Aunque siempre deberías recordar esta cita, especialmente cuando te descubres a ti mismo en la cuesta resbaladiza de optimización prematura, algunas veces necesitas determinar donde tu programa gasta todo su tiempo, para ver si puedes mejorar el desempeño de esas secciones.

Un perfilador recoge información que te permite ver cuáles partes del programa consumen memoria y cuáles métodos consumes el tiempo máximo. Algunos perfiladores aun te permiten a desactivar el recolector de basuras para ayudar a determinar patrones de asignación de memoria.

Un perfilador también puede ser una herramienta útil en detectar hilos muertos en tu programa.

Rastreando consumo de memoria

Aquí está el tipo de información que un perfilador puede mostrar para el uso de memoria:

- El número de asignaciones del objeto para un tipo específico.
- Lugares donde las asignaciones del objeto toman lugar.
- Métodos involucrados en asignación de instancias de esta clase.
- Objetos vagabundos: Los objetos que no son ubicados, usados, y no está recolectada en la basura. Estos se mantienen aumentando el tamaño del montón de JVM y representan fugas de memoria, lo cual puede causar un error de memoria apagada o unos costos operativos excesivos en el recolector de basuras.
- La asignación excesiva de objetos temporales que aumentan el trabajo del colector de basuras y así reduce el desempeño de la aplicación.
- El fracaso a liberar instancias añadidas a una colección y no es removida (ésta es una causa especial de objetos vagabundos).

Rastreando uso de la CPU

Los perfiladores también siguen la pista de cuánto tiempo el CPU gasta en partes diversas de tu código. Te pueden decir:

- El número de veces que un método fue invocado.
- El porcentaje de tiempo CPU utilizado por cada método. Si este método llama otros métodos, el perfilador te puede decir la cantidad de tiempo transcurrido en estos otros métodos.
- Totaliza tiempo transcurrido absoluto por cada método, incluyendo el tiempo espera de E/S, bloqueos, etc. Esta vez depende de los recursos disponibles del sistema.

Por aquí puedes decidirte que secciones de tu código necesitan optimización.

Pruebas de cobertura

La prueba de cobertura muestra las líneas de código que se ejecutó durante la prueba. Esto puede extraer tu atención al código que no es usado y por consiguiente podría ser un candidato para la remoción o redescomposición.

Para obtener información de prueba de cobertura para **SimpleDebugging.java**, usas la orden:

```
java –Xrunjcov:type=M SimpleDebugging
```

Como una prueba, pone las líneas de código que no será ejecutado en **SimpleDebugging.java** (tendrás que estar algo listo acerca de esto ya que el compilador puede detectar líneas inalcanzables de código).

Interfaz de Perfilado JVM

El agente perfilador comunica los eventos en los que está interesado para el JVM. El interfaz de perfil JVM soporta los siguientes eventos:

- Entrar y salir un método
- Asignar, mover, y liberar un objeto
- Crear y suprimir una arena del montón
- Comienza y finaliza un ciclo de recolección de basura
- Asigna y libera una referencia global JNI
- Ubica y libera una referencia global débil JNI
- Carga y descarga un método compilado
- Comienza y termina un flujo de ejecución
- Los datos del archivo de clase preparan para la instrumentación
- Carga y descarga una clase
- Para un monitor Java bajo argumentación: Espera a Entrar, entrado, y salida
- Para un monitor crudo bajo la argumentación: Espera a Entrar, entrado, y salida
- Para un monitor no contendido Java: Espera y esperado
- Descarga del Monitor

- Descarga del Montón
- Descarga del Objeto
- Solicita descargar o reanudar perfilado de datos
- La inicialización JVM y el cierre

Al perfilar, el JVM envía estos eventos al agente perfilador, lo cual luego traslada la información deseada a la sección de entrada de perfilador, el cual puede ser un proceso funcionando con otra máquina, si se desea.

Usando HPROF

El ejemplo en esta sección demuestra cómo puedes correr el perfilador que se embarca con el JDK. Aunque la información de este perfilador está en la forma algo cruda de archivos del texto en vez de la representación gráfica que la mayoría de perfiladores comerciales producen, todavía provee ayuda valiosa en determinar las características de tu programa.

Corres el perfilador pasándole un argumento adicional al JVM cuando invocas el programa. Este argumento debe ser un único **string**, sin tener espacios después de las comas, como éste (aunque debería estar en una sola línea, se ha enrollado en el libro):

```
java –Xrunhprof:heap=sites, cpu=samples, depth=10, monitor=y,
thread=y, doe=y ListPerformance
```

- El **heap=sites** pide que el perfilador escriba información acerca de la utilización de memoria en el montón, indicando dónde fue ubicado.
- **cpu=samples** pide que el perfilador haga muestreo estadístico para determinar el uso de la CPU.
- **depth=10** indica la profundidad del rastro para hilos.
- **thread=y** pide que el perfilador identifique los hilos en las huellas de la pila.
- **doe=y** pide que el perfilador produzca descarga de perfilado de datos en la salida.

El siguiente listado contiene sólo una porción del archivo producido por HPROF. El archivo de salida es creado en el directorio actual y es llamado **java.hprof.txt**.

El comienzo de **java.hprof.txt** describe los detalles de las secciones restantes en el archivo. Los datos producidos por el perfilador están en secciones diferentes; por ejemplo, **TRACE** representa una sección del rastro en el archivo. Verás muchas secciones **TRACE**, cada uno numerado a fin de que puedan ser referenciadas más tarde.

La sección **SITES** muestra sitios de asignación de memoria. La sección tiene varias filas, ordenadas por el número de bytes que son ubicados y están siendo referenciadas – los bytes en vivo. La memoria se encuentra enumerada en bytes. El ego de la columna representa el porcentaje de memoria tomada por este sitio, la siguiente columna, **accum**, representa el porcentaje acumulativo de memoria. Las columnas **live bytes** y **live objects** representan el número

de bytes en vivo en este sitio y el número de objetos que fueron creados que consume estos bytes. Los **allocated bytes** y **objects** representan el número total de objetos y los bytes que son instanciados, incluyendo los únicos siendo usados y los que no son usados. La diferencia en el número de bytes listados en ubicados y vivo representan los bytes que pueden ser basura recolectada. La columna **trace** realmente establece referencias para un **TRACE** en el archivo. La primera fila establece referencias para trace 668 como se muestra debajo. El **name** representa la clase cuya instancia fue creada.

```
SITES BEGIN (ordered by live bytes) Thu Jul 18 11:23:06 2002
          percent          live       alloc'ed  stack class
 rank   self  accum    bytes objs   bytes objs trace name
    1 59.10% 59.10%   573488    3  573488    3   668 java.lang.Object
    2  7.41% 66.50%    71880  543   72624  559     1 [C
    3  7.39% 73.89%    71728    3   82000   10   649 java.lang.Object
    4  5.14% 79.03%    49896  232   49896  232     1 [B
    5  2.53% 81.57%    24592  310   24592  310     1 [S

TRACE 668: (thread=1)
       java.util.Vector.ensureCapacityHelper(Vector.java:222)
       java.util.Vector.insertElementAt(Vector.java:564)
       java.util.Vector.add(Vector.java:779)
       java.util.AbstractList$ListItr.add(AbstractList.java:495)
       ListPerformance$3.test(ListPerformance.java:40)
       ListPerformance.test(ListPerformance.java:63)
       ListPerformance.main(ListPerformance.java:93)
```

Este trace demuestra la secuencia de llamada de método que ubica la memoria. Si pasas a través del rastro como indicado por los números de la línea, te encontrarás con que una asignación del objeto toma lugar en la línea número 222 de **Vector.java**:

```
elementData = new Object[newCapacity];
```

Esto te ayuda a descubrir partes del programa que usa arriba de cantidades significativas de memoria (59.10 %, en este caso).

Note el **[C** en **SITE 1** representa el tipo primitivo **char**. Ésta es la representación interna del JVM para los tipos primitivos.

Desempeño del hilo

La sección **CPU SAMPLES** demuestra la utilización de la CPU. Aquí es en parte de un rastro de esta sección.

```
SITES END
CPU SAMPLES BEGIN (total = 514) Thu Jul 18 11:23:06 2002
rank   self  accum   count trace method
   1 28.21% 28.21%     145   662 java.util.AbstractList.iterator
   2 12.06% 40.27%      62   589 java.util.AbstractList.iterator
   3 10.12% 50.39%      52   632 java.util.LinkedList.listIterator
```

```
   4  7.00% 57.39%      36   231 java.io.FileInputStream.open
   5  5.64% 63.04%      29   605 ListPerformance$4.test
   6  3.70% 66.73%      19   636 java.util.LinkedList.addBefore
```

La organización de este listado es similar a la organización de los listados **SITES**. Las filas son ordenadas por la utilización de la UPC. La fila en la parte superior tiene la máxima utilización de la UPC, como indicada en la columna **self**. La columna **accum** lista la utilización acumulativa de la UPC. El campo **count** especifica que el número de por esta huella fue activo. Las siguientes dos columnas especifican el número de la huella y el método que tomó esta vez.

Considera la primera fila de la sección **CPU SAMPLES**. 28.12% de tiempo total CPU fue utilizado en el método **java.util.AbstractList.iterator**(), y fue designada 145 veces. Los detalles de esta llamada pueden verse mirando el rastro número 662:

```
TRACE 662: (thread=1)
      java.util.AbstractList.iterator(AbstractList.java:332)
      ListPerformance$2.test(ListPerformance.java:28)
      ListPerformance.test(ListPerformance.java:63)
      ListPerformance.main(ListPerformance.java:93)
```

Puedes deducir que iterar a través de una lista toma una cantidad significativa de tiempo.

Para proyectos grandes que es a menudo más útil tener la información representada en forma gráfica. Un número de perfiladores produce despliegues gráficos, pero la cobertura de estos está más allá del alcance de este libro.

Directivas de optimización

- Evita sacrificar legibilidad de código para el desempeño.
- El desempeño no debería ser considerado en aislamiento. Pesa la cantidad de esfuerzo requerido versus la ventaja ganada.
- El desempeño puede ser una preocupación en proyectos grandes pero no es a menudo un asunto para proyectos pequeños.
- Obtener un programa para trabajar debería tener una prioridad superior que estudiar más a fondo el desempeño del programa. Una vez que tienes un programa de trabajo puedes usar el perfilador para hacerle más eficiente. El desempeño debería ser considerado durante el proceso inicial del diseño/desarrollo sólo si - se determina - es un factor crítico.
- No hagas suposiciones acerca de donde están los cuellos de botella. Ejecute un perfilador para obtener los datos.
- Siempre que sea posible trata explícitamente de descartar una instancia estableciéndola a **null**. Éste a veces puede ser un indicio útil para el recolector de basuras.
- El tamaño del programa tiene importancia. La optimización de desempeño es generalmente de valor sólo cuando el tamaño del proyecto es grande, corre por mucho tiempo y la velocidad es un asunto.

- Las variables **static final** pueden ser optimizadas por el JVM para mejorar la velocidad de programa. Las constantes de programa de ese modo deberían ser declaradas como **static** y **final**.

Doclets

Aunque podría estar un poco asombrando para pensar acerca de una herramienta que fue desarrollada para soporte de documentación como algo que te ayuda a seguirle la pista a los problemas en tus programas, doclets pueden ser sorprendentemente útiles. Porque un doclet interconecta en el analizador gramatical del javadoc, tiene información disponible para ese analizador sintáctico. Con esto, programáticamente puedes examinar los nombres de clase, los nombres del campo, y firmas de método en tu código y los problemas de potencial de la bandera.

El proceso de producir la documentación JDK de los archivos fuentes Java involucra el análisis sintáctico del archivo fuente y el formateo de este archivo analizado gramaticalmente usando el doclet estándar. Puedes escribir un doclet personalizado para personalizar el formateo de tus comentarios del javadoc. Sin embargo, doclets te permiten hacer mucho más que simplemente formateando el comentario porque un doclet tiene disponible mucho de la información acerca del archivo fuente que está siendo analizado gramaticalmente.

Puedes extraer información acerca de todos los miembros de la clase: Los campos, constructores, métodos, y los comentarios se asociaron con cada uno de los miembros (alas, el cuerpo de código de método no está disponible). Los detalles acerca de los miembros son objetos especiales interiores encapsulados, el cual contiene información acerca de las propiedades del miembro (privado, estático, final, etc.). Esta información puede ser de ayuda en detectar código pobremente escrito, como las variables del miembro que deberían ser privadas pero son públicas, parámetros de método sin comentarios, e identificadores que no siguen convenciones de nombramiento.

Javadoc no puede capturar todos los errores de compilación. Divisará errores de sintaxis, algo semejante como un refuerzo irreemplazable, pero no puede capturar errores semánticos. El acercamiento más seguro es correr el compilador Java en tu código antes de tratar de usar una herramienta basado en doclet.

El mecanismo de análisis sintáctico previsto por javadoc analiza gramaticalmente el archivo fuente entero y lo almacena en la memoria en un objeto de clase **RootDoc**. El punto de entrada para el doclet enviado al javadoc es **start**(**RootDoc doc**). Es comparable para el **main**(**String Args**) de un programa normal Java. Puedes atravesar a través del objeto **RootDoc** y puedes extraer la información necesaria. El siguiente ejemplo demuestra cómo escribir un doclet simple; Simplemente imprime a todos los miembros de cada clase que fue analizada gramaticalmente:

```
//: c15:PrintMembersDoclet.java
```

```
// Doclet that prints out all members of the class.
import com.sun.javadoc.*;

public class PrintMembersDoclet {
  public static boolean start(RootDoc root) {
    ClassDoc[] classes = root.classes();
    processClasses(classes);
    return true;
  }
  private static void processClasses(ClassDoc[] classes) {
    for(int i = 0; i < classes.length; i++) {
      processOneClass(classes[i]);
    }
  }
  private static void processOneClass(ClassDoc cls) {
    FieldDoc[] fd = cls.fields();
    for(int i = 0; i < fd.length; i++)
      processDocElement(fd[i]);
    ConstructorDoc[] cons = cls.constructors();
    for(int i = 0; i < cons.length; i++)
      processDocElement(cons[i]);
    MethodDoc[] md = cls.methods();
    for(int i = 0; i < md.length; i++)
      processDocElement(md[i]);
  }
  private static void processDocElement(Doc dc) {
    MemberDoc md = (MemberDoc)dc;
    System.out.print(md.modifiers());
    System.out.print(" " + md.name());
    if(md.isMethod())
      System.out.println("()");
    else if(md.isConstructor())
      System.out.println();
  }
} ///:~
```

Puedes usar el doclet para imprimir los miembros como éste:

```
javadoc -doclet PrintMembersDoclet -private
PrintMembersDoclet.java
```

Esto invoca **javadoc** en el último argumento en el comando, lo cual quiera decir que analizará gramaticalmente el archivo **PrintMembersDoclet.java**. La opción **-doclet** pide que javadoc use el doclet personalizado **PrintMembersDoclet**. La etiqueta **-private** instruye javadoc para también imprimir miembros privados (el defecto es imprimir sólo miembros protegidos y públicos).

RootDoc contiene una colección de **ClassDoc** que mantiene toda la información acerca de la clase. Clases como **MethodDoc**, **FieldDoc**, y **ConstructorDoc** contienen información referente a los métodos, campos, y constructores, respectivamente. El método **processOneClass**() extrae la lista

de estos miembros y los imprime.

También puedes crear taglets, el cual te permite implementar etiquetas personalizadas del javadoc. La documentación JDK presenta un ejemplo que implementa una etiqueta **@todo**, lo cual despliega su texto en amarillo en la salida resultante Javadoc. Busca "**taglet**" en la documentación JDK para más detalles.

Resumen

Este capítulo introdujo de lo que me he percatado puede ser el asunto más esencial en programar, superceder asuntos de lenguaje de sintaxis y del diseño: ¿Cómo aseguras que tu código es correcto, y lo mantienes de esa manera?

La experiencia reciente ha demostrado que la herramienta más útil y práctica hasta la fecha es prueba de unidades, el cual, como se muestra en este capítulo, puede estar combinado muy eficazmente con Diseño por contrato. Hay otros tipos de pruebas igualmente, algo semejante como la prueba de conformidad para comprobar que tus historias de casos/usuario de uso todas han sido implementadas. Pero por alguna razón, nosotros en el pasado hemos relegado poner a prueba para terminar después por alguien más. La programación extrema insiste en que las pruebas de la unidad estén escritas antes del código; Creas el cuadro de trabajo de prueba para la clase, y luego la clase misma (en una o dos ocasiones he hecho esto exitosamente, pero estoy generalmente encantado si la prueba aparece en alguna parte durante el proceso inicial de codificación). Permanece la resistencia para probar, usualmente por esos que no han hecho un intento y creen que pueden escribir buen código sin experimentar. Pero la mayor experiencia que tengo, lo mayor lo repito a mí mismo:

Si no está probado, está hecho pedazos.

Es un mantra importante, especialmente cuando estás pensando en reducir esquinas. El mayor de tus problemas que descubres, lo más adjunto aumenta para la seguridad de pruebas incorporadas.

Los sistemas de construcción (en particular, **Ant** también) y el *control de revisión* (*CVS*) fueron introducidos en este capítulo porque proveen estructura para tu proyecto y sus pruebas. Para mí, la meta principal de Programación Extrema es la velocidad – la habilidad para rápidamente mover tu proyecto hacia adelante (pero en una moda fidedigna), y rápidamente refactorizarla cuando te das cuenta de que puede ser mejorado. La velocidad requiere que una estructura del soporte te dé confianza que las cosas no caerán a través de los cracks cuando comienzas a hacer cambios grandes para tu proyecto. Esto incluye a un depositario confiable, lo cual te permite rodar de regreso a cualquier versión previa, y un sistema automático de la construcción que, una vez configurado, garantizan que el proyecto puede ser compilado y probado en un único paso.

Una vez que tienes motivo para creer que tu programa es sano, el registro de

actividades provee una forma para monitorear su pulso, y aun (como se muestra en este capítulo) para automáticamente enviarte un email si algo comienza a salir mal. Cuando lo hace, depurar y perfilar le ayuda a seguir la pista a los problemas y asuntos de desempeño.

Quizá es la naturaleza de programación de computadoras para querer una respuesta única, evidente, concreta. Después de todo, trabaja con unos y ceros, que no tiene límites indistintos (realmente lo hacen, pero los ingenieros electrónicos han hecho todo lo posible para darnos el modelo que queremos). En lo que se refiere a soluciones, es genial creer que aquí hay una respuesta. Pero me he encontrado con que hay límites para cualquier técnica, y comprensión donde esos límites son mucho más poderosos que cualquier único acercamiento puede ser, porque te permite usar un método donde su máxima fuerza miente, y para combinarla con otros acercamientos donde no es tan fuerte. Por ejemplo, en este capítulo el Diseño por contrato fue presentado en combinación con prueba de unidades de la caja blanca, y como creaba el ejemplo, descubrí que los dos funcionamientos en el concierto fue bastante más útil que ya sea uno a solas.

He encontrado esta idea para ser verdadero en más que simplemente el asunto de descubrir problemas, sino que también en construir sistemas en primer lugar. Por ejemplo, usar un sencillo lenguaje de programación o herramienta para solucionar tu problema es atractivo del punto de vista de consistencia, pero a menudo me he encontrado con que puedo solucionar ciertos problemas bastante más rápidamente y eficazmente usando el lenguaje de programación Python en lugar de Java, para el beneficio general del proyecto. También puedes descubrir que **Ant** trabaja en algunos lugares, y en otros, **make** es más útil. O, si tus clientes están en plataformas *Windows*, puede ser sensato hacer la decisión radical de usar *Delphi* o *Visual BASIC* para desarrollar programas del lado cliente más rápidamente que lo podrías hacer en Java. El asunto importante es mantener una amplitud de ideas y recordar que estás tratando de conseguir resultados, no necesariamente usas una cierta herramienta o técnica. Esto puede ser difícil, pero si recuerdas que la frecuencia de fallas de proyecto es realmente alta y tus oportunidades de éxito son proporcionalmente bajas, podría ser un poco más abierto para las soluciones que podrían ser más productivas. Una de mis frases favoritas de Programación Extrema (y uno que me encuentro con que desobedece a menudo por razones usualmente absurdas) es "*haz la cosa más simple que posiblemente podría trabajar*." La mayoría de las veces, lo más simple y el acercamiento más expediente, si lo puedes descubrir, es lo mejor.

Ejercicios

1. Crea una clase conteniendo una cláusula **static** que lanza una excepción si las aserciones no está habilitados. Demuestre que esta prueba trabaje correctamente.

2. Modifica el ejercicio anterior para usar el acercamiento en **LoaderAssertions.java** para activar aserciones en lugar de lanzar una excepción. Demuestre que éste trabaje correctamente.

3. En **LoggingLevels.java**, comenta fuera del código que establece el nivel de severidad de los manipuladores del registrador de la raíz y verifica que los mensajes de nivel **CONFIG** y debajo no es reportado.

4. Hereda de **java.util.Logging.Level** y define tu nivel con un valor menos de **FINEST**. Modifica **LoggingLevels.java** para usar tu nuevo nivel y demuestre que los mensajes en tu niv el no aparecerán cuando el nivel de registro de actividades es **FINEST**.

5. Asocia a un **FileHandler** con el registrador de la raíz.

6. Modifica **FileHandler** a fin de que el formato de salida sea en un archivo del texto simple.

7. Modifica **MultipleHandlers.java** a fin de que genere salida en formato simple de texto en lugar de XML.

8. Modifica a **LoggingLevels.java** para colocar niveles diferentes de registro de actividades para los manipuladores asociados con el registrador de la raíz.

9. Escribe un programa simple que coloca el nivel de registro de actividades del registrador de la raíz basado en un argumento de línea de comando.

10. Escribe un ejemplo usando **Formatters** y **Handlers** para devolver un archivo de registro como HTML.

11. Escribe a un ejemplo usando **Handlers** y **Filters** para registrar mensajes con cualquier nivel de severidad sobre **INFO** en un archivo y cualquier nivel de severidad inclusive y debajo de **INFO** en otro archivo. Los archivos deberían escribirse en texto simple.

12. Modifica log.prop para agregar una clase adicional de inicialización que inicializa a un Formatter personalizado para el com del registrador.

13. Corre a JDB en SimpleDebugging.java, pero no des la orden catch Exception. Demuestra que todavía captura la excepción.

14. ¡Agrega una referencia no inicializada para SimpleDebugging.java (tendrás que hacerla de un modo que el compilador no capture el error!) y use JDB para seguirle la pista al problema.

15. Realiza la prueba descrita en la sección “Prueba s de cobertura”.

16. Crea un doclet que despliega identificadores que no podría seguir a la convención de nombramiento Java comprobando cómo las letras mayúsculas sirven para aquellos identificadores.

16: Programación Distribuida

Históricamente, la programación a través de múltiples máquinas ha sido fuente de error, difícil y compleja.

El programador tenía que conocer muchos detalles sobre la red y, en ocasiones, incluso el hardware. Generalmente era necesario comprender las distintas "capas" del protocolo de red, y había muchas funciones diferentes en cada una de las bibliotecas de la red involucradas con el establecimiento de conexiones, el empaquetado y desempaquetado de bloques de información; el reenvío y recepción de esos bloques; y el establecimiento de acuerdos. Era una tarea tremenda.

Sin embargo, la idea básica de la computación distribuida no es tan complicada, y está abstraída de forma muy elegante en las bibliotecas de Java. Se desea:

- Conseguir algo de información de una máquina lejana y moverla a la máquina local, o viceversa. Esto se logra con programación básica de red.
- Conectarse a una base de datos, que puede residir en otra máquina. Esto se logra con la *Java DataBase Connectivity* (JDBC), que es una abstracción alejada de los detalles difíciles y específicos de cada plataforma de *SQL* (el lenguaje de consulta estructurado o *Structured Query Language* que se usa en la mayoría de transacciones de bases de datos).
- Proporcionar servicios vía un servidor web. Esto se logra con los *servlets* de Java y las *Java Server Pages* (JSP).
- Ejecutar métodos sobre objetos Java que residan en máquinas remotas, de forma transparente, como si estos objetos residieran en máquinas locales. Esto se logra con el *Remote Method Invocation* (RMI) de Java.
- Utilizar código escrito en otros lenguajes, que está en ejecución en otras arquitecturas. Esto se logra usando la *Common Object Request Broker Architecture* (CORBA), soportado directamente por Java.
- Aislar la lógica de negocio de aspectos de conectividad, especialmente en conexiones con bases de datos, incluyendo la gestión de transacciones y la seguridad. Esto se logra usando Enterprise JavaBeans (EJB). Los EJB no son una arquitectura distribuida, sino que las aplicaciones resultantes suelen usarse en un sistema cliente-servidor en red.
- Fácilmente, dinámicamente, añadir y quitar dispositivos de una red que representa un sistema local. Esto se logra con Jini de Java.

En este capítulo se dará a cada tema una ligera introducción. Nótese, por favor, que cada tema es bastante voluminoso y de por sí puede ser fuente de libros

enteros, por lo que este capítulo sólo pretende familiarizar al lector con estos temas, y no convertirlo en un experto (sin embargo, se puede recorrer un largísimo camino en la programación en red, *servlets* y *JSP* con la información que se presenta aquí).

Programación en red

Una de las mayores fortalezas de Java es su funcionamiento inocuo en red. Los diseñadores de bibliotecas de red de Java han hecho que trabajar en red sea bastante similar a la escritura y lectura de archivo, excepto en que el "archivo" exista en una máquina remota y la máquina remota pueda decidir exactamente qué desea hacer con la información que se le envía o solicita. Siempre que sea posible, se han abstraído los detalles de la red subyacente, dejándose para la JVM y la instalación de Java que haya en la máquina local. El modelo de programación que se usa es el de un archivo; de hecho, se envuelve la conexión de red (un "socket") con objetos flujo, por lo que se acaban usando las mismas llamadas a métodos que con el resto de flujos. Además, el multihilo inherente a Java es extremadamente útil al tratar con otro aspecto de red: la manipulación simultánea de múltiples conexiones.

Esta sección introduce el soporte de red de Java usando ejemplos fáciles de entender.

Identificar una máquina

Por supuesto, para comunicarse con una máquina desde otra y para asegurarse de estar conectado con una máquina particular, debe haber alguna forma de identificar de forma unívoca las máquinas de una red. Las primeras redes se diseñaron de forma que proporcionaran nombres únicos de máquinas dentro de la red local. Sin embargo, Java trabaja dentro de Internet, lo que requiere una forma de identificar una máquina unívocamente de entre todas las demás máquinas del mundo. Esto se logra con la dirección IP (Internet Protocol) que puede existir de dos formas:

1. La forma familiar DNS (Domain Name System). Nuestro nombre de dominio es bruceeckel.com, y de tener un computador denominado Opus en nuestro dominio, su nombre de dominio habría sido Opus.bruceeckel.com. Éste es exactamente el tipo de nombre que se usa al enviar un correo a la gente, y suele incorporarse en una dirección World Wide Web.
2. Alternativamente, puede usarse la forma del "cuarteto punteado", que consta de cuatro números separados por puntos, como 123.255.28.120.

En ambos casos, la dirección IP se representa internamente como un número de 32' bits (de forma que cada uno de los cuatro números no puede exceder de 255), y se puede lograr un objeto Java especial para que represente este número

de ninguna de las formas de arriba, usando el método **static InetAddress.getByName**() que está en java.net. El resultado es un objeto de tipo **InetAddress**, que puede usarse para construir un "socket" como se verá después.

Como un simple ejemplo del uso de **InetAddress.getByName**(), considérese lo que ocurre si se tiene un proveedor de servicios de Internet (*ISP* o *Internet Service Provider*) vía telefónica. Cada vez que uno llama, le asignan una dirección IP temporal. Pero mientras se está conectado, la dirección IP de cada uno tiene la misma validez que cualquier otra dirección IP de la red. Si alguien se conecta a la máquina utilizando su dirección IP puede conectarse a un servidor web o a un servidor *FTP* en ejecución en tu máquina. Por supuesto, necesitan saber la dirección IP y cómo éste varía cada vez que uno se conecta, ¿cómo puede saberse?

[1] Esto implica un máximo de algo más de cuatro millones de números, que se está agotando rápidamente. El nuevo estándar para direcciones IP usará un número de 128 bits, que debería producir direcciones IP únicas suficientes para el futuro predecible.

El programa siguiente usa **InetAddress.getByName**() para producir la dirección IP actual. Para usarla, hay que saber el nombre del computador que se esté usando. En Windows 95/98 hay que ir a "*Configuración*", "*Panel de Control*", "*Red*" y después seleccionar la solapa "*Identificación*". El campo "*Nombre de PC*" es el que hay que poner en la línea de comandos:

```
//: c15:WhoAmI.java
// Finds out your network address when
// you're connected to the Internet.
import java.net.*;

public class WhoAmI {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

En este caso, la máquina se denomina " peppy ". Por tanto, una vez que nos hemos conectado a nuestro ISP ejecutamos el programa:

```
java WhoAmI peppy
```

Recibimos un mensaje como éste (por supuesto, la dirección es distinta cada vez):

```
peppy/199.190.87.75
```

Si le decimos a un amigo esta dirección y tenemos un servidor web en ejecución en nuestro PC, éste puede conectarse a la misma acudiendo a la URL *http://199.190.87.75* (sólo mientras continuemos conectados durante esa sesión). Ésta puede ser una forma útil de distribuir información a alguien más, o probar la configuración de un sitio web antes de enviarlo a un servidor "real".

Servidores y clientes

Todo el sentido de una red es permitir a dos máquinas conectarse y comunicarse. Una vez que ambas máquinas se han encontrado mutuamente, pueden tener una conversación bidireccional. Pero, ¿cómo se identifican mutuamente? Es como perderse en un parque de atracciones: una máquina tiene que estar en un lugar y escuchar mientras la otra máquina dice: "Eh, ¿dónde estás?" A la máquina que "sigue en un sitio" se le denomina el servidor, y a la que la busca se le denomina el cliente. Esta distinción es importante sólo mientras el cliente está intentando conectarse al servidor. Una vez que se han conectado, se convierte en un proceso de comunicación bidireccional y no vuelve a ser importante el hecho de si alguno desempeñó el papel de servidor y el otro el de cliente.

Por tanto, el trabajo del servidor es esperar una conexión, y ésta se lleva a cabo por parte del objeto servidor especial que se crea. El trabajo del cliente es intentar hacer una conexión al servidor, y esto lo logra el objeto cliente especial que se crea. Una vez que se hace la conexión se verá que en ambos extremos, servidor y cliente, la conexión se convierte mágicamente en un objeto flujo E/S, y a partir de ese momento se puede tratar la conexión como si simplemente se estuviera leyendo y escribiendo a un archivo. Por consiguiente, tras hacer la conexión, simplemente se usarán los comandos de E/S familiares del Capítulo 12. Ésta es una de las mejores facetas del funcionamiento en red de Java.

Probar programas sin una red

Por muchas razones, se podría no tener una máquina cliente, una máquina servidora y una red disponibles para probar los programas. Se podrían estar llevando a cabo ejercicios en una situación parecida a la de una clase, o se podrían estar escribiendo programas que no son aún lo suficientemente estables como para ponerlos en la red. Los creadores del *Internet Protocol* eran conscientes de este aspecto, y crearon una dirección especial denominada

localhost para ser una dirección IP "*bucle local*" para hacer pruebas sin red. La forma genérica de producir esta dirección en Java es:

```
InetAddress addr = InetAddress.getByName(null);
```

Si se pasa un **null** a **getByName**(), éste pasa por defecto a usar el **localhost**. La **InetAddress** es lo que se usa para referirse a la máquina particular, y hay que producir este nombre antes de seguir avanzando. No se pueden manipular los contenidos de una **InetAddress** (pero se pueden imprimir, como se verá en el ejemplo siguiente). La única forma de crear un **InetAddress** es a través de uno de los siguientes métodos miembro **static** sobrecargados: **getByName**() (que es el que se usará generalmente), **getAllByNarne**() o **getLocalHost**().

También se puede producir la dirección del bucle local pasándole el **String localhost**:

```
InetAddress.getByName("localhost");
```

(asumiendo que "**localhost**" está configurado en la tabla de "**hosts**" de la máquina), o usando su forma de cuarteto punteado para nombrar el número IP reservado del bucle:

```
InetAddress.getByName("127.0.0.1");
```

Las tres formas producen el mismo resultado.

Puerto: un lugar único dentro de la máquina

Una dirección IP no es suficiente para identificar un único servidor, puesto que pueden existir muchos servidores en una misma máquina. Cada máquina IP también contiene puertos, y cuando se establece un cliente o un servidor hay que elegir un puerto en el que tanto el cliente como el servidor acuerden conectarse; si se encuentra alguno, la dirección IP es el barrio, y el puerto es el bar.

El puerto no es una ubicación física en una máquina, sino una abstracción software (fundamentalmente para propósitos de reservas). El programa cliente sabe como conectarse a la máquina vía su dirección IP, pero, (¿cómo se conecta a un servicio deseado? (potencialmente uno de muchos en esa máquina). Es ahí donde aparecen los puertos como un segundo nivel de direccionamiento. La idea es que si se pregunta por un puerto en particular, se está pidiendo el servicio asociado a ese número. La hora del día es un ejemplo simple de un servicio. Depende del cliente el que éste conozca el número de puerto sobre el que se está ejecutando el servicio deseado.

Los servicios del sistema se reservan el uso de los puertos del 1 al 1.024, por lo que no se deberían usar estos números o cualquier otro puerto que se sepa que está en uso. La primera elección para los ejemplos de este libro será el puerto 8080 (en memoria del venerable chip Intel 8080 de 8 bits de nuestro primer computador, una máquina CP/M).

Sockets

El socket es la abstracción software que se usa para representar los "terminales" de una conexión entre dos máquinas. Para una conexión dada, hay un socket en cada máquina, y se puede imaginar un "cable" hipotético entre las dos máquinas, estando cada uno de los extremos del "cable" enchufados al socket. Por supuesto, se desconoce completamente el hardware físico y el cableado entre máquinas. Lo fundamental de esta abstracción es que no hay que saber nada más que lo necesario.

En Java, se crea un socket para hacer una conexión a la otra máquina, después se logra un **InputStream** y un **OutputStream** (o, con los convertidores apropiados, un **Reader** y un **Writer**) a partir del socket para poder tratar la conexión como un objeto flujo de E/S. Hay dos clases de socket basadas en flujos: un **ServerSocket** que usa un servidor para "escuchar" eventos entrantes y un **Socket** que usa un cliente para iniciar una conexión. Una vez que un cliente hace una conexión socket, el **ServerSocket** devuelve (vía el método **accept**()) un **Socket** correspondiente a través del cual tendrán lugar las comunicaciones en el lado servidor. A partir de ese momento se tiene una auténtica conexión **Socket** a **Socket** y se tratan ambos extremos de la misma forma, puesto que son iguales. En este momento se usan los métodos **getInputStream**() y **getOutputStream**() para producir los objetos **InputStream** y **OutputStream** correspondientes de cada **Socket**. Éstos deben envolverse en espacios de almacenamiento intermedio y clases formateadoras exactamente igual que cualquier otro objeto flujo descrito en el Capítulo 11.

El uso del término **ServerSocket** habría parecido ser otro ejemplo de un esquema confuso de nombres en las bibliotecas de Java. Podría pensarse que habría sido mejor denominar al **ServerSocket**, "**ServerConnector**" o algo sin la palabra "**Socket**". También se podría pensar que, tanto **ServerSocket** como **Socket**, deberían ser heredados de alguna clase base común. Sin duda, ambas clases tienen varios métodos en común, pero no son suficientes como para darles a ambos una clase base común. En vez de ello, el trabajo de **ServerSocket** es esperar hasta que se le conecte otra máquina, y devolver después un **Socket**. Por esto, **ServerSocket** parece no tener un nombre muy adecuado para lo que hace, pues su trabajo no es realmente ser un **socket**, sino construir un objeto **Socket** cuando alguien se conecta al mismo.

Sin embargo, el **ServerSocket** crea un "servidor" físico o socket oyente en la máquina host. Este socket queda esperando a posibles conexiones entrantes y devuelve un socket de "establecimiento" (con los puntos de **f**inalización local y remoto definidos) vía el método **accept**(). La parte confusa es que estos dos sockets (el oyente y el establecimiento están asociados con el mismo socket servidor). El socket oyente sólo puede aceptar nuevas peticiones de conexión, y no paquetes de datos. Por tanto, mientras **ServerSocket** no tenga mucho sentido desde el punto de vista programático, sí lo tiene "físicamente".

Cuando se crea un **ServerSocket**, sólo se le da un número de puerto. No hay que dar una dirección IP, porque ya está en la máquina que representa. Sin embargo, cuando se crea un **Socket** hay que dar, tanto una dirección IP como un número de puerto al que se está intentando conectar. (Sin embargo, el **Socket** que vuelve de **ServerSocket.accept**() ya contiene toda esta información.)

Un servidor y cliente simples

Este ejemplo hace el uso más simple de servidores y clientes basados en sockets. Todo lo que hace el servidor es esperar a una conexión, después usa el **Socket** producido por esa conexión para crear un **InputStrearn** y un **OutputStream**. Éstos se convierten en un **Reader** y un **Writer**, y después envueltos en un **BufferedReader** y un **PrintWriter**. Después de esto, todo lo que se lee de **BufferedReader** se reproduce en el **PrintWriter** hasta que recibe la línea "FIN", momento en el que se cierra la conexión.

El cliente hace la conexión al servidor, después crea un **OutputStream** y lleva a cabo la misma envoltura que el servidor. Las líneas de texto se envían a través del **PrintWriter** resultante. El cliente también crea un **InputStream** (de nuevo, con conversiones y envolturas apropiadas) para escucharlo que esté diciendo el servidor (que, en este caso, es simplemente una reproducción de las mismas palabras).

Tanto el servidor como el cliente usan el mismo número de puerto, y el cliente usa la dirección del bucle local para conectar al servidor en la misma máquina, con lo que no hay que probarlo a través de la red. (En algunas configuraciones, podría ser necesario estar conectado a una red para que este programa funcione, incluso si no se está comunicando a través de una red.)

He aquí el servidor:

```
//: c15:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;
```

```
public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

Se puede ver que el **ServerSocket** simplemente necesita un número de puerto, no una dirección IP (puesto que se está ejecutando en esta máquina). Cuando se llama a **accept**() el método se *bloquea* hasta que algún cliente intente conectarse al mismo. Es decir, está ahí esperando a una conexión, pero los otros procesos pueden ejecutarse (véase Capítulo 14). Cuando se hace una conexión, **accept**() devuelve un objeto **Socket** que representa esa conexión.

La responsabilidad de limpiar los sockets se enfoca aquí cuidadosamente. Si falla el constructor **Serversocket**, el programa simplemente finaliza (nótese que tenemos que asumir que el constructor de **ServerSocket** no deje ningún socket de red abierto sin control si falla). En este caso, **main**() lanza **IOException**, por lo que no es necesario un bloque **try**. Si el constructor **ServerSocket** tiene éxito, hay que guardar otras llamadas a métodos en un bloque **try**-**finally** para asegurar que, independientemente de cómo se deje el bloque, se cierre correctamente el **ServerSocket**.

Para el **Socket** devuelto por **accept**() se usa la misma lógica. Si falla **accept**(), tenemos que asumir que el **Socket** no existe o guarda recursos, por lo que no es necesario limpiarlo. Sin embargo, si tiene éxito, las siguientes sentencias tendrán que estar en un bloque **try**-**finally**, de forma que si fallan, se seguirá limpiando el **Socket**. Es necesario tener cuidado aquí porque los sockets usan recursos importantes no relacionados con la memoria, por lo que hay que ser diligente para limpiarlos (puesto que no hay ningún destructor que lo haga en Java).

Tanto el **ServerSocket** como el **Socket** producidos por **accept**() se imprimen a **System.out**. Esto significa que se invoca automáticamente a sus métodos **toString**(). Éstos producen:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

La siguiente parte del programa parece simplemente como si sólo se abrieran y escribieran archivos, de no ser porque el **InputStream** y **OutputStream** se crean a partir del objeto **Socket**. Tanto el objeto **InputStream** como el objeto **OutputStream** se convierten a objetos **Reader** y **Writer** usando las clases "conversoras" **InputStreamReader** y **OutputStreamWriter**, respectivamente. También se podrían haber usado directamente las clases **InputStream** y **OutputStream** de Java 1.0, pero con la salida, hay una ventaja añadida al usar el enfoque **Writer**. Éste aparece con **PrintWriter**, que tiene un constructor sobrecargado que toma un segundo argumento, un indicador **boolean** que indica si hay que vaciar automáticamente la salida al final de cada sentencia pr**i**ntln() (pero no **print**()).

Cada vez que se escribe a out, hay que vaciar su espacio de almacenamiento intermedio, de forma que la información fluya por la red. El vaciado es importante en este ejemplo particular, porque tanto el cliente como el servidor, esperan ambos a una línea proveniente de la otra parte antes de proceder. Si no se da el vaciado, la información no saldrá a la red hasta llenar el espacio de almacenamiento intermedio, causando muchos problemas en este ejemplo.

Cuando se escriban programas de red hay que tener cuidado con el uso del vaciado automático.

Cada vez que se vacía el espacio de almacenamiento intermedio, hay que crear y enviar un paquete. En este caso, eso es exactamente lo que queremos, pues si no se envía el paquete que contiene la línea, se detendría el intercambio amistoso bidireccional entre el cliente y el servidor. Dicho de otra forma, el final de línea es el fin de cada mensaje. Pero en muchos casos, los mensajes no están delimitados por líneas, por lo que es mucho más eficiente no usar el autovaciado y dejar en su lugar que el espacio de almacenamiento intermedio incluido decida cuándo construir y enviar un paquete. De esta forma se pueden enviar paquetes mayores, y el procesamiento será más rápido.

Nótese que, como ocurre con casi todos los flujos que se abren, éstos tienen sus espacios de almacenamiento intermedio. Hay un ejercicio al final de este capítulo para mostrar al lector lo que ocurre si no se utilizan los espacios de almacenamiento intermedio (todo se ralentiza).

El bucle **while** infinito lee líneas del **BufferedReader** entrada y escribe información a **System.out** y al **PrintWriter** salida. Nótese que entrada y salida podrían ser cualquier flujo, lo que ocurre simplemente es que están conectados a la red.

Cuando el cliente envía la línea "FIN", el programa sale del bucle y cierra el **Socket**.
He aquí el cliente:

```
//: c15:JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
```

```
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
} ///:~
```

En el método **main**() se pueden ver las tres formas de producir la **InetAddress** de la dirección IP del bucle local: utilizando **null**, **localhost** o la dirección explícitamente reservada 127.0.0.1. Por supuesto, si se desea establecer una conexión con una máquina que está en la red, deberá sustituirse esa dirección por la dirección IP de la máquina. Al imprimir la **InetAddress addr** (vía la llamada automática a su método **toString**()) el resultado es:

```
localhost/127.0.0.1
```

Pasando a **getByName**() un **null**, éste encontraba por defecto el **localhost**, lo que producía la dirección especial 127.0.0.1.

Nótese que el **Socket** de nombre socket se crea tanto con la **InetAddress** como con el número de puerto. Para entender lo que significa imprimir uno de estos objetos **Socket**, recuérdese que una conexión a Internet viene determinada exclusivamente por estos cuatro fragmentos de datos: **clientHost**, **clientPortNumber**, **serverHost** y **serverPortNumber**. Cuando aparece un servidor, toma su puerto asignado (8080) en el bucle local (127.0.0.1). Al aparecer el cliente, se le asigna el siguiente puerto disponible en la máquina, en este caso el 1077, que resulta estar en la misma máquina que el servidor. Ahora,

para que los datos se muevan entre el cliente y el servidor, cada uno de los extremos debe saber a dónde enviarlos. Por consiguiente, durante el proceso de conexión al servidor "conocido", el cliente envía una "dirección de retorno" de forma que el servidor pueda saber a dónde enviar sus datos. Esto es lo que se aprecia en la salida ejemplo para el lado servidor:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Esto significa que el servidor acaba de aceptar una conexión de 127.0.0.1 en el puerto 1077 al escuchar en su puerto local (8080). En el lado cliente:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

lo que significa que el cliente hizo una conexión con la 127.0.0.1 en el puerto 8080 usando el puerto local 1077.

Se verá que cada vez que se arranca de nuevo el cliente, se incrementa el número de puerto local. Empieza en el 1025 (uno más del bloque de puertos reservados) y va creciendo hasta volver a arrancar la máquina, momento en el que vuelve a empezar en el 1025. (En las máquinas UNM, una vez que se llega al límite superior del rango de sockets, los números vuelven también al número mínimo disponible.)

Una vez que se ha creado el objeto **Socket**, el proceso de convertirlo en un **BufferedReader** y en un **PrintWriter** es el mismo que en el servidor (de nuevo, en ambas ocasiones se empieza con un **Socket**). Aquí, el cliente inicia la conversación enviando la cadena "hola" seguida de un número. Nótese que es necesario volver a vaciar el espacio de almacenamiento intermedio (cosa que ocurre automáticamente vía el segundo argumento al constructor **PrintWriter**). Si no se vacía el espacio de almacenamiento intermedio, se colgaría toda la conversación, pues nunca se llegaría a enviar el primer "hola" (el espacio de almacenamiento intermedio no estaría lo suficientemente lleno como para que se produjera automáticamente un envío). Cada línea que se envíe de vuelta al servidor se escribe en **System.out** para verificar que todo está funcionando correctamente. Para terminar la conversación, se envía el "FIN" preacordado. Si el cliente simplemente cuelga, el servidor enviará una excepción.

Se puede ver que en el ejemplo se tiene el mismo cuidado para asegurar que los recursos de red representados por el **Socket** se limpien adecuadamente, mediante el uso de un bloque **try**-**finally**.

Los sockets producen una conexión "dedicada" que persiste hasta que sea desconectada explícitamente. (La conexión dedicada puede seguir siendo desconectada de forma no explícita si se cuelga alguno de los lados o intermediarios.) Esto significa que las dos partes están bloqueadas en la comunicación, estando la conexión constantemente abierta. Esto parece un

enfoque lógico para las redes, pero añade algo de sobrecarga a la propia red. Más adelante, en este mismo capítulo se verá un enfoque distinto al funcionamiento en red, en el que las conexiones son únicamente temporales.

Servir a múltiples clientes

El **Servidorparlante** funciona, pero solamente puede manejar a un cliente en cada momento. En un servidor típico, se deseará poder tratar con muchos clientes simultáneamente. La respuesta es el multihilo, y en los lenguajes que no lo soportan directamente esto significa todo tipo de complicaciones. En el Capítulo 14 se vio que en Java el multihilo es tan simple como se pudo, considerando que es de por sí un aspecto complejo. Dado que la capacidad de usar hilos en Java es bastante directa, construir un servidor que gestione múltiples clientes es relativamente sencillo.

El esquema básico es construir un **ServerSocket** único en el servidor, e invocar a **accept**() para que espere a una nueva conexión. Cuando **accept**() devuelva un **Socket**, se toma éste y se usa para crear un nuevo hilo cuyo trabajo es servir a ese cliente en particular. Después, se llama de nuevo a **accept**() para esperar a un nuevo cliente.
En el siguiente código servidor, puede verse que tiene una apariencia similar a la de **JabberServer.java**, excepto en que todas las operaciones que sirven a un cliente en particular han quedado desplazadas al interior de una clase hilo separado:

```
//: c15:MultiJabberServer.java
// A server that uses multithreading
// to handle any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
```

```
            socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
    }
  }
}

public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        } catch(IOException e) {
          // If it fails, close the socket,
          // otherwise the thread will close it:
          socket.close();
        }
      }
    } finally {
      s.close();
    }
  }
```

```
} ///:~
```

El hilo **ServeOneJabber** toma el objeto **Socket** producido por **accep**() en el método **main**() cada vez que un cliente nuevo haga una conexión. Posteriormente, y como antes, crea un **BufferedReader** y un objeto **PrintWriter** con autovaciado utilizando el **Socket**. Finalmente, llama al método **start**( ) especial de **Thread**, que lleva a cabo la inicialización de hilos e invoca después a **run**(). Éste hace el mismo tipo de acción que en el ejemplo anterior: leer algo del socket y después reproducirlo de vuelta hasta leer la señal especial "FIN".

La responsabilidad de cara a limpiar el socket debe ser diseñada nuevamente con cuidado. En este caso, el socket se crea fuera de **ServeOneJabber**, por lo que es posible compartir esta responsabilidad. Si el constructor **ServeOneJabber** falla, simplemente lanzará la excepción al llamador, que a continuación limpiará el hilo. Pero si el constructor tiene éxito, será el objeto **ServeOneJabber** el que tome la responsabilidad de limpiar el hilo, en su **run**().

Nótese la simplicidad del **MultiJabberServer**. Como antes, se crea un **ServerSocket** y se invoca a **accept**() para permitir una nueva conexión. Pero en esta ocasión, se pasa el valor de retorno de **accept**() (un **Socket**) al constructor de **ServeOneJabber**, que crea un nuevo hilo para manejar esa conexión. Cuando acaba la conexión, el hilo simplemente desaparece.

Si falla la creación de **ServerSocket**, se lanza de nuevo la excepción a través del método **main**().
Pero si la creación tiene éxito, el **try**-**finally** externo garantiza su limpieza. El **try**-**catch** interno simplemente previene del fallo del constructor de **ServeOneJabber**; si el constructor tiene éxito, el hilo **ServeOneJabber** cerrará el socket asociado.

Para probar que el servidor verdaderamente gestiona múltiples clientes, he aquí el siguiente programa, que crea muchos clientes (usando hilos) que se conectan al mismo servidor. El máximo número de hilos permitido viene determinado por **final int MAX_THREADS**.

```
//: c15:MultiJabberClient.java
// Client that tests the MultiJabberServer
// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
```

```
  private static int threadcount = 0;
  public static int threadCount() {
    return threadcount;
  }
  public JabberClientThread(InetAddress addr) {
    System.out.println("Making client " + id);
    threadcount++;
    try {
      socket =
        new Socket(addr, MultiJabberServer.PORT);
    } catch(IOException e) {
      System.err.println("Socket failed");
      // If the creation of the socket fails,
      // nothing needs to be cleaned up.
    }
    try {
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Enable auto-flush:
      out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())), true);
      start();
    } catch(IOException e) {
      // The socket should be closed on any
      // failures other than the socket
      // constructor:
      try {
        socket.close();
      } catch(IOException e2) {
        System.err.println("Socket not closed");
      }
    }
    // Otherwise the socket will be closed by
    // the run() method of the thread.
  }
  public void run() {
    try {
      for(int i = 0; i < 25; i++) {
        out.println("Client " + id + ": " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } catch(IOException e) {
      System.err.println("IO Exception");
```

```
    } finally {
      // Always close it:
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
      threadcount--; // Ending this thread
    }
  }
}

public class MultiJabberClient {
  static final int MAX_THREADS = 40;
  public static void main(String[] args)
      throws IOException, InterruptedException {
    InetAddress addr =
      InetAddress.getByName(null);
    while(true) {
      if(JabberClientThread.threadCount()
         < MAX_THREADS)
        new JabberClientThread(addr);
      Thread.currentThread().sleep(100);
    }
  }
} ///:~
```

El constructor **JabberClientThread** toma un **InetAddress** y lo usa para abrir un **Socket**. Probablemente se empiece a ver el patrón: siempre se usa el **Socket** para crear algún tipo de objeto **Reader** y/o **Writer** (o **InputStream** y/u **OutputStream**), que es la única forma de usar el **Socket**. (Por supuesto, puede escribir una o dos clases que automaticen este proceso en vez de llevar a cabo todo el tecleado cuando éste se vuelve molesto.) De nuevo, **start**() lleva a cabo la inicialización de hilos e invoca a **run**(). Aquí, se envían los mensajes al servidor y se reproduce esa información proveniente del servidor en la pantalla. Sin embargo, el hilo tiene un tiempo de vida limitado y suele acabarse. Nótese que el socket se limpia si falla el constructor después de haber creado el socket, pero antes de que acabe el constructor. De otra forma, la responsabilidad de llamar al **close**() para el socket se relega al método **run**().

El **threadcount** mantiene un seguimiento de cuántos objetos **JabberClientThread** existen en cada momento. Se incrementa como parte del constructor y se disminuye al acabar **run**() (lo que significa que el hilo está finalizando). En **MultiJabberClient.main**() puede verse que se prueba el número de hilos, y si hay demasiados no se crean más. Después, el método se duerme. De esta forma, algunos hilos acabarían eventualmente y se podrían crear más. Se puede experimentar con **MAX_THREADS** para ver dónde empieza a tener problemas un sistema en particular por la existencia de demasiadas conexiones.

Datagramas

Los ejemplos vistos hasta el momento usan el *Transmission Control Protocol* (TCP, conocido también como sockets basados en flujos), diseñado para garantizar la fiabilidad de manera que la información llegue siempre. Permite la retransmisión de datos perdidos, proporciona múltiples caminos a través de distintos enrutadores en caso de que uno falle, y los bytes se entregan en el mismo orden en que son enviados. Todo este control y fiabilidad tiene un coste: TCP supone una gran sobrecarga.

Hay un segundo protocolo, denominado *User Datagram Protocol* (UDP), que no garantiza que los paquetes se entreguen y que lleguen en el orden en que son enviados. Se llama "protocolo no orientado a la conexión" (TCP es un "protocolo orientado a la conexión"), lo cual no suena bien, pero dado que es muchísimo más rápido, en ocasiones es útil. Hay algunas aplicaciones, como una señal de audio, en las que no es crítico el hecho de que se puedan perder algunos paquetes, pero la velocidad es vital. O considérese un servidor de hora del día, en el que no importa verdaderamente el que se pierda algún mensaje. Además, algunas aplicaciones deberían ser capaces de disparar un mensaje UDP al servidor para asumir con posterioridad, si no hay respuesta en un periodo razonable de tiempo, que el mensaje se ha perdido.

Generalmente, se trabajará directamente programando bajo TCP, y sólo ocasionalmente se hará uso de UDP Hay un tratamiento de UDP más completo, incluyendo un ejemplo, en la primera edición de este libro (disponible en el CD ROM adjunto a este libro, o como descarga gratuita de *http://www.BruceEckel.com*).

Utilizar URL en un applet

Es posible que un applet pueda mostrar cualquier URL a través del navegador web en el que se esté ejecutando. Esto se puede lograr con la línea:

```
getAppletContext().showDocument(u);
```

donde **u** es el objeto URL. He aquí un ejemplo simple que le remite a otra página web. Aunque simplemente se logra una redirección a una página HTML, también se podría remitir a la salida de un programa CGI.

```
//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet {
  JButton send = new JButton("Go");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    send.addActionListener(new Al());
    cp.add(send);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // This could be a CGI program instead of
        // an HTML page.
        URL u = new URL(getDocumentBase(),
          "FetcherFrame.html");
        // Display the output of the URL using
        // the Web browser, as an ordinary page:
        getAppletContext().showDocument(u);
      } catch(Exception e) {
        l.setText(e.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new ShowHTML(), 100, 50);
  }
} ///:~
```

La belleza de la clase URL radica en cuánto hace por ti. Es posible conectarse a servidores web sin saber mucho o nada de lo que está ocurriendo verdaderamente.

Leer un archivo de un servidor

Una variación del programa anterior lee un archivo ubicado en el servidor. En este caso, el archivo es especificado por el cliente:

```
//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet {
  JButton fetchIt= new JButton("Fetch the Data");
  JTextField f =
    new JTextField("Fetcher.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    fetchIt.addActionListener(new FetchL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(fetchIt);
  }
  public class FetchL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
        t.setText(url + "\n");
        InputStream is = url.openStream();
        BufferedReader in = new BufferedReader(
          new InputStreamReader(is));
        String line;
        while ((line = in.readLine()) != null)
          t.append(line + "\n");
      } catch(Exception ex) {
        t.append(ex.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new Fetcher(), 500, 300);
  }
} ///:~
```

La creación del objeto URL es similar al ejemplo anterior - **getDocumentBase**() es como antes el punto de comienzo, pero en esta ocasión, el nombre del archivo se lee de **JTextField**. Una vez que se crea el objeto URL, se ubica su versión **String** en el **JTextArea**, de forma que se pueda ver la apariencia que tiene. Después, se proporciona un **InputStream** desde el URL, que en este caso simplemente producirá un flujo de los caracteres del archivo. Después de convertir a un **Reader**, y del tratamiento por espacios de almacenamiento intermedio, se lee cada línea para ser añadida al **JTextArea**. Nótese que el

JTextArea se ha ubicado dentro de un **JScrollPane**, de forma que todo el desplazamiento de pantallas se gestiona automáticamente.

Más aspectos de redes

Además de lo cubierto en este tratamiento introductorio hay, de hecho, bastantes más aspectos relacionados con las redes. La capacidad de red de Java también proporciona gran soporte para URL, incluyendo gestores de protocolos para distintos tipos de contenidos que pueden descubrirse en los distintos sitios Internet. Se pueden encontrar más facetas de tratamiento de red de Java descritas con gran detalle en Java Network Programming, de Elliote Rusty Harold (O'Reilly, 1997).

Conectividad a Bases de Datos de Java (JDBC)

Se ha estimado que la mitad de todos los desarrollos software involucran aplicaciones cliente/servidor. Una gran promesa de Java ha sido la habilidad de construir aplicaciones de base de datos cliente/servidor independientes de la plataforma. En realidad esto se ha conseguido gracias a la Conectividad a Bases de Datos de Java UDBC, Java DataBase Connectivity).

Uno de los mayores problemas de las bases de datos ha sido la guerra de características entre las propias compañías de bases de datos. Hay un lenguaje de bases de datos "estándar", el Structure Query Languaje (SQL-92), pero probablemente todo el mundo conoce el proveedor de la base de datos con la que trabaja a pesar del estándar. JDBC está diseñado para ser independiente de la plataforma, por lo que en tiempo de programación no hay que preocuparse por la base de datos que se esté utilizando. Sin embargo, sigue siendo posible construir llamadas específicas de ese fabricante desde JDBC, por lo que no se puede decir que haya restricción alguna a la hora de hacer lo que se tenga hacer.

Un lugar en el que los programadores pueden necesitar usar nombres del tipo SQL es en la sentencia **TABLE CREATE** de SQL al crear una nueva tabla de base de datos y definir el tipo SQL de cada columna. Desgraciadamente hay bastantes variaciones entre los tipos de SQL soportados por los distintos productos de base de datos. Bases de datos distintas que soportan tipos SQL con la misma semántica y estructura puede que den a esos tipos distintos nombres.

La mayoría de las bases de datos principales soportan un tipo de datos SQL para valores binarios grandes: en Oracle, a este tipo se le denomina LONG RAW, Sybase lo llama IMAGE, Informix lo llama BYTE, y DB2 le llama LONG VARCHAR FOR BIT DATA. Por consiguiente, si la portabilidad de la base de datos es una de

las metas a lograr, debería intentarse utilizar exclusivamente identificadores de tipos SQL genéricos.

La portabilidad es importante al escribir para un libro en el que los lectores pueden estar probando los ejemplos con cualquier tipo de almacén de datos desconocido. Hemos intentado escribir estos ejemplos para que sean lo más portables posible. El lector también podría darse cuenta de que se ha aislado el código específico de la base de datos para centralizar cualquier cambio que haya que realizar para hacer que estos ejemplos sean operativos en algún otro entorno.

JDBC, como muchas de las API de Java, está diseñado persiguiendo la simplicidad. Las llamadas a métodos que se hagan se corresponden con las operaciones lógicas que uno pensaría que debe hacer al recopilar datos desde la base de datos: conectarse a la base de datos, crear una sentencia y ejecutar la petición, y tener acceso al conjunto resultado.

Para permitir esta independencia de la plataforma, JDBC proporciona un gestor de controladores que mantiene dinámicamente todos los objetos controlador que puedan requerir las consultas a la base de datos. Por tanto, si se tienen tres tipos distintos de bases de datos a las que se desea establecer comunicación, se necesitarán tres objetos controlador. Estos objetos se registran a sí mismos en el gestor de controladores en el momento de su carga, y se puede forzar esta carga utilizando **Class.forName**( ).

Para abrir una base de datos, hay que crear un "URL de base de datos" que especifica:

1. Que se está usando JDBC con "jdbc".
2. El "subprotocolo": el nombre del controlador o el nombre de un mecanismo de conectividad de base de datos. Puesto que el diseño de JDBC se inspiró en ODBC, el primer subprotocolo disponible es el puente "jdbc-odbc", denominado "odbc".
3. El identificador de la base de datos. Éste varía en función del controlador de base de datos que se use, pero generalmente proporciona un nombre lógico al que el software administrador de la base de datos vincula a un directorio físico en el que están almacenadas las tablas de la base de datos. Para que un identificador de base de datos tenga algún significado hay que registrar el nombre utilizando el software de administración de base de datos. (Este proceso de registro varía de plataforma a plataforma.)

Toda esta información se combina en un **String**, el "URL de base de datos". Por ejemplo, para conectarse a través del subprotocolo ODBC a una base de datos denominada "gente", el URL de la base de datos podría ser:

```
String dbUrl = "jdbc:odbc:people";
```

Si se está estableciendo una conexión a través de una red, el URL de la base de datos debe contener la información de conexión que identifique la máquina remota, pudiendo resultar algo intimidatorio. He aquí un ejemplo de una base de datos Fuga a la que se invoca desde un cliente remoto utilizando RMI:

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

La URL de la base de datos es verdaderamente dos llamadas a jdbc en una. La primera parte, "*jdbc:rmi://192.168.170.27:1099/*" usa RMI para hacer la conexión al motor de base de datos remota escuchando en el puerto 1099 de la dirección IP *192.168.170.27*. La segunda parte del URL, "*jdbc:cloudscape:db*" conlleva los ajustes más típicos al usar el subprotocolo y el nombre de la base de datos, pero esto sólo ocurrirá una vez que la primera sección haya hecho la conexión vía RMI a la máquina remota.

Cuando uno está listo para conectarse a la base datos, hay que invocar al método **static DriverManager.getConnection**() y pasarle el URL de la base de datos, el nombre de usuario y la contraseña para entrar en la base de datos. A cambio, se recibe un objeto **Connection** que puede ser usado posteriormente para preguntar y manipular la base de datos.

El ejemplo siguiente abre una base de datos de información de contacto y busca el apellido de una persona, suministrado en línea de comandos. Sólo selecciona los nombres de la gente que tienen dirección de correo electrónico, y después imprime todos los que casen con el apellido suministrado:

```
//: c15:jdbc:Lookup.java
// Looks up email addresses in a
// local database using JDBC.
import java.sql.*;

public class Lookup {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    // Load the driver (registers itself)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, user, password);
    Statement s = c.createStatement();
    // SQL code:
    ResultSet r =
      s.executeQuery(
```

```
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // Capitalization doesn't matter:
      System.out.println(
        r.getString("Last") + ", "
        + r.getString("fIRST")
        + ": " + r.getString("EMAIL") );
    }
    s.close(); // Also closes ResultSet
  }
} ///:~
```

Puede verse que la creación del URL de la base de datos se hace como se describió anteriormente. En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que los campos nombre de usuario y contraseña son cadenas de caracteres vacías.

Una vez que se hace la conexión con **DriverManager.getConnection**(), se puede usar el objeto **Connection** resultante para crear un objeto **Statement** utilizando el método **createStatement**( ).

Con el **Statement** resultante, se puede llamar a **executeQuery**( ), pasándole una cadena de caracteres que contenga cualquier sentencia SQL compatible con el estándar SQL-92. (En breve se verá cómo se puede generar esta sentencia automáticamente, evitando tener que saber mucho SQL.)

El método **executeQuery**() devuelve un objeto **ResultSet**, que es un iterador: el método **next**() mueve el iterador al siguiente recurso de la sentencia, o devuelve **false** si se ha alcanzado el fin del conjunto resultado. Una ejecución de **executeQuery**() siempre genera un objeto **ResultSet** incluso si la solicitud conlleva un conjunto vacío (es decir, no se lanza una excepción). Nótese que hay que llamar a **next**() una vez, por lo menos antes de intentar leer ningún registro. Si el conjunto resultado está vacío, esta primera llamada a **next**() devolverá **false**. Por cada registro del conjunto resultado, es posible seleccionar los campos usando (entre otros enfoques) el nombre del campo como cadena de caracteres. Nótese también que se ignora el uso de mayúsculas o minúsculas en el nombre del campos - con una base de datos SQL no importa. El tipo de dato que se devuelve se determina invocando a **getInt**(), **getString**(), **getFloat**(), etc. En este momento, se tienen los datos de la base de datos en formato Java nativo y se puede hacer lo que se desee con ellos usando código Java ordinario.

Hacer que el ejemplo funcione

Con JDBC, entender el código es relativamente fácil. La parte complicada es hacer que funcione en un sistema en particular. La razón por la que es complicada es que requiere que cada lector averigüe cómo cargar adecuadamente su controlador JDBC, y cómo configurar una base de datos utilizando su software de administración de base de datos. Por supuesto, este proceso puede variar radicalmente de máquina a máquina, pero el proceso que sabíamos usar para que funcionara bajo un Windows de 32 bits puede dar algunas pistas para que cada uno se enfrente a su propia situación:

Paso 1: Encontrar el Controlador JDBC

El programa de arriba contiene la sentencia:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Ésta implica una estructura de directorios que es determinante. Con esta instalación particular del JDK 1.1, no había ningún archivo llamado **JdbcOdbcDriver.class**, por lo que si se echa un vistazo a este ejemplo y se empieza a buscarlo, uno se frustra. Otros ejemplos publicados usan un pseudo-nombre, como "**miControlador.ClassName**", que es aún menos útil. De hecho, la sentencia de carga de arriba para el controlador jdbc-odbc (el único que, de hecho, viene con el JDK) sólo aparece en unos pocos sitios en la documentación en línea (en particular, en una página de nombre "JDBC-ODBC Bridge Driver"). Si la sentencia de carga de arriba no funciona, entonces puede que el nombre haya cambiado como parte de los cambios de versión de Java, por lo que habría que volver a recorrerse toda la documentación.

Si la sentencia de carga está mal, se obtendrá justo en ese momento una excepción. Para probar si la sentencia de carga del controlador está funcionando correctamente o no, puede marcarse como comentario la sentencia; si el programa no lanza excepciones, será señal de que el controlador se está cargando correctamente.

Paso 2: Configurar la base de datos

Esto es, de nuevo, específico al Windows de 32 bits; puede ser que alguien tenga que investigar si está trabajando con otra plataforma.

En primer lugar, abra el panel de control. Se verá que hay dos iconos que dicen "ODBC". Hay que usar el que dice "Fuentes de Datos ODBC 32 bits", puesto que el otro es para lograr retrocompatibilidad con software ODBC de 16 bits y no generará ningún resultado en el caso de JDBC. Cuando se abre el icono "Fuentes de Datos ODBC 32 bits", se verá una caja de diálogo con solapas, entre las que

se incluyen "DSN de Usuario", "DSN de Sistema", "DSN de Archivo", etc., donde DSN significa "*Data Source Name*" ("Nombre de la fuente de datos"). Resulta que para el puente JDBC-ODBC, el único lugar en el que es importante configurar la base de datos es "DSN del Sistema", pero también se deseará probar la configuración y hacer consultas, y para ello también será necesario configurar la base de datos en "DSN de Archivo". Esto permitirá a la herramienta Microsoft Query (que viene con Microsoft Office) encontrar la base de datos. Nótese que existen también otras herramientas de otros vendedores.

La base de datos más interesante es una que esté actualmente en uso. El ODBC estándar soporta varios formatos de archivos distintos, incluyendo algunos tan venerables como DBase. Sin embargo, también incluye el formato simple "ASCII separado por comas", que generalmente toda herramienta de datos tiene la capacidad de escribir. Simplemente tomamos la base de datos de "gente", mantenida por nosotros durante años usando herramientas de gestión de contactos, y la exportamos a un archivo ASCII separado por comas (que suelen tener extensión .csv). En la sección "DSN de sistema", seleccionamos "Agregar", elegimos el controlador de texto para gestionar nuestro archivo ASCII separado por comas, y después deseleccionamos "usar directorio actual" para que me permitiera especificar el directorio al que exportar el archivo de datos.

Se verá al hacer esto que verdaderamente no se especifica un archivo, sólo un directorio. Eso es porque una base de datos suele representarse como una colección de archivos bajo un único directorio (aunque podría estar representada también de otra forma). Cada archivo suele contener una única tabla, y las sentencias SQL pueden producir resultados provenientes de múltiples tablas de la base de datos (a esto se le llama **join**). A una base de datos que sólo contiene una base de datos (como mi base de datos "gente") se le suele llamar *flat-file database*. La mayoría de problemas que van más allá del simple almacenamiento y recuperación de datos suelen requerir de varias tablas, que deben estar relacionadas mediante **joins** para producir los resultados deseados, y a éstas se las denomina bases de datos relacionales.

Paso 3: Probar la configuración

Para probar la configuración, será necesario disponer de alguna forma de descubrir si la base de datos es visible desde un programa que hace consultas a la misma. Por supuesto, se puede ejecutar simplemente el programa ejemplo JDBC de arriba, incluyendo la sentencia:

```
Connection c = DriverManager.getConnection(
  dbUrl, user, password);
```

Si se lanza una excepción, la configuración era incorrecta.
Sin embargo, es útil hacer que una herramienta de generación de consultas se vea involucrada en esto. Utilizamos Microsoft Query, que viene con Microsoft

Office, pero podría preferirse alguna otra. La herramienta de consultas debería saber dónde está la base de datos y Microsoft Query exigía que fuera al campo Administrador de ODBC de la etiqueta "DSN de Archivo" y añadiera una nueva entrada ahí, especificando de nuevo el controlador de texto y el directorio en el que reside nuestra base de datos. Se puede nombrar a la entrada como se desee, pero es útil usar el mismo nombre usado en "DSN de Sistema".

Una vez hecho esto se verá que la base de datos está disponible al crear una nueva consulta usando la herramienta de consultas.

Paso 4: Generar la consulta SQL

La consulta que creamos usando Microsoft Query no sólo nos mostraba que la base de datos estaba ahí y en orden correcto, sino que también creaba automáticamente el código SQL que necesitaba insertar en nuestro programa Java. Queríamos una consulta que buscara los registros que tuvieran el mismo apellido que se tecleara en la línea de comandos al arrancar el programa Java. Por tanto, y como punto de partida, buscamos un apellido específico, "Eckel".

También queríamos que se mostraran sólo aquellos nombres que tuvieran alguna dirección de correo electrónico asociada. Los pasos que seguimos para crear esta consulta fueron:

1. Empezar una nueva consulta y utilizar el *Query Wizard*. Seleccionar la base de datos "*gente*". (Esto equivale a abrir la conexión de base de datos usando el URL de base de datos apropiado).
2. Seleccionar la tabla "*gente*" dentro de la base de datos. Desde dentro de la tabla, seleccionar las columnas PRIMERO, SEGUNDO y CORREO.
3. Bajo "*Filtro de Datos*", seleccionar SEGUNDO y elegir "*equals*" con un argumento "*Eckel*".
4. Hacer clic en el botón de opción "*hd*".
5. Seleccionar CORREO y elegir "*Is not Null*".
6. Bajo "*Sort by*", seleccionar PRIMERO.

Los resultados de esta consulta mostrarán si se está obteniendo lo deseado.

Ahora se puede presionar el botón SQL y sin hacer ningún tipo de investigación, aparecerá el código SQL correcto, listo para ser cortado y pegado. Para esta consulta, sería algo así:

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

Es posible que las cosas vayan mal, especialmente si las consultas son más complicadas, pero usando una herramienta de generación de consultas, se pueden probar éstas de forma interactiva y generar automáticamente el código correcto.

Es difícil defender la postura de hacer esto a mano.

Paso 5: Modificar y cortar en una consulta

Se verá que el código de arriba tiene distinto aspecto del que se usó en el programa. Eso es porque la herramienta de consultas usa especificación completa de todos los nombres, incluso cuando sólo esté involucrada una tabla. (Cuando hay más de una tabla, la especificación completa evita colisiones entre columnas de distintas tablas que pudieran tener igual nombre.) Puesto que esta consulta simplemente involucra a una tabla, se puede eliminar el especificador "gente" de la mayoría de los nombres, así:

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

Además, no se deseará codificar todo el programa para que sólo busque un nombre en concreto. En vez de ello, se debería buscar el nombre proporcionado como parámetro de línea de comandos. Hacer estos cambios y convertir la sentencia SQL en un **String** de creación dinámica produce:

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL tiene otra forma de insertar nombres en una consulta, denominado procedimientos almacenados, que se usan para lograr incrementos de velocidad. Pero como experimentación de base de datos para un primer enfoque, construir las cadenas de consulta en Java es una opción más que correcta.
A partir de este ejemplo se puede ver que usando las herramientas actualmente disponibles – en particular la herramienta de construcción de consultas - es bastante directo programar con SQL y JDBC.

Una versión con IGU del programa de búsqueda

Es más útil dejar que el programa de búsqueda se ejecute continuamente y simplemente cambiar al mismo y teclear un nombre cuando se desee buscar a alguien. El siguiente programa crea el programa de búsqueda como una aplicación/applet y también añade finalización automática, de forma que los datos se mostrarán sin forzar al lector a teclear el apellido completo:

```
//: c15:jdbc:VLookup.java
// GUI version of Lookup.java.
// <applet code=VLookup
// width=500 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;

public class VLookup extends JApplet {
  String dbUrl = "jdbc:odbc:people";
  String user = "";
  String password = "";
  Statement s;
  JTextField searchFor = new JTextField(20);
  JLabel completion =
    new JLabel("                         ");
  JTextArea results = new JTextArea(40, 20);
  public void init() {
    searchFor.getDocument().addDocumentListener(
      new SearchL());
    JPanel p = new JPanel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(results, BorderLayout.CENTER);
    try {
      // Load the driver (registers itself)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
      s = c.createStatement();
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  class SearchL implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
```

```
    public void insertUpdate(DocumentEvent e){
      textValueChanged();
    }
    public void removeUpdate(DocumentEvent e){
      textValueChanged();
    }
  }
  public void textValueChanged() {
    ResultSet r;
    if(searchFor.getText().length() == 0) {
      completion.setText("");
      results.setText("");
      return;
    }
    try {
      // Name completion:
      r = s.executeQuery(
        "SELECT LAST FROM people.csv people " +
        "WHERE (LAST Like '" +
        searchFor.getText()  +
        "%') ORDER BY LAST");
      if(r.next())
        completion.setText(
          r.getString("last"));
      r = s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE (LAST='" +
        completion.getText() +
        "') AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    } catch(Exception e) {
      results.setText(
        searchFor.getText() + "\n");
      results.append(e.toString());
      return;
    }
    results.setText("");
    try {
      while(r.next()) {
        results.append(
          r.getString("Last") + ", "
          + r.getString("fIRST") +
          ": " + r.getString("EMAIL") + "\n");
      }
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  public static void main(String[] args) {
```

```
    Console.run(new VLookup(), 500, 200);
  }
} ///:~
```

Mucha de la lógica de la base de datos es la misma, pero puede verse que se añade un **DocumentListener** al **JTextField** (véase la entrada **javax.swing.JTextFie1d** de la documentación Java HTML de *http://www.java.sun.com* para encontrar detalles), de forma que siempre que se teclee un nuevo carácter, primero se intenta terminar el apellido buscándolo en la base de datos y usando el primero que se muestre. (Se coloca en la **JLabel conclusion**, y se usa como texto de búsqueda). De esta forma, tan pronto como se hayan tecleado los suficientes caracteres para que el programa encuentre el apellido buscado de manera unívoca, se puede parar el mismo.

Por qué el API JDBC parece tan complejo

Cuando se accede a la documentación en línea de JDBC puede asustar. En concreto, en la interfaz **DatabaseMetaData** - que es simplemente vasta, contrariamente al resto de interfaces de Java- hay métodos como **dataDefinitionCausesTransactionCommit**(), **getMaxColumnNameLength**(), **getMaxStatementLength**(), **storesMixedCaseQuotedIdentiñers**(), **supportsANSI92IntermediateSQL**(), **supportsLhnitedOuterJoins**() y demás. ¿De qué va todo esto?

Como se mencionó anteriormente, las bases de datos, desde su creación, siempre han parecido una fuente constante de tumultos, especialmente debido a la demanda de aplicaciones de bases de datos, y por consiguiente, las herramientas de bases de datos suelen ser tremendamente grandes.

Recientemente -y sólo recientemente- ha habido una convergencia en el lenguaje común SQL (y hay muchos otros lenguajes comunes de bases de datos de uso frecuente).

Pero incluso con un SQL "estándar" hay tantas variaciones del tema que JDBC debe proporcionar la gran interfaz **DatabaseMetaData**, de forma que el código pueda descubrir las capacidades del SQL estándar en particular que usa la base de datos a la que se está actualmente conectado.

Abreviadamente, puede escribirse SQL simple y transportable, pero si se desea optimizar la velocidad, la codificación se multiplicará tremendamente al investigar las capacidades de la base de datos de un vendedor concreto.

Esto, por supuesto, no es culpa de Java. Las discrepancias entre los productos de base de datos son simplemente algo que JDBC intenta ayudar a compensar. Pero recuerde que le puede facilitar las cosas la escritura de sentencias de base de

datos (queries) genéricas, sin apenas preocupar al rendimiento, o, si debe afinar en el rendimiento, conozca la plataforma en la que está escribiendo para evitar tener que escribir todo ese código de investigación.

Un ejemplo más sofisticado

Un segundo ejemplo **[2]** más interesante involucra una base de datos multitabla que reside en un servidor. Aquí, la base de datos está destinada a proporcionar un repositorio de actividades de la comunidad y permitir a la gente apuntarse a esos eventos, de forma que se le llama Base de Datos de Interés de la Comunidad (CID, o Community Interests Database). Este ejemplo sólo proporcionará un repaso de la base de datos y su implementación. Hay muchos libros, seminarios y paquetes software que nos ayudarán a diseñar y desarrollar una base de datos.

Además, este ejemplo presupone que anteriormente se ha instalado una base de datos SQL en un servidor (aunque también podría ejecutarse en una maquina local), y que se ha integrado y descubierto un controlador JDBC apropiado para esa base de datos. Existen varias bases de datos SQL gratuitas disponibles, y algunas se instalan incluso automáticamente con varias versiones de Linux.

Cada una es responsable de elegir la base de datos y de ubicar el controlador JDBC; este ejemplo está basado en una base de datos SQL llamada "Fuga".

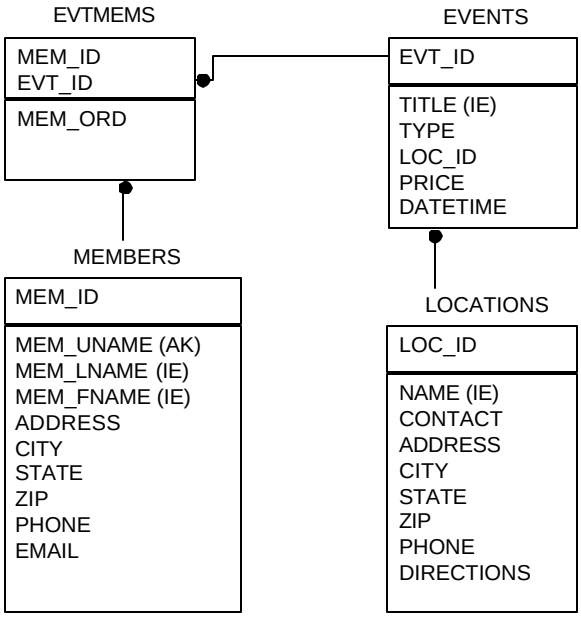
Para facilitar los cambios en la información de conexión, el controlador de la base de datos, el URL de la misma, el nombre de usuario y la contraseña se colocaron en una clase diferente:

```
//: c15:jdbc:CIDConnect.java
// Database connection information for
// the community interests database (CID).

public class CIDConnect {
  // All the information specific to CloudScape:
  public static String dbDriver =
    "COM.cloudscape.core.JDBCDriver";
  public static String dbURL =
    "jdbc:cloudscape:d:/docs/_work/JSapienDB";
  public static String user = "";
  public static String password = "";
} ///:~
```

En este ejemplo, no hay protección por contraseñas en la base de datos, por lo que el nombre de usuario y la contraseña están vacíos.

[2] Creado por Dave Bartlett.

La base de datos consiste en un conjunto de tablas con la estructura que se muestra:



"Miembros" contiene la información de miembros de la comunidad, "Eventos" y "Localizaciones" contienen información sobre las actividades y cuándo tendrán lugar, y "Evtmiems" conecta los eventos con los miembros a los que les gustaría asistir. Puede verse que un miembro de datos de una tabla produce una clave en la otra.

La clase siguiente contiene las cadenas SQL que crearán estas tablas de bases de datos (véase una guía de SQL si se desea obtener explicación del código SQL):

```
//: c15:jdbc:CIDSQL.java
// SQL strings to create the tables for the CID.

public class CIDSQL {
  public static String[] sql = {
    // Create the MEMBERS table:
    "drop table MEMBERS",
    "create table MEMBERS " +
    "(MEM_ID INTEGER primary key, " +
    "MEM_UNAME VARCHAR(12) not null unique, "+
    "MEM_LNAME VARCHAR(40), " +
    "MEM_FNAME VARCHAR(20), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE CHAR(4), " +
    "ZIP CHAR(5), " +
    "PHONE CHAR(12), " +
    "EMAIL VARCHAR(30))",
```

```
    "create unique index " +
    "LNAME_IDX on MEMBERS(MEM_LNAME)",
    // Create the EVENTS table
    "drop table EVENTS",
    "create table EVENTS " +
    "(EVT_ID INTEGER primary key, " +
    "EVT_TITLE VARCHAR(30) not null, " +
    "EVT_TYPE VARCHAR(20), " +
    "LOC_ID INTEGER, " +
    "PRICE DECIMAL, " +
    "DATETIME TIMESTAMP)",
    "create unique index " +
    "TITLE_IDX on EVENTS(EVT_TITLE)",
    // Create the EVTMEMS table
    "drop table EVTMEMS",
    "create table EVTMEMS " +
    "(MEM_ID INTEGER not null, " +
    "EVT_ID INTEGER not null, " +
    "MEM_ORD INTEGER)",
    "create unique index " +
    "EVTMEM_IDX on EVTMEMS(MEM_ID, EVT_ID)",
    // Create the LOCATIONS table
    "drop table LOCATIONS",
    "create table LOCATIONS " +
    "(LOC_ID INTEGER primary key, " +
    "LOC_NAME VARCHAR(30) not null, " +
    "CONTACT VARCHAR(50), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE VARCHAR(4), " +
    "ZIP VARCHAR(5), " +
    "PHONE CHAR(12), " +
    "DIRECTIONS VARCHAR(4096))",
    "create unique index " +
    "NAME_IDX on LOCATIONS(LOC_NAME)",
  };
} ///:~
```

El programa siguiente usa la información de **ConectarCID** y **CIDSQL** para cargar el controlador JDBC y establecer la conexión a la base de datos y crear después la estructura de tablas del diagrama de arriba. Para conectar con la base de datos se invoca al método **static DriverManager.getConnection**(), pasándole el URL de la base de datos, el nombre de usuario y una contraseña para entrar en la misma. Al retornar, se obtiene un objeto **Connection** que puede usarse para hacer consultas y manipular la base de datos. Una vez establecida la conexión se puede simplemente meter el SQL en la base de datos, en este caso, recorriendo el array **CIDSQL**. Sin embargo, la primera vez que se ejecute el programa, el comando "drop table" fallará, causando una excepción, que es capturada, y de la que se informa, para finalmente ignorarla. La razón del comando "drop table" es

permitir una experimentación sencilla: se puede modificar el SQL que define las tablas y después volver a ejecutar el programa, lo que hará que las viejas tablas sean reemplazadas por las nuevas.

En este ejemplo, tiene sentido dejar que se lancen las excepciones a la consola:

```
//: c15:jdbc:CIDCreateTables.java
// Creates database tables for the
// community interests database.
import java.sql.*;

public class CIDCreateTables {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException,
  IllegalAccessException {
    // Load the driver (registers itself)
    Class.forName(CIDConnect.dbDriver);
    Connection c = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    Statement s = c.createStatement();
    for(int i = 0; i < CIDSQL.sql.length; i++) {
      System.out.println(CIDSQL.sql[i]);
      try {
        s.executeUpdate(CIDSQL.sql[i]);
      } catch(SQLException sqlEx) {
        System.err.println(
          "Probably a 'drop table' failed");
      }
    }
    s.close();
    c.close();
  }
} ///:~
```

Nótese que se pueden controlar todos los cambios en la base de datos, cambiando **Strings** en la tabla CIDSQL, sin modificar **CrearTablasCID**.

El método **executeUpdate**() devolverá generalmente el número de las filas afectadas por la sentencia SQL. Este método se usa más frecuentemente para ejecutar sentencias `INSERT`, `UPDATE` o `DELETE`, para modificar Una O más columnas. Para sentencias Como `CREATE TABLE`, `DROP TABLE`, y `CREATE INDEX`, **executeUpdate**() siempre devuelve cero.

Para probar la base de datos, se carga con algunos datos de ejemplo. Esto requiere una serie de `INSERTS` seguidos de una `SELECT` para producir un resultado conjunto. Para facilitar las adiciones y cambios a los datos de prueba, éste se dispone en un array bidimensional de **Objects**, y el método **executeInsert**()

puede usar después la información de una columna de la tabla para crear el comando SQL apropiado.

```
//: c15:jdbc:LoadDB.java
// Loads and tests the database.
import java.sql.*;

class TestSet {
  Object[][] data = {
    { "MEMBERS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MEMBERS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MEMBERS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
      "123 Downunder Lane",
      "Sydney", "NSW", "12345",
      "123.456.7890", "rcastaneda@you.net" },
    { "LOCATIONS", new Integer(1),
      "Center for Arts",
      "Betty Wright", "123 Elk Ave.",
      "Crested Butte", "CO", "81224",
      "123.456.7890",
      "Go this way then that." },
    { "LOCATIONS", new Integer(2),
      "Witts End Conference Center",
      "John Wittig", "123 Music Drive",
      "Zoneville", "PA", "19123",
      "123.456.7890",
      "Go that way then this." },
    { "EVENTS", new Integer(1),
      "Project Management Myths",
      "Software Development",
      new Integer(1), new Float(2.50),
      "2000-07-17 19:30:00" },
    { "EVENTS", new Integer(2),
      "Life of the Crested Dog",
      "Archeology",
      new Integer(2), new Float(0.00),
      "2000-07-19 19:00:00" },
    // Match some people with events
    {  "EVTMEMS",
      new Integer(1),  // Dave is going to
      new Integer(1),  // the Software event.
```

```
      new Integer(0) },
    { "EVTMEMS",
      new Integer(2),  // Bruce is going to
      new Integer(2),  // the Archeology event.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(3),  // Robert is going to
      new Integer(1),  // the Software event.
      new Integer(1) },
    { "EVTMEMS",
      new Integer(3), // ... and
      new Integer(2), // the Archeology event.
      new Integer(1) },
  };
  // Use the default data set:
  public TestSet() {}
  // Use a different data set:
  public TestSet(Object[][] dat) { data = dat; }
}

public class LoadDB {
  Statement statement;
  Connection connection;
  TestSet tset;
  public LoadDB(TestSet t) throws SQLException {
    tset = t;
    try {
      // Load the driver (registers itself)
      Class.forName(CIDConnect.dbDriver);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    connection = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    statement = connection.createStatement();
  }
  public void cleanup() throws SQLException {
    statement.close();
    connection.close();
  }
  public void executeInsert(Object[] data) {
    String sql = "insert into "
      + data[0] + " values(";
    for(int i = 1; i < data.length; i++) {
      if(data[i] instanceof String)
        sql += "'" + data[i] + "'";
      else
        sql += data[i];
      if(i < data.length - 1)
```

```
        sql += ", ";
      }
      sql += ')';
      System.out.println(sql);
      try {
        statement.executeUpdate(sql);
      } catch(SQLException sqlEx) {
        System.err.println("Insert failed.");
        while (sqlEx != null) {
          System.err.println(sqlEx.toString());
          sqlEx = sqlEx.getNextException();
        }
      }
    }
    public void load() {
      for(int i = 0; i< tset.data.length; i++)
        executeInsert(tset.data[i]);
    }
    // Throw exceptions out to console:
    public static void main(String[] args)
    throws SQLException {
      LoadDB db = new LoadDB(new TestSet());
      db.load();
      try {
        // Get a ResultSet from the loaded database:
        ResultSet rs = db.statement.executeQuery(
          "select " +
          "e.EVT_TITLE, m.MEM_LNAME, m.MEM_FNAME "+
          "from EVENTS e, MEMBERS m, EVTMEMS em " +
          "where em.EVT_ID = 2 " +
          "and e.EVT_ID = em.EVT_ID " +
          "and m.MEM_ID = em.MEM_ID");
        while (rs.next())
          System.out.println(
            rs.getString(1) + "  " +
            rs.getString(2) + ", " +
            rs.getString(3));
      } finally {
        db.cleanup();
      }
    }
} ///:~
```

La clase **PruebaConjunto** contiene un conjunto de datos por defecto producido al usar el constructor por defecto; sin embargo, también se puede crear un objeto **PruebaConjunto** usando un conjunto de datos alternativo con el segundo constructor. El conjunto de datos se guarda en un array bidimensional de **Object** porque puede ser de cualquier tipo, incluidos **Strings** o tipos numéricos. El método **ejecutarInsertar**() utiliza RTTI para distinguir entre datos **String** (que

deben ir entre comillas) y los no-**String** a medida que construye el comando SQL a partir de los datos. Tras imprimir este programa en la consola, se usa **executeUpdate**( ) para enviarlo a la base de datos.

El constructor de **CargarBD** hace la conexión, y **cargar**() recorre los datos y llama a **ejecutarInsertar**() por cada registro. El método **limpiar**() cierra la sentencia y la conexión; para garantizar que se invoque, éste se ubica dentro de una cláusula **finally**.

Una vez cargada la base de datos, una sentencia **executeQuery**() produce el resultado conjunto de ejemplo. Puesto que la consulta combina varias tablas, es un ejemplo de un **join**.

Hay más información de JDBC disponible en los documentos electrónicos que vienen como parte de la distribución de Java de Sun. Además, se puede encontrar más en el libro JDBC Database Access with Java (Hamilton, Cattel, y Fisher, Addison-Wesley, 1997). También suelen aparecer otros libros sobre este tema con bastante frecuencia.

Servlets

El acceso de clientes desde Internet o intranets corporativas es una forma segura de permitir a varios usuarios acceder a datos y recursos de forma sencilla **[3]**. Este tipo de acceso está basado en el uso de los estándares Hypertext Markup Language (HTML) e Hypertext Transfer Protocol (HTTP) de la World Wide Web por parte de los clientes. El conjunto de API Servlet abstrae un marco de solución común para responder a peticiones H'ITP.

[3] Dave Bartlett contribuyó desarrollo de este material, y también en la sección JSP.

Tradicionalmente, la forma de gestionar un problema como el de permitir a un cliente de Internet actualizar una base de datos es crear una página HTML con campos de texto y un botón de "enviar". El usuario teclea la información apropiada en los campos de texto y presiona el botón "enviar". Los datos se envían junto con una URL que dice al servidor qué hacer con los datos especificando la ubicación de un programa Common Gateway Interface (CGI) que ejecuta el servidor, proporcionando al programa los datos al ser invocado. El programa CGI suele estar escrito en Perl, Python, C, C++ o cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar. Esto es todo lo que es proporcionado por el servidor web: se invoca al programa CGI, y se usan flujos estándar (u, opcionalmente en caso de entrada, una variable de entorno) para la entrada y la salida. El programa CGI es responsable de todo lo demás. Primero, mira a los datos y decide si el formato es correcto. Si no, el programa CGI debe producir HTML para describir el problema; esta página se

pasa al servidor Web (vía salida estándar del programa CGI), que lo vuelve a enviar al usuario. El usuario debe generalmente salvar la página e intentarlo de nuevo. Si los datos son correctos, el programa CGI los procesa de forma adecuada, añadiéndolos quizás a una base de datos. Después, debe producir una página HTML apropiada para que el servidor web se la de- vuelva al usuario.

Sería ideal ir a una solución completamente basada en Java para este ejemplo - un applet en el lado cliente que valide y envíe los datos, y un servlet en el lado servidor para recibir y procesar los datos. Desgraciadamente, aunque se ha demostrado que los applets son una tecnología con gran soporte, su uso en la Web ha sido problemático porque no se puede confiar en que en un navegador cliente web haya una versión particular de Java disponible; de hecho, jni siquiera se puede confiar en que un navegador web soporte Java! En una Intranet, se puede requerir que esté disponible cierto soporte, lo que permite mucha mayor flexibilidad en lo que se puede hacer, pero en la Web el enfoque más seguro es manejar todo el proceso en el lado servidor y entregar HTML plano al cliente.

De esa forma, no se negará a ningún cliente el uso del sitio porque no tenga el software apropiado instalado.

Dado que los servlets proporcionan una solución excelente para soporte en el lado servidor, son una de las razones más populares para migrar a Java. No sólo proporcionan un marco que sustituye a la programación CGI (y elimina muchos problemas complicados de los CGIs), sino que todo el código gana portabilidad de plataforma gracias al uso de Java, teniendo acceso a todos los APIs de Java (excepto, por supuesto, a los que producen IGU, como Swing).

El servlet básico

La arquitectura del API servlet es la de un proveedor de servicios clásico con un método **service**() a través del cual se enviarán todas las peticiones del cliente por parte del software contenedor del servlet, y los métodos de ciclo de vida **hit**() y **destroy**(), que se invocan sólo cuando se carga y descarga el servlet (esto raramente ocurre).

```
public interface Servlet {
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

El único propósito de **getServletConfig**() es devolver un objeto **ServletConfig** que contenga los parámetros de inicialización y arranque de este servidor. El método **getServletInfo**() devuelve una cadena de caracteres que contiene información sobre el servlet, como el autor, la versión y los derechos del autor.

La clase **GenericServlet** es una implementación genérica de esta interfaz y no suele usarse. La clase **HttpServlet** es una extensión de **GenericServlet** y está diseñada específicamente para manejar el protocolo HTTP - **HttpServlet** es la que se usará la mayoría de veces.

El atributo más conveniente de la API de servlets lo constituyen los objetos auxiliares que vienen con la clase **HttpServlet** para darle soporte. Si se mira al método **service**() de la interfaz **Servlet**, se verá que tiene dos parámetros: **ServletRequest** y **ServletResponse**. Con la clase **HttpServlet** se extienden estos dos objetos para HTTP: **HttpServletRequest** y **HttpServletResponse**. He aquí un ejemplo simple que muestra el uso de **HttpServletResponse**:

```
//: c15:servlets:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
  int i = 0; // Servlet "persistence"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.print("<HEAD><TITLE>");
    out.print("A server-side strategy");
    out.print("</TITLE></HEAD><BODY>");
    out.print("<h1>Servlets Rule! " + i++);
    out.print("</h1></BODY>");
    out.close();
  }
} ///:~
```

ReglasServlets es casi tan simple como puede serlo un servlet. El servlet sólo se inicializa una vez llamando a su método **init**(), al cargar el servidor tras arrancar por primera vez el contenedor de servlets. Cuando un cliente hace una petición a una URL que resulta representar un servlet, el contenedor de servlets intercepta la petición y hace una llamada al método **service**(), tras configurar los objetos **HttpServletRequest** y **HttpServletResponse**.

La responsabilidad principal del método **service**() es interactuar con la petición HTTP que ha enviado el cliente, y construir una respuesta HTTP basada en los

atributos contenidos dentro de la petición. **ReglasServlets** sólo manipula el objeto respuesta sin mirar a lo que el cliente puede haber enviado.

Tras configurar el tipo de contenido de la respuesta (cosa que debe hacerse siempre antes de procurar el **Writer** u **OutputStream**), el método **getWriter**() del objeto respuesta produce un objeto **PrintWriter**, que se usa para escribir información de respuesta basada en caracteres (alternativamente, **getOutputStream**() produce un **OutputStream**, usado para respuesta binaria, que sólo se usa en soluciones más especializadas).

El resto del programa simplemente manda HTML de vuelta al cliente (se asume que el lector entiende HTML, por lo que no se explica esa parte) como una secuencia de **Strings**. Sin embargo, nótese la inclusión del "contador de accesos" representado por la variable **i**. Este se convierte automáticamente en un **String** en la sentencia **print**().

Cuando se ejecuta el programa, se verá que se mantiene el valor de i entre las peticiones al servidor. Ésta es una propiedad esencial de los servlets: puesto que en el contenedor sólo se carga un servlet de cada clase, y éste nunca se descarga (a menos que finalice el contenedor de servlets, lo cual es algo que normalmente sólo ocurre si se reinicia la máquina servidora), cualquier campo de esa clase servidora se convierte en un objeto persistente! Esto significa que se pueden mantener sin esfuerzo valores entre peticiones servlets, mientras que con CGI había que escribir los valores al disco para preservarlos, lo que requería una cantidad de trabajo bastante elevada para que funcionara con éxito, y solía producir soluciones exclusivas para una plataforma.

Por supuesto, en ocasiones, el servidor web, y por consiguiente, el contenedor de servlets, tienen que ser reiniciados como parte del mantenimiento o por culpa de un fallo de corriente. Para evitar perder cualquier información persistente, se invoca automáticamente a los métodos **init**() y **destroy**() del servlet siempre que se carga o descarga el servlet, proporcionando la oportunidad de salvar los datos durante el apagado, y restaurarlos tras el nuevo arranque. El contenedor de servlets llama al método **destroy**() al terminarse a sí mismo, por lo que siempre se logra una oportunidad de salvar la información valiosa, en la medida en que la máquina servidora esté configurada de forma inteligente.

Hay otro aspecto del uso de **HttpServlet**. Esta clase proporciona métodos **doGet**() y **dopost**(), que diferencian entre un envío CGI “GET” del cliente, y un CGI "POST". GET y POST simplemente varían en los detalles de la forma en que envían los datos, que es algo que preferimos ignorar. Sin embargo, la mayoría de información publicada que hemos visto parece ser favorable a la creación de métodos **doGet**() y **dopost**() separados en vez de un método **service**() genérico, que maneje los dos casos. Este favoritismo parece bastante común, pero nunca lo he visto explicado de forma que nos haga creer que se deba a algo más que a la inercia de los programadores de CGI que están habituados a prestar atención a si

se está usando **GET** o **POST**. Por tanto, y por mantener el espíritu de "hacer todo siempre de la forma más simple que funcione" **[4]** o, simplemente usaremos el método **service**() en estos ejemplos, y que se encargue de los **GET** frente a **POST**. Sin embargo, hay que mantener presente que podríamos dejarnos algo, por lo que de hecho sí que podría haber una buena razón para usar en su lugar **doGet**() o **dopost**().

[4] Uno de los eslóganes principales de la Programación Extrema **(XP).** Ver http:www.xprogramming.com.

Siempre que se envía un formulario a un servidor, el **HttpServletRequest** viene precargado con todos los datos del formulario, almacenados como pares clave-valor. Si se conocen los nombres de los campos, pueden usarse directamente con el método **getParameter**() para buscar los valores. También se puede lograr un **Enumeration** (la forma antigua del **Iterator**) para los nombres de los campos, como se muestra en el ejemplo siguiente. Este ejemplo también demuestra cómo se puede usar un único servlet para producir la página que contiene el formulario y para responder a la pagina (más adelante se verá una solución mejor, con JSP). Si la **Enumeration** está vacía, no hay campos; esto significa que no se envió ningún formulario. En este caso, se produce el formulario y el botón de enviar reinvocará al mismo servlet. Sin embargo, si los campos existen, se muestran.

```
//: c15:servlets:EchoForm.java
// Dumps the name-value pairs of any HTML form
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    Enumeration flds = req.getParameterNames();
    if(!flds.hasMoreElements()) {
      // No form submitted -- create one:
      out.print("<html>");
      out.print("<form method=\"POST\"" +
        " action=\"EchoForm\">");
      for(int i = 0; i < 10; i++)
        out.print("<b>Field" + i + "</b> " +
          "<input type=\"text\""+
          " size=\"20\" name=\"Field" + i +
          "\" value=\"Value" + i + "\"><br>");
      out.print("<INPUT TYPE=submit name=submit"+
      " Value=\"Submit\"></form></html>");
    } else {
```

```
      out.print("<h1>Your form contained:</h1>");
      while(flds.hasMoreElements()) {
        String field= (String)flds.nextElement();
        String value= req.getParameter(field);
        out.print(field + " = " + value+ "<br>");
      }
    }
    out.close();
  }
} ///:~
```

Una pega que se verá aquí es que Java no parece haber sido diseñado con el procesamiento de cadenas de caracteres en mente -el formateo de la página de retorno no supone más que quebraderos de cabeza debido a los saltos de línea, las marcas de escape y los signos "+" necesarios para construir objetos **String**. Con una página HTML extensa no sería razonable codificarla directamente en Java. Una solución es mantener la página como un archivo de texto separado, y abrirla y pasársela al servidor web. Si se tiene que llevar a cabo cualquier tipo de sustitución de los contenidos de la página, la solución no es mucho mejor debido al procesamiento tan pobre de las cadenas de texto en Java. En estos casos, probablemente se hará mejor usando una solución más apropiada (nuestra elección sería Phyton; hay una versión que se fija en Java llamada JPython) para generar la página de respuesta.

Servlets y multihilo

El contenedor de servlets tiene un conjunto de hilos que irá despachando para gestionar las peticiones de los clientes. Es bastante probable que dos clientes que lleguen al mismo tiempo puedan ser procesados por **service**() a la vez. Por consiguiente, el método **service**() debe estar escrito de forma segura para hilos. Cualquier acceso a recursos comunes (archivos, bases de datos) necesitará estar protegido haciendo uso de la palabra clave **synchronized**.

El siguiente ejemplo simple pone una cláusula **synchronized** en torno al método **sleep**() del hilo.

Éste bloqueará al resto de los hilos hasta que se haya agotado el tiempo asignado (5 segundos). Al probar esto, deberíamos arrancar varias instancias del navegador y acceder a este servlet tan rápido como se pueda en cada una -se verá que cada una tiene que esperar hasta que le llega su turno.

```
//: c15:servlets:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class ThreadServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
  }
} ///:~
```

También es posible sincronizar todo el servlet poniendo la palabra clave **synchronized** delante del método **service**(). De hecho, la única razón de usar la cláusula **synchronized** en su lugar es por si la sección crítica está en un cauce de ejecución que podría no ejecutarse. En ese caso, se podría evitar también la sobrecarga de tener que sincronizar cada vez utilizando una cláusula **synchronized**. De otra forma, todos los hilos tendrían que esperar de todas formas, por lo que también se podría sincronizar todo el método.

Gestionar sesiones con servlets

HTTP es un protocolo "sin sesión", por lo que no se puede decir desde un acceso al servidor a otro si se trata de la misma persona que está accediendo repetidamente al sitio, o si se trata de una persona completamente diferente. Se ha invertido mucho esfuerzo en mecanismos que permitirán a los desarrolladores web llevar a cabo un seguimiento de las sesiones. Las compañías no podrían hacer comercio electrónico sin mantener un seguimiento de un cliente y por ejemplo, de los elementos que éste ha introducido en su carro de la compra.

Hay bastantes métodos para llevar a cabo el seguimiento de sesiones, pero el más común es con "cookies" persistentes, que son una parte integral de los estándares Internet. El Grupo de Trabajo HTTP de la Internet Engineering Task Force ha descrito las cookies en el estándar oficial en RFC 2109 (*ds.internic.net/rfc/rfc2109.txt* o compruebe *www.cookiecentral.com*).

Una cookie no es más que una pequeña pieza de información enviada por un servidor web a un navegador. El navegador almacena la cookie en el disco local y cuando se hace otra llamada al URL con la que está asociada la cookie, éste se envía junto con la llamada, proporcionando así que la información deseada vuelva

a ese servidor (generalmente, proporcionando alguna manera de que el servidor pueda determinar que es uno el que llama). Los clientes, sin embargo, pueden desactivar la habilidad del navegador para aceptar cookies. Si el sitio debe llevar un seguimiento de un cliente que ha desactivado las cookies, hay que incorporar a mano otro método de seguimiento de sesiones (reescritura de URL o campos de formulario ocultos), puesto que las capacidades de seguimiento de sesiones construidas en el API servlet están diseñadas para cookies.

La clase Cookie

El API servlet (en versiones 2.0 y superior) proporciona la clase **Cookie**. Esta clase incorpora todos los detalles de cabecera HTTP y permite el establecimiento de varios atributos de cookie. Utilizar la cookie es simplemente un problema de añadirla al objeto respuesta. El constructor toma un nombre de cookie como primer parámetro y un valor como segundo. Las cookies se añaden al objeto respuesta antes de enviar ningún contenido.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

Las cookies suelen recubrirse invocando al método **getCookies**() del objeto **HttpServletRequest**, que devuelve un array de objetos cookie.

```
Cookie[] cookies = req.getCookies();
```

Después se puede llamar a **getValue**() para cada cookie, para producir un **String** que contenga los contenidos de la cookie. En el ejemplo de arriba, **getValue**('"TIJava") producirá un **String** de valor "2000".

La clase Session

Una sesión es una o más solicitudes de páginas por parte de un cliente a un sitio web durante un periodo definido de tiempo. Si uno compra, por ejemplo, ultramarinos en línea, se desea que una sesión dure todo el periodo de tiempo desde que se añade al primer elemento a "Mi carrito de la compra" hasta el momento en el que se compruebe todo. Cada elemento que se añada al carrito de la compra vendrá a producir una nueva conexión HTTP, que no tiene conocimiento de conexiones previas o de los elementos del carrito de la compra. Para compensar esta falta de información, los mecanismos suministrados por la especificación de cookies permiten al servlet llevar a cabo seguimiento de sesiones.

Un objeto **Session servlet** vive en la parte servidora del canal de comunicación; su meta es capturar datos útiles sobre este cliente a medida que el cliente recorre e interactúa con el sitio web. Esta información puede ser pertinente para la sesión actual, como los elementos del carro de la compra, o pueden ser datos como la

información de autentificación introducida cuando el cliente entró por primera vez en el sitio web, y que no debería ser reintroducida durante un conjunto de transacciones particular.

La clase **Session** del API servlet usa la clase **Cookie** para hacer su trabajo. Sin embargo, todo el objeto **Session** necesita algún tipo de identificador único almacenado en el cliente y que se pasa al servidor. Los sitios web también pueden usar otros tipos de seguimiento de sesión, pero estos mecanismos serán más difíciles de implementar al no estar encapsulados en el API servlet (es decir, hay que escribirlos a mano para poder enfrentarse a la situación de deshabilitación de las cookies por parte del cliente)

He aquí un ejemplo que implementa seguimiento de sesión con el API servlet:

```
//: c15:servlets:SessionPeek.java
// Using the HttpSession class.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retrieve Session Object before any
    // output is sent to the client.
    HttpSession session = req.getSession();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HEAD><TITLE> SessionPeek ");
    out.println(" </TITLE></HEAD><BODY>");
    out.println("<h1> SessionPeek </h1>");
    // A simple hit counter for this session.
    Integer ival = (Integer)
      session.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    session.setAttribute("sesspeek.cntr", ival);
    out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
    out.println("<h2>");
    out.println("Saved Session Data </h2>");
    // Loop through all data in the session:
    Enumeration sesNames =
      session.getAttributeNames();
    while(sesNames.hasMoreElements()) {
```

```
      String name =
        sesNames.nextElement().toString();
      Object value = session.getAttribute(name);
      out.println(name + " = " + value + "<br>");
    }
    out.println("<h3> Session Statistics </h3>");
    out.println("Session ID: "
      + session.getId() + "<br>");
    out.println("New Session: " + session.isNew()
      + "<br>");
    out.println("Creation Time: "
      + session.getCreationTime());
    out.println("<I>(" +
      new Date(session.getCreationTime())
      + ")</I><br>");
    out.println("Last Accessed Time: " +
      session.getLastAccessedTime());
    out.println("<I>(" +
      new Date(session.getLastAccessedTime())
      + ")</I><br>");
    out.println("Session Inactive Interval: "
      + session.getMaxInactiveInterval());
    out.println("Session ID in Request: "
      + req.getRequestedSessionId() + "<br>");
    out.println("Is session id from Cookie: "
      + req.isRequestedSessionIdFromCookie()
      + "<br>");
    out.println("Is session id from URL: "
      + req.isRequestedSessionIdFromURL()
      + "<br>");
    out.println("Is session id valid: "
      + req.isRequestedSessionIdValid()
      + "<br>");
    out.println("</BODY>");
    out.close();
  }
  public String getServletInfo() {
    return "A session tracking servlet";
  }
} ///:~
```

Dentro del método **service**(), se invoca a **getSession**() para el objeto petición, que devuelve el objeto **Session** asociado con esta petición. El objeto **Session** no viaja a través de la red, sino que en vez de ello, vive en el servidor asociado con un cliente y sus peticiones.

El método **getSession**() viene en dos versiones: la de sin parámetros, usada aquí, y **getSession**(boolean). Usar **getSession**(true) equivale a **getSession**(). La única razón del **boolean** es para establecer si se desea crear el objeto sesión

si no es encontrado. La llamada más habitual es **getSession**(true), razón de la existencia de **getSession**().

El objeto **Session**, si no es nuevo, nos dará detalles sobre el cliente provenientes de sus visitas anteriores. Si el objeto **Session** es nuevo, el programa comenzará a recopilar información sobre las actividades del cliente en esta visita. La captura de esta información del cliente se hace mediante los métodos **setAttribute**() y **getAttribute**() del objeto de sesión.

```
java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name,
                  java.lang.Object value)
```

El objeto **Session** utiliza un emparejamiento nombre-valor simple para cargar información. El nombre es un **String**, y el valor puede ser cualquier objeto derivado de **java.lang.0bject**. **SeguirSesion** mantiene un seguimiento de las veces que ha vuelto el cliente durante esta sesión. Esto se hace con un objeto **Integer** denominado **sesspeek.cntr**. Si no se encuentra el nombre se crea un **Integer** de valor uno, si no, se crea un **Integer** con el valor incrementado respecto del **Integer** anteriormente guardado. Si se usa la misma clave en una llamada a setA**t**tribute(), el objeto nuevo sobrescribe el viejo. El contador incrementado se usa para mostrar el número de veces que ha visitado el cliente durante esta sesión.

El método **getAttributeNames**() está relacionado con **getAttribute**() y **setAttribute**(); devuelve una enumeración de los nombres de objetos vinculados al objeto **Session**. Un bucle **while** en **Seguirsesion** muestra este método en acción.

Uno podría preguntarse durante cuánto tiempo puede permanecer inactivo un objeto **Session**. La respuesta depende del contenedor de servlets que se esté usando; generalmente vienen por defecto a 30 minutos (1.800 segundos), que es lo que debería verse desde la llamada a **SeguirSesion** hasta **getMaxInactiveInterval**(). Las pruebas parecen producir resultados variados entre contenedores de servlets. En ocasiones, el objeto **Session** puede permanecer inactivo durante toda la noche, pero nunca hemos visto ningún caso en el que el objeto **Session** desaparezca en un tiempo menor al especificado por el intervalo de inactividad. Esto se puede probar estableciendo el valor de este intervalo con **setMaxInactiveInterval**() a 5 segundos y ver si el objeto **Session** se cuelga o es eliminado en el tiempo apropiado. Éste puede constituir un atributo a investigar al seleccionar un contenedor de servlets.

Ejecutar los ejemplos de servlets

Si el lector no está trabajando con un servidor de aplicaciones que maneje automáticamente las tecnologías servlet y JSP de Sun, puede descargar la implementación Tomcat de los servlets y JSPs de Java, que es una implementación gratuita, de código fuente abierto, y que es la implementación de referencia oficial de Sun. Puede encontrarse en *jakarta.apache.org.*
Siga las instrucciones de instalación de la implementación Tomcat, después edite el archivo **server.xml** para que apunte a la localización de su árbol de directorios en el que se ubicarán los servlets. Una vez que se arranque el programa Tomcat, se pueden probar los programas de servlets.

Ésta sólo ha sido una somera introducción a los servlets; hay libros enteros sobre esta materia. Sin embargo, esta introducción debería proporcionarse las suficientes ideas como para que se inicie.

Además, muchas ideas de la siguiente sección son retrocompatibles con los servlets.

Java Server Pages

Las Java Server Pages (JSP) son una extensión del estándar Java definido sobre las Extensiones de servlets. La meta de las JSP es la creación y gestión simplificada de páginas web dinámicas. La implementación de referencia Tomcat anteriormente mencionada y disponible gratuitamente en jakarta.apache.org soporta JSP automáticamente.

Las JSP permiten combinar el HTML de una página web con fragmentos de código Java en el mismo documento. El código Java está rodeado de etiquetas especiales que indican al contenedor de JSP que debería usar el código para generar un servlet o parte de uno. El beneficio de las JSP es que se puede mantener un documento único que representa tanto la página como el código Java que habilita. La pega es que el mantenedor de la página JSP debe dominar tanto HTML como Java (sin embargo, los entornos constructores de IGU para JSP deberían aparecer en breve).

La primera vez que el contenedor de JSP carga un JSP (que suele estar asociado con, o ser parte de, un servidor web) se genera, compila y carga automáticamente en el contenedor de servlets el código servlet necesario para cumplimentar las etiquetas JSP. Las porciones estáticas de la página HTML se producen enviando objetos **String** estáticos a **write**(). Las porciones dinámicas se incluyen directamente en el servlet.

A partir de ese momento, y mientras el código JSP de la página no se modifique, se comporta como si fuera una página HTML estática con servlets asociados (sin embargo, el servlet genera todo el código HTML). Si se modifica el código fuente de la JSP, ésta se recompila y recarga automáticamente la siguiente vez que se

solicite esa página. Por supuesto, debido a todo este dinamismo, se apreciará una respuesta lenta en el acceso por primera vez a una JSP. Sin embargo, dado que una JSP suele usarse mucho más a menudo que ser cambiada, normalmente uno no se verá afectado por este retraso.

La estructura de una página JSP está a caballo entre la de un servlet y la de una página HTML. Las etiquetas JSP empiezan y acaban con *"2'* y ">", como las etiquetas HTML, pero las etiquetas también incluyen símbolos de porcentaje, de forma que todas las etiquetas JSP se delimitan por:

```
<% JSP code here %>
```

El signo de porcentaje precedente puede ir seguido de otros caracteres que determinen el tipo específico de código JSP de la etiqueta.

He aquí un ejemplo extremadamente simple que usa una llamada a la biblioteca estándar Java para lograr la hora actual en milisegundos, que es después dividida por mil para producir la hora en segundos. Dado que se usa una expresión JSP (la <%=), el resultado del cálculo se fuerza a un **Stnng** y se coloca en la página web generada:

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

En los ejemplos de JSP de este libro no se incluyen la primera y última líneas en el archivo de código extraído y ubicado en el árbol de códigos fuente del libro.
Cuando el cliente crea una petición de la página JSP hay que haber configurado el servidor web para confiar en la petición del contenedor de JSP que posteriormente invoca a la página. Como se mencionó arriba, la primera vez que se invoca la página, el contenedor de JSP genera y compila los componentes especificados por la página como uno o más servlets. En el ejemplo de arriba, el servlet contendrá código para configurar el objeto **HttpServletResponse**, producir un objeto **PrintWriter** (que siempre se denomina out), y después convertir el cómputo de tiempo en un **String** que es enviado a out. Como puede verse, todo esto se logra con una sentencia muy sucinta, pero el programador HTML/diseñador web medio no tendrá las aptitudes necesarias para escribir semejante código.

Objetos implícitos

Los servlets incluyen clases que proporcionan utilidades convenientes, como **HttpServletRequest**, **HttpServletResponse**, **Session**, etc. Los objetos de estas clases están construidos en la especificación JSP y automáticamente disponibles para ser usados en un JSP, sin tener que escribir ninguna línea extra de código. Los objetos implícitos de un JSP se detallan en la tabla siguiente:

Variable implícita	**De tipo (javax.servlet)**	**Descripción**	**Ámbito**
request	Subtipo de protocolo dependiente de **HttpServletRequest**	La petición que dispara la invocación del servicio.	petición
response	Subtipo de protocolo dependiente de **HttpServletResponse**	La respuesta a la petición.	página
pageContext	**jsp.PageContext**	El contexto de la página encapsula facetas dependientes de la implementación y proporciona métodos de conveniencia y acceso de espacio de nombres a este JSP.	página
session	Subtipo de protocolo dependiente de **http.HttpSession**	El objeto sesión creado para el cliente que hace la petición. Ver el objeto servlet **Session**.	sesión
application	**ServletContext**	El contexto de servlet obtenido del objeto de configuración del servlet (por ejemplo, **getServletConfig**( ), **getContext**( )).	aplicación
out	**jsp.JspWriter**	El objeto que escribe en el flujo de salida.	página
config	**ServletConfig**	El **ServletConñg** de este JSP.	página
page	**java.lang.Object**	La instancia de la	página

		clase de implementación de esta página que procese la petición actual.	

El ámbito de cada objeto puede variar significativamente. Por ejemplo, el objeto **session** tiene un ámbito que excede al de una página, pues puede abarcar varias peticiones de clientes y páginas. El objeto **application** puede proporcionar servicios a un grupo de páginas JSP que representan juntas una aplicación web.

Directivas JSP

Las directivas son mensajes al contenedor de JSP y se delimitan por la "@":

```
<%@ directive {attr="value"}* %>
```

Las directivas no envían nada al flujo out, pero son importantes al configurar los atributos y dependencias de una página JSP con el contenedor de JSP. Por ejemplo, la línea:

```
<%@ page language="java" %>
```

dice que el lenguaje de escritura de guiones que se está usando en la página JSP es Java. De hecho, la especificación de Java sólo describe las semánticas de guiones para atributos de lenguaje iguales a "Java". La intención de esta directiva es aportar flexibilidad a la tecnología JSP. En el futuro, si hubiera que elegir otro lenguaje, como Python (un buen lenguaje de escritura de guiones), entonces este lenguaje debería soportar el Entorno de Tiempo de Ejecución de Java, exponiendo el modelo de objetos de la tecnología Java al entorno de escritura de guiones, especialmente las variables implícitas definidas arriba, las propiedades de los JavaBeans y los métodos públicos.

La directiva más importante es la directiva de página. Define un número de atributos dependientes de la página y comunica estos atributos al contenedor de JSP. Entre estos atributos se incluye: **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** y **errorpage**. Por ejemplo:

```
<%@ page session=”true” import=”java.util.*” %>
```

La primera línea indica que la página requiere participación en una sesión HTTP. Dado que no hemos establecido directiva de lenguaje, el contenedor de JSP usa por defecto Java y la variable de lenguaje de escritura denominada **session** es de tipo **javax.servlet.http.HttpSession.** Si la directiva hubiera sido falsa, la

variable implícita **session** no habría estado disponible. Si no se especifica la variable **session**, se pone a "**true**" por defecto.

El atributo **import** describe los tipos disponibles al entorno de escritura de guiones. Este atributo se usa igual que en el lenguaje de programación Java, por ejemplo, una lista separada por comas de expresiones **import** normales. Esta lista es importada por la implementación de la página JSP traducida y está disponible para el entorno de escritura de guiones. De nuevo, esto sólo está definido verdaderamente cuando el valor de la directiva de lenguaje es "**java**".

Elementos de escritura de guiones JSP

Una vez que se han usado las directivas para establecer el entorno de escritura de guiones se puede usar los elementos del lenguaje de escritura de guiones. JSP 1.1 tiene tres elementos de lenguaje de escritura de guiones -declaraciones, *scriptlets* y expresiones. Una declaración declarará elementos, un *scriptlet* es un fragmento de sentencia y una expresión es una expresión completa del lenguaje. En JSP cada elemento de escritura de guiones empieza por "<%". La sintaxis de cada una es:

```
<%! declaration %>
<%  scriptlet   %>
<%= expression  %>
```

El espacio en blanco tras "<%!", "<%, "<%=" y antes de "%>" es opcional.

Todas estas etiquetas se basan en XML; se podría incluso decir que una página JSP puede corresponderse con un documento XML. La sintaxis equivalente en XML para los elementos de escritura de guiones de arriba sería:

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet>   scriptlet   </jsp:scriptlet>
<jsp:expression>  expression  </jsp:expression>
```

Además, hay dos tipos de comentarios:

```
<%-- jsp comment --%>
<!-- html comment -->
```

La primera forma permite añadir comentarios a las páginas fuente JSP que no aparecerán de ninguna forma en el HTML que se envía al cliente. Por supuesto, la segunda forma de comentario no es específica de los JSP - e s simplemente un comentario HTML ordinario. Lo interesante es que se puede insertar código JSP dentro de un comentario HTML, y el comentario se producirá en la página resultante, incluyendo el resultado del código JSP.

Las declaraciones se usan para declarar variables y métodos en el lenguaje de escritura de guiones (actualmente sólo Java) usado en una página JSP. La declaración debe ser una sentencia Java completa y no puede producir ninguna salida en el flujo salida. En el ejemplo **Hola.jsp** de debajo, las declaraciones de las variables **cargaHora**, **cargaFecha** y **conteoAccesos** son sentencias Java completas que declaran e inicializan nuevas variables.

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<%-- These are declarations: --%>
<%!
    long loadTime= System.currentTimeMillis();
    Date loadDate = new Date();
    int hitCount = 0;
%>
<html><body>
<%-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- A "scriptlet" that writes to the server
console and to the client page.
Note that the ';' is required: --%>
<%
   System.out.println("Goodbye");
   out.println("Cheerio");
%>
</body></html>
///:~
```

Cuando se ejecute este programa se verá que las variables **cargaHora**, **cargaFecha** y **conteoAccesos** guardan sus valores entre accesos a la página, por lo que son claramente campos y no variables locales.

Al final del ejemplo hay un scriptlet que escribe "Adios" a la consola servidora web y "Cheerio" al objeto **JspWriter** implícito **out**. Los scriptlets pueden contener

cualquier fragmento de código que sean sentencias Java válidas. Los scriptlets se ejecutan bajo demanda en tiempo de procesamiento.

Cuando todos los fragmentos de scritplet en un JSP dado se combinan en el orden en que aparecen en la página JSP, deberían conformar una sentencia válida según la definición del lenguaje de programación Java. Si producen o no alguna salida al flujo out depende del código del scriptlet. Uno debería ser consciente de que los scriptlets pueden producir efectos laterales al modificar objetos que son visibles para ellos.

Las expresiones JSP pueden entremezclarse con el HTML en la sección central de **Hola.jsp**. Las expresiones deben ser sentencias Java completas, que son evaluadas, convertidas a un **String** y enviadas a out. Si el resultado no puede convertirse en un **String**, se lanza una **ClassCastException**.

Extraer campos y valores

El ejemplo siguiente es similar a uno que se mostró anteriormente en la sección de servlets. La primera vez que se acceda a la página, se detecta que no se tienen campos y se devuelve una página que contiene un formulario, usando el mismo código que en el ejemplo de los servlets, pero en formato JSP. Cuando se envía el formulario con los campos rellenos al mismo URL de JSP, éste detecta los campos y los muestra. Ésta es una técnica brillante pues permite tener tanto la página que contiene el formulario para que el usuario la rellene como el código de respuesta para esa página en un único archivo, facilitando así la creación y mantenimiento.

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form. --%>
<%-- This JSP also generates the form. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { // No fields %>
    <form method="POST"
    action="DisplayFormData.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Field<%=i%>: <input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
<%  } %>
    <INPUT TYPE=submit name=submit
    value="Submit"></form>
<%} else {
    while(flds.hasMoreElements()) {
      String field = (String)flds.nextElement();
```

```
      String value = request.getParameter(field);
%>
      <li><%= field %> = <%= value %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

El aspecto más interesante de este ejemplo es que demuestra cómo puede entremezclarse el código scriptlet con el código HTML incluso hasta el punto de generar HTML dentro de un bucle **for** de Java. Esto es especialmente conveniente para construir cualquier tipo de formulario en el que, de lo contrario, se requeriría código HTML repetitivo.

Atributos JSP de página y su ámbito

Merodeando por la documentación HTML de servlets y JSP, se pueden encontrar facetas que dan información sobre el servlet o el JSP actualmente en ejecución. El ejemplo siguiente muestra uno de estos fragmentos de datos:

```
//:! c15:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {  %>
    <H3>Scope: <%= scope %> </H3>
<%  Enumeration e =
      pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

Este ejemplo también muestra el uso tanto del HTML embebido como de la escritura en out para sacar la página HTML resultante.

El primer fragmento de información que se produce es el nombre del servlet, que probablemente será simplemente "JSP, pero depende de la implementación. También se puede descubrir la versión actual del contenedor de servlets usando el objeto aplicación. Finalmente, tras establecer un atributo de sesión, se muestran los "nombres de atributo" de un ámbito en particular. Los ámbitos no se usan mucho en la mayoría de programas JSP; simplemente se muestran aquí para añadir interés al ejemplo.

Hay cuatro ámbitos de atributos, que son: el ámbito de página (ámbito 1), el ámbito de petición (ámbito 2), el ámbito de sesión (ámbito 3) -aquí, el único elemento disponible en ámbito de sesión es "Mi perro", añadido justo antes del bucle **for**, y el ámbito de aplicación (ámbito 4), basado en el objeto **ServletContext** Sólo hay un **ServletContext** por "aplicación web" por cada Máquina Virtual Java. (Una "aplicación web" es una colección de servlets y contenido instalados bajo un subconjunto del espacio de nombres URL del servidor, como **/catalog**. Esto se suele establecer utilizando un archivo de configuración.) En el ámbito de aplicación se verán objetos que representan rutas para el directorio de trabajo y el directorio temporal.

Manipular sesiones en JSP

Las sesiones se presentaron en la sección anterior de los servlets, y también están disponibles dentro de los JSP El ejemplo siguiente ejercita el objeto **session** y permite manipular la cantidad de tiempo antes de que la sesión se vuelva no válida.

```
//:! c15:jsp:SessionObject.jsp
<%--Getting and setting session object values--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>Old MaxInactiveInterval =
  <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>New MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>If the session object "My dog" is
still around, this value will be non-null:<H2>
<H3><li>Session value for "My dog" =
<%= session.getAttribute("My dog") %></li></H3>
<%-- Now add the session object "My dog" --%>
<% session.setAttribute("My dog",
                    new String("Ralph")); %>
<H1>My dog's name is
```

```
<%= session.getAttribute("My dog") %></H1>
<%-- See if "My dog" wanders to another form --%>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit
Value="Invalidate"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Keep Around"></FORM>
</body></html>
///:~
```

El objeto **session** se proporciona por defecto, por lo que está disponible sin necesidad de codificación extra. Las llamadas a **getId**(), **getCreationTime**() y **getMaxInactiveInterval**() se usan para mostrar información sobre este objeto **session**.
Cuando se trae por primera vez esta sesión se verá un **MaxInactiveInterval** de, por ejemplo, 1800 segundos (30 minutos). Esto dependerá de la forma en que esté configurado el contenedor de JSP/servlets. El **MaxInactiveInterval** se acorta a 5 segundos para que las cosas parezcan interesantes. Si se refresca la página antes de que expire el intervalo de 5 segundos, se verá:

```
Session value for "My dog" = Ralph
```

Pero si se espera más que eso, "Ralph" se convertirá en **null**.

Para ver cómo se puede traer la información de sesión a otras páginas, y para ver también el efecto de invalidar un objeto de sesión frente a simplemente dejar que expire, se crean otros dos JSP. El primero (al que se llega presionando el botón "invalidar" de **ObjetoSesion.jsp**) lee la información de sesión y después invalida esa sesión explícitamente:

```
//:! c15:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

Para experimentar con esto, refresque **ObjetoSesion.jsp**, y después pulse inmediatamente en el objeto "invalidar" para ir a **ObjetoSesion2,jsp**. En este momento se verá "Ralph", y después (antes de que haya expirado el intervalo de 5 segundos), refresque **ObjetoSesion2.jsp** para ver que la sesión ha sido invalidada a la fuerza y que "Ralph" ha desaparecido.

Si se vuelve a **ObjetoSesion.jsp**, refresque la página, de forma que se tenga un nuevo intervalo de 5 segundos, después presione el botón "Mantener", que le llevará a la siguiente página, **ObjetoSesion3.jsp**, que **NO** invalida la sesión:

```
//:! c15:jsp:SessionObject3.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

Dado que esta página no invalida la sesión, "Ralph" merodeará por ahí mientras se siga refrescando la página, antes de que expire el intervalo de 5 segundos. Esto es semejante a una mascota "Tomagotchi" -en la medida en que se juega con "Ralph", sigue vivo, si no, expira.

Crear y modificar cookies

Las cookies se presentaron en la sección anterior relativa a servlets. De nuevo, la brevedad de los JSP hace que jugar con las cookies aquí sea mucho más sencillo que con el uso de servlets. El ejemplo siguiente muestra esto cogiendo las cookies que vienen con la petición, leyendo y modificando sus edades máximas (fechas de expiración) y adjuntando una **Cookie** a la respuesta saliente:

```
//:! c15:jsp:Cookies.jsp
<%--This program has different behaviors under
 different browsers! --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie name: <%= cookies[i].getName() %> <br>
  value: <%= cookies[i].getValue() %><br>
  Old max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
  New max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int count = 0; int dcount = 0; %>
<% response.addCookie(new Cookie(
```

```
    "Bob" + count++, "Dog" + dcount++)); %>
</body></html>
///:~
```

Dado que cada navegador almacena las cookies a su manera, se pueden ver distintos comportamientos con distintos navegadores (no puede asegurarse, pero podría tratarse de algún tipo de error solucionado para cuando se lea el presente texto).

También podrían experimentarse resultados distintos si se apaga el navegador y se vuelve a arrancar en vez de visitar simplemente una página distinta y volver a **Cookies.jsp**. Nótese que usar objetos sesión parece ser más robusto que el uso directo de cookies.

Tras mostrar el identificador de sesión, se muestra cada cookie del array de **cookies** que viene con el objeto **request**, junto con su edad máxima. Después se cambia la edad máxima y se muestra de nuevo para verificar el nuevo valor. A continuación se añade una nueva cookie a la respuesta. Sin embargo, el navegador puede parecer ignorar la edad máxima; merece la pena jugar con este programa y modificar el valor de la edad máxima para ver el comportamiento bajo distintos navegadores.

Resumen de JSP

Esta sección sólo ha sido un recorrido breve por los JSP, e incluso con lo aquí cubierto (junto con lo que se ha aprendido en el resto del libro, y el conocimiento que cada uno tenga de HTML) se puede empezar a escribir páginas web sofisticadas vía JSP. La sintaxis de JSP no pretende ser excepcionalmente profunda o complicada, por lo que si se entiende lo que se ha presentado en esta sección, uno ya puede ser productivo con JSP Se puede encontrar más información en los libros más actuales sobre servlets o en java.sun.com.

Es especialmente bonito tener disponibles los JSP, incluso si la meta es producir servlets. Se descubrirá que si se tiene una pregunta sobre el comportamiento de una faceta servlet, es mucho más fácil y sencillo escribir un programa de pruebas de JSP para responder a esa cuestión, que escribir un servlet. Parte del beneficio viene de tener que escribir menos código y de ser capaz de mezclar el HTML con el código Java, pero la mayor ventaja resulta especialmente obvia cuando se ve que el contenedor JSP maneja toda la recompilación y recarga de JSP automáticamente siempre que se cambia el código fuente.

Siendo los JSP tan terroríficos, sin embargo, merece la pena ser conscientes de que la creación de JSP requiere de un nivel de talento más elevado que el necesario para simplemente programar en Java o crear páginas web. Además, depurar una página JSP que no funciona no es tan fácil como depurar un

programa Java, puesto que (actualmente) los mensajes de error son más oscuros. Esto podría cambiar al mejorar los sistemas de desarrollo, pero también puede que veamos otras tecnologías construidas sobre Java y la Web que se adapten mejor a las destrezas del diseñador de sitios web.

RMI (Invocation Remote Method)

Los enfoques tradicionales a la ejecución de código en otras máquinas a través de una red siempre han sido confusos a la vez que tediosos y fuentes de error a la hora de su implementación. La mejor forma de pensar en este problema es que algún objeto resulte que resida en otra máquina, y que se pueda enviar un mensaje al objeto remoto y obtener un resultado exactamente igual que si el objeto viviera en la máquina local. Esta simplificación es exactamente lo que permite hacer el *Remote Method Invocation* (RMI) de Java. Esta sección recorre los pasos necesarios para que cada uno cree sus propios objetos RMI.

Interfaces remotos

RMI hace un uso intensivo de las interfaces. Cuando se desea crear un objeto remoto, se enmascara la implementación subyacente pasando una interfaz. Por consiguiente, cuando el cliente obtiene una referencia a un objeto remoto, lo que verdaderamente logra es una referencia a una interfaz, que resulta estar conectada a algún fragmento de código local que habla a través de la red. Pero no hay que pensar en esto, sino simplemente en enviar mensajes vía la referencia a la interfaz.

Cuando se cree una interfaz remota, hay que seguir estas normas:

1. La interfaz remota debe ser **public** (no puede tener "acceso de paquete" es decir, no puede ser "amigo". De otra forma, el cliente obtendría un error al intentar cargar un objeto remoto que implemente la interfaz remota.
2. La interfaz remota debe extender la interfaz **java.rmi.Remote.**
3. Cada método de la interfaz remota debe declarar **java.rmi.RemoteException** en su cláusula **throws,** además de cualquier excepción específica de la aplicación.
4. Todo objeto remoto pasado como parámetro o valor de retorno (bien directamente o bien embebido en un objeto local) debe declararse como la interfaz remota, no como la clase implementación.

He aquí una interfaz remota simple que representa un servicio de tiempo exacto:

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;
```

```
interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

Tiene el mismo aspecto que cualquier otra interfaz, excepto por extender Remote y porque todos sus métodos lanzan **RemoteException**. Recuérdese que una interfaz y todos sus métodos son automáticamente **public**.

Implementar la interfaz remota

El servidor debe contener una clase que extienda **UnicastRemoteObject** e implementar la interfaz remota. Esta clase también puede tener métodos adicionales, pero sólo los métodos de la interfaz remota están disponibles al cliente, por supuesto, dado que el cliente sólo obtendrá una referencia al interfaz, y no a la clase que lo implementa.

Hay que definir explícitamente el constructor para el objeto remoto, incluso si sólo se está definiendo un constructor por defecto que invoque al constructor de la clase base. Hay que escribirlo puesto que debe lanzar **RemoteException**.

He aquí la implementación de la interfaz remota **ITiempoPerfecto**:

```
//: c15:rmi:PerfectTime.java
// The implementation of
// the PerfectTime remote object.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor
  // to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving. Throw
  // exceptions out to the console.
  public static void main(String[] args)
```

```
  throws Exception {
    System.setSecurityManager(
    new RMISecurityManager());
    PerfectTime pt = new PerfectTime();
    Naming.bind(
      "//peppy:2005/PerfectTime", pt);
    System.out.println("Ready to do time");
  }
} ///:~
```

Aquí, **main**() maneja todos los detalles de establecimiento del servidor. Cuando se sirven objetos RMI, en algún momento del programa hay que:

1. Crear e instalar un gestor de seguridad que soporte RMI. El único disponible para RMI como parte de la distribución JAVA es **RMISecurityManager**.
2. Crear una o más instancias de un objeto remoto. Aquí, puede verse la creación del objeto **TiempoPerfecto**.
3. Registrar al menos uno de los objetos remotos con el registro de objetos remotos RMI para propósitos de reposición. Un objeto remoto puede tener métodos que produzcan referencias a otros objetos remotos. Esto permite configurarlo de forma que el cliente sólo tenga que ir al registro una vez para lograr el primer objeto remoto.

Configurar el registro

Aquí se ve una llamada al método **static Naming.bind**(). Sin embargo, esta llamada requiere que el registro se esté ejecutando como un proceso separado en el computador. El nombre del servidor de registros es **rmiregistry**, y bajo Windows de 32 bits se dice:

```
start rmiregistry
```

para que arranque en segundo plano. En Unix, el comando es:

```
rmiregistry &
```

Como muchos programas de red, el **rmiregistry** está ubicado en la dirección IP de cualquier máquina que lo arranque, pero también debe estar escuchando por un puerto. Si se invoca al **rmiregistry** como arriba, sin argumentos, el puerto del registro por defecto será 1099. Si se desea que esté en cualquier otro puerto, se añade un argumento a la línea de comandos para especificar el puerto. Para este ejemplo, el puerto está localizado en el 2005, de forma que bajo Windows de 32 bits el **rmiregistry** debería empezarse así:

```
start rmiregistry 2005
```

o en el caso de Unix:

```
rmiregistry 2005 &
```

La información sobre el puerto también debe proporcionarse al comando **bind**(), junto con la dirección IP de la máquina en la que está ubicado el registro. Pero esto puede ser un problema frustrante si se desea probar programas RMI de forma local de la misma forma en que se han probado otros programas de red hasta este momento en el presente capítulo. En la versión 1.1 del JDK, hay un par de problemas **[5]**:

1. **localhost** no funciona con RMI. Por consiguiente, para experimentar con RMI en una única máquina, hay que proporcionar el nombre de la máquina. Para averiguar el nombre de la máquina bajo Windows de 32 bits, se puede ir al panel de control y seleccionar "Red". Después, se selecciona la solapa "Identificación", y se dispondrá del nombre del computador. En nuestro caso, llamamos a mi computador "Pepe". El uso de mayúsculas y minúsculas parece ignorarse.
2. RMI no funcionará a menos que el computador tenga una conexión TCP/IP activa, incluso si todos los componentes simplemente se comunican entre sí en la máquina local. Esto significa que hay que conectarse al proveedor de servicios de Internet antes de intentar ejecutar el programa o se obtendrán algunos mensajes de excepción siniestros.

[5] Para descubrir esta información fueron muchas las neuronas que sufrieron una muerte agónica.

Con todo esto en mente, el comando **bind**() se convierte en :

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

Si se está usando el puerto por defecto, el 1099, no hay que especificar un puerto, por lo que podría decirse:

```
Naming.bind("//peppy/PerfectTime", pt);
```

Se deberían poder hacer pruebas locales usando sólo el identificador:

```
Naming.bind("PerfectTime", pt);
```

El nombre del servicio es arbitrario; resulta que en este caso es **TiempoPerfecto**, exactamente igual que el nombre de la clase, pero se le podría dar el nombre que se desee. Lo importante es que sea un nombre único en el registro que el cliente conozca, para buscar el objeto remoto. Si el nombre ya está en el registro, se obtiene una **AlreadyBoundException**. Para evitar esto, se puede usar siempre **rebind**() en vez de **bind**(), puesto que **rebind**(), o añade una nueva entrada o reemplaza una ya existente.

Incluso aunque exista **main**( ), el objeto se ha creado y registrado, por lo que se mantiene vivo por parte del registro, esperando a que venga un cliente y lo solicite. Mientras se esté ejecutando el **rmiregistry** y no se invoque a **Narning.unbind**() para ese nombre, el objeto estará ahí. Por esta razón, cuando se esté desarrollando código hay que apagar el **rmiregistry** y volver a arrancarlo al compilar una nueva versión del objeto remoto.

Uno no se ve forzado a arrancar **rmiregistry** como un proceso externo. Si se sabe que una aplicación es la única que va a usar el registro, se puede arrancar dentro del programa con la línea:

```
LocateRegistry.createRegistry(2005);
```

Como antes, 2005 es el número de puerto que usamos en este ejemplo. Esto equivale a ejecutar **rmiregistry** 2005 desde la línea de comandos, pero a menudo puede ser más conveniente cuando se esté desarrollando código RMI, pues elimina los pasos adicionales de arrancar y detener el registro. Una vez ejecutado este código, se puede invocar **bind**() usando **Naming** como antes.

Crear stubs y skeletons

Si se compila y ejecuta **TiempoPerfecto.java**, no funcionará incluso aunque el **rmiregistry** se esté ejecutando correctamente. Esto se debe a que todavía no se dispone de todo el marco para RMI.

Hay que crear primero los **stubs** y **skeletons** que proporcionan las operaciones de conexión de red y que permiten fingir que el objeto remoto es simplemente otro objeto local de la máquina.
Lo que ocurre tras el telón es complejo. Cualquier objeto que se pase o que sea devuelvo por un objeto remoto debe implementar **Serializable** (si se desea pasar referencias remotas en vez de objetos enteros, los parámetros objeto pueden implementar **Remote**), por lo que se puede imaginar que los **stubs** y **skeletons** están llevando a cabo operaciones de serialización y deserialización automáticas, al ir mandando todos los parámetros a través de la red, y al devolver el resultado. Afortunadamente, no hay por qué saber nada de esto, pero si que hay que crear los **stubs** y **skeletons**. Este proceso es simple: se invoca a la herramienta **rmic** para el código compilado, y ésta crea los archivos necesarios. Por tanto, el único requisito es añadir otro paso al proceso de compilación.

Sin embargo, la herramienta **rmic** tiene un comportamiento particular para paquetes y **classpaths**. **TiempoPerfecto.java** está en el **package cl5.rmi**, e incluso si se invoca a **rmic** en el mismo directorio en el que está localizada **TiempoPerfecto.class**, **rmic** no encontrará el archivo, puesto que busca el **classpath**. Por tanto, hay que especificar las localizaciones distintas al **classpath**, como en:

```
rmic c15.rmi.PerfectTime
```

No es necesario estar en el directorio que contenga **TiempoPerfecto.class** cuando se ejecute este comando, si bien los resultados se colocarán en el directorio actual.

Cuando **rmic** se ejecuta con éxito, se tendrán dos nuevas clases en el directorio:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

correspondientes al **stub** y al **skeleton**. Ahora, ya estamos listos para que el cliente y el servidor se intercomuniquen.

Utilizar el objeto remoto

Toda la motivación de RMI es simplificar el uso de objetos remotos. Lo único extra que hay que hacer en el programa cliente es buscar y capturar la interfaz remota desde el servidor. A partir de ese momento, no hay más que programación Java ordinaria: envío de mensajes a objetos. He aquí el programa que hace uso de **TiempoPerfecto**:

```
//: c15:rmi:DisplayPerfectTime.java
// Uses remote object PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
      new RMISecurityManager());
    PerfectTimeI t =
      (PerfectTimeI)Naming.lookup(
        "//peppy:2005/PerfectTime");
    for(int i = 0; i < 10; i++)
      System.out.println("Perfect time = " +
        t.getPerfectTime());
  }
} ///:~
```

La cadena de caracteres ID es la misma que la que se usó para registrar el objeto con **Naming**, y la primera parte representa al URL y al número de puerto. Dado

que se está usando una URL también se puede especificar una máquina en Internet.

Lo que se devuelve de **Naming.lookup**() hay que convertirlo a la interfaz remota, no a la clase. Si se usa la clase en su lugar, se obtendrá una excepción.

En la llamada a método:

```
t.getPerfectTime()
```

puede verse que una vez que se tiene una referencia al objeto remoto, la programación con él no difiere de la programación con un objeto local (con una diferencia: los métodos remotos lanzan **RemoteException**).

CORBA

En aplicaciones distribuidas grandes, las necesidades pueden no verse satisfechas con estos enfoques que acabamos de describir. Por ejemplo, uno podría querer interactuar con almacenes de datos antiguos, o podría necesitar servicios de un objeto servidor independientemente de su localización física. Estas situaciones requieren de algún tipo de *Remote Procedure Call* (RPC), y posiblemente de independencia del lenguaje. Es aquí donde CORBA puede ser útil.

CORBA no es un aspecto del lenguaje; es una tecnología de integración. Es una especificación que los fabricantes pueden seguir para implementar productos de integración compatibles con CORBA.

Éste es parte del esfuerzo de la OMG para definir un marco estándar para interoperabilidad de objetos distribuidos independientemente del lenguaje.

CORBA proporciona la habilidad de construir llamadas a procedimientos remotos en objetos Java y no Java, y de interactuar con sistemas antiguos de forma independiente de la localización. Java añade soporte a redes y un lenguaje orientado a objetos perfecto para construir aplicaciones gráficas o no. El modelo de objetos de Java y el de la OMG se corresponden perfectamente entre sí; por ejemplo, ambos implementan el concepto de interfaz y un modelo de objetos de referencia.

Fundamentos de CORBA

A la especificación de la interoperabilidad entre objetos desarrollada por la OMG se le suele denominar la Arquitectura de Gestión de Objetos (OMA, *Object Management Architecture*). La OMA define dos conceptos: el *Core Object Model* y la *OlMA Reference Architecture*. El primero establece los conceptos básicos de un

objeto, interfaz, operación, etc. (CORBA es un refinamiento del *Core Object Model*). La *OMA Reference Architecture* define una infraestructura de servicios y mecanismos subyacentes que permiten interoperar a los objetos. Incluye el *Object Request Broker* (ORB), *Object Services* (conocidos también como CORBA services) y facilidades generales.

El ORB es el canal de comunicación a través del cual unos objetos pueden solicitar servicios a otros, independientemente de su localización física. Esto significa que lo que parece una llamada a un método en el código cliente es, de hecho, una operación compleja. En primer lugar, debe existir una conexión con el objeto servidor, y para crear la conexión el ORB debe saber dónde reside el código que implementa ese servidor. Una vez establecida la conexión, hay que pasar los parámetros del método, por ejemplo, convertidos en un flujo binario que se envía a través de la red. También hay que enviar otra información como el nombre de la máquina servidora, el proceso servidor y la identidad del objeto servidor dentro de ese proceso. Finalmente, esta información se envía a través de un protocolo de bajo nivel, se decodifica en el lado servidor y se ejecuta la llamada. El ORB oculta toda esta complejidad al programador y hace la operación casi tan simple como llamar a un método de un objeto local. No hay ninguna especificación que indique cómo debería implementarse un núcleo ORB, pero para proporcionar compatibilidad básica entre los ORB de los diferentes vendedores, la OMG define un conjunto de servicios accesibles a través de interfaces estándar.

Lenguaje de Definición de Interfaces CORBA (IDL)

COBA está diseñado para lograr la transparencia del lenguaje: un objeto cliente puede invocar a métodos de un objeto servidor de distinta clase, independientemente del lenguaje en que estén implementados. Por supuesto, el objeto cliente debe conocer los nombres y signaturas de los métodos que expone el objeto servidor. Es aquí donde interviene el IDL. El COBA IDL es una forma independiente del lenguaje de especificar tipos de datos, atributos, operaciones, interfaces y demás. La sintaxis IDL es semejante a la de C++ o Java.

La tabla siguiente muestra la correspondencia entre algunos de los conceptos comunes a los tres lenguajes, que pueden especificarse mediante CORBA IDL:

CORBA IDL	**Java**	**C++**
Module	Package	Namespace
Interface	Interface	Pure abstract class
Method	Method	Member function

También se soporta el concepto de herencia, utilizando el operador "dos puntos" como en C++. El programador escribe una descripción IDL de los atributos, métodos e interfaces implementados y usados por el servidor y los clientes.

Después, se compila el IDL mediante un compilador IDL/Java proporcionado por un fabricante, que lee el fuente IDL y genera código Java.

El compilador IDL es una herramienta extremadamente útil: no genera simplemente un código fuente Java equivalente al IDL, sino que también genera el código que se usará para pasar los parámetros a métodos y para hacer llamadas remotas. A estos códigos se les llama **stub** y **skeleton**, y están organizados en múltiples archivos fuente Java, siendo generalmente parte del mismo paquete Java.

El servicio de nombres

El servicio de nombres es uno de los servicios CORBA fundamentales. A un objeto CORBA se accede a través de una referencia, un fragmento de información que no tiene significado para un lector humano. Pero a las referencias pueden asignarse nombres o cadenas de caracteres definidas por el programador. A esta operación se le denomina encadenar la referencia, y uno de los componentes de OMA, el Servicio de Nombres, se dedica exclusivamente a hacer conversiones y correspondencias cadena y objeto-a-cadena. Puesto que el servicio de nombres actúa como un directorio de teléfonos y, tanto servidores como clientes pueden consultarlo y manipularlo, se ejecuta como un proceso aparte. A la creación de una correspondencia objeto-a-cadena se le denomina vinculación de un objeto y a la eliminación de esta correspondencia de le denomina desvinculación. A la obtención de una referencia de un objeto pasando un **String** se le denomina resolución de un nombre.

Por ejemplo, al arrancar, una aplicación servidora podría crear un objeto servidor, establecer una correspondencia en el servicio de nombres y esperar después a que los clientes hagan peticiones. Un cliente obtiene primero una referencia al objeto servidor, resuelve el **string**, y después puede hacer llamadas al servidor haciendo uso de la referencia.

De nuevo, la especificación del servicio de nombres es parte de CORBA, pero la aplicación que lo proporciona está implementada por el fabricante del ORB. La forma de acceder a la funcionalidad del Servicio de Nombres puede variar de un fabricante a otro.

Un ejemplo

El código que se muestra aquí no será muy elaborado ya que distintos ORB tienen distintas formas de acceder a servicios CORBA, por lo que los ejemplos son específicos de los fabricantes. (El ejemplo de debajo usa JavaIDL, un producto gratuito de Sun que viene con un ORB ligero, un servicio de nombres, y un compilador de IDL a Java.) Además, dado que Java es un lenguaje joven y en

evolución, no todas las facetas de CORBA están presentes en los distintos productos Java/CORBA.

Queremos implementar un servidor, en ejecución en alguna máquina, al que se pueda preguntar la hora exacta. También se desea implementar un cliente que pregunte por la hora exacta. En este caso, se implementarán ambos programas en Java, pero también se podrían haber usado dos lenguajes distintos (lo cual ocurre a menudo en situaciones reales).

Escribir el IDL fuente

El primer paso es escribir una descripción IDL de los servicios proporcionados. Esto lo suele hacer el programador del servidor, que es después libre de implementar el servidor en cualquier lenguaje en el que exista un compilador CORBA IDL.

El archivo IDL se distribuye al programador de la parte cliente y se convierte en el puente entre los distintos lenguajes.

El ejemplo de debajo muestra la descripción IDL de nuestro servidor **TiempoExacto**:

```
//: c15:corba:ExactTime.idl
//# You must install idltojava.exe from
//# java.sun.com and adjust the settings to use
//# your local C preprocessor in order to compile
//# This file. See docs at java.sun.com.
module remotetime {
   interface ExactTime {
      string getTime();
   };
}; ///:~
```

Ésta es la declaración de una interfaz **TiempoExacto** de dentro del espacio de nombres **tiemporemoto**.

La interfaz está formada por un único método que devuelve la hora actual en formato **string**.

Crear stubs y skeletons

El segundo paso es compilar el IDL para crear el código stub y el skeleton de Java que se usará para implementar el cliente y el servidor. La herramienta que viene con el producto **JavaIDL** es **idltojava**:

```
idltojava remotetime.idl
```

Esto generará automáticamente el código, tanto para el stub como para el skeleton. **Idltojava** genera un package Java cuyo nombre se basa en el del módulo IDL, **tiemporemoto**, y los archivos Java generados se ponen en el subdirectorio **tiemporemoto**. El skeleton es - **TiempoExactoImplBase.java**, que se usará par implementar el objeto servidor, y - **TiempoExactoStub.java** se usará para el cliente. Hay representaciones Java de la interfaz IDL en **TiempoExacto.java** y un par de otros archivos de soporte que se usan, por ejemplo, para facilitar el acceso a operaciones de servicio de nombres.

Implementar el servidor y el cliente

Debajo puede verse el código correspondiente al lado servidor. La implementación del objeto servidor está en la clase **ServidorTiempoExacto**. El **ServidorTiempoRemoto** es la aplicación que crea un objeto servidor, lo registra en el ORB, le da un nombre a la referencia al objeto, y después queda a la espera de peticiones del cliente.

```
//: c15:corba:RemoteTimeServer.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Server object implementation
class ExactTimeServer extends _ExactTimeImplBase {
  public String getTime(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
              System.currentTimeMillis()));
  }
}

// Remote application implementation
public class RemoteTimeServer {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws Exception {
    // ORB creation and initialization:
    ORB orb = ORB.init(args, null);
    // Create the server object and register it:
    ExactTimeServer timeServerObjRef =
      new ExactTimeServer();
    orb.connect(timeServerObjRef);
    // Get the root naming context:
```

```
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references(
          "NameService");
      NamingContext ncRef =
        NamingContextHelper.narrow(objRef);
      // Assign a string name to the
      // object reference (binding):
      NameComponent nc =
        new NameComponent("ExactTime", "");
      NameComponent[] path = { nc };
      ncRef.rebind(path, timeServerObjRef);
      // Wait for client requests:
      java.lang.Object sync =
        new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }
    }
  } ///:~
```

Como puede verse, implementar el objeto servidor es simple; es una clase normal Java que se hereda del código skeleton generado por el compilador IDL. Las cosas se complican cuando hay que interactuar con el ORB y otros servicios CORBA.

Algunos servicios CORBA

He aquí una breve descripción de lo que está haciendo el código **JavaIDL** (ignorando principalmente la parte del código CORBA dependiente del vendedor). La primera línea del método **main**() arranca el ORB, y por supuesto, esto se debe a que el objeto servidor necesitará interactuar con él.

Justo después de la inicialización del ORB se crea un objeto servidor. De hecho, el término correcto sería un objeto sirviente transitorio: un objeto que recibe peticiones de clientes, cuya longevidad es la misma que la del proceso que lo crea. Una vez que se ha creado el objeto sirviente transitorio, se registra en el ORB, lo que significa que el ORE3 sabe de su existencia y que puede ahora dirigirle peticiones.

Hasta este punto, todo lo que tenemos es **RefObjServidofliempo**, una referencia a objetos conocida sólo dentro del proceso servidor actual. El siguiente paso será asignarle un nombre en forma de **string** a este objeto sirviente; los clientes usarán ese nombre para localizarlo. Esta operación se logra haciendo uso del Servicio de Nombres. Primero, necesitamos una referencia a objetos al Servicio de Nombres; la llamada a **resolve-initial-referentes**() toma la referencia al objeto pasado a **string** del Servicio de Nombres, que es "**NameService**", en **JavaIDL**, y devuelve una referencia al objeto. Ésta se

convierte a una referencia **NamingContext** específica usando el método **narrow**().

Ahora ya pueden usarse los servicios de nombres. Para vincular el objeto sirviente con una referencia a objetos pasada a **string**, primero se crea un objeto **NameComponent**, inicializado con "**TiempoExacto**", la cadena de caracteres que se desea vincular al objeto sirviente.

Después, haciendo uso del método **rebind**(), se vincula la referencia pasada a **string** a la referencia al objeto. Se usa **rebind**() para asignar una referencia, incluso si ésta ya existe, mientras que **bind**() provoca una excepción si la referencia ya existe. En CORBA un nombre está formado por varios **NameContexts** - por ello se usa un array para vincular el nombre a la referencia al objeto. Finalmente, el objeto sirviente ya está listo para ser usado por los clientes. En este punto, el proceso servidor entra en un estado de espera. De nuevo, esto se debe a que es un sirviente transitorio, por lo que su longevidad podría estar confinada al proceso servidor. **JavaIDL** no soporta actualmente objetos persistentes - objetos que sobreviven a la ejecución del proceso que los crea.

Ahora que ya se tiene una idea de lo que está haciendo el código servidor, echemos un vistazo al código cliente:

```
//: c15:corba:RemoteTimeClient.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
  // Throw exceptions to console:
  public static void main(String[] args)
  throws Exception {
    // ORB creation and initialization:
    ORB orb = ORB.init(args, null);
    // Get the root naming context:
    org.omg.CORBA.Object objRef =
      orb.resolve_initial_references(
        "NameService");
    NamingContext ncRef =
      NamingContextHelper.narrow(objRef);
    // Get (resolve) the stringified object
    // reference for the time server:
    NameComponent nc =
      new NameComponent("ExactTime", "");
    NameComponent[] path = { nc };
    ExactTime timeObjRef =
      ExactTimeHelper.narrow(
        ncRef.resolve(path));
```

```
    // Make requests to the server object:
    String exactTime = timeObjRef.getTime();
    System.out.println(exactTime);
  }
} ///:~
```

Las primeras líneas hacen lo mismo que en el proceso servidor: se inicializa el ORB y se resuelve una referencia al servidor del Servicio de Nombres. Después se necesita una referencia a un objeto para el objeto sirviente, por lo que le pasamos una referencia a objeto pasada a **String** al método **resolve**(), y convertimos el resultado en una referencia a la interfaz **TiempoExacto** usando el método **narrow**(). Finalmente, invocamos a **obtenerTiempo**().

Activar el proceso de servicio de nombres

Finalmente, tenemos una aplicación servidora y una aplicación cliente listas para interoperar. Hemos visto que ambas necesitan el servicio de nombres para vincular y resolver referencias a objetos pasadas a **String**. Hay que arrancar el proceso de servicio de nombres antes de ejecutar o el servidor o el cliente. En **JavaIDL** el servicio de nombres es una aplicación Java que viene con el paquete del producto, pero puede ser distinta en cada producto. El servicio de nombres **JavaIDL** se ejecuta dentro de una instancia de la JVM y escucha por defecto al puerto 900 de red.

Activar el servidor y el cliente

Ahora ya se pueden arrancar las aplicaciones servidor y cliente (en este orden, pues el servidor es temporal). Si todo se configura bien, lo que se logra es una única línea de salida en la ventana de consola del cliente, con la hora actual. Por supuesto, esto puede que de por sí no sea muy excitante, pero habría que tener algo en cuenta: incluso si están en la misma máquina física, la aplicación cliente y la servidora se están ejecutando en máquinas virtuales distintas y se pueden comunicar vía una capa de integración subyacente, el ORB y el Servicio de Nombres.

Éste es un ejemplo simple, diseñado para trabajar sin red, pero un O B suele estar configurado para buscar transparencia de la localización. Cuando el servidor y el cliente están en máquinas distintas, el ORB puede resolver referencias pasadas a **Strings**, remotas, utilizando un componente denominado el Repositorio de implementaciones. Aunque éste es parte COBA, casi no hay ninguna especificación relativa al mismo, por lo que varía de fabricante en fabricante.

Como puede verse, hay mucho más de CORBA que lo que se ha cubierto hasta aquí, pero uno ya debería haber captado la idea básica. Si se desea más información sobre CORBA un buen punto de partida es el sitio web de OMG, en

http://www.omg.org. Ahí hay documentación, white papers, procedimientos y referencias a otras fuentes y productos relacionados con COBA.

Applets de Java y CORBA

Los applets de Java pueden actuar como clientes CORBA. De esta forma, un applet puede acceder a información remota y a servicios expuestos como objetos CORBA. Pero un applet sólo se puede conectar con el servidor desde el que fue descargado, por lo que todos los objetos CORBA con los que interactúe el applet deben estar en ese servidor. Esto es lo contrario a lo que intenta hacer CORBA: ofrecer una transparencia total respecto a la localización.

Éste es un aspecto de la seguridad de la red. Si uno está en una Intranet, una solución es disminuir las restricciones de seguridad en el navegador. O establecer una política de firewall para la conexión con servidores externos.

Algunos productos Java ORB ofrecen soluciones propietarias a este problema. Por ejemplo, algunos implementan lo que se denomina Tunneling HTTP, mientras que otros tienen sus propias facetas firewall.

Éste es un tema demasiado complejo como para cubrirlo en un apéndice, pero definitivamente es algo de lo que habría que estar pendientes.

CORBA frente a RMI

Se ha visto que una de las facetas más importantes de COBA es el soporte WC, que permite a los objetos locales invocar a métodos de objetos remotos. Por supuesto, ya hay una faceta nativa Java que hace exactamente lo mismo: RMI (véase Capítulo 15). Mientras que RMI hace posibles RFT entre objetos Java, CORBA las hace posibles entre objetos implementados en cualquier lenguaje. La diferencia es, pues, enorme.

Sin embargo, RMI puede usarse para invocar a servicios en código remoto no Java. Todo lo que se necesita es algún tipo de envoltorio de objeto Java en torno al código no Java del lado servidor. El objeto envoltorio se conecta externamente a clientes Java vía RMI, e internamente se conecta al código no Java usando una de las técnicas mostradas anteriormente, como JNI o J/Direct.

Este enfoque requiere la escritura de algún tipo de capa de integración, que es exactamente lo que hace CORBA automáticamente, pero de esta forma no se necesitaría un ORB elaborado por un tercero.

Enterprise JavaBeans

Supóngase **[6]** que hay que desarrollar una aplicación multicapa para visualizar y actualizar registros de una base de datos a través de una interfaz web. Se puede escribir una aplicación de base de datos usando JDBC, una interfaz web usando JSP/servlets, y un sistema distribuido usando CORBA/RMI.

[6] Sección a la que ayudó Robert Castaneda, ayudado a su vez por Dave Bartlett.

Pero, ¿qué consideraciones extra hay que hacer al desarrollar un sistema de objetos distribuido además de simplemente conocer las API? He aquí los distintos aspectos:

Rendimiento: los objetos distribuidos que uno crea deben ejecutarse con buen rendimiento, pues potencialmente podrían servir a muchos clientes simultáneamente. Habrá que usar técnicas de optimización como la captura y organización de recursos como conexiones a base de datos. También habrá que gestionar el ciclo de vida de los objetos distribuidos.

Escalabilidad: los objetos distribuidos deben ser también escalables. La escalabilidad en una aplicación distribuida significa que se pueda incrementar el número de instancias de los objetos distribuidos, trasladándose a máquinas adicionales sin necesidad de modificar ningún código.

Seguridad: un objeto distribuido suele tener que gestionar la autorización de los clientes que lo acceden. Idealmente, se pueden añadir nuevos usuarios y roles al mismo sin tener que recompilar.

Transacciones distribuidas: un objeto distribuido debería ser capaz de referenciar a transacciones distribuidas de forma transparente. Por ejemplo, si se está trabajando con dos base de datos separadas, habría que poder actualizarlas simultáneamente dentro de la misma transacción, o deshacer una transacción si no se satisface determinado criterio.

Reusabilidad: el objeto distribuido ideal puede moverse sin esfuerzo a otro servidor de aplicaciones de otro vendedor. Sería genial si se pudiera revender un componente objeto distribuido sin tener que hacerle modificaciones especiales, o comprar un componente de un tercero y usarlo sin tener que recompilarlo ni reescribirlo.

Disponibilidad: si una de las máquinas del sistema se cae, los clientes deberían dirigirse automáticamente a copias de respaldo de esos objetos, residentes en otras máquinas.

Estas consideraciones, además del problema de negocio que se trata de solucionar, pueden hacer un proyecto inabordable. Sin embargo, todos los

aspectos excepto los del problema de negocio son redundantes -hay que reinventar soluciones para cada aplicación de negocio distribuida.

Sun, junto con otros fabricantes de objetos distribuidos líderes, se dio cuenta de que antes o después todo equipo de desarrollo estaría reinventando estas soluciones particulares, por lo que crearon la especificación de los Enterprise JavaBeans (EJB). Esta especificación describe un modelo de componentes del lado servidor que toma en consideración todos los aspectos mencionados utilizando un enfoque estándar que permite a los desarrolladores crear componentes de negocio denominados EJBs, que se aíslan del código "pesado" de bajo nivel y se enfocan sólo en proporcionar lógica de negocio. Dado que los EJB están definidos de forma estándar, pueden ser independientes del fabricante.

JavaBeans frente a EJB

Debido a la similitud de sus nombres, hay mucha confusión en la relación entre el modelo de componentes JavaBeans y la especificación de los Enterprise JavaBeans. Mientras que, tanto unos como otros comparten los mismos objetivos de promocionar la reutilización y portabilidad del código Java entre las herramientas de desarrollo y diseminación con el uso de patrones de diseño estándares, los motivos que subyacen tras cada especificación están orientados a solucionar problemas diferentes.

Los estándares definidos en el modelo de comportamiento de los JavaBeans están diseñados para crear componentes reusables que suelen usarse en herramientas de desarrollo IDE y comúnmente, aunque no exclusivamente, en componentes visuales.

La especificación de los JavaBeans define un modelo de componentes para desarrollar código Java del lado servidor. Dado que los EJBs pueden ejecutarse potencialmente en distintas plataformas de la parte servidor -incluyendo servidores que no tienen presentaciones visuales - un EJB no puede hacer uso de las bibliotecas gráficas, como la AWT o Swing.

La especificación EJB

La especificación de Enterprise JavaBeans describe un modelo de componentes del lado servidor. Define seis papeles que se usan para llevar a cabo las tareas de desarrollo y distribución además de definir los componentes del sistema. Estos papeles se usan en el desarrollo, distribución, y ejecución de un sistema distribuido. Los fabricantes, administradores y desarrolladores van jugando los distintos papeles, para permitir la división del conocimiento técnico y del dominio. El fabricante proporciona un marco de trabajo de sonido en su parte técnica, y los desarrolladores crean componentes específicos del dominio; por ejemplo, un componente "contable". Una misma parte puede llevar a cabo uno o varios

papeles. Los papeles definidos en la especificación de EJB se resumen en la tabla siguiente:

Papel	**Responsabilidad**
Suministrador de Enterprise Bean	El desarrollador responsable de crear componentes EJB reusables. Estos componentes se empaquetan en un archivo jar especial (archivo ejb-jar)
Ensamblador de aplicaciones	Crea y ensambla aplicaciones a partir de una colección de archivos ejb-jar. Incluye la escritura de aplicaciones que usan la colección de EJB (por ejemplo, servlets, JSP, Swing, etc., etc.).
Distribuidor	Toma la colección de archivos ejb-jar del ensamblador y/o suministrador de Beans y los distribuye en un entorno de tiempo de ejecución: uno o más contenedores EJB.
Proveedor de contenedores/ servidores de EJB	Proporciona un entorno de tiempo de ejecución y herramientas que se usan para distribuir, administrar y ejecutar componentes EJB.
Administrador del sistema	Gestiona los distintos componentes y servicios, de forma que estén configurados e interactúen correctamente, además de asegurar que el sistema esté activo y en ejecución.

Componentes EJB

Los componentes EJB son elementos de lógica de negocio reutilizable que se adhieren a estándares estrictos y patrones de diseño definidos en la especificación EJB. Esto permite la portabilidad de los componentes. También permite poder llevar a cabo otros servicios -como la seguridad, la gestión de cachés, y las transacciones distribuidas- por parte de los componentes. El responsable del desarrollo de componentes EJB es el Enterprise Bean Provider.

Contenedor & Servidor EJB

El Contenedor EJB es un entorno de tiempo de ejecución que contiene y ejecuta componentes EJB y les proporciona un conjunto de servicios estándares. Las responsabilidades del contenedor de EJB están definidas de forma clara en la especificación, en aras de lograr neutralidad respecto del fabricante. El contenedor de EJB proporciona el "peso pesado" de bajo nivel del EJB, incluyendo transacciones distribuidas, seguridad, gestión del ciclo de vida de los beans,

gestión de caché, hilado y gestión de sesiones. El proveedor de contenedor EJB es el responsable de su provisión.

Un Servidor EJB se define como un servidor de aplicación que contiene y ejecuta uno o más contenedores de EJB. El Proveedor de Servicios EJB es responsable de proporcionar un Servidor EJB. Generalmente se puede asumir que el Contenedor de EJBs y el Servidor de EJBs son el mismo.

Interfaz Java para Nombres y Directorios (JNDI) [7]

Interfaz usado en los Enterprise JavaBeans como el servicio de nombres para los componentes EJB de la red y para otros servicios de contenedores, como las transacciones. JNDI establece una correspondencia muy estricta con otros estándares de nombres y directorios como CORBA CosNaming y puede implementarse, realmente, como un envoltorio realmente de éstos.

[7] N. del traductor: Java Naming and Directory Interface.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS se usa en las Enterprise JavaBeans como el API transaccional. Un proveedor de Enterpise Beans puede usar JTS para crear código de transacción, aunque el contenedor EJB suele implementar transacciones EJB de parte de los componentes EJB. El distribuidor puede definir los atributos transaccionales de un componente EJB en tiempo de distribución. El Contenedor de EJB es el responsable de gestionar la transacción, sea local o distribuida. La especificación JTS es la correspondencia Java al CORBA OTS (Object Transaction Service).

CORBA y RMI/IIOP

La especificación EJB define la interoperabilidad con CORBA a través de la compatibilidad con protocolos CORBA. Esto se logra estableciendo una correspondencia entre servicios EJB como JTS y JNDI con los servicios CORBA correspondientes, y la implementación de RMI sobre el protocolo IIOP de CORBA.

El uso de CORBA y RMI/IIOP en Enterprise JauaBeans está implementado en el contenedor de EJB y es el responsable del proveedor de contenedores EJB. El uso de CORBA y de RMI/IIOP sobre el contenedor de EJB está oculto desde el propio componente EJB.

Esto significa que el proveedor de Enterprise Beans puede escribir su componente EJB y distribuirlo a cualquier contenedor de EJB sin que importe qué protocolo de comunicación se esté utilizando.

Las partes de un componente EJB

Un EJB está formado por varias piezas, incluyendo el propio Bean, la implementación de algunas interfaces, y un archivo de información. Todo se empaqueta junto en un archivo jar especial.

Enterprise Bean

El Enterprise Bean es una clase Java que desarrolla el Enterprise Bean Provider. Implementa una interfaz de Enterprise Bean y proporciona la implementación de los métodos de negocio que va a llevar a cabo el componente. La clase no implementa ninguna autorización, autentificación, multihilo o código transaccional.

Interfaz local

Todo Enterprise Bean que se cree debe tener su interfaz local asociado. Esta interfaz se usa como fábrica del EJB. Los clientes lo usan para encontrar una instancia del EJB o crear una nueva.

Interfaz remota

Se trata de una interfaz Java que refleja los métodos del Enterprise Bean que se desea exponer al mundo exterior. Esta interfaz juega un papel similar al de una interfaz CORBA IDL.

Descriptor de distribución

El descriptor de distribución es un archivo XML que contiene información sobre el EJB. El uso de XML permite al distribuidor cambiar fácilmente los atributos del EJB.

Los atributos configurables definidos en el descriptor de distribución incluyen:

- Los nombres de interfaz Local y Remota requeridos por el EJB.
- El nombre a publicar en el JNDI para la interfaz local del EJB.
- Atributos transaccionales para cada método de EJB.
- Listas de control de acceso para la autenticación.

Archivo EJB-Jar

Es un archivo jar normal que contiene el EJB, las interfaces local y remota, y el descriptor de distribución.

Funcionamiento de un EJB

Una vez que se tiene un archivo EJB-Jar que contiene el Bean, las interfaces Local y Remota, y el descriptor de distribución, se pueden encajar todas las piezas y a la vez entender por qué se necesitan las interfaces Local y Remota, y cómo los usa el contendor de EJB.

El Contenedor implementa las interfaces Local y Remoto del archivo EJB-Jar. Como se mencionó anteriormente, la interfaz Local proporciona métodos para crear y localizar el EJB. Esto significa que el contenedor de EJB es responsable de la gestión del ciclo de vida del EJB. Este nivel de indirección permite que se den optimizaciones. Por ejemplo, 5 clientes podrían solicitar simultáneamente la creación de un EJB a través de una interfaz Local, y el contenedor de EJB respondería creando sólo un EJB y compartiéndolo entre los 5 clientes. Esto se logra a través de la interfaz Remota, que también está implementado por el Contenedor de EJB. El objeto Remoto implementado juega el papel de objeto Proxy al EJB.
Todas las llamadas al EJB pasan por el proxy a través del contenedor de EJB vía las interfaces Local y Remota. Esta indirección es la razón por la que el contenedor de EJB puede controlar la seguridad y el comportamiento transaccional.

Tipos de EJB

La especificación de Enterprise JavaBeans define distintos tipos de EJB con características y comportamientos diferentes. En la especificación se han definido dos categorías de EJB: Beans de sesión y Beans de entidad, y cada categoría tiene sus propias variantes.

Beans de Sesión

Se usan para representar minúsculos casos de uso o flujo de trabajo por parte de un cliente. Representan operaciones sobre información persistente, pero no a la información persistente en sí. Hay dos tipos de Beans de sesión: los que no tienen estado y los que lo tienen. Todos los Beans de sesión deben implementar la interfaz **javax.ejb.SessionBean**. El contenedor de EJB gobierna la vida de un Bean de Sesión.

Beans de Sesión sin estado: son el tipo de componente EJB más simple de implementar. No mantienen ningún estado conversacional con clientes entre invocaciones a métodos por lo que son fácilmente reusables en el lado servidor, y dado que pueden ser almacenados en cachés, se escalan bien bajo demanda. Cuando se usen, hay que almacenar toda información de estado fuera del EJB.

Beans de Sesión con estado: mantienen el estado entre invocaciones. Tienen una correspondencia lógica de uno a uno con el cliente y pueden mantener el estado entre ellas. El responsable de su almacenamiento y paso a caché es el

Contenedor de EJB, que lo logra mediante su Pasivación y Activación. Si el contenedor de EJB falla, se perdería toda la información de los Beans de sesión con estado. Algunos Contenedores de EJB avanzados proporcionan posibilidad de recuperación para estos beans.

Beans de entidad

Son componentes que representan información persistente y comportamiento de estos datos. Estos Beans los pueden compartir múltiples clientes, de la misma forma que puede compartirse la información de una base de datos. El Contenedor de EJB es el responsable de gestionar en la caché los Beans de Entidad, y de mantener su integridad. La vida de estos beans suele ir más allá de la vida del Contenedor de EJB por lo que si éste falla, se espera que el Bean de Entidad siga disponible cuando el contenedor vuelva a estar activo.

Hay dos tipos de Beans de Entidad: los que tienen Persistencia Gestionada por el Contenedor y los de Persistencia Gestionada por el Bean.

Persistencia Gestionada por el Contenedor (**CMP o Container Managed Persistence**). Un Bean de Entidad CMP tiene su persistencia implementada en el Contenedor de EJB. Mediante atributos especificados en el descriptor de distribución, el Contenedor de EJB establecerá correspondencias entre los atributos del Bean de Entidad y algún almacenamiento persistente (generalmente -aunque no siempre- una base de datos). CMP reduce el tiempo de desarrollo del EJB, además de reducir dramáticamente la cantidad de código necesaria.

Persistencia Gestionada por el Bean (**BMP o Bean Managed Persistence**). Un Bean de entidad BMP tiene su persistencia implementada por el proveedor de Enterprise Beans. Éste es responsable de implementar la lógica necesaria para crear un nuevo EJB, actualizar algunos atributos de los EJB, borrar un EJB, y encontrar un EJB a partir de un almacén persistente. Esto suele implicar la escritura de código JDBC para interactuar con una base de datos u otro almacenamiento persistente. Con BMP, el desarrollado tiene control total sobre la gestión de la persistencia del Bean de Entidad.

BMP también proporciona flexibilidad allí donde puede no es posible implementar un CMP. Por ejemplo, si se deseara crear un EJB que envolviera algún código existente en un servidor, se podría escribir la persistencia usando CORBA.

Desarrollar un EJB

Como ejemplo, se implementará como componente EJB el ejemplo 'Tiempo Perfecto" de la sección RMI anterior. El ejemplo será un sencillo Bean de Sesión sin Estado.

Como se mencionó anteriormente, los componentes EJB consisten en al menos una clase (el EJB) y dos interfaces: el Remoto y el Local. Cuando se crea una interfaz remota para un EJB, hay que seguir las siguientes directrices:

1. La interfaz Remota debe ser **public.**
2. La interfaz Remota debe extender a la interfaz **javax.ejb.EJB0bject.**
3. Cada método de la interfaz Remota debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de cualquier excepción específica de la aplicación.
4. Cualquier objeto que se pase como parámetro o valor de retorno (bien directamente o embebido en algún objeto local) debe ser un tipo de datos RMI-IIOP válido (incluyendo otros objetos EJB).

He aquí una interfaz remota simple para el EJB **TiempoPerfecto**:

```
//: c15:ejb:PerfectTime.java
//# You must install the J2EE Java Enterprise
//# Edition from java.sun.com and add j2ee.jar
//# to your CLASSPATH in order to compile
//# this file. See details at java.sun.com.
// Remote Interface of PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
  public long getPerfectTime()
    throws RemoteException;
} ///:~
```

La interfaz Local es la fábrica en la que se creará el componente. Puede definir métodos **create**, para crear instancias de EJB, o **métodosfinder** que localizan EJB existentes y se usan sólo para Beans de entidad. Cuando se crea una interfaz Local para un EJB hay que seguir estas directrices:

1. La interfaz **Local** debe ser **public.**
2. La interfaz **Local** debe extender el interfaz **javax.ejb.EJBHome.**
3. Cada método **create** de la interfaz **Local** debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de una **javax.ejb.CreateException.**
4. El valor de retorno de un método **create** debe ser una interfaz Remota.
5. El valor de retorno de un método **finder** (sólo para Beans de Entidad) debe ser una interfaz Remota o **java.util.Enumeration** o **java.util.Collection.**
6. Cualquier objeto que se pase como parámetro (bien directamente o embebido en un objeto local) debe ser un tipo de datos RMI-IIOP válido (esto incluye otros objetos EJB).

La convención estándar de nombres para las interfaces Locales es tomar el nombre de la interfaz Remoto y añadir "Home" al final. He aquí la interfaz Local para el EJB **TiempoPerfecto**:

```
//: c15:ejb:PerfectTimeHome.java
// Home Interface of PerfectTimeBean.
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTimeHome extends EJBHome {
  public PerfectTime create()
    throws CreateException, RemoteException;
} ///:~
```

Ahora se puede implementar la lógica de negocio. Cuando se cree la clase de implementación del EJB, hay que seguir estas directrices, (nótese que debería consultarse la especificación de EJB para lograr una lista completa de las directrices de desarrollo de Enterprise JauaBeans):

1. La clase debe ser **public.**
2. La clase debe implementar una interfaz EJB (o **javax.ejb.SessionBean** o **javax.ejb.EntityBean)**.
3. La clase debería definir métodos que se correspondan directamente con los métodos de la interfaz Remota. Nótese que la clase no implementa la interfaz Remota; hace de espejo de los métodos de la interfaz Remota pero **no** lanza **java.rmi.RemoteException.**
4. Definir uno o más métodos **ejbCreate(**) para inicializar el EJB.
5. El valor de retorno y los parámetros de todos los métodos deben ser tipos de datos RMI-IIOP válidos.

```
//: c15:ejb:PerfectTimeBean.java
// Simple Stateless Session Bean
// that returns current system time.
import java.rmi.*;
import javax.ejb.*;

public class PerfectTimeBean
  implements SessionBean {
  private SessionContext sessionContext;
  //return current time
  public long getPerfectTime() {
     return System.currentTimeMillis();
  }
  // EJB methods
  public void ejbCreate()
  throws CreateException {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void
  setSessionContext(SessionContext ctx) {
    sessionContext = ctx;
  }
```

```
}///:~
```

Dado que este ejemplo es muy sencillo, los métodos EJB (**ejbCreate**(), **ejbRemove**(), **ejbActivate**(), **ejbPassivate**()) están todos vacíos. Estos métodos son invocados por el Contenedor de EJB y se usan para controlar el estado del componente. El método **setSesionContext**() pasa un objeto **javax.ejb.SessionContext** que contiene información sobre el contexto del componente, como información de la transacción actual y de seguridad.

Tras crear el Enterprise JavaBean, tenemos que crear un descriptor de distribución. Éste es un archivo XML que describe el componente EJB, y que debería estar almacenado en un archivo denominado **ebj-jar.xml**.

```
//:! c15:ejb:ejb-jar.xml
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-
jar_1_1.dtd'>

<ejb-jar>
  <description>Example for Chapter 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>PerfectTime</ejb-name>
      <home>PerfectTimeHome</home>
      <remote>PerfectTime</remote>
      <ejb-class>PerfectTimeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
///:~
```

Pueden verse el componente, la interfaz Remota y la interfaz Local definidos dentro de la etiqueta <**session**> de este descriptor de distribución. Estos descriptores pueden ser generados automáticamente usando las herramientas de desarrollo de EJB.

Junto con el descriptor de distribución estándar **ejb-jar.xml**, la especificación EJB establece que cualquier etiqueta específica de un vendedor debería almacenarse en un archivo separado. Esto pretende lograr una alta portabilidad entre componentes y distintos tipos de contenedores de EJB.

Los archivos deben guardarse dentro de un Archivo Java estándar (JAR). Los descriptores de distribución deberían ubicarse dentro del subdirectorio **/META-INF** del archivo Jar.

Una vez que se ha definido el componente EJB dentro del descriptor de distribución, el distribuidor debería distribuir el componente EJB en el Contendor de EJB. Al escribir esto, el proceso de distribución era bastante "intensivo respecto a I G U y específico de cada contendor de EJB individual, por lo que este repaso no documenta ese proceso. Todo contenedor de EJB, sin embargo, tendrá un proceso documentado para la distribución de un EJB.

Dado que un componente EJB es un objeto distribuido, el proceso de distribución también debería crear algunos **stubs** de cliente para invocar al componente. Estas clases deberían ubicarse en el **classpath** de la aplicación cliente. Dado que los componentes EJB pueden implementarse sobre RMI-IIOP (CORBA) o sobre RMI-JRMP, los **stubs** generados podrían variar entre contenedores de EJB; de cualquier manera, son clases generadas.

Cuando un programa cliente desea invocar a un EJB, debe buscar el componente EJB dentro de la JNDI y obtener una referencia al interfaz local del componente EJB. Esta interfaz se usa para crear una instancia del EJB.

En este ejemplo, el programa cliente es un programa Java simple, pero debería recordarse que podría ser simplemente un servlet, un JSP o incluso un objeto distribuido CORBA o RMI.

```
//: c15:ejb:PerfectTimeClient.java
// Client program for PerfectTimeBean

public class PerfectTimeClient {
public static void main(String[] args)
throws Exception {
  // Get a JNDI context using
  // the JNDI Naming service:
  javax.naming.Context context =
    new javax.naming.InitialContext();
  // Look up the home interface in the
  // JNDI Naming service:
  Object ref = context.lookup("perfectTime");
  // Cast the remote object to the home interface:
  PerfectTimeHome home = (PerfectTimeHome)
    javax.rmi.PortableRemoteObject.narrow(
      ref, PerfectTimeHome.class);
  // Create a remote object from the home interface:
  PerfectTime pt = home.create();
  // Invoke  getPerfectTime()
  System.out.println(
```

```
      "Perfect Time EJB invoked, time is: " +
      pt.getPerfectTime() );
    }
} ///:~
```

La secuencia de este ejemplo se explica mediante comentarios. Nótese el uso del método **narrow**() para llevar a cabo una conversión del objeto antes de llevar a cabo una conversión Java autentica. Esto es muy semejante a lo que ocurre en CORBA. Nótese también que el objeto **Home** se convierte en una fábrica de objetos **TiempoPerfecto**.

Resumen de EJB

La especificación de Enterprise JavaBeans es un terrible paso adelante de cara a la estandarización y simplificación de la computación de objetos distribuidos. Es una de las piezas más importantes de la plataforma Java 2 Enterprise Edition J2EE) y está recibiendo mucho soporte de la comunidad de objetos distribuidos. Actualmente hay muchas herramientas disponibles, y si no lo estarán en breve, para ayudar a acelerar el desarrollo de componentes EJB.

Este repaso sólo pretendía ser un breve tour por los EJB. Para más información sobre la especificación de EJB puede acudirse a la página oficial de los Enterprise JavaBeans en http:/hava.sun.com/products/ejb donde puede descargarse la última especificación y la implementación de referencia J2EE. Éstas pueden usarse para desarrollar y distribuir componentes EJB propios.

Jini: servicios distribuidos

Esta sección **[8]** da un repaso a la tecnología Jini de Sun Microsystems. Describe algunas de las ventajas e inconvenientes de Jini y muestra cómo su arquitectura ayuda a afrontar el nivel de abstracción en programación de sistemas distribuidos, convirtiendo de forma efectiva la programación en red en programación orientada a objetos.

[8] Esta sección se llevó a cabo con la ayuda de Bill Venners (*http://www.artima.com*).

Jini en contexto

Tradicionalmente, se han diseñado los sistemas operativos con la presunción de que un computador tendría un procesador, algo de memoria, y un disco. Cuando se arranca un computador lo primero que hace es buscar un disco. Si no lo encuentra, no funciona como computador. Cada vez más, sin embargo, están apareciendo computadores de otra guisa: como dispositivos embebidos que

tienen un procesador, algo de memoria, y una conexión a la red -pero no disco. Lo primero que hace un teléfono móvil al arrancarlo, por ejemplo, es buscar una red de telefonía. Si no la encuentra, no puede funcionar como teléfono móvil. Esta tendencia en el entorno hardware, de centrados en el disco a centrados en la red, afectará a la forma que tenemos de organizar el software -y es aquí donde Jini cobra sentido.

Jini es un intento de replantear la arquitectura de los computadores, dada la importancia creciente de la red y la proliferación de procesadores en dispositivos sin unidad de disco. Estos dispositivos, que vendrán de muchos fabricantes distintos, necesitarán interactuar por una red. La red en sí será muy dinámica -regularmente se añadirán y eliminarán dispositivos y servicios. Jini proporciona mecanismos para permitir la adición, eliminación y búsqueda de directorios y servicios de forma sencilla a través de la red. Además, Jini proporciona un modelo de programación que facilita a los programadores hacer que sus dispositivos se comuniquen a través de la red.

Construido sobre Java, la serialización de objetos, y RMI (que permiten, entre todos, que los objetos se muevan por la red de máquina virtual en máquina virtual), Jini intenta extender los beneficios de la programación orientada a objetos a la red. En vez de pedir a fabricantes de dispositivos que se pongan de acuerdo en protocolos de red para que sus dispositivos interactúen, Jini habilita a los dispositivos para que se comuniquen mutuamente a través de interfaces con objetos.

¿Qué es Jini?

Jini es un conjunto de API y protocolos de red que pueden ayudar a construir y desplegar sistemas distribuidos organizados como federaciones de servicios. Un servicio puede ser cualquier cosa que resida en la red y que esté listo para llevar a cabo una función útil. Los dispositivos hardware, el software, los canales de comunicación -e incluso los propios seres humanos- podrían ser servicios. Una unidad de disco habilitada para Jini, por ejemplo, podría ofrecer un servicio de "almacenamiento". Una impresora habilitada para Jini podría ofrecer un servicio de "impresión". Por consiguiente, una federación de servicios es un conjunto de servicios disponible actualmente en la red, que un cliente (entendiendo por tal un programa, servicio o usuario) puede juntar para que le ayuden a lograr alguna meta.

Para llevar a cabo una tarea, un cliente lista la ayuda de varios servicios. Por ejemplo, un programa cliente podría mandar a un servidor fotos del almacén de imágenes de una cámara digital, descargar las fotos a un servicio de almacenamiento persistente ofrecido por una unidad de disco, y enviar una página de versiones del tamaño de una de las fotos al servicio de impresión de una impresora a color. En este ejemplo, el programa cliente construye un sistema

distribuido que consiste en sí mismo, el servicio de almacenamiento de imágenes, el servicio de almacenamiento persistente, y el servicio de impresión en color. El cliente y los servicios de este sistema distribuido trabajan juntos para desempeñar una tarea: descargar y almacenar imágenes de una cámara digital e imprimir una página de copias diminutas.

La idea que hay tras la palabra federación es que la visión de Jini de la red no implique una autoridad de control central. Dado que no hay ningún servicio al mando, el conjunto de todos los servicios disponibles en la red conforman una federación -un grupo formado por iguales. En vez de una autoridad central, la infraestructura de tiempo de ejecución de Jini simplemente proporciona una forma para que los clientes y servicios se encuentren entre sí (vía un servicio de búsqueda que almacena un directorio de los servicios actualmente disponibles). Una vez que los servicios se localizan entre sí ya pueden funcionar. El cliente y sus servicios agrupados llevan a cabo su tarea independientemente de la infraestructura de tiempo de ejecución de Jini. Si el servicio de búsqueda de Jini se cae, cualquier sistema distribuido conformado vía el servicio de búsqueda antes de que se produjera el fallo puede continuar con su trabajo. Jini incluso incluye un protocolo de red que pueden usar los clientes para encontrar servicios en la ausencia de un servicio de búsqueda.

Cómo funciona Jini

Jini define una infraestructura de tiempo de ejecución que reside en la red y proporciona mecanismos que permiten a cada uno añadir, eliminar, localizar y acceder a servicios. La infraestructura de tiempo de ejecución reside en tres lugares: en servicios de búsqueda que residen en la red, en los proveedores de servicios (como los dispositivos habilitados para Jini), y en los clientes. Los Servicios de búsqueda son el mecanismo de organización central de los sistemas basados en Jini. Cuando aparecen más servicios en la red, se registran a sí mismos con un servicio de búsqueda. Cuando los clientes desean localizar un servicio para ayudar con alguna tarea, consultan a un servicio de búsqueda.

La infraestructura de tiempo de ejecución usa un protocolo de nivel de red denominado discovery, y dos protocolos de nivel de red, llamados join y lookup. **Discovery** permite a los clientes y servicios localizar los servicios de búsqueda. Join permite a un servicio registrarse a sí mismo en un servicio de búsqueda. Lookup permite a un cliente preguntar por los servicios que pueden ayudar a lograr sus metas.

El proceso de discovery

Discovery funciona así: imagínese que se tiene una unidad de disco habilitada para Jini que ofrece un servicio de almacenamiento persistente. Tan pronto como se conecte el disco a la red, éste lanza en multidifusión un anuncio de presencia

depositando un paquete de multidifusión en un puerto bien conocido. En este anuncio de presencia va incluida la dirección IP y el número de puerto en el que el servicio de búsqueda puede localizar la unidad de disco.

Los servicios de búsqueda monitorizan el puerto bien conocido en espera de paquetes de anuncio de presencia. Cuando un servicio de búsqueda recibe un anuncio de presencia, lo abre e inspecciona el paquete. Éste contiene información que permite al servicio de búsqueda determinar si debería o no contactar al emisor del paquete. En caso afirmativo, contacta con el emisor directamente haciendo una conexión TCP a la dirección IP y número de puerto extraídos del paquete. Usando RMI, el servicio de búsqueda envía un objeto denominado registrador de servicios, a través de la red, a la fuente del paquete. El propósito del servicio registrador de objetos es facilitar una comunicación más avanzada con el servicio de búsqueda. Al invocar a los métodos de este objeto, el emisor del paquete de anuncio puede unirse y buscar en el servicio de búsqueda. En el caso de la unidad de disco, el servicio de búsqueda haría una conexión TCP a la unidad de disco y le enviaría un objeto registrador de servicios, a través del cual registraría después la unidad de disco su servicio de almacenamiento persistente vía el proceso join.

El proceso join

Una vez que un proveedor de servicio tiene un objeto registrador de servicios, como resultado del **discovery**, está listo para hacer una join - convertirse en parte de la federación de servicios registrados en el servicio de búsqueda. Para ello, el proveedor del servicio invoca al método **register**() del objeto registrador de servicios, pasándole como parámetro un objeto denominado un elemento de servicio, un conjunto de objetos que describen el servicio. El método **register**() envía una copia del elemento de servicio al servicio de búsqueda, donde se almacena el elemento de servicio.

Una vez completada esta operación, el proveedor del servicio ha acabado el proceso **join**: su servicio ya está registrado en el servicio de búsqueda.

Este elemento de servicio es un contenedor de varios objetos, incluyendo uno llamado un objeto de servicio, que pueden usar los clientes para interactuar con el servicio. El elemento de servicio también puede incluir cualquier número de atributos, que pueden ser cualquier objeto. Algunos atributos potenciales son iconos, clases que proporcionan IGU para el servicio, y objetos que proporcionan más información del servicio.

Los objetos de servicio suelen implementar uno o más interfaces a través de los cuales los clientes interactúan con el servicio. Por ejemplo, un servicio de búsqueda es un servicio Jini, y su objeto de servicio está en el registrador de servicios. El método **register**() invocado por los proveedores de servicio durante

el **join** se declara en la interfaz **ServiceRegistrar** (miembro del paquete **net.jini.core.lookup**), que implementan todos los objetos registradores de servicios. Los clientes y proveedores de servicios se comunican con el servicio de búsqueda a través del objeto registrador de servicios invocando a los métodos declarados en la interfaz **ServiceRegistrar**. De manera análoga, una unidad de disco proporcionaría un objeto servicio que implementaba alguna interfaz de servicio de almacenamiento bien conocido. Los clientes buscarían e interactuarían con la unidad de disco a través de esta interfaz de servicio de almacenamiento.

El proceso lookup

Una vez que se ha registrado un servicio con un servicio **lookup** vía el proceso **join**, ese servicio está disponible para ser usado por clientes que pregunten al servicio de búsqueda. Para construir un sistema distribuido que trabaje en conjunción para desempeñar alguna tarea un cliente debe localizar y agrupar la ayuda de servicios individuales. Para encontrar un servicio, los clientes preguntan a servicios de búsqueda vía un proceso llamado **lookup**.

Para llevar a cabo una búsqueda, un cliente invoca al método **lookup**() de un objeto registrador de servicios. (Un cliente, como un proveedor de servicios, logra un registrador de servicios a través del proceso de **discovery** anteriormente descrito). El cliente pasa una plantilla de servicio como parámetro al **lookup**(). Esta plantilla es un objeto que sirve como criterio de búsqueda para la consulta. La plantilla puede incluir una referencia a un array de objetos **Class**. Estos objetos **Class** indican al servicio de búsqueda el tipo (o tipos) Java del objeto servicio deseados por el cliente. La plantilla también puede incluir un ID de servicio, que identifica un servicio de manera unívoca, y atributos, que deben casar exactamente con los atributos proporcionados por el proveedor de servicios en el elemento de servicio. La plantilla de servicio también puede contener comodines para alguno de estos campos. Un comodín en el campo ID del servicio, por ejemplo, casará con cualquier ID de servicio.

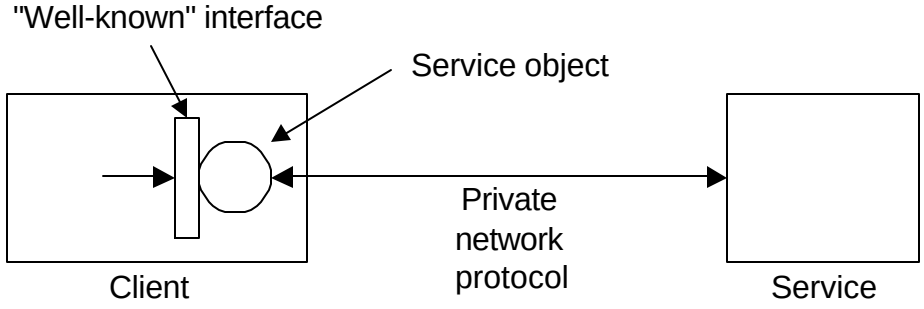
El método **lookup**() envía la plantilla al servicio **lookup**, que lleva a cabo la consulta y devuelve cero a cualquier objeto de servicio que case. El cliente logra una referencia a los objetos de servicio que casen como valor de retorno del método **lookup**().
En el caso general, un cliente busca un servicio por medio de un tipo Java, generalmente una interfaz. Por ejemplo, si un cliente quisiese usar una impresora, compondría una plantilla de servicio que incluiría un objeto **Class** para una interfaz bien conocida. El servicio de búsqueda devolvería un objeto (o varios) de servicio que implementa esta interfaz. En la plantilla de servicio pueden incluirse atributos para reducir el número de coincidencias para una búsqueda de este tipo. El cliente usaría el servicio de impresión invocando a los métodos de la interfaz de impresión bien conocida del objeto de servicio.

Separación de interfaz e implementación

La arquitectura de Jini acerca la programación orientada a la red permitiendo a los servicios de red tomar ventaja de uno de los aspectos fundamentales de los objetos: la separación de interfaz e implementación. Por ejemplo, un objeto de servicio puede garantizar a los clientes acceso al servicio de muchas formas. El objeto puede representar, de hecho, todo el servicio, que es descargado al cliente durante la búsqueda, y después ejecutado localmente. Después, cuando el cliente invoca a métodos del objeto servicio, envía las peticiones al servidor a través de la red, que hace el trabajo real. Una tercera opción es que el objeto servicio local y un servidor remoto hagan parte del trabajo cada uno.

Una consecuencia importante de la arquitectura de Jini e s que el protocolo de red usado para comunicarse entre un objeto de servicio intermediario y un servidor remoto no tiene por qué ser conocido por el cliente. Como se muestra en la siguiente figura, el protocolo de red es parte de la implementación del servicio. El cliente puede comunicarse con el servicio vía el protocolo privado porque el servicio inyecta parte de su propio código (el objeto servicio) en el espacio de reacciones del cliente. El objeto de servicio inyectado podría comunicarse con el servicio vía RMI, CORBA, DCOM, algún protocolo casero construido sobre sockets y flujos, o cualquier otra cosa. El cliente simplemente no tiene que preocuparse de protocolos de red, puede comunicarse con la interfaz bien conocida que implementa el objeto de servicio. El objeto de servicio se encarga de cualquier comunicación necesaria por la red.

Distintas implementaciones de una misma interfaz de servicio pueden usar enfoques y protocolos de red completamente diferentes. Un servicio puede usar hardware especializado para completar las solicitudes del cliente, o puede hacer todo su trabajo vía software. De hecho, el enfoque de implementación tomado por un único servicio puede evolucionar con el tiempo. El cliente puede estar seguro de tener un objeto de servicio que entiende la implementación actual del servicio, porque el cliente recibe el objeto de servicio (por parte del servicio de búsqueda) del propio proveedor de servicios. Para el cliente, un servicio tiene el aspecto de una interfaz bien conocida, independientemente de cómo esté implementado el servicio.



El cliente se comunica con el servicio a través de una interfaz bien conocida.

Abstraer sistemas distribuidos

Jini intenta elevar el nivel de abstracción para programación de sistemas distribuidos, desde el nivel del protocolo de red al nivel de interfaz del objeto. En la proliferación emergente de dispositivos embebidos conectados a redes puede haber muchas piezas de un sistema distribuido que provengan de distintos fabricantes. Jini hace innecesario que los fabricantes de dispositivos se pongan de acuerdo en los protocolos de nivel de red que permitan a sus dispositivos interactuar. En vez de ello, los fabricantes deben estar de acuerdo en las interfaces Java a través de las cuales pueden interactuar sus dispositivos. Los procesos de **discovery**, **join**, y **lookup** proporcionados por la infraestructura de tiempo de ejecución de Jini permiten a los dispositivos localizarse entre sí a través de la red. Una vez que se localizan, los dispositivos pueden comunicarse mutuamente a través de interfaces Java.

Resumen

Junto con Jini para redes de dispositivos locales, este capítulo ha presentado algunos, pero no todos los componentes a los que Sun denomina J2EE: la Java 2 Enterprise Edition. La meta de J2EE es construir un conjunto de herramientas que permitan a un desarrollador Java construir aplicaciones basadas en servidores mucho más rápido, y de forma independiente de la plataforma. Construir aplicaciones así no sólo es difícil y consume tiempo, sino que es especialmente difícil construirlas de forma que puedan ser portadas fácilmente a otras plataformas, y además mantener la lógica de negocio separada de los detalles de implementación subyacentes. J2EE proporciona un marco de trabajo para ayudar a crear aplicaciones basadas en servidores; estas aplicaciones tienen gran demanda hoy en día, y esa demanda parece, además, creciente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento The Thinking in Java Annotated Solution Guide, disponible a bajo coste en ***http://www.BruceEckel.com.***

1. Compilar y ejecutar los programas **Servidorparlante** y **Clienteparlante** de este capítulo. Ahora, editar los archivos para eliminar todo espacio de almacenamiento intermedio en las entradas y salidas, después compilar y volver a ejecutarlos para observar los resultados.

2. Crear un servidor que pida una contraseña, después un archivo y enviarlo a través de la conexión de red. Crear un cliente que se conecte al servidor,

proporcionar la contraseña adecuada, y después capturar y salvar el archivo. Probar los dos programas en la misma máquina usando el **localhost** (la dirección IP de bucle local 127.0.0.1 producida al invocar a **InetAddress.getByName(null)).**

3. Modificar el servidor del Ejercicio 2, de forma que use el multihilo para manejar múltiples clientes.

4. Modificar **ClientePar1ante.java** de forma que no se dé el vaciado de la salida y se observe su efecto.

5. Modificar **ServidorMultiParlante** de forma que use *hilos cooperativos.* En vez de lanzar un hilo cada vez que un cliente se desconecte, el hilo debería pasar a un "espacio de hilos disponibles". Cuando se desee conectar un nuevo cliente, el servidor buscará en este espacio un hilo que gestione la petición, y si no hay ninguno disponible, construirá uno nuevo. De esta forma, el número de hilos necesario irá disminuyendo en cuanto a la cantidad necesaria. La aportación de hilos cooperativos es que no precisa de sobrecarga para la creación y destrucción de hilos nuevos por cada cliente.

6. A partir de **MostrarHTML.java**, crear un applet que sea una pasarela protegida por contraseña a una porción particular de un sitio web.
7. Modificar **CrearTablasCID.java**, de forma que lea las cadenas de caracteres SQL de un texto en vez de CIDSQL.

8. Configurar el sistema para que se pueda ejecutar con éxito **CrearTablasCID.java** y **CargarBD.java**.

9. Modificar **ReglasServlets.java** superponiendo el método **destroy**() para que salve el valor de i a un archivo, y el método hit( ) para que restaure el valor. Demostrar que funciona volviendo a arrancar el contenedor de servlets. Si no se tiene un contenedor de servlets, habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

10. Crear un servlet que añada un *cookie* al objeto respuesta, almacenándolo en el lado cliente. Añadir al servlet el código que recupera y muestra el *cookie.* Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org para poder ejecutar servlets.*

11. Crear un servlet que use un objeto **Session** para almacenar la información de sesión que se seleccione. En el mismo servlet, retirar y mostrar la información de sesión. Si no se tiene un contenedor de servlets habrá que

descargar, instalar y ejecutar Tomcat de http://jakarta.apache.org para poder ejecutar servlets.

12. Crear un servlet que cambie el intervalo inactivo de una sesión a 5 segundos invocando a **getMaxInactiveInterval**(). Probar que la función llegue a expirar tras 5 segundos. Si no se tiene un contenedor de servlets habrá que descargar, instalar y ejecutar Tomcat de *http:/jakarta.apache.org* para poder ejecutar servlets.

13. Crear una página JSP que imprima una línea de texto usando la etiqueta **<H1>**. Poner el color de este texto al azar, utilizando código Java embebido en la página JSP Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de http://jakarta.apache.org para poder ejecutar servlets.

14. Modificar el valor de la edad máxima de **Cookies.jsp** y observar el comportamiento bajo dos navegadores diferentes. Notar también la diferencia entre revisitar la página y apagar y volver a arrancar el navegador. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

15. Crear un JSP con un campo que permita al usuario introducir el tiempo de expiración de la sesión y un segundo campo que guarde la fecha almacenada en la sesión. El botón de envío refresca la página y captura la hora de expiración actual además de la información de la sesión y las coloca como valores por defecto en los campos antes mencionados. Si no se tiene un contenedor de JSP habrá que descargar, instalar y ejecutar Tomcat de *http://jakarta.apache.org* para poder ejecutar servlets.

16. (Más complicado) tomar el programa **BuscarV.java** y modificarlo, de forma que al hacer clic en el nombre resultante tome automáticamente el nombre y lo copie al portapapeles (de forma que se pueda simplemente pegar en el correo electrónico). Habrá que echar un vistazo al Capítulo 13 para recordar como usar el portapapeles en JFC.

17: Patrones de diseño

Este capítulo introduce los acercamientos de "patrones" importantes y todavía poco tradicionales para el diseño de programas.

Probablemente el paso más importante adelante en el diseño orientado a objetos es los movimientos de *"patrones de diseño"*, puesto en crónicas en *Patrones de Diseño*, por *Gama*, *Helm*, *Johnson* y *Vlissides* (*Addison-Wesley* 1995). **[1]** Este el libro le muestra 23 soluciones diferentes a clases particulares de problemas. En este capítulo, los conceptos básicos de patrones de diseño serán introducidos junto con varios ejemplos. Esto debería agudizar tu apetito para leer los Patrones de Diseño (una fuente de que ahora se ha convertido en un vocabulario esencial, casi obligatorio, para programadores OOP).

[1] Pero esté advertido: Los ejemplos están en C++.

La parte más reciente de este capítulo contiene un ejemplo del proceso de evolución del diseño, comenzando con una solución inicial y moviéndose a través de la lógica y el proceso de desarrollar la solución para los diseños más correctos. El programa mostrado (una simulación de ordenación de basura) ha evolucionado con el paso del tiempo, y puedes considerar esa evolución como un prototipo para la manera que tu propio diseño puede arrancar como una solución adecuada para un problema particular y puede evolucionar en un acercamiento flexible para una clase de problemas.

El concepto de patrón

Inicialmente, puedes pensar acerca de un patrón como una forma especialmente brillante y penetrante de solucionar una clase particular de problemas. Es decir, tal parece ser que un gran número de personas ha resuelto todos los ángulos de un problema y ha sacado de entre manos la solución más general, flexible para eso. El problema podría ser uno que has visto y has solucionado antes, excepto que tu solución probablemente no tuvo el tipo de integridad que verás envuelto en un patrón.

Aunque son llamados "patrones de diseño", realmente no están ocupados con el área de diseño. Un patrón parece destacarse sobre la forma de pensar tradicional acerca de análisis, diseño, e implementación. En lugar de eso, un patrón encarna una idea completa dentro de un programa, y así algunas veces puede aparecer en la fase de análisis o la fase del diseño de alto nivel. Esto es interesante porque un patrón tiene una implementación directa en código y así es que no podrías esperar que eso aparezca antes de implementación o diseño de bajo nivel (y de hecho no podrías darte cuenta de que necesitas un patrón particular hasta que te acerques a esas fases).

El concepto básico de un patrón también puede ser visto como el concepto básico de diseño de programa: Agregando una capa de abstracción. Cada vez que abstraes algo que aísla detalles particulares, y una de las motivaciones más apremiantes detrás de éste son separar cosas que cambian de cosas que permanecen igual. Otra forma de poner esto es que una vez que encuentras alguna parte de tu programa que tiene probabilidad de cambiar pues uno razona u otro, querrás librar esos cambios de propagar otros cambios a todo lo largo de tu código. No sólo le hace a esta marca el código mucho más barato para mantener, sino que también resulta que es usualmente más simple entender (que dé como resultado costos reducidos).

A menudo, la parte más difícil de desarrollar un diseño elegante y barato para mantener está en descubrir lo que llamo “el vector de cambio.” (Aquí, “vector” se refiere al máximo gradiente y no a una colección de la clase.) Esto significa encontrar la cosa más importante que cambia en tu sistema, o pone de otro modo, descubriendo dónde está tu costo máximo. Una vez que descubres el vector de cambio, tienes el punto focal alrededor del cual estructura tu diseño.

Así es que la meta de patrones de diseño es aislar cambios en tu código. Si lo miras así, has estado viendo a algunos diseñar patrones ya en este libro. Por ejemplo, la herencia puede ser considerada como un patrón de diseño (si bien uno implementado por el compilador). Te permite expresar diferencias en el comportamiento en objetos (esa es la cosa que cambia) que todos tiene la misma interfaz (eso es lo que permanece igual). La composición también puede ser considerada un patrón, desde que te permite cambiar – dinámicamente o estáticamente – los objetos que implementan tu clase, y así la manera que trabaja la clase.

También ya has visto otro patrón que aparece en Patrones de Diseño: El iterador (Java 1.0 y 1.1 caprichosamente le llaman **Enumeration**; las colecciones Java 1.2 usan “iterador”). Esto esconde la implementación particular de la colección como estás dando un paso enteramente y seleccionando los elementos uno por uno. El iterador te permite escribir código genérico que realiza una operación en todos los elementos en una secuencia sin hacer caso de la forma que la secuencia es construida. Así tu código genérico puede ser usado con cualquier colección que puede producir un iterador.

El singleton

Posiblemente el patrón más simple del diseño es el singleton, lo cual es una forma para proveer a la única e incomparable instancia de un objeto. Esto es usado en las librerías Java, pero aquí hay un ejemplo más directo :

```
//: SingletonPattern.java
// The Singleton design pattern: you can
// never instantiate more than one.
package c16;

// Since this isn't inherited from a Cloneable
```

```
// base class and cloneability isn't added,
// making it final prevents cloneability from
// being added in any derived classes:
final class Singleton {
  private static Singleton s = new Singleton(47);
  private int i;
  private Singleton(int x) { i = x; }
  public static Singleton getHandle() {
    return s;
  }
  public int getValue() { return i; }
  public void setValue(int x) { i = x; }
}

public class SingletonPattern {
  public static void main(String[] args) {
    Singleton s = Singleton.getHandle();
    System.out.println(s.getValue());
    Singleton s2 = Singleton.getHandle();
    s2.setValue(9);
    System.out.println(s.getValue());
    try {
      // Can't do this: compile-time error.
      // Singleton s3 = (Singleton)s2.clone();
    } catch(Exception e) {}
  }
} ///:~
```

La clave para crear a un singleton es impedirle al programador del cliente tener de cualquier forma crear un objeto excepto las formas que provees. Debes hacer a todos los constructores **private**, y debes crear al menos un constructor para impedirle al compilador sintetizar a un constructor predeterminado para ti (el cual lo creará como **friendly**).

En este punto, decides cómo vas a crear tu objeto. Aquí, es creado estáticamente, pero también puedes esperar hasta que el programador cliente le pide uno y lo cree a petición. En todo caso, el objeto debería guardarse privadamente. Provees acceso a través de los métodos públicos. Aquí, **getHandle**() produce el manipulador para el objeto **Singleton**. El resto de la interfaz (**getValue**() y **setValue**()) es la interfaz normal de clase.

Java también permite la creación de objetos a través de la clonación. En este ejemplo, hacer la clase final impide clonación. Ya que **Singleton** es heredado directamente de **Object**, el método **clone**() permanece protegido así es que no puede ser usado (haciéndolo de esta forma produce un error de fase de compilación). Sin embargo, si heredas de una jerarquía de clase que ya ha sobrescrito **clone**() como **Cloneable** público e implementado, la forma para impedir la clonación es sobrescribir a **clone**() y lanzar a un **CloneNotSupportedException** como se describió en el Capítulo 12. (También podrías sobrepujar a **clone**() y simplemente retorna esto, pero eso sería engañoso ya que el cliente programador pensaría que clonaban el objeto, pero en lugar de eso todavía se ocuparían del original.)

Noto que no estás restringido a la creación de un sólo objeto. Ésta es también una técnica para crear una alberca limitada de objetos. En esa situación, sin embargo, puedes ser enfrentado con el problema de compartir objetos en la alberca. Si éste es un asunto, puedes crear una solución involucrando una comprobación y el registro de los objetos compartidos.

Clasificando patrones

El libro de *Patrones de Diseño* discute 23 patrones diferentes, clasificados bajo tres propósitos (todo el cual se refiere al aspecto particular que puede variar). Los tres propósitos son:

1. **Creational**: Cómo puede ser un objeto creado. Esto a menudo involucra a aislar los detalles de creación del objeto así es que tu código no está bajo la dependencia de lo que allí son los tipos de objetos y así no tiene que variarse cuando agregas un tipo nuevo de objeto. Dicho **Singleton** está clasificado como un patrón creacional, y más adelante en este capítulo verás ejemplos de Método de Fábrica y el Prototipo.
2. **Estructural**: Diseñando objetos para satisfacer restricciones particulares de proyecto. Estos surten efecto con la forma que los objetos están conectados con otros objetos para asegurar que cambios en el sistema no requieren cambios a esas conexiones.
3. **Conductista**: Los objetos que manejan tipos de detalle de acciones dentro de un programa. Estos narran de forma resumida procesos que quieres realizar, como interpretar un lenguaje, cumpliendo con una petición, moviéndose a través de una secuencia (como en un iterador), o implementando un algoritmo. Este capítulo contiene ejemplos de los patrones **Observador** y **Visitador**.

El libro de *Patrones de Diseño* tiene una sección en cada uno de sus 23 patrones junto con uno o más ejemplos para cada uno, típicamente en C++ pero a veces en *Smalltalk*. (Te encontrarás con que éste no tiene mucha importancia desde que fácilmente puedas traducir los conceptos de ya sea lenguaje dentro de Java.) Este libro no repetirá todos los patrones mostrados en Patrones de Diseño ya que ese libro mantiene su propio rumbo y debería ser estudiado separadamente. En lugar de eso, este capítulo dará algunos ejemplos que te deberían proveer de que una percepción aceptable de qué se tratan los patrones y por qué son tan importantes.

El patrón del observador

El patrón del observador soluciona un problema medianamente común: ¿Qué ocurre si un grupo de objetos necesita actualizarse cuando algún objeto cambia el estado? Esto puede verse en el aspecto "*vista modelo*" del *MVC* de *Smalltalk* (el controlador de vista modelo), o el casi equivalente "Arquitectura de Vista de Documento." Supongo que tienes algunos datos (el "documento") y más que una vista, dices un plan y una vista textual. Cuando cambias los datos, los dos puntos de vista deben saber actualizarse ellos mismos, y que es lo que el observador facilita. Es un problema bastante común que su solución ha sido hecha una parte de la librería estándar del **java.util**.

Hay dos tipos de objetos usados para implementar el patrón del observador en Java. La clase **Observable** le sigue la pista a todo el que quiere serle informado cuando un cambio ocurre, si el "estado" ha cambiado o no. Cuando alguien dice *"OK, todo el mundo los debería comprobar y potencialmente se debería actualizar*", la clase **Observable** realiza esta tarea llamando el método **notifyObservers**() para cada uno en la lista. El método **notifyObservers**() es en parte de la clase base **Observable**.

Hay realmente dos "*cosas que cambian*" en el patrón del observador: La cantidad de objetos observadores y la forma que ocurre una actualización. Es decir, el patrón del observador te permite modificar ambos de estos sin afectar el código circundante.

El siguiente ejemplo es similar al ejemplo **ColorBoxes** de Capítulo 14. Las cajas son colocadas en una cuadrícula en la pantalla y cada uno es inicializado a un color aleatorio. Además, cada caja implementa la interfaz **Observer** y está registrada con un objeto **Observable**. Cuando das un clic sobre una caja, todos los demás cajas son notificadas que un cambio ha estado hecho porque el objeto **Observable** automáticamente llama el método **update**() de cada objeto del **Observable**. Dentro de este método, la caja inspecciona para ver si está adyacente para el que fue hecho clic, y si es así cambia su color para corresponder a la caja hecha clic.

```
//: BoxObserver.java
// Demonstration of Observer pattern using
// Java's built-in observer classes.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// You must inherit a new type of Observable:
class BoxObservable extends Observable {
  public void notifyObservers(Object b) {
    // Otherwise it won't propagate changes:
    setChanged();
    super.notifyObservers(b);
  }
}

public class BoxObserver extends Frame {
  Observable notifier = new BoxObservable();
  public BoxObserver(int grid) {
    setTitle("Demonstrates Observer pattern");
    setLayout(new GridLayout(grid, grid));
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        add(new OCBox(x, y, notifier));
  }
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
      grid = Integer.parseInt(args[0]);
    Frame f = new BoxObserver(grid);
```

```
    f.setSize(500, 400);
    f.setVisible(true);
    f.addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      });
  }
}

class OCBox extends Canvas implements Observer {
  Observable notifier;
  int x, y; // Locations in grid
  Color cColor = newColor();
  static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  OCBox(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    this.notifier = notifier;
    addMouseListener(new ML());
  }
  public void paint(Graphics  g) {
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      notifier.notifyObservers(OCBox.this);
    }
  }
  public void update(Observable o, Object arg) {
    OCBox clicked = (OCBox)arg;
    if(nextTo(clicked)) {
      cColor = clicked.cColor;
      repaint();
    }
  }
  private final boolean nextTo(OCBox b) {
    return Math.abs(x - b.x) <= 1 &&
```

```
            Math.abs(y - b.y) <= 1;
  }
} ///:~
```

Cuándo primero miras la documentación en línea para **Observable**, un poco confunde porque parece que puedes usar un objeto **Observable** común para manejar las actualizaciones. Pero esto no trabaja; Pruébalo – dentro de **BoxObserver**, crea un objeto **Observable** en lugar de un objeto **BoxObservable** y vea qué ocurre: Nada. Obtener un efecto, debes heredar de Observable y en alguna parte de tu código de clase derivada llama a **setChanged**(). Éste es el método que coloca la bandera "cambiada", lo cual quiere decir que cuando llamas a **notifyObservers**() todos los observadores, de hecho, serán notificados. En el ejemplo arriba de **setChanged**() es simplemente llamado dentro de **notifyObservers**(), pero podrías usas cualquier criterio que quieres decidir cuando llamas a **setChanged**().

BoxObserver contiene un sencillo objeto Observable llamado **notifier**, y cada vez que un objeto del **OCBox** es creado, está atado a **notifier**. En **OCBox**, cada vez que das un clic sobre el ratón el método **notifyObservers**() es llamado, pasando el objeto hecho clic como un argumento así es que todas las cajas que reciben el método **message** (en su **update**()) conoce quién fue hecho clic y puede decidirse ya sea a cambiarse o no. Usando una combinación de código en **notifyObservers**() y **update**() puedes trabajar fuera de algunos esquemas medianamente complicados.

Podría parecer que la forma como los observadores le son notificados debe estar congelado en la fase de compilación en el método **notifyObservers**(). Sin embargo, si ves más de cerca en el código arriba verás que el único lugar en **BoxObserver** u **OCBox** donde estás consciente de que estés trabajando con un **BoxObservable** está a punto de la creación del objeto **Observable** – desde entonces todos usan la interfaz **Observable** básica. Esto quiere decir que podrías heredar otras clases **Observable** y las puedes intercambiar en el tiempo de ejecución si quieres cambiar el comportamiento de notificación luego.

Simulando el reciclado de basura

La naturaleza de este problema es que la basura es tirado sin clasificar en un sencillo depósito, así es que la información específica de tipo se pierde. Excepto más tarde, el tipo especifica la información que debe ser recobrada para ordenar correctamente la basura. En la solución inicial, RTTI (descrito en Apéndice B) es usado.

Éste no es un diseño trivial porque tiene una restricción añadida. Lo que lo hace interesante – es más como los problemas confusos que tienes probabilidad de encontrar en tu trabajo. La restricción adicional es que la basura llega a la planta de reciclaje de basura todo mezclado. El programa debe modelar la ordenación de esa basura. Ésta es donde RTTI entra: Tienes un montón de pedazos anónimos de basura, y el programa entiende exactamente qué tipo son.

```
//: RecycleA.java
// Recycling with RTTI
package c16.recyclea;
import java.util.*;
import java.io.*;

abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  abstract double value();
  double weight() { return weight; }
  // Sums the value of Trash in a bin:
  static void sumValue(Vector bin) {
    Enumeration e = bin.elements();
    double val = 0.0f;
    while(e.hasMoreElements()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)e.nextElement();
      // Polymorphism in action:
      val += t.weight() * t.value();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.weight());
    }
    System.out.println("Total value = " + val);
  }
}

class Aluminum extends Trash {
  static double val  = 1.67f;
  Aluminum(double wt) { super(wt); }
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

class Paper extends Trash {
  static double val = 0.10f;
  Paper(double wt) { super(wt); }
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

class Glass extends Trash {
  static double val = 0.23f;
  Glass(double wt) { super(wt); }
```

```
  double value() { return val; }
  static void value(double newval) {
    val = newval;
  }
}

public class RecycleA {
  public static void main(String[] args) {
    Vector bin = new Vector();
    // Fill up the Trash bin:
    for(int i = 0; i < 30; i++)
      switch((int)(Math.random() * 3)) {
        case 0 :
          bin.addElement(new
            Aluminum(Math.random() * 100));
          break;
        case 1 :
          bin.addElement(new
            Paper(Math.random() * 100));
          break;
        case 2 :
          bin.addElement(new
            Glass(Math.random() * 100));
      }
    Vector
      glassBin = new Vector(),
      paperBin = new Vector(),
      alBin = new Vector();
    Enumeration sorter = bin.elements();
    // Sort the Trash:
    while(sorter.hasMoreElements()) {
      Object t = sorter.nextElement();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.addElement(t);
      if(t instanceof Paper)
        paperBin.addElement(t);
      if(t instanceof Glass)
        glassBin.addElement(t);
    }
    Trash.sumValue(alBin);
    Trash.sumValue(paperBin);
    Trash.sumValue(glassBin);
    Trash.sumValue(bin);
  }
} ///:~
```

La primera cosa que notarás es la declaración **package**:

```
package c16.recyclea;
```

Esto quiere decir que en los listados de código fuentes disponibles para el libro,

este archivo será colocado en el subdirectorio **recyclea** que se bifurca del subdirectorio **c16** (para el Capítulo 16). La herramienta de desempaquetado en el Capítulo 18 se encarga de colocarlo dentro del subdirectorio correcto. La razón para hacer esto es que este capítulo reescribe este ejemplo particular un número de veces y metiendo cada versión en su propio **package** los nombres de clase no estarán en conflicto.

Varios objetos Vectoriales se le crean contener manipuladores de **Trash**. Por supuesto, **Vector** realmente contienen **Object** así es que contendrán cualquier cosa en absoluto. La razón por la que creen que **Trash** (o algo derivado de **Trash**) es único porque has tenido cuidado para no echar cualquier cosa excepto **Trash**. Si metieses algo "equivocado" en el Vector, no obtendrás cualquier error o advertencias de fases de compilación – te enterarás sólo por una excepción en el tiempo de ejecución.

Cuando las agarraderas **Trash** se agregan, pierden sus identidades específicas y se convierten en simplemente agarraderas **Object** (son dirigidas hacia arriba). Sin embargo, por el polimorfismo el comportamiento correcto todavía ocurre cuando los métodos dinámicamente comprometidos son llamados a través del **Enumeration sorter**, una vez el **Object** resultante se ha proyectado de regreso para **Trash sumValue**() también usa un **Enumeration** para realizar operaciones en cada objeto en el Vector.

Se ve tonto lanzar los tipos de **Trash** en una colección conteniendo manipuladores de tipo base, y luego vuelve y deja de lanzar. ¿Por qué no simplemente pone la basura en el receptáculo apropiado en primer lugar? (Ciertamente, éste es el enigma completo de reciclaje). En este programa sería fácil de reparar, pero algunas veces la estructura de un sistema y una flexibilidad pueden grandemente aprovecharse del downcasting.

El programa satisface los requisitos del diseño: trabaja. Esto podría estar bien con tal de que sea una solución instantánea. Sin embargo, un programa útil tiende a evolucionar con el paso del tiempo, así es que debes preguntar, *"¿Qué ocurre si la situación cambia?*" Por ejemplo, el cartón es ahora un artículo de comercio valioso del reciclable, así es cómo eso será integrado en el sistema (especialmente si el programa es grande y complicado). Ya que la anteriormente citada codificación de verificación de tipo en la declaración **switch** podría esparcirse a todo lo largo del programa, debes pasarte a encontrar todo ese código cada vez que un tipo nuevo se agrega, y si pierdes uno el compilador no te dará ayuda alguna apuntándolo como un error.

La clave para el uso indebido de *RTTI* aquí es que cada tipo es probado. Si andas buscando sólo un subconjunto de tipos porque ese subconjunto necesita tratamiento especial, eso es probablemente bueno. Pero si vas en busca de cada tipo dentro de una declaración **switch**, entonces pierdes probablemente un punto importante, y definitivamente hace tu código menos mantenible. En la siguiente sección que miraremos cómo este programa evolucionó sobre varias etapas para ponerse mucho más flexible. Esto debería probar un ejemplo valioso en diseño de programa.

Mejorando el diseño

Las soluciones en *Patrones de Diseño* son organizadas alrededor de la pregunta "*¿Qué cambiará a medida que este programa evoluciona?*" Éste es usualmente la pregunta más importante que puedes preguntar acerca de cualquier diseño. Si puedes construir tu sistema alrededor de la respuesta, los resultados serán de dos puntas: No sólo tu sistema permitirá mantenimiento fácil (y barato), sino que también podrías producir componentes que son reusables, a fin de que otros sistemas pueden ser construidos más baratos. Ésta es la promesa de programación orientada a objetos, pero no ocurre automáticamente; Requiere concepto y comprensión en su parte. En este pasaje veremos cómo puede ocurrir este proceso durante el refinamiento de un sistema.

La respuesta para la pregunta "*¿Qué cambiará?*" Para el sistema de reciclaje está uno en común: Más tipos serán añadidos al sistema. La meta del diseño, entonces, es hacer esta adición de tipos tan indoloras como posibles. En el programa de reciclaje, nos gustaría encapsular todos los lugares dónde la información específica de tipo es mencionado, así (si por ningún otra razón) cualquier cambios pueden ser localizados para esas encapsulaciones. Resulta que este proceso también limpia el resto de código considerablemente.

"hacer más objetos"

Esto trae a colación un principio general de diseño orientado a objetos que primero escuché hablado por Grady Booch: "*Si el diseño es demasiado complicado, haga más objetos*". Esto es simultáneamente contra intuitivo y ridículamente simple, y es la línea directiva más útil que he encontrado. (Usted podría comentar que "*hacer más objetos*" es a menudo equivalente a "*agregue otro nivel de indirección*"). En general, si usted encuentra un lugar con código desordenado, considere qué tipo de clase limpiaría eso. A menudo el efecto secundario de hacer la limpieza del código será un sistema que deba estructurar y que sea más flexible.

Considere primero el lugar donde los objetos **Trash** son creados, lo cual es una declaración **switch** dentro de **main**():

```
for(int i = 0; i < 30; i++)
      switch((int)(Math.random() * 3)) {
        case 0 :
          bin.addElement(new
            Aluminum(Math.random() * 100));
          break;
        case 1 :
          bin.addElement(new
            Paper(Math.random() * 100));
          break;
        case 2 :
          bin.addElement(new
            Glass(Math.random() * 100));
}
```

Esto es definitivamente desordenado, y también un lugar donde usted debe cambiar código cada vez que un tipo nuevo se agrega. Si los tipos nuevos se agregan comúnmente, una mejor solución es un sencillo método que toma toda la información necesaria y produce un manipulador a un objeto del tipo correcto, ya lanzado a un objeto de basura. En *Patrones de Diseño* está ampliamente referido a como un patrón creacional (del cual hay varios). El patrón específico que será aplicado aquí es una variante del Método de Fábrica. Aquí, el método de fábrica es un miembro estático de **Trash**, pero más comúnmente es un método que es sobrescrito en la clase derivada.

La idea del método de la fábrica es que usted le pasa a eso la información esencial que necesita saber para crear su objeto, luego esta puesto de regreso y espera el manipulador (ya lanzado para el tipo base) para salir de improviso como el valor de retorno. Desde entonces, usted trata con el objeto polimorficamente. Así, usted nunca necesita saber aun el tipo exacto de objeto que es creado. De hecho, el método de fábrica lo esconde de usted para impedir el uso indebido accidental. Si quiere usar el objeto sin polimorfismo, explícitamente debe usar a *RTTI* y señalarlo .

Pero hay un pequeño problema, especialmente cuando usa el acercamiento (no mostrado aquí) más complicado de hacer el método de fábrica en la clase base y sobrescribirlo en las clases derivadas. ¿Qué ocurre si la información requerida en la clase derivada requiere más o diferentes argumentos? “Creando más objetos” soluciona este problema. Para implementar el método de fábrica, la clase **Trash** llama un nuevo método **factory**. Para esconder los datos creacionales, hay una nueva clase llamada **Info** que contiene toda la información necesaria para el método de fábrica para crear el objeto correcto **Trash**. Aquí está una implementación simple de **Info**:

```
class Info {
  int type;
  // Must change this to add another type:
  static final int MAX_NUM = 4;
  double data;
  Info(int typeNum, double dat) {
    type = typeNum % MAX_NUM;
    data = dat;
  }
}
```

El único trabajo de un objeto **Info** es sujetar información para el método **factory**(). Ahora, si hay una situación en la cual **factory**() necesita más o diferente información para crear un tipo nuevo de objeto **Trash**, la interfaz **factory**() no necesita variarse. La clase **Info** puede variarse agregando datos nuevos y constructores nuevos, o en la moda más orientada a objetos típica de subclasificación.

El método **factory**() para este ejemplo simple se parece a éste:

```
static Trash factory(Info i) {
    switch(i.type) {
      default: // To quiet the compiler
```

```
      case 0:
        return new Aluminum(i.data);
      case 1:
        return new Paper(i.data);
      case 2:
        return new Glass(i.data);
      // Two lines here:
      case 3:
        return new Cardboard(i.data);
    }
}
```

Aquí, la determinación del tipo exacto de objeto es simple, pero usted puede imaginar un sistema más complicado en el cual **factory**() usa un algoritmo complicado. El caso es que está ahora escondido en un lugar, y usted sabe alcanzar este lugar cuando agrega tipos nuevos.

La creación de objetos nuevos es ahora mucho más simple en **main**():

```
for(int i = 0; i < 30; i++)
      bin.addElement(
        Trash.factory(
          new Info(
            (int)(Math.random() * Info.MAX_NUM),
                  Math.random() * 100)));
```

Un objeto **Info** es creado para pasar los datos en **factory**(), lo cual a su vez produce alguna clase de objeto **Trash** en el montón y devuelve el manipulador que ha añadido el **Vector bin**. Por supuesto, si cambia la cantidad y tipo de argumento, esta declaración todavía necesitará ser modificada, pero eso puede ser eliminado si la creación del objeto **Info** está automatizada. Por ejemplo, un Vector de argumentos puede estar aprobado en el constructor de una llamada del objeto **Info** (o directamente en un **factory**(), respecto a eso). Esto pide que los argumentos sean analizados gramaticalmente y comprobados en ejecución, pero provee la máxima flexibilidad.

Puede ver de este código lo que el problema del "*vector de cambio*" la fábrica es responsable de solucionar: Si usted le añade tipos nuevos al sistema (el cambio), el único código que debe estar modificado está dentro de la fábrica, así la fábrica aísla el efecto de ese cambio.

Un patrón para la creación del prototipado

Un problema con el diseño de arriba es que todavía requiere una posición central donde todos los tipos de los objetos deben ser conocidos: dentro del método **factory**(). Si los tipos nuevos están regularmente siendo añadidos al sistema, el método **factory**() debe variarse para cada tipo nuevo. Cuando usted descubre algo así como esto, es útil tratar de ir un paso más allá y mover toda la información acerca del tipo – incluyendo su creación – en la clase representando ese tipo. Así, la única cosa que usted necesita hacer para

añadirle un tipo nuevo al sistema debe heredar una clase única.

Para mover la información concerniendo creación de tipo en cada tipo específico de **Trash**, el patrón "prototipo" (del libro *Patrones de Diseño*) será usado. La idea general es que usted tiene una secuencia maestra de objetos, uno de cada tipo en el que usted está interesado en la confección. Los objetos en esta secuencia son usados sólo para hacer objetos nuevos, usar una operación que no está a diferencia del esquema **clone**() construido en la clase raíz de Java **Object**. En este caso, nombraremos el método de clonación **tClone**(). Cuando está listo para hacer un nuevo objeto, probablemente usted tiene alguna suerte de información que establece el tipo de objeto que quiere crear, luego usted se mueve a través de la secuencia maestra comparando su información con no importar qué información apropiada está en los objetos del prototipo en la secuencia maestra. Cuando usted encuentra uno que corresponde a sus necesidades, lo clona.

En este esquema no hay información de código duro para la creación. Cada objeto sabe cómo exponer información apropiada y cómo clonarse. Así, el método **factory**() no necesita variarse cuando un tipo nuevo es añadido al sistema.

Un acercamiento para el problema de prototipado es agregar un número de métodos para soportar la creación de objetos nuevos. Sin embargo, en Java 1.1 ya hay soporte para crear objetos nuevos si usted tiene un manipulador para el objeto **Class**. Con Java 1.1 la reflexión (introducido en el Capítulo 10) usted puede llamar a un constructor aun si usted tiene sólo un manipulador para el objeto **Class**. Ésta es la solución perfecta para el problema del prototipado.

La lista de prototipos será representada indirectamente por una lista de agarraderas para todos los objetos **Class** que usted quiere crear. Además, si el prototyping deja de operar, el método **factory**() asumirá que es porque un objeto particular **Class** no estaba en la lista, y tratará de cargarla. Cargando los prototipos dinámicamente como este, la clase **Trash** no necesita conocer qué tipos están funcionando, así es que no necesita modificaciones cuando usted agregue tipos nuevos. Esto le permite ser fácilmente reusado a lo largo del resto del capítulo.

```
//: Trash.java
// Base class for Trash recycling examples
package c16.trash;
import java.util.*;
import java.lang.reflect.*;

public abstract class Trash {
  private double weight;
  Trash(double wt) { weight = wt; }
  Trash() {}
  public abstract double value();
  public double weight() { return weight; }
  // Sums the value of Trash in a bin:
  public static void sumValue(Vector bin) {
    Enumeration e = bin.elements();
```

```
    double val = 0.0f;
    while(e.hasMoreElements()) {
      // One kind of RTTI:
      // A dynamically-checked cast
      Trash t = (Trash)e.nextElement();
      val += t.weight() * t.value();
      System.out.println(
        "weight of " +
        // Using RTTI to get type
        // information about the class:
        t.getClass().getName() +
        " = " + t.weight());
    }
    System.out.println("Total value = " + val);
  }
  // Remainder of class provides support for
  // prototyping:
  public static class PrototypeNotFoundException
      extends Exception {}
  public static class CannotCreateTrashException
      extends Exception {}
  private static Vector trashTypes =
    new Vector();
  public static Trash factory(Info info)
      throws PrototypeNotFoundException,
      CannotCreateTrashException {
    for(int i = 0; i < trashTypes.size(); i++) {
      // Somehow determine the new type
      // to create, and create one:
      Class tc =
        (Class)trashTypes.elementAt(i);
      if (tc.getName().indexOf(info.id) != -1) {
        try {
          // Get the dynamic constructor method
          // that takes a double argument:
          Constructor ctor =
            tc.getConstructor(
              new Class[] {double.class});
          // Call the constructor to create a
          // new object:
          return (Trash)ctor.newInstance(
            new Object[]{new Double(info.data)});
        } catch(Exception ex) {
          ex.printStackTrace();
          throw new CannotCreateTrashException();
        }
      }
    }
    // Class was not in the list. Try to load it,
    // but it must be in your class path!
    try {
      System.out.println("Loading " + info.id);
      trashTypes.addElement(
```

```
        Class.forName(info.id));
    } catch(Exception e) {
      e.printStackTrace();
      throw new PrototypeNotFoundException();
    }
    // Loaded successfully. Recursive call
    // should work this time:
    return factory(info);
  }
  public static class Info {
    public String id;
    public double data;
    public Info(String name, double data) {
      id = name;
      this.data = data;
    }
  }
} ///:~
```

La clase básica **Trash** y **sumValue**() quedan como antes. El resto de la clase soporta el patrón del prototipado. Usted primero ve dos clases internas (el cual es hecha estática, así es que son clases internas sólo para propósitos de organización de código) describiendo excepciones que puedan ocurrir. Esto es seguido por un **Vector trashTypes**, lo cual se usa para contener los manipuladores **Class**.

En **Trash.factory**(), el **String** dentro del objeto **Info id** (una versión diferente de la clase **Info** que ese del anterior argumento) contiene el nombre de tipo del **Trash** para ser creado; Este **String** es comparado con los nombres **Class** en la lista. Si hay un encuentro, entonces ese es el objeto a crear. Por supuesto, hay muchas formas para determinar qué objeto quiere hacer. Este es usado así que la información leída de un archivo pueden ser convertidos en objetos.

Una vez que has descubierto el tipo **Trash** a crear, entonces los métodos de reflexión entran en juego. El método **getConstructor**() toma un argumento que es un montón de manipuladores **Class**. Este arreglo representa los argumentos, en su orden correcta, para el constructor que andas buscando. Aquí, el arreglo es dinámicamente creado usando la sintaxis de creación de arreglo Java 1.1:

```
new Class[] {double.class}
```

Este código da por supuesto que cada tipo **Trash** tiene a un constructor que toma a un doble (y notese que **double.class** son distintos de **Double.class**). Es también posible, para una solución más para flexible, llamar a **getConstructors**(), el cual devuelve un arreglo de los constructores posibles.

Lo que regresa de **getConstructor**() es un manipulador para un objeto Constructor (parte de **java.lang.reflect**). Llamas al constructor dinámicamente con el método **newInstance**(), lo cual toma a un montón de **Object** conteniendo argumentos actuales. Este arreglo es otra vez creado

usando la sintaxis Java 1.1:

```
new Object[]{new Double(info.data)}
```

En este caso, sin embargo, el **double** debe ser colocado dentro de una clase envoltorio a fin de que pueda ser de este arreglo de objetos. El proceso de llamar a **newInstance**() extrae al **double**, pero puedes ver que está un poco confuso – un argumento podría ser un **double** o un **Double**, pero cuando haces la llamada siempre debes pasar en un **Double**. Afortunadamente, este asunto existe sólo para los tipos primitivos.

Una vez que entiendes cómo hacer eso, el proceso de crear un objeto nuevo dado solo un manipulador **Class** es notablemente simple. La reflexión también te permite llamar métodos en esta misma moda dinámica.

Por supuesto, el manipulador correcto **Class** no podría estar en la lista de **trashTypes**. En este caso, el retorno en el bucle interno nunca es ejecutado y te darás de baja al final. Aquí, el programa trata de rectificar la situación cargando el objeto **Class** dinámicamente y añadiéndola a la lista de **trashTypes**. Si todavía no puede ser encontrado algo que está realmente mal, pero si la carga tiene éxito entonces el método de fábrica es llamado recursivamente para hacer un intento de nuevo.

Como verás, la belleza de este diseño es que este código no necesita ser cambiado, a pesar de las situaciones diferentes en las que será usada (dado que todas las subclases **Trash** contengan un constructor que toma un argumento único **double**).

Subclases de Trash

Para caber dentro del esquema del prototipado, lo único que es requerido de cada subclase nueva de **Trash** es que contiene un constructor que toma un argumento **double**. La reflexión Java 1.1 maniobra todo lo demás.

Aquí están los tipos diferentes de **Trash**, cada uno en su archivo pero en parte del paquete **Trash** (otra vez, para facilitar aprovechamiento dentro del capítulo):

```
//: Aluminum.java
// The Aluminum class with prototyping
package c16.trash;

public class Aluminum extends Trash {
  private static double val = 1.67f;
  public Aluminum(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
//: Paper.java
// The Paper class with prototyping
```

```
package c16.trash;

public class Paper extends Trash {
  private static double val = 0.10f;
  public Paper(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
//: Glass.java
// The Glass class with prototyping
package c16.trash;

public class Glass extends Trash {
  private static double val = 0.23f;
  public Glass(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
```

Y aquí está un tipo nuevo de **Trash**:

```
//: Cardboard.java
// The Cardboard class with prototyping
package c16.trash;

public class Cardboard extends Trash {
  private static double val = 0.23f;
  public Cardboard(double wt) { super(wt); }
  public double value() { return val; }
  public static void value(double newVal) {
    val = newVal;
  }
} ///:~
```

Puedes ver eso, otro que el constructor, allí es nada especial acerca de cualquiera de estas clases.

Analizando Trash desde un archivo externo

La información acerca de objetos **Trash** será leída de un archivo exterior. El archivo tiene toda la información necesaria acerca de cada pedazo de basura en una línea única en la forma **Trash:weight**, como:

```
c16.Trash.Glass:54
c16.Trash.Paper:22
c16.Trash.Paper:11
```

```
c16.Trash.Glass:17
c16.Trash.Aluminum:89
c16.Trash.Paper:88
c16.Trash.Aluminum:76
c16.Trash.Cardboard:96
c16.Trash.Aluminum:25
c16.Trash.Aluminum:34
c16.Trash.Glass:11
c16.Trash.Glass:68
c16.Trash.Glass:43
c16.Trash.Aluminum:27
c16.Trash.Cardboard:44
c16.Trash.Aluminum:18
c16.Trash.Paper:91
c16.Trash.Glass:63
c16.Trash.Glass:50
c16.Trash.Glass:80
c16.Trash.Aluminum:81
c16.Trash.Cardboard:12
c16.Trash.Glass:12
c16.Trash.Glass:54
c16.Trash.Aluminum:36
c16.Trash.Aluminum:93
c16.Trash.Glass:93
c16.Trash.Paper:80
c16.Trash.Glass:36
c16.Trash.Glass:12
c16.Trash.Glass:60
c16.Trash.Paper:66
c16.Trash.Aluminum:36
c16.Trash.Cardboard:22
```

Note que la ruta de la clase debe ser incluida cuando otorgas los nombres de la clase, de otra manera la clase no será encontrada.

Para analizar gramaticalmente esto, la línea es leída y el método **String indexOf**() produce el índice de **':'**. Esto es primero usado con el método **String substring**() para extraer el nombre del tipo de basura, y obtener el peso que es convertido en un doble con el método estático **Double.valueOf**(). El método **trim**() quita espacio en blanco a ambos extremos de un **string**.

El analizador sintáctico **Trash** es colocado en un archivo separado ya que será reusado a lo largo de este capítulo:

```
//: ParseTrash.java
// Open a file and parse its contents into
// Trash objects, placing each into a Vector
package c16.trash;
import java.util.*;
import java.io.*;

public class ParseTrash {
  public static void
```

```
  fillBin(String filename, Fillable bin) {
    try {
      BufferedReader data =
        new BufferedReader(
          new FileReader(filename));
      String buf;
      while((buf = data.readLine())!= null) {
        String type = buf.substring(0,
          buf.indexOf(':')).trim();
        double weight = Double.valueOf(
          buf.substring(buf.indexOf(':') + 1)
          .trim()).doubleValue();
        bin.addTrash(
          Trash.factory(
            new Trash.Info(type, weight)));
      }
      data.close();
    } catch(IOException e) {
      e.printStackTrace();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  // Special case to handle Vector:
  public static void
  fillBin(String filename, Vector bin) {
    fillBin(filename, new FillableVector(bin));
  }
} ///:~
```

En **RecycleA.java**, un Vector fue usado para contener los objetos **Trash**. Sin embargo, otros tipos de colecciones pueden ser usados igualmente. Para tener en cuenta esto, la primera versión de **fillBin**() le lleva un manipulador a un **Fillable** , lo cual es simplemente una interfaz que soporta un método llamado **addTrash**():

```
//: Fillable.java
// Any object that can be filled with Trash
package c16.trash;

public interface Fillable {
  void addTrash(Trash t);
} ///:~
```

Cualquier cosa que soporta esta interfaz puede ser usada con **fillBin**. Por supuesto, **Vector** no implementa **Fillable**, así es que no trabajará. Ya que Vector es usado en la mayor parte de los ejemplos, tiene sentido sumar un segundo método sobrecargado **fillBin**() que toma un **Vector**. El **Vector** puede ser utilizado como un objeto **Fillable** usando una clase adaptador:

```
//: FillableVector.java
// Adapter that makes a Vector Fillable
```

```
package c16.trash;
import java.util.*;

public class FillableVector implements Fillable {
  private Vector v;
  public FillableVector(Vector vv) { v = vv; }
  public void addTrash(Trash t) {
    v.addElement(t);
  }
} ///:~
```

Puedes ver que la única actividad de esta clase es asociar el método **addTrash**() de **Fillable** al **addElement**() de **Vector**. Con esta clase en mano, el método sobrecargado **fillBin**() puede ser usado con un Vector en **ParseTrash.java**:

Este acercamiento surte efecto por cualquier clase colección que es usada frecuentemente. Alternativamente, la clase colección puede proveer su propio adaptador que implementa **Fillable**. (Verás esto más tarde, en **DynaTrash.java**.)

Reciclaje con prototipado

Ahora puedes ver la versión revisada de **RecycleA.java** usando la técnica del prototipado:

```
//: RecycleAP.java
// Recycling with RTTI and Prototypes
package c16.recycleap;
import c16.trash.*;
import java.util.*;

public class RecycleAP {
  public static void main(String[] args) {
    Vector bin = new Vector();
    // Fill up the Trash bin:
    ParseTrash.fillBin("Trash.dat", bin);
    Vector
      glassBin = new Vector(),
      paperBin = new Vector(),
      alBin = new Vector();
    Enumeration sorter = bin.elements();
    // Sort the Trash:
    while(sorter.hasMoreElements()) {
      Object t = sorter.nextElement();
      // RTTI to show class membership:
      if(t instanceof Aluminum)
        alBin.addElement(t);
      if(t instanceof Paper)
        paperBin.addElement(t);
      if(t instanceof Glass)
```

```
        glassBin.addElement(t);
    }
    Trash.sumValue(alBin);
    Trash.sumValue(paperBin);
    Trash.sumValue(glassBin);
    Trash.sumValue(bin);
  }
} ///:~
```

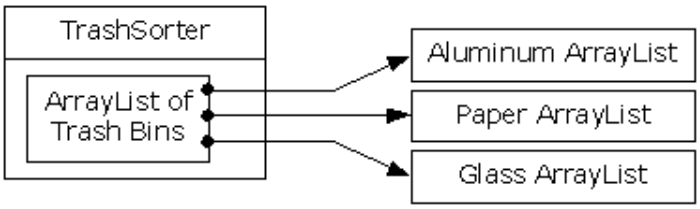
Todos los objetos **Trash**, así como también el **ParseTrash** y clases de soporte, son ahora en parte del paquete **c16.trash** así es que son simplemente importados.

El proceso de abrir el archivo de datos conteniendo descripciones **Trash** y el análisis sintáctico de ese archivo ha estado envuelto en el método estático **ParseTrash.fillBin**, así que ahora no es una parte de nuestro diseño **focus**(). Verás que a lo largo del resto de capítulo, no importa qué clases nuevas estén añadidas, **ParseTrash.fillBin**() continuará trabajando sin cambio, el cual indica un buen diseño.

En términos de creación del objeto, este diseño hace por cierto localiza con severidad los cambios que necesitas hacer para añadirle un tipo nuevo al sistema. Sin embargo, hay un problema significativo en el uso de RTTI que aparece claramente aquí. ¡El programa parece correr muy bien, y aún así nunca detecta cualquier cartón, si bien hay cartón en la lista! Esto ocurre por el uso de RTTI, lo cual busca sólo los tipos que le dicen que busque. La pista en la que RTTI está siendo despilfarrado está que *cada tipo en el sistema* está siendo probado, en vez de un tipo único o subconjunto de tipos. Como verás más adelante, hay formas para usar polimorfismo en lugar de cuando experimentas para cada tipo. Pero si usas a muy a menudo RTTI, y le añades un tipo nuevo a tu sistema, fácilmente puedes olvidar hacer los cambios necesarios en tu programa y produciría un problema difícil de encontrar. Así que cuesta tratar de eliminar a RTTI en este caso, no sólo por las razones estéticas – produce más código mantenible.

Abstrayendo el uso

Con la creación aparte, es hora de abordar el resto del diseño : donde las clases son usadas. Desde que es el hecho de ordenación en depósitos que es en particular feo y arriesgado, ¿por qué no tomar ese proceso y ocultarlo dentro de una clase? Éste es el principio de “Si debes hacer algo feo, al menos localiza la fealdad dentro de una clase.” Se parece a esto:



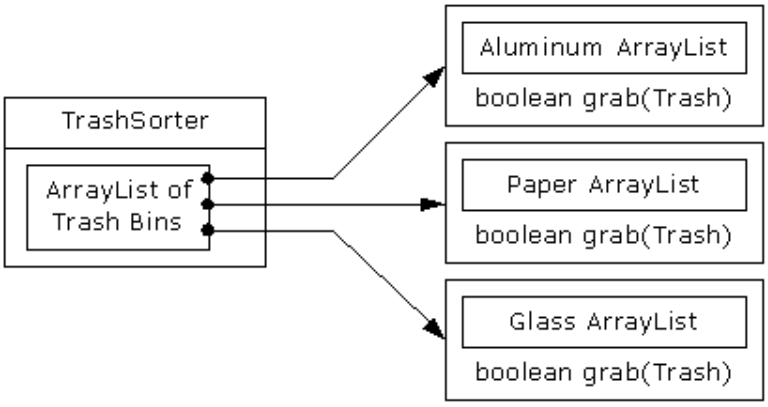
(Este diagrama usa al **ArrayList** más moderno, pero también describe la implementación Vector.) La inicialización del objeto **TrashSorter** ahora debe variarse cada vez que un tipo nuevo de **Trash** es añadido al modelo. Podrías suponerte que la clase **TrashSorter** podría verse algo así como éste:

```
class TrashSorter extends Vector {
  void sort(Trash t) { /* ... */ }
}
```

Es decir, **TrashSorter** es un Vector de manipuladores para **Vectors** de manipuladores **Trash**, y con **addElement**() que puedes instalar otro, de esta manera:

```
TrashSorter ts = new TrashSorter();
ts.addElement(new Vector());
```

Ahora, sin embargo, **sort**() se convierte en un problema. ¿Cómo distribuye el método estáticamente codificado con el hecho que un tipo nuevo se ha agregado? Para solucionar esto, el tipo de información debe ser removida de **sort**() a fin de que todo lo que necesita hacer es llamar un método genérico que se encarga de los detalles del tipo. Esto, claro está, es describir un método dinámicamente comprometido. Así **sort**() simplemente activará a través de la secuencia y llamará un método dinámicamente comprometido para cada Vector. Ya que el trabajo de este método es agarrar los pedazos de basura en los que está interesado, esto es llamado **grab**(**Trash**). La estructura ahora se ve como:



(Este diagrama usa al **ArrayList** más moderno, pero también describe la implementación Vector.) **TrashSorter** necesita llamar cada método **grab**() y obtener un resultado diferente a merced de la clase de **Trash** que el Vector actual contiene. Es decir, cada Vector debe darse cuenta del tipo que contiene. El acercamiento clásico para este problema es crear una clase base de depósito **Trash** y heredar una clase derivada nueva para cada tipo diferente que quieres contener. Si Java tuviera un mecanismo de tipo parametrizado que probablemente sería el acercamiento más franco. Excepto en vez de la codificación de mano todas las clases de las cuales un mecanismo debería

construir para nosotros, una mejor observación puede producir un mejor acercamiento.

Un principio básico del diseño OOP es “Usa los miembros de datos para la variación en estado, y usa el polimorfismo para la variación en el comportamiento.” Tu primer pensamiento podría ser que el método **grab**() ciertamente se comporta diferentemente para un **Vector** que contiene a **Paper** para uno que contiene a **Glass**. Pero qué hace esto estrictamente dependiente en el tipo, y nada más. Esto podría ser interpretado como un estado diferente, y desde que Java tiene una clase para representar tipo (**Class**) ésta puede ser usada para determinar al tipo de **Trash** que un **Tbin** particular contendrá.

El constructor para este **Tbin** requiere que le pases el **Class** de tu elección. Esto dice al **Vector** qué tipo está supuesto a contener. Luego el método **grab**() usa a **Class BinType** y RTTI para ver si el objeto **Trash** que le has dado corresponde al tipo que está supuesto a captar.

Aquí está el programa entero. Los números comentados (ejm (*1 *)) señala secciones que estarán descritas siguiendo el código.

```
//: RecycleB.java
// Adding more objects to the recycling problem
package c16.recycleb;
import c16.trash.*;
import java.util.*;

// A vector that admits only the right type:
class Tbin extends Vector {
  Class binType;
  Tbin(Class binType) {
    this.binType = binType;
  }
  boolean grab(Trash t) {
    // Comparing class types:
    if(t.getClass().equals(binType)) {
      addElement(t);
      return true; // Object grabbed
    }
    return false; // Object not grabbed
  }
}

class TbinList extends Vector { //(*1*)
  boolean sort(Trash t) {
    Enumeration e = elements();
    while(e.hasMoreElements()) {
      Tbin bin = (Tbin)e.nextElement();
      if(bin.grab(t)) return true;
    }
    return false; // bin not found for t
  }
  void sortBin(Tbin bin) { // (*2*)
    Enumeration e = bin.elements();
```

```
    while(e.hasMoreElements())
      if(!sort((Trash)e.nextElement()))
        System.out.println("Bin not found");
  }
}

public class RecycleB {
  static Tbin bin = new Tbin(Trash.class);
  public static void main(String[] args) {
    // Fill up the Trash bin:
    ParseTrash.fillBin("Trash.dat", bin);

    TbinList trashBins = new TbinList();
    trashBins.addElement(
      new Tbin(Aluminum.class));
    trashBins.addElement(
      new Tbin(Paper.class));
    trashBins.addElement(
      new Tbin(Glass.class));
    // add one line here: (*3*)
    trashBins.addElement(
      new Tbin(Cardboard.class));

    trashBins.sortBin(bin); // (*4*)

    Enumeration e = trashBins.elements();
    while(e.hasMoreElements()) {
      Tbin b = (Tbin)e.nextElement();
      Trash.sumValue(b);
    }
    Trash.sumValue(bin);
  }
} ///:~
```

1. **TbinList** contiene un conjunto de manipuladores **Tbin**, a fin de que **sort**() pueda iterar a través de los **Tbins** cuando anda buscando un encuentro para el objeto **Trash** que le has pasado.
2. **SortBin**() te permite pasar a un **Tbin** entero adentro, y se mueve a través del **Tbin**, escoge cada pedazo de **Trash**, y lo ordena en el **Tbin** específico apropiado. Note la genericidad de este código: No cambia en absoluto si los tipos nuevos se agregan. Si la masa de tu código no necesita cambiar cuando un tipo nuevo se agrega (o algún otro cambio ocurre) entonces tienes un sistema fácilmente extensible.
3. Ahora puedes ver qué tan fácil es agregar un tipo nuevo. Pocas líneas deben variarse para soportar la adición. Si esto es realmente importante, puedes dejar aun más limpio por promover manipular el diseño.
4. Una llamada de método causa que el contenido de depósito sea ordenado en los respectivos depósitos específicamente tipeados.

Múltiples despachos

El diseño de arriba es ciertamente satisfactorio. Añadiéndole tipos nuevos al sistema consta de añadir o modificar clases distintas sin causar cambios en el código para ser propagado a lo largo del sistema. Además, RTTI no es "despilfarrado" como estaba en **RecycleA.java**. Sin embargo, cabe pasarse un paso más allá y tomar un punto de vista del purista acerca de RTTI y el punto de vista que debería ser eliminado enteramente de la operación de ordenar la basura en depósitos.

Para lograr esto, primero debes tomar la perspectiva de que todas las actividades dependientes en tipo – como detectar el tipo de la misma clase de basura y meterlo en el depósito correcto – deberían estar controladas a través del polimorfismo y ligamento dinámico.

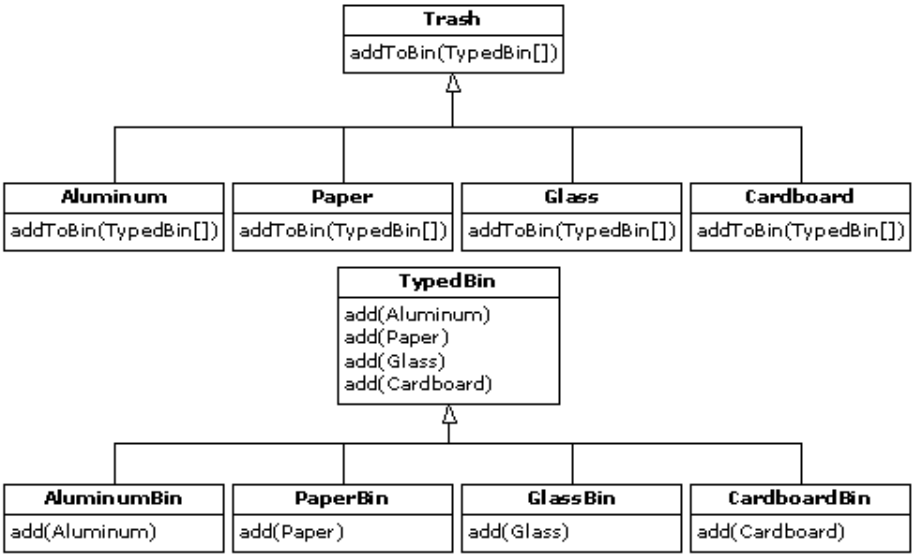
Los ejemplos anteriores primero ordenados por el tipo, luego actuado en las secuencias de elementos que fueron todo de un tipo particular. Pero cada vez que te encuentras escogiendo tipos particulares, detente y piensa. La idea integral de polimorfismo (las llamadas de método dinámicamente comprometidas) es maniobrar información específica en tipo para usted. ¿Entonces por qué estás yendo en busca de tipos?

La respuesta es algo acerca de lo que probablemente no piensas: Java realiza solamente despacho único. Es decir, si realizas una operación en más de un objeto cuyo tipo es desconocido, Java invocará el mecanismo dinámico de ligamento en sólo uno de esos tipos. Esto no soluciona el problema, así es que terminas detectando algunos tipos manualmente y produces eficazmente tu comportamiento dinámico de ligamento.

La solución es el llamado despacho múltiple, lo cual significa establecer una configuración algo semejante a una llamada única de método que produce más que una llamada dinámica de método y así determinan más de un tipo durante el proceso. Para obtener este efecto, necesitas trabajar con más de una jerarquía de tipo: Necesitarás una jerarquía de tipo para cada despacho. El siguiente ejemplo trabaja con dos jerarquías: La familia existente **Trash** y una jerarquía de los tipos de depósitos de basura de los que la basura será colocada adentro. Esta segunda jerarquía no está todo el tiempo obvio y en este caso es necesario ser creada para producir despacho múltiple (en este caso habrá sólo dos despachos, lo cual es referido como *doble despachamiento*).

Implementando el despacho doble

Recuerde que el polimorfismo puede ocurrir sólo por llamadas de método, así si quieres que ocurra el despacho doble, debe haber dos llamadas de método: Uno usado para determinar el tipo dentro de cada jerarquía. En la jerarquía **Trash** estará un nuevo método llamado **addToBin**(), lo cual toma un argumento de un arreglo de **TypedBin**. Usa este arreglo para dar un paso completamente y tratar de agregarse a sí mismo para el depósito correcto, y es donde verás el despacho doble.



La jerarquía nueva es **TypedBin**, y contiene su propio método llamado **add**() que es también usado polimorficamente . Pero aquí hay una tendencia adicional: **add**() es sobrecargado para tomar argumentos de tipos diferentes de basura. Así una parte esencial del esquema de despacho doble también involucra sobrecarga.

Rediseñar el programa produce un dilema: Es ahora necesario que la clase base **Trash** contenga un método **addToBin**(). Un método es copiar todo el código y cambiar la clase base. Otro acercamiento, el cual puedes tomar cuando no tienes control del código fuente, es poner el método **addToBin**() en una interfaz, dejar solo a **Trash**, y heredar nuevos tipos específicos de **Aluminum**, **Paper**, **Glass**, y **Cardboard**. Éste es el método que será tomado aquí.

La mayor parte de las clases en este diseño deben ser públicas, así es que son colocadas en sus archivos. Aquí está la interfaz:

```
//: TypedBinMember.java
// An interface for adding the double dispatching
// method to the trash hierarchy without
// modifying the original hierarchy.
package c16.doubledispatch;

interface TypedBinMember {
  // The new method:
  boolean addToBin(TypedBin[] tb);
} ///:~
```

En cada subtipo particular de **Aluminum**, **Paper**, **Glass**, y **Cardboard**, el método **addToBin**() en la interfaz **TypedBinMember** son implementados, pero tal parece ser que el código es exactamente igual en cada caso:

```
//: DDAluminum.java
// Aluminum for double dispatching
```

```
package c16.doubledispatch;
import c16.trash.*;

public class DDAluminum extends Aluminum
    implements TypedBinMember {
  public DDAluminum(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
//: DDPaper.java
// Paper for double dispatching
package c16.doubledispatch;
import c16.trash.*;

public class DDPaper extends Paper
    implements TypedBinMember {
  public DDPaper(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
//: DDGlass.java
// Glass for double dispatching
package c16.doubledispatch;
import c16.trash.*;

public class DDGlass extends Glass
    implements TypedBinMember {
  public DDGlass(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
//: DDCardboard.java
// Cardboard for double dispatching
package c16.doubledispatch;
import c16.trash.*;

public class DDCardboard extends Cardboard
    implements TypedBinMember {
  public DDCardboard(double wt) { super(wt); }
  public boolean addToBin(TypedBin[] tb) {
    for(int i = 0; i < tb.length; i++)
```

```
      if(tb[i].add(this))
        return true;
    return false;
  }
} ///:~
```

El código en cada **addToBin**() llama a **add**() para cada objeto **TypedBin** en el arreglo. Pero nótese el argumento: **this**. El tipo de **this** es diferente para cada subclase de **Trash**, así es que el código es diferente. (Aunque este código se beneficiará si un mecanismo de tipo parameterizado es para siempre añadido a Java.) De tal manera este es la primera parte del despacho doble, porque una vez que están dentro de este método que conoces están **Aluminum**, o **Paper**, etc. Durante la llamada para **add**(), esta información es pasada por el tipo de **this**. El compilador resuelve la llamada para la versión sobrecargada correcta de **add**(). Pero ya que **tb[i]** produce un manipulador para el tipo base **TypedBin**, esta llamada terminará llamando un método diferente a merced del tipo de **TypedBin** que está actualmente seleccionado. Este es el segundo despacho.

Aquí está la clase base para **TypedBin**:

```
//: TypedBin.java
// Vector that knows how to grab the right type
package c16.doubledispatch;
import c16.trash.*;
import java.util.*;

public abstract class TypedBin {
  Vector v = new Vector();
  protected boolean addIt(Trash t) {
    v.addElement(t);
    return true;
  }
  public Enumeration elements() {
    return v.elements();
  }
  public boolean add(DDAluminum a) {
    return false;
  }
  public boolean add(DDPaper a) {
    return false;
  }
  public boolean add(DDGlass a) {
    return false;
  }
  public boolean add(DDCardboard a) {
    return false;
  }
} ///:~
```

Puedes ver que los métodos sobrecargados **add**() todos retornan **false**. Si el método no es sobrecargado en una clase derivada, continuará retornando

false, y la persona que llama (**addToBin**(), en este caso) asumirá que el objeto actual **Trash** no se ha agregado exitosamente a una colección, y continúa yendo en busca de la colección correcta.

En cada una de las subclases de **TypedBin**, sólo un método sobrecargado es sobrescrito, según el tipo de depósito que está siendo creado. Por ejemplo, **CardboardBin** sobrescribe a **add**(**DDCardboard**). El método sobrescrito le añade el objeto de basura a su colección y retorna **true**, mientras todo el resto de los métodos **add**() en **CardboardBin** continúan retornando **false**, desde que no han sido sobrescritos. Éste es otro caso en el cual un mecanismo de tipo parametrizado en Java permitiría la generación automática de código. (Con plantillas C++, no tendrías que explícitamente escribir las subclases o colocar el método **addToBin**() en **Trash**.)

Ya que para este ejemplo los tipos de basura han sido hechos a la medida y colocados en un directorio diferente, necesitarás que un fichero de datos diferente de basura para hacerlo trabajar. Aquí hay un posible **DDTrash.dat**:

```
c16.DoubleDispatch.DDGlass:54
c16.DoubleDispatch.DDPaper:22
c16.DoubleDispatch.DDPaper:11
c16.DoubleDispatch.DDGlass:17
c16.DoubleDispatch.DDAluminum:89
c16.DoubleDispatch.DDPaper:88
c16.DoubleDispatch.DDAluminum:76
c16.DoubleDispatch.DDCardboard:96
c16.DoubleDispatch.DDAluminum:25
c16.DoubleDispatch.DDAluminum:34
c16.DoubleDispatch.DDGlass:11
c16.DoubleDispatch.DDGlass:68
c16.DoubleDispatch.DDGlass:43
c16.DoubleDispatch.DDAluminum:27
c16.DoubleDispatch.DDCardboard:44
c16.DoubleDispatch.DDAluminum:18
c16.DoubleDispatch.DDPaper:91
c16.DoubleDispatch.DDGlass:63
c16.DoubleDispatch.DDGlass:50
c16.DoubleDispatch.DDGlass:80
c16.DoubleDispatch.DDAluminum:81
c16.DoubleDispatch.DDCardboard:12
c16.DoubleDispatch.DDGlass:12
c16.DoubleDispatch.DDGlass:54
c16.DoubleDispatch.DDAluminum:36
c16.DoubleDispatch.DDAluminum:93
c16.DoubleDispatch.DDGlass:93
c16.DoubleDispatch.DDPaper:80
c16.DoubleDispatch.DDGlass:36
c16.DoubleDispatch.DDGlass:12
c16.DoubleDispatch.DDGlass:60
c16.DoubleDispatch.DDPaper:66
c16.DoubleDispatch.DDAluminum:36
c16.DoubleDispatch.DDCardboard:22
```

Aquí está el resto de programa:

```
//: DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
package c16.doubledispatch;
import c16.trash.*;
import java.util.*;

class AluminumBin extends TypedBin {
  public boolean add(DDAluminum a) {
    return addIt(a);
  }
}

class PaperBin extends TypedBin {
  public boolean add(DDPaper a) {
    return addIt(a);
  }
}

class GlassBin extends TypedBin {
  public boolean add(DDGlass a) {
    return addIt(a);
  }
}

class CardboardBin extends TypedBin {
  public boolean add(DDCardboard a) {
    return addIt(a);
  }
}

class TrashBinSet {
  private TypedBin[] binSet = {
    new AluminumBin(),
    new PaperBin(),
    new GlassBin(),
    new CardboardBin()
  };
  public void sortIntoBins(Vector bin) {
    Enumeration e = bin.elements();
    while(e.hasMoreElements()) {
      TypedBinMember t =
        (TypedBinMember)e.nextElement();
      if(!t.addToBin(binSet))
        System.err.println("Couldn't add " + t);
    }
  }
  public TypedBin[] binSet() { return binSet; }
}

public class DoubleDispatch {
```

```
  public static void main(String[] args) {
    Vector bin = new Vector();
    TrashBinSet bins = new TrashBinSet();
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("DDTrash.dat", bin);
    // Sort from the master bin into the
    // individually-typed bins:
    bins.sortIntoBins(bin);
    TypedBin[] tb = bins.binSet();
    // Perform sumValue for each bin...
    for(int i = 0; i < tb.length; i++)
      Trash.sumValue(tb[i].v);
    // ... and for the master bin
    Trash.sumValue(bin);
  }
} ///:~
```

TrashBinSet encapsula todos los tipos diferentes de **TypedBins**, junto con el método **sortIntoBins**(), el cual es donde todo el despacho doble toma lugar. Puedes ver que una vez que la estructura es establecida, la ordenación en varios **TypedBins** es notablemente fácil. Además, la eficiencia de dos llamadas dinámicas de método es probablemente mejor que de cualquier otra manera pudieras ordenar.

Note la facilidad de uso de este sistema en **main**(), así como también la independencia completa de cualquier información específica de tipo dentro de **main**(). Todos los demás métodos que hablan sólo para la interfaz de clase base **Trash** serán igualmente invulnerables para los cambios en tipos **Trash**.

Los cambios necesarios para agregar un tipo nuevo están relativamente aislados: Heredas el tipo nuevo de **Trash** con su método **addToBin**(), luego recibes una herencia de un **TypedBin** nuevo (éste es realmente una copia equitativa y edición simple), y finalmente agregas un tipo nuevo en la inicialización del agregado para **TrashBinSet**.

El patrón visitante

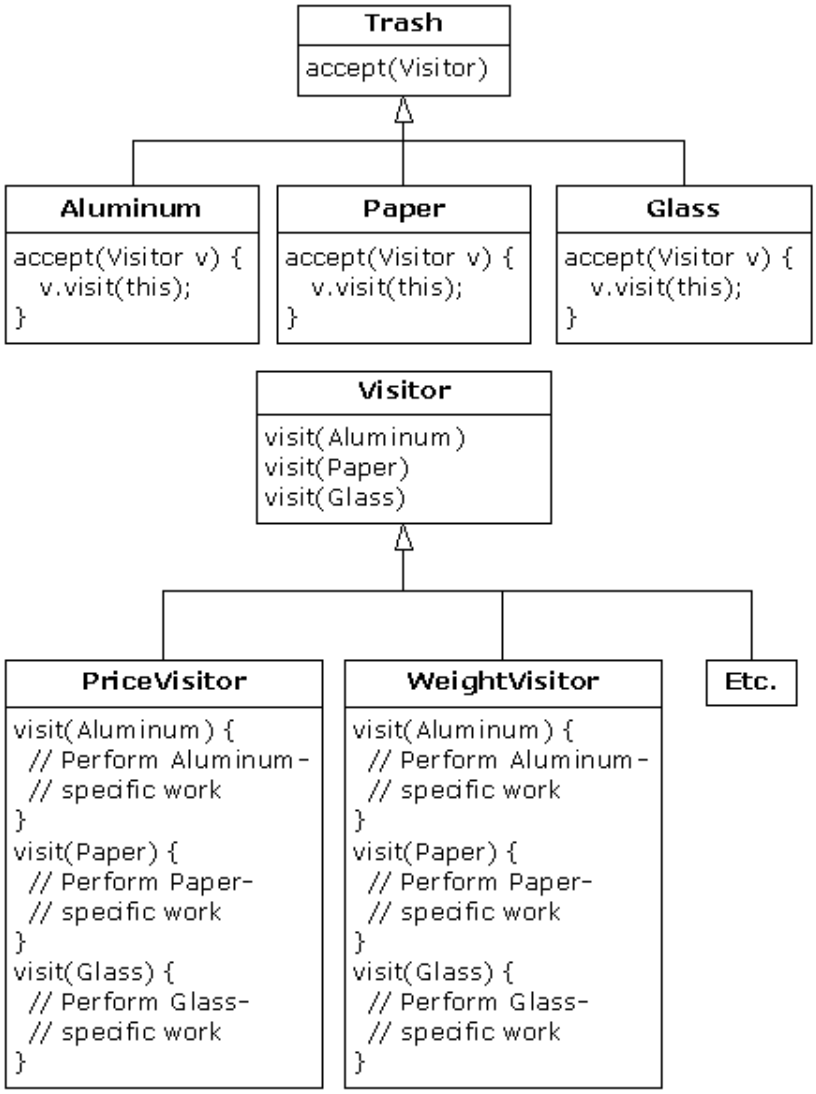
Ahora considera aplicarle un patrón de diseño con una meta completamente diferente al problema de ordenación de basura.

Para este patrón, ya no estamos preocupados con optimizar la adición de tipos nuevos de **Trash** para el sistema. Ciertamente, este patrón hace agregar un tipo nuevo de **Trash** más complicado. La suposición es que tienes una jerarquía primaria de clase que es fija; Quizá es de otro vendedor y no puedes hacer cambios a esa jerarquía. Sin embargo, te gustaría añadirle los métodos polimorfos nuevos a esa jerarquía, lo cual quiere decir que normalmente tendrías que añadirle algo a la interfaz de clase base. Así es que el dilema es que necesitas añadirle los métodos a la clase base, pero no puedes tocar la clase base. ¿Cómo omites esto?

El patrón de diseño que soluciona esta clase de problema se llama *visitante*, y

se fundamenta en el esquema de despacho doble enseñado en la última sección (el definitivo en el libro de *Patrones de Diseño*).

El patrón visitante te permite extender la interfaz del tipo primario creando una jerarquía separada de clase de tipo **Visitor** para virtualizar las operaciones realizadas en el tipo primario. Los objetos del tipo primario simplemente aceptan al visitante, luego llaman el método dinámicamente obligado del visitante. Se parece a esto:



Ahora, si **v** es un manipulador **Visitable** para un objeto **Aluminum**, el código:

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

Causa dos llamadas polimorfas de método: primero selecciona la versión de **Aluminum** de **accept**(), y el segundo dentro de **accept**() cuando la versión específica de **visit**() es llamada dinámicamente usando el manipulador v de la clase base **Visitor**.

Esta configuración quiere decir que la funcionabilidad nueva puede ser añadida al sistema en forma de las subclases nuevas de **Visitor**. La jerarquía **Trash** no

necesita ser tocada. Éste es el beneficio de primera del patrón visitante: Le puedes añadir la funcionabilidad polimórfica nueva a una jerarquía de clase sin tocar esa jerarquía de métodos (una vez el **accept**() haya sido instalado). Note que el beneficio es de ayuda aquí pero no es exactamente lo que emprendimos a lograr, así es que a primera vista podrías decidir que ésta no es la solución deseada.

Pero mira una cosa que ha sido lograda: La solución visitante evita ordenación de la secuencia maestra **Trash** en las secuencias individuales tipeadas. Así, puedes dejar todo en la secuencia maestra sola y simplemente atraviesa esa secuencia usando la visita apropiada para lograr la meta. Aunque este comportamiento parece ser un efecto secundario de visitante, eso nos da lo que queremos (evitando a *RTTI*).

El despacho doble en el patrón del visitante se encarga de determinar ambos el tipo de **Trash** y el tipo de **Visitor**. En el siguiente ejemplo, hay dos implementaciones de **Visitor**: **PriceVisitor** para ambos determina y suma el precio, y **WeightVisitor** para seguirle la pista a los pesos.

Puedes ver todo esto implementado en la versión nueva, perfeccionada del programa de reciclaje. Al igual que con **DoubleDispatch.java**, la clase **Trash** queda a solas y una interfaz nueva es creada para agregar el método **accept**():

```
//: Visitable.java
// An interface to add visitor functionality to
// the Trash hierarchy without modifying the
// base class.
package c16.trashvisitor;
import c16.trash.*;

interface Visitable {
  // The new method:
  void accept(Visitor v);
} ///:~
```

Los subtipos de **Aluminum**, **Paper**, **Glass**, y **Cardboard** implementan el método **accept**():

```
//: VAluminum.java
// Aluminum for the visitor pattern
package c16.trashvisitor;
import c16.trash.*;

public class VAluminum extends Aluminum
    implements Visitable {
  public VAluminum(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
//: VPaper.java
// Paper for the visitor pattern
```

```
package c16.trashvisitor;
import c16.trash.*;

public class VPaper extends Paper
    implements Visitable {
  public VPaper(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
//: VGlass.java
// Glass for the visitor pattern
package c16.trashvisitor;
import c16.trash.*;

public class VGlass extends Glass
    implements Visitable {
  public VGlass(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
//: VCardboard.java
// Cardboard for the visitor pattern
package c16.trashvisitor;
import c16.trash.*;

public class VCardboard extends Cardboard
    implements Visitable {
  public VCardboard(double wt) { super(wt); }
  public void accept(Visitor v) {
    v.visit(this);
  }
} ///:~
```

Como nadie toma consistencia en la clase base **Visitor**, puede ser creado como una **interfaz**:

```
//: Visitor.java
// The base interface for visitors
package c16.trashvisitor;
import c16.trash.*;

interface Visitor {
  void visit(VAluminum a);
  void visit(VPaper p);
  void visit(VGlass g);
  void visit(VCardboard c);
} ///:~
```

Otra vez los tipos personalizados **Trash** han sido creados en un subdirectorio diferente. El fichero de datos nuevo **Trash** es **VTrash.dat** y se parece a esto:

```
c16.TrashVisitor.VGlass:54
c16.TrashVisitor.VPaper:22
c16.TrashVisitor.VPaper:11
c16.TrashVisitor.VGlass:17
c16.TrashVisitor.VAluminum:89
c16.TrashVisitor.VPaper:88
c16.TrashVisitor.VAluminum:76
c16.TrashVisitor.VCardboard:96
c16.TrashVisitor.VAluminum:25
c16.TrashVisitor.VAluminum:34
c16.TrashVisitor.VGlass:11
c16.TrashVisitor.VGlass:68
c16.TrashVisitor.VGlass:43
c16.TrashVisitor.VAluminum:27
c16.TrashVisitor.VCardboard:44
c16.TrashVisitor.VAluminum:18
c16.TrashVisitor.VPaper:91
c16.TrashVisitor.VGlass:63
c16.TrashVisitor.VGlass:50
c16.TrashVisitor.VGlass:80
c16.TrashVisitor.VAluminum:81
c16.TrashVisitor.VCardboard:12
c16.TrashVisitor.VGlass:12
c16.TrashVisitor.VGlass:54
c16.TrashVisitor.VAluminum:36
c16.TrashVisitor.VAluminum:93
c16.TrashVisitor.VGlass:93
c16.TrashVisitor.VPaper:80
c16.TrashVisitor.VGlass:36
c16.TrashVisitor.VGlass:12
c16.TrashVisitor.VGlass:60
c16.TrashVisitor.VPaper:66
c16.TrashVisitor.VAluminum:36
c16.TrashVisitor.VCardboard:22
```

El resto del programa crea tipos específicos **Visitor** y los envía a través de una lista única de objetos **Trash**:

```
//: TrashVisitor.java
// The "visitor" pattern
package c16.trashvisitor;
import c16.trash.*;
import java.util.*;

// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(VAluminum al) {
```

```
    double v = al.weight() * al.value();
    System.out.println(
      "value of Aluminum= " + v);
    alSum += v;
  }
  public void visit(VPaper p) {
    double v = p.weight() * p.value();
    System.out.println(
      "value of Paper= " + v);
    pSum += v;
  }
  public void visit(VGlass g) {
    double v = g.weight() * g.value();
    System.out.println(
      "value of Glass= " + v);
    gSum += v;
  }
  public void visit(VCardboard c) {
    double v = c.weight() * c.value();
    System.out.println(
      "value of Cardboard = " + v);
    cSum += v;
  }
  void total() {
    System.out.println(
      "Total Aluminum: $" + alSum + "\n" +
      "Total Paper: $" + pSum + "\n" +
      "Total Glass: $" + gSum + "\n" +
      "Total Cardboard: $" + cSum);
  }
}

class WeightVisitor implements Visitor {
  private double alSum; // Aluminum
  private double pSum; // Paper
  private double gSum; // Glass
  private double cSum; // Cardboard
  public void visit(VAluminum al) {
    alSum += al.weight();
    System.out.println("weight of Aluminum = "
        + al.weight());
  }
  public void visit(VPaper p) {
    pSum += p.weight();
    System.out.println("weight of Paper = "
        + p.weight());
  }
  public void visit(VGlass g) {
    gSum += g.weight();
    System.out.println("weight of Glass = "
        + g.weight());
  }
  public void visit(VCardboard c) {
```

```
    cSum += c.weight();
    System.out.println("weight of Cardboard = "
        + c.weight());
  }
  void total() {
    System.out.println("Total weight Aluminum:"
        + alSum);
    System.out.println("Total weight Paper:"
        + pSum);
    System.out.println("Total weight Glass:"
        + gSum);
    System.out.println("Total weight Cardboard:"
        + cSum);
  }
}

public class TrashVisitor {
  public static void main(String[] args) {
    Vector bin = new Vector();
    // ParseTrash still works, without changes:
    ParseTrash.fillBin("VTrash.dat", bin);
    // You could even iterate through
    // a list of visitors!
    PriceVisitor pv = new PriceVisitor();
    WeightVisitor wv = new WeightVisitor();
    Enumeration it = bin.elements();
    while(it.hasMoreElements()) {
      Visitable v = (Visitable)it.nextElement();
      v.accept(pv);
      v.accept(wv);
    }
    pv.total();
    wv.total();
  }
} ///:~
```

Note que la forma de **main**() ha vuelto a cambiar. Ahora hay sólo un depósito único **Trash**. Los dos objetos **Visitor** son aceptados en cada elemento en la secuencia, y realizan sus operaciones. Las visitas conservan sus datos internos para cuadrar los pesos totales y los precios.

Finalmente, no hay identificación de tipo de tiempo de ejecución aparte del molde inevitable para **Trash** cuando tiran cosas fuera de la secuencia. Esto, también, podría ser eliminado con la implementación de tipos parametrizado en Java.

Una forma que puedes distinguir esta solución de la solución de despacho doble descrita previamente es notar eso, en la solución de despacho doble, sólo uno de los métodos sobrecargados, **add**(), fue sobrescrito cuando cada subclase fue creada, mientras que aquí cada uno de los métodos sobrecargados **visit**() son sobrescritos en cada subclase de **Visitor**.

¿Más acoplador?

Hay mucho más código aquí, y allí es definitivo acoplarse entre la jerarquía **Trash** y la jerarquía **Visitor**. Sin embargo, hay también alta cohesión dentro de los conjuntos respectivos de clases: Ellos cada uno hacen una única cosa (**Trash** describe a **Trash**, mientras **Visitor** describe acciones realizadas en **Trash**), lo cual es un señalizador de un buen diseño. Por supuesto, en este caso marcha bien sólo si agregas nuevos **Visitors**, pero se pone en medio del camino cuando agregas tipos nuevos de **Trash**.

El acoplador bajo entre las clases y la alta cohesión dentro de una clase es definitivamente una meta importante del diseño. Aplicado descuidadamente, sin embargo, te puede impedir lograr un diseño más elegante. Tal parece ser que algunas clases inevitablemente tienen una cierta intimidad con otro. Éstos a menudo ocurren a pares que quizá podrían ser llamados pareados, por ejemplo, las colecciones y los iteradores (**Enumerations**). El par **Trash**-**Visitor** de arriba parece ser tan pareado.

¿RTTI considerado dañino?

Varios diseños en este capítulo tratan de quitar a RTTI, lo cual te podría dar la impresión que es “considerada dañina” (la condenación destinada para pobre, malo y predestinado **goto**, el cual nunca fue de tal manera puesto en Java). Esto no es cierto; El uso indebido de RTTI es el problema. La razón por la que nuestros diseños quitaron a RTTI es porque la aplicación errada de esa característica impidió extensibilidad, mientras la meta indicada fue poder añadirle un tipo nuevo al sistema con poco impacto en código circundante como sea posible. Desde que RTTI es a menudo despilfarrado haciéndole buscar cada tipo único en tu sistema, causa el código a ser poco extensible: Cuando agregas un tipo nuevo, tienes que andar a la caza de todo el código en el cual RTTI es usado, y si lo pierdes noobtendrás ayuda del compilador.

Sin embargo, RTTI automáticamente no crea código poco extensible. Volvamos a visitar el reciclador de basura otra vez. Esta vez, una herramienta nueva será introducida, el cual se llama a un **TypeMap**. Contiene un **Hashtable** que contiene **Vectors**, pero la interfaz es simple: Puedes **add**() un objeto nuevo, y puedes **get**() un Vector conteniendo todos los objetos de un tipo particular. Las claves para el **Hashtable** contenido son los tipos en el Vector asociado. La belleza de este diseño (propuesto por Larry O'Brien) es que el **TypeMap** dinámicamente agrega un par nuevo cada vez que encuentra un tipo nuevo, así es que cada vez que le añades un tipo nuevo al sistema (aun si agregas el tipo nuevo en el tiempo de ejecución), se adapta.

Nuestro ejemplo otra vez se fundamentará en la estructura de los tipos **Trash** en paquete **c16.Trash** (y el archivo **Trash.dat** usado allá puede ser usado aquí sin cambio):

```
//: DynaTrash.java
// Using a Hashtable of Vectors and RTTI
```

```
// to automatically sort trash into
// vectors. This solution, despite the
// use of RTTI, is extensible.
package c16.dynatrash;
import c16.trash.*;
import java.util.*;

// Generic TypeMap works in any situation:
class TypeMap {
  private Hashtable t = new Hashtable();
  public void add(Object o) {
    Class type = o.getClass();
    if(t.containsKey(type))
      ((Vector)t.get(type)).addElement(o);
    else {
      Vector v = new Vector();
      v.addElement(o);
      t.put(type,v);
    }
  }
  public Vector get(Class type) {
    return (Vector)t.get(type);
  }
  public Enumeration keys() { return t.keys(); }
  // Returns handle to adapter class to allow
  // callbacks from ParseTrash.fillBin():
  public Fillable filler() {
    // Anonymous inner class:
    return new Fillable() {
      public void addTrash(Trash t) { add(t); }
    };
  }
}

public class DynaTrash {
  public static void main(String[] args) {
    TypeMap bin = new TypeMap();
    ParseTrash.fillBin("Trash.dat",bin.filler());
    Enumeration keys = bin.keys();
    while(keys.hasMoreElements())
      Trash.sumValue(
        bin.get((Class)keys.nextElement()));
  }
} ///:~
```

Aunque poderosa, la definición para **TypeMap** es simple. Contiene un **Hashtable**, y el método **add**() hace más del trabajo. Cuando **add**() un objeto nuevo, el manipulador para el objeto **Class** para ese tipo es extraído. Esto es utilizado como una clave para determinar si un **Vector** conteniendo objetos de ese tipo está ya presente en el **Hashtable**. Si es así, ese **Vector** es extraído y el objeto es añadido al **Vector**. En caso de que no, el objeto **Class** y un **Vector** nuevo se agregan como un par de valor de clave.

Puedes traer un **Enumeration** de todos los objetos **Class** desde **keys**(), y puedes usar cada objeto **Class** para ir a traer al Vector correspondiente con **get**(). Y todo está allí para eso.

El método **filler**() es interesante porque se aprovecha del diseño de **ParseTrash.fillBin**(), lo cual simplemente no trata de llenar un Vector pero en lugar de eso cualquier cosa que implementa al **Fillable** interactúan con su método **addTrash**(). Todo lo que **filler**() necesita hacer es devolverle un manipulador a una interfaz que implementa **Fillable**, y entonces este manipulador puede ser utilizado como un argumento para **fillBin**() como éste:

```
ParseTrash.fillBin("Trash.dat", bin.filler());
```

Para producir este manipulador, una clase interna anónima (descrita en el Capítulo 8) es usada. Nunca necesitas que una clase nombrada implemente a **Fillable** , simplemente necesitas un manipulador para un objeto de esa clase, así que éste es un uso apropiado de clases internas anónimas.

Una cosa que llama la atención acerca de este diseño es que si bien no es creado para manejar la ordenación, **fillBin**() realiza un orden cada vez que introduce a un objeto **Trash** en depósito.

Muchas de las clases **DynaTrash** deberían ser familiar desde los ejemplos previos. Esta vez, en lugar de colocar a los objetos nuevos **Trash** dentro de un depósito de tipo Vector, el depósito es de tipo **TypeMap**, así es que cuando la basura es lanzada en depósito es inmediatamente ordenada por mecanismo interno de ordenación de **TypeMap**. Dar un paso a través del **TypeMap** y operar en cada Vector individual se convierte en un asunto simple:

```
Enumeration keys = bin.keys();
while(keys.hasMoreElements())
   Trash.sumValue(
 bin.get((Class)keys.nextElement()));
```

Como puedes ver, añadiéndole un tipo nuevo al sistema no afectará este código en absoluto, ni el código en **TypeMap**. Ésta es ciertamente la solución menor para el problema, y discutiblemente el más elegante igualmente. Eso confía con exceso en RTTI, pero note que cada par de valor clave en el **Hashtable** anda buscando sólo un tipo. Además, no hay forma de que puedas “olvidar” añadirle el código correcto a este sistema cuando agregas un tipo nuevo, ya que no hay ningún código que necesites agregar.

Resumen

Sacar de entre manos un diseño como **TrashVisitor.java** que contiene una mayor cantidad de código que los anteriores diseños puede parecer al principio ser contraproductivo. Conviene notar lo que estás tratando de lograr con varios diseños. Los patrones de diseño en general se esfuerzan por separar las cosas que cambian de las cosas que permanecen igual. Las “cosas que cambian”

pueden referirse a muchos tipos diferentes de cambios. Quizá el cambio ocurre porque el programa es colocado dentro de un ambiente nuevo o porque algo en el ambiente actual cambia (éste podría ser: "el usuario quiere añadirle una forma nueva al diagrama actualmente en la pantalla"). O, como en este caso, el cambio podría ser la evolución del cuerpo del código. Mientras las versiones previas del ejemplo de ordenación de basura enfatizaron la adición de tipos nuevos de **Trash** para el sistema, **TrashVisitor.java** te permite fácilmente agregar funcionabilidad nueva sin incomodar la jerarquía **Trash**. Hay más código en **TrashVisitor.java**, pero añadiéndole nueva funcionabilidad a **Visitor** es de mal gusto. Si esto es algo que ocurre bastante, entonces vale el código y esfuerzo extra para hacerlo ocurrir más fácilmente.

El descubrimiento del vector de cambio no es asunto trivial; No es algo que un analista usualmente puede detectar antes de que el programa vea su diseño inicial. La información necesaria probablemente no lo hará aparecer hasta posteriores fases en el proyecto: Algunas veces sólo en las fases de diseño o implementación te hacen descubrir una necesidad más profunda o más sutil en tu sistema. En caso de agregar tipos nuevos (el cual fue el foco de la mayor parte de los ejemplos de "reciclaje") podrías darte cuenta de que necesitas una jerarquía particular de la herencia sólo cuando estás en la fase de mantenimiento y empezáis a prolongar el sistema!

Una de las cosas más importantes que aprenderás estudiando patrones de diseño parece ser un cambio radical de actitud de lo que ha sido promovido en lo que va de este libro. Es decir: "OOP es todo acerca del polimorfismo." Esta declaración puede producir el síndrome de "dos años de edad con un martillo" (todo se parece a una uña). Puesto de cualquier otro modo, es duramente suficiente "obtener" polimorfismo, y una vez que lo haces, tratas de lanzar todos tus diseños en ese único molde particular.

Lo que patrones de diseño dice es que OOP no se trata justamente de polimorfismo. Se trata de "separar las cosas que cambian de las cosas que permanecen igual." El polimorfismo es una forma especialmente importante para hacer esto, y resulta ser de ayuda si el lenguaje de programación directamente soporta polimorfismo (así es que no tienes que interceptarlo en ti mismo, el cual tendería a hacerlo prohibitivamente costoso). Pero los patrones de diseño en general demuestran otras formas para lograr la meta básica, y una vez que tus ojos hayan sido abiertos para éste comenzarás a buscar más diseños creativos.

Ya que el libro de *Patrones de Diseño* salió afuera e hizo tal impacto, las personas han estado yendo en busca de otros patrones. Puedes esperar ver que más de estos aparecen como pasa el tiempo. Aquí hay algunos sitios recomendados por Jim Coplien, de fama C++ (*http://www.bell-labs.com/~cope*), quien es uno de los proponentes principales del movimiento de patrones:

http://st-www.cs.uiuc.edu/users/patterns
http://c2.com/cgi/wiki
http://c2.com/ppr
http://www.bell-labs.com/people/cope/Patterns/Process/index.html
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic

http://www.cs.wustl.edu/~schmidt/patterns.html
http://www.espinc.com/patterns/overview.html

También nótese que ha habido una convención anual en patrones del diseño, llamado PLAF, eso produce unas actas publicadas, la tercera parte de la cual salió afuera después de 1997 (todo publicado por Addison-Wesley).

Ejercicios

1. Usando **SingletonPattern.java** como un punto de partida, crea una clase que maneja un número fijo de sus propios objetos.
2. Añádele una clase **Plastic** a **TrashVisitor.java**.
3. Añádele una clase **Plastic** a **DynaTrash.java**.

18: Proyectos

Este capítulo incluye un conjunto de proyectos que se construyen en el material presentado en este libro o de otra manera no ajustada en capítulos anteriores.

La mayor parte de estos proyectos son significativamente más complicados que los ejemplos en el resto de libro, y a menudo demuestran usos y técnicas nuevas de bibliotecas de clase.

Procesar texto

Si vienes de un C o C++ de fondo, podrías ser escéptico en primero del poder de Java en lo que se refiere a manejar texto. Ciertamente, un inconveniente es que la velocidad de ejecución es más lenta y eso podría entorpecer una cierta cantidad de tus esfuerzos. Sin embargo, las herramientas (en detalle la clase **String**) son muy poderosas, como los ejemplos en esta sección muestran (y las mejoras de desempeño han sido prometidas para Java).

Como verás, estos ejemplos fueron creados para solucionar problemas que se levantaron en la creación de este libro. Sin embargo, no están restringidos para eso y las soluciones que le ofrecen fácilmente pueden ser adaptadas a otras situaciones. Además, demuestran el poder de Java en un área que previamente no ha sido enfatizado en este libro.

Extrayendo listados de código

Sin duda has notado que cada listado completo (no el fragmento de código) de código en este libro comienza y finaliza con marcas especiales de etiqueta de comentario '//:' y '///:~'. Esta meta información es incluida a fin de que el código pueda ser automáticamente extraído del libro en archivos de código fuente del compilable. En mi libro previo, tuve un sistema que me permitió automáticamente incorporarse archivos probados de código en el libro. En este libro, sin embargo, descubrí que fue a menudo más fácil pegar el código en el libro una vez que estaba inicialmente probado y, ya que es difícil de poner lo correcto la primera vez, para realizar ediciones para el código dentro del libro. ¿Pero cómo extraerlo y probar el código? Este programa es la respuesta, y podría venir bien cuando te dispones a solucionar un problema de procesamiento del texto. También demuestra muchas de las características de clase **String**.

Primero guardo el libro entero en formato del texto de ASCII en un archivo separado. El programa **CodePackager** tiene dos modos (el cual puedes ver descrito en **usageString**): Si usas la bandera **-p**, espera ver un archivo de entrada conteniendo el texto de ASCII del libro. Pasará a través de este archivo y usará las marcas de la etiqueta del comentario para extraer el

código, y usa el nombre de archivo en la primera línea para determinar el nombre del archivo. Además, busca la declaración del paquete en el caso de que necesita colocar el archivo en un directorio especial (escogido por la ruta indicada por la declaración **package**).

Pero eso no es todo. También espera el cambio en los capítulos siguiéndole la pista a los nombres del paquete. Ya que todos los paquetes para cada capítulo empiezan con **c02**, **c03**, **c04**, etc. para indicar el capítulo donde pertenecen (excepto por esos a partir de **com**, los cuales son ignorados con el objeto de seguirle la pista a los capítulos), con tal de que el primer listado en cada capítulo contenga una declaración **package** con el número de capítulo, el programa **CodePackager** puede seguir la pista de cuando el capítulo cambió y metió todos los subsiguientes archivos en el subdirectorio nuevo **chapter**.

Como cada archivo es extraído, es colocado dentro de un objeto **SourceCodeFile** que es luego colocado dentro de una colección. (Este proceso estará descrito más a fondo después.) Estos objetos **SourceCodeFile** simplemente podrían guardarse en archivos, pero eso nos trae para el segundo uso para este proyecto. Si invocas a **CodePackager** sin la bandera -p espera un archivo "empacado" como la entrada, que luego se extraerá en archivos separados. Así la bandera -p quiere decir que los archivos extraídos serán encontrados "empacados" en este único archivo.

¿Por qué la molestia con el archivo empacado? Porque las plataformas diferentes de la computadora tienen formas diferentes de almacenar información del texto en archivos. Un gran asunto es el caracter fin de línea o caracteres, pero los otros asuntos también pueden existir. Sin embargo, Java tiene un tipo de especial del flujo IO – el **DataOutputStream** – que promete eso, a pesar de qué máquina los datos vienen, el almacenamiento de esa información estará en una forma que puede ser correctamente recuperada por cualquier otra máquina usando a un **DataInputStream**. Es decir, Java maneja todos los detalles específicos en la plataforma, lo cual es una gran parte de la promesa de Java. Así la bandera -p almacena todo en un único archivo en un formato universal. Haces un download de este archivo y el programa Java de la Web, y cuando corres **CodePackager** en este archivo sin la bandera **-p** los archivos todos serán extraídos a lugares correctos en tu sistema. (Puedes especificar un subdirectorio alterno; De otra manera los subdirectorios solamente serán creados en el directorio actual.) Para asegurar que ningún resto específico de sistema de formatos, los objetos File son usados en todas partes como una ruta o un archivo es descrito. Además, hay una comprobación de cordura: Un archivo vacío es colocado en cada subdirectorio; El nombre de ese archivo indica cuántos archivos deberías encontrar en ese subdirectorio.

Aquí está el código, lo cual será relatado detalladamente al final del listado:

```
//: CodePackager.java
// "Packs" and "unpacks" the code in "Thinking
// in Java" for cross-platform distribution.
/* Commented so CodePackager sees it and starts
   a new chapter directory, but so you don't
   have to worry about the directory where this
   program lives:
package c17;
```

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

class Pr {
  static void error(String e) {
    System.err.println("ERROR: " + e);
    System.exit(1);
  }
}

class IO {
  static BufferedReader disOpen(File f) {
    BufferedReader in = null;
    try {
      in = new BufferedReader(
        new FileReader(f));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static BufferedReader disOpen(String fname) {
    return disOpen(new File(fname));
  }
  static DataOutputStream dosOpen(File f) {
    DataOutputStream in = null;
    try {
      in = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream(f)));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static DataOutputStream dosOpen(String fname) {
    return dosOpen(new File(fname));
  }
  static PrintWriter psOpen(File f) {
    PrintWriter in = null;
    try {
      in = new PrintWriter(
        new BufferedWriter(
          new FileWriter(f)));
    } catch(IOException e) {
      Pr.error("could not open " + f);
    }
    return in;
  }
  static PrintWriter psOpen(String fname) {
    return psOpen(new File(fname));
  }
```

```
  static void close(Writer os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
  static void close(DataOutputStream os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
  static void close(Reader os) {
    try {
      os.close();
    } catch(IOException e) {
      Pr.error("closing " + os);
    }
  }
}

class SourceCodeFile {
  public static final String
    startMarker = "//:", // Start of source file
    endMarker = "} ///:~", // End of source
    endMarker2 = "}; ///:~", // C++ file end
    beginContinue = "} ///:Continued",
    endContinue = "///:Continuing",
    packMarker = "###", // Packed file header tag
    eol = // Line separator on current system
      System.getProperty("line.separator"),
    filesep = // System's file path separator
      System.getProperty("file.separator");
  public static String copyright = "";
  static {
    try {
      BufferedReader cr =
        new BufferedReader(
          new FileReader("Copyright.txt"));
      String crin;
      while((crin = cr.readLine()) != null)
        copyright += crin + "\n";
      cr.close();
    } catch(Exception e) {
      copyright = "";
    }
  }
  private String filename, dirname,
    contents = new String();
  private static String chapter = "c02";
  // The file name separator from the old system:
```

```
public static String oldsep;
public String toString() {
  return dirname + filesep + filename;
}
// Constructor for parsing from document file:
public SourceCodeFile(String firstLine,
    BufferedReader in) {
  dirname = chapter;
  // Skip past marker:
  filename = firstLine.substring(
      startMarker.length()).trim();
  // Find space that terminates file name:
  if(filename.indexOf(' ') != -1)
    filename = filename.substring(
        0, filename.indexOf(' '));
  System.out.println("found: " + filename);
  contents = firstLine + eol;
  if(copyright.length() != 0)
    contents += copyright + eol;
  String s;
  boolean foundEndMarker = false;
  try {
    while((s = in.readLine()) != null) {
      if(s.startsWith(startMarker))
        Pr.error("No end of file marker for " +
          filename);
      // For this program, no spaces before
      // the "package" keyword are allowed
      // in the input source code:
      else if(s.startsWith("package")) {
        // Extract package name:
        String pdir = s.substring(
          s.indexOf(' ')).trim();
        pdir = pdir.substring(
          0, pdir.indexOf(';')).trim();
        // Capture the chapter from the package
        // ignoring the 'com' subdirectories:
        if(!pdir.startsWith("com")) {
          int firstDot = pdir.indexOf('.');
          if(firstDot != -1)
            chapter =
              pdir.substring(0,firstDot);
          else
            chapter = pdir;
        }
        // Convert package name to path name:
        pdir = pdir.replace(
          '.', filesep.charAt(0));
        System.out.println("package " + pdir);
        dirname = pdir;
      }
      contents += s + eol;
      // Move past continuations:
```

```
      if(s.startsWith(beginContinue))
        while((s = in.readLine()) != null)
          if(s.startsWith(endContinue)) {
            contents += s + eol;
            break;
          }
      // Watch for end of code listing:
      if(s.startsWith(endMarker) ||
         s.startsWith(endMarker2)) {
        foundEndMarker = true;
        break;
      }
    }
    if(!foundEndMarker)
      Pr.error(
        "End marker not found before EOF");
    System.out.println("Chapter: " + chapter);
  } catch(IOException e) {
    Pr.error("Error reading line");
  }
}
// For recovering from a packed file:
public SourceCodeFile(BufferedReader pFile) {
  try {
    String s = pFile.readLine();
    if(s == null) return;
    if(!s.startsWith(packMarker))
      Pr.error("Can't find " + packMarker
        + " in " + s);
    s = s.substring(
      packMarker.length()).trim();
    dirname = s.substring(0, s.indexOf('#'));
    filename = s.substring(s.indexOf('#') + 1);
    dirname = dirname.replace(
      oldsep.charAt(0), filesep.charAt(0));
    filename = filename.replace(
      oldsep.charAt(0), filesep.charAt(0));
    System.out.println("listing: " + dirname
      + filesep + filename);
    while((s = pFile.readLine()) != null) {
      // Watch for end of code listing:
      if(s.startsWith(endMarker) ||
         s.startsWith(endMarker2)) {
        contents += s;
        break;
      }
      contents += s + eol;
    }
  } catch(IOException e) {
    System.err.println("Error reading line");
  }
}
public boolean hasFile() {
```

```
    return filename != null;
  }
  public String directory() { return dirname; }
  public String filename() { return filename; }
  public String contents() { return contents; }
  // To write to a packed file:
  public void writePacked(DataOutputStream out) {
    try {
      out.writeBytes(
        packMarker + dirname + "#"
        + filename + eol);
      out.writeBytes(contents);
    } catch(IOException e) {
      Pr.error("writing " + dirname +
        filesep + filename);
    }
  }
  // To generate the actual file:
  public void writeFile(String rootpath) {
    File path = new File(rootpath, dirname);
    path.mkdirs();
    PrintWriter p =
      IO.psOpen(new File(path, filename));
    p.print(contents);
    IO.close(p);
  }
}

class DirMap {
  private Hashtable t = new Hashtable();
  private String rootpath;
  DirMap() {
    rootpath = System.getProperty("user.dir");
  }
  DirMap(String alternateDir) {
    rootpath = alternateDir;
  }
  public void add(SourceCodeFile f){
    String path = f.directory();
    if(!t.containsKey(path))
      t.put(path, new Vector());
    ((Vector)t.get(path)).addElement(f);
  }
  public void writePackedFile(String fname) {
    DataOutputStream packed = IO.dosOpen(fname);
    try {
      packed.writeBytes("###Old Separator:" +
        SourceCodeFile.filesep + "###\n");
    } catch(IOException e) {
      Pr.error("Writing separator to " + fname);
    }
    Enumeration e = t.keys();
    while(e.hasMoreElements()) {
```

```
      String dir = (String)e.nextElement();
      System.out.println(
        "Writing directory " + dir);
      Vector v = (Vector)t.get(dir);
      for(int i = 0; i < v.size(); i++) {
        SourceCodeFile f =
          (SourceCodeFile)v.elementAt(i);
        f.writePacked(packed);
      }
    }
    IO.close(packed);
  }
  // Write all the files in their directories:
  public void write() {
    Enumeration e = t.keys();
    while(e.hasMoreElements()) {
      String dir = (String)e.nextElement();
      Vector v = (Vector)t.get(dir);
      for(int i = 0; i < v.size(); i++) {
        SourceCodeFile f =
          (SourceCodeFile)v.elementAt(i);
        f.writeFile(rootpath);
      }
      // Add file indicating file quantity
      // written to this directory as a check:
      IO.close(IO.dosOpen(
        new File(new File(rootpath, dir),
          Integer.toString(v.size())+".files")));
    }
  }
}

public class CodePackager {
  private static final String usageString =
  "usage: java CodePackager packedFileName" +
  "\nExtracts source code files from packed \n" +
  "version of Tjava.doc sources into " +
  "directories off current directory\n" +
  "java CodePackager packedFileName newDir\n" +
  "Extracts into directories off newDir\n" +
  "java CodePackager -p source.txt packedFile" +
  "\nCreates packed version of source files" +
  "\nfrom text version of Tjava.doc";
  private static void usage() {
    System.err.println(usageString);
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) usage();
    if(args[0].equals("-p")) {
      if(args.length != 3)
        usage();
      createPackedFile(args);
```

```
  }
  else {
    if(args.length > 2)
      usage();
    extractPackedFile(args);
  }
}
private static String currentLine;
private static BufferedReader in;
private static DirMap dm;
private static void
createPackedFile(String[] args) {
  dm = new DirMap();
  in = IO.disOpen(args[1]);
  try {
    while((currentLine = in.readLine())
        != null) {
      if(currentLine.startsWith(
          SourceCodeFile.startMarker)) {
        dm.add(new SourceCodeFile(
                currentLine, in));
      }
      else if(currentLine.startsWith(
          SourceCodeFile.endMarker))
        Pr.error("file has no start marker");
      // Else ignore the input line
    }
  } catch(IOException e) {
    Pr.error("Error reading " + args[1]);
  }
  IO.close(in);
  dm.writePackedFile(args[2]);
}
private static void
extractPackedFile(String[] args) {
  if(args.length == 2) // Alternate directory
    dm = new DirMap(args[1]);
  else // Current directory
    dm = new DirMap();
  in = IO.disOpen(args[0]);
  String s = null;
  try {
     s = in.readLine();
  } catch(IOException e) {
    Pr.error("Cannot read from " + in);
  }
  // Capture the separator used in the system
  // that packed the file:
  if(s.indexOf("###Old Separator:") != -1 ) {
    String oldsep = s.substring(
      "###Old Separator:".length());
    oldsep = oldsep.substring(
      0, oldsep. indexOf('#'));
```

```
      SourceCodeFile.oldsep = oldsep;
    }
    SourceCodeFile sf = new SourceCodeFile(in);
    while(sf.hasFile()) {
      dm.add(sf);
      sf = new SourceCodeFile(in);
    }
    dm.write();
  }
} ///:~
```

Primero notarás la declaración **package** que se dejó comentado. Ya que éste es el primer programa en el capítulo, la declaración **package** hay que decirle a **CodePackager** que el capítulo ha cambiado, pero meterla en un paquete sería un problema. Cuando creas un paquete, amarras el programa resultante a una estructura particular del directorio, lo cual es bueno para muchos de los ejemplos en este libro. Aquí, sin embargo, el programa **CodePackager** debe ser compilado y ejecutado desde un directorio arbitrario, así es que la declaración **package** se deja comentada. Todavía se parecerá a una declaración común **package** para **CodePackager**, sin embargo, ya que el programa no es lo suficientemente sofisticado para detectar comentarios multilínea. (No tiene necesidad para tal sofisticación, un hecho que viene bien aquí.)

Las primeras dos clases son clases del soporte/utilidad diseñadas para hacer el resto de programa más consistente para escribir y más fácil para leer. La primera parte, **Pr**, es similar a la biblioteca ANSI C **perror**, ya que imprime un mensaje de error (pero sale del programa). La segunda categoría narra de forma resumida la creación de archivos, un proceso que fue demostrado en el Capítulo 12 como una que rápidamente se vuelve poco conciso y molesto. En el Capítulo 12, la solución propuesta creó clases nuevas, pero aquí las llamadas estáticas de método son usadas. Dentro de esos métodos las excepciones apropiadas son percibidas y distribuidas. Estos métodos le hacen al resto de código mucho más claro para leer.

La primera clase que ayuda a solucionar el problema es **SourceCodeFile**, que represente toda la información (incluyendo el contenido, nombre de archivo, y directorio) para un archivo de código fuente en el libro. También contiene un conjunto de constantes **String** representando los marcadores que empiezan y finalizan un archivo, un marcador usado dentro el archivo empacado, el separador de fin de línea del sistema actual y el separador de la ruta del archivo (note el uso de **System.getProperty**() para obtener la versión local), y un aviso de derecho de autor, que es extraído del siguiente archivo Copyright.txt.

```
//////////////////////////////////////////////
// Copyright (c) Bruce Eckel, 1998
// Source code file from the book "Thinking in Java"
// All rights reserved EXCEPT as allowed by the
// following statements: You may freely use this file
// for your own work (personal or commercial),
// including modifications and distribution in
```

```
// executable form only. Permission is granted to use
// this file in classroom situations, including its
// use in presentation materials, as long as the book
// "Thinking in Java" is cited as the source.
// Except in classroom situations, you may not copy
// and distribute this code; instead, the sole
// distribution point is http://www.BruceEckel.com
// (and official mirror sites) where it is
// freely available. You may not remove this
// copyright and notice. You may not distribute
// modified versions of the source code in this
// package. You may not use this file in printed
// media without the express permission of the
// author. Bruce Eckel makes no representation about
// the suitability of this software for any purpose.
// It is provided "as is" without express or implied
// warranty of any kind, including any implied
// warranty of merchantability, fitness for a
// particular purpose or non-infringement. The entire
// risk as to the quality and performance of the
// software is with you. Bruce Eckel and the
// publisher shall not be liable for any damages
// suffered by you or any third party as a result of
// using or distributing software. In no event will
// Bruce Eckel or the publisher be liable for any
// lost revenue, profit, or data, or for direct,
// indirect, special, consequential, incidental, or
// punitive damages, however caused and regardless of
// the theory of liability, arising out of the use of
// or inability to use software, even if Bruce Eckel
// and the publisher have been advised of the
// possibility of such damages. Should the software
// prove defective, you assume the cost of all
// necessary servicing, repair, or correction. If you
// think you've found an error, please email all
// modified files with clearly commented changes to:
// Bruce@EckelObjects.com. (please use the same
// address for non-code errors found in the book).
//////////////////////////////////////////////////
```

Al extraer archivos de un archivo empacado, el separador del archivo del sistema que empacó el archivo es también notable, así es que puede ser reemplazado con el correcto para el sistema local.

El nombre del subdirecto rio para el capítulo actual es guardado en el campo **chapter**, lo cual es inicializado a **c02**. (Notarás que el listado en el Capítulo 2 no contiene una declaración **package**.) La única vez que el campo de capítulo cambia es cuando una declaración **package** es descubierta en el archivo corriente.

Construyendo un archivo empacado

El primer constructor está acostumbrado a extraer un archivo de la versión del texto de ASCII de este libro. El código llamado (que aparece más abajo en el listado) lee cada línea hasta que encuentra uno que corresponde al comienzo de un listado. En ese momento, crea un objeto nuevo **SourceCodeFile**, pasándolos a la primera línea (que ya ha sido leídos por el código llamado) y el objeto **BufferedReader** del cual extrae el resto de listado de código fuente.

En este punto, comienzas a ver el uso pesado de los métodos **String**. Para extraer el nombre de archivo, la versión sobrecargada de **substring**() es llamada que tomas el comienzo del desplazamiento y va al final del **String**. Este índice de comienzo se produce encontrando al **length**() del **startMarker**. **trim**() remueve espacios en blanco de ambos finales del **String**. La primera línea también puede tener palabras después del nombre del archivo; Estos son detectados usando **indexOf**(), lo cual retorna -1 si no pudiera encontrar el carácter que buscas y el valor donde la primera instancia de ese carácter es encontrada si lo hace. Note que allí está también una versión sobrecargada de **indexOf**() que toma a un **String** en lugar de un carácter.

Una vez que el nombre de archivo es analizado gramaticalmente y almacenado, la primera línea es colocada dentro del contenido **String** (que se usa para contener el texto entero del listado de código fuente). En este punto, el resto de las líneas son leídas y concatenadas en el contenido **String**. Es no completamente tan simple, ya que en ciertas situaciones requieren un manejo especial. Un caso es la detección de errores: Si lo ejecutas en un **startMarker**, quiere decir que ningún marcador de fin estaba posado al final del listado que está actualmente siendo coleccionado. Ésta es una condición de error que aborta el programa.

La segunda causa especial es la palabra clave **package**. Aunque Java es un lenguaje de forma independiente, este programa pide que la palabra clave **package** esté en el comienzo de la línea. Cuando la palabra clave **package** se ve, el nombre del paquete es extraído buscando el espacio al comienzo y el punto y coma al final. (Note que esto también pudo haber sido realizado en una operación única usando al **substring**() sobrecargado que toma ambos los índices del comienzo e índices finales.) Luego los puntos en el nombre del paquete son reemplazados por el separador del archivo, aunque una suposición está hecha aquí es que el separador del archivo es sólo un carácter largo. Esto es probablemente cierto en todos los sistemas, pero es un lugar para ver si hay problemas.

El comportamiento predeterminado es concatenar cada línea para contenido, junto con el **string** fin de línea, hasta que el **endMarker** es descubierto, el cual señala que el constructor debería terminar. Si el fin del archivo es encontrado antes de que el **endMarker** se vea, ese es un error.

Extrayendo desde un archivo empaquetado

El segundo constructor se usa para recuperar los archivos de código fuente desde un archivo empaquetado. Aquí, el método llamado no tiene que preocuparse por saltar por encima del texto intermedio. El archivo contiene todos los archivos de código fuente, colocado extremo a extremo. Todo lo que

necesitas dar para este constructor es el **BufferedReader** donde la información viene, y el constructor la toma de allí. Hay algo de meta información, sin embargo, al principio de cada listado, y esto es denotado por el **packMarker**. Si el **packMarker** no está allí, quiere decir que la persona que llama está equivocadamente tratando de usar a este constructor donde no es apropiado.

Una vez que el **packMarker** es encontrado, es quitado y el nombre del directorio (terminado por un '#') y el nombre de archivo (que vaya al fin de la línea) son extraídos. En ambos casos, el carácter viejo del separador es reemplazado por uno que está reciente para esta máquina usando el método **String replace**(). El separador viejo está posado al principio del archivo empaquetado, y verás cómo es extraído este después en el listado.

El resto del constructor es muy simple. Lee y concatena cada línea para los **contents** hasta que el **endMarker** sea encontrado.

Acceder y escribir los listados

El siguiente conjunto de métodos son asesores simples: **directory**(), **filename**() (nota el método puedes tener lo mismo ortografía y la capitalización como el campo) y **contents**(), y **hasFile**() para indicar si este objeto contiene un archivo o no. (La necesidad para esto se verá más adelante.)

Los tres métodos finales están preocupados con escribir a este listado de código en un archivo, ya sea un archivo empaquetado por **writePacked**() o un archivo fuente Java por **writeFile**(). Todo lo que **writePacked**() necesita es el **DataOutputStream**, lo cual fue abierto en otra parte, y representa el archivo que está siendo escrito. Pone la información del encabezado en la primera línea y luego llama a **writeBytes**() a escribir a **contents** en un formato "universal".

Al escribir el archivo fuente Java, el archivo debe ser creado. Esto se hace por **IO.psOpen**(), dándole un objeto **File** que contiene no sólo el nombre de archivo pero también la ruta. Pero la pregunta ahora es: ¿Existe esta ruta? El usuario tiene la opción de colocar todos los directorios de código de la fuente dentro de un subdirectorio completamente diferente, lo cual aun no podría existir. Así es que antes de que cada archivo sea escrito, **File.mkdirs**() es llamado con la ruta en la que quieres escribir el archivo. Esto hará la ruta entera al mismo tiempo.

Conteniendo la colección entera de listados

Conviene organizar los listados como subdirectorios mientras la colección entera se construye en la memoria. Una razón es otra comprobación de cordura: Como cada subdirectorio de listados es creado, un archivo adicional será añadido cuyo nombre contiene el número de archivos en ese directorio.

La clase **DirMap** produce este efecto y demuestra el concepto de un "**multimapa**." Esto es implementado usando a un **Hashtable** cuyas claves son

los subdirectorios siendo creados y cuyos valores son objetos Vector conteniendo los objetos **SourceCodeFile** en ese directorio particular. Así, en lugar de asociar una clave para un valor único, el "**multimapa**" asocia una clave para un conjunto de valores por el Vector asociado. Aunque esto suena complejo, es notablemente franco implementar. Verás que la mayor parte del tamaño de la clase **DirMap** es debido a las porciones que escriben a archivos, no para la implementación "**multimapa**".

Hay dos formas que puedes hacer un **DirMap**: El constructor predeterminado asume que quieres que los directorios se ramifiquen completamente de lo actual, y el segundo constructor te deja especificar una ruta absoluta alterna para el directorio que empieza.

El método **add**() es donde un montón de acción densa ocurre. Primero, el **directory**() es extraído del **SourceCodeFile** que quieres agregar, y luego el **Hashtable** es examinado para ver si ya contiene esa clave. De lo contrario, un **Vector** nuevo es añadido al **Hashtable** y asociado con esa clave. En este punto, el **Vector** está allí, de una u otra manera, y es extraído así es que el **SourceCodeFile** puede agregarse. Porque los **Vectors** pueden estar fácilmente combinados con **Hashtables** como éste, el poder de ambos es amplificado.

Escribir un archivo empaquetado involucra a abrir el archivo para escribir (como un **DataOutputStream** así los datos son universalmente recuperables) y escribiendo la información del encabezado acerca del separador viejo en la primera línea. Después, un **Enumeration** de las claves **Hashtable** es producido y dado un paso hasta el final para seleccionar cada directorio e irle a traer al Vector asociado con ese directorio así cada **SourceCodeFile** en que a Vector le pueden ser escritos para el archivo empaquetado.

Escribiendo los archivos de origen Java a sus directorios en **write**() son casi idénticos para **writePackedFile**() desde que ambos métodos simplemente llamen el método apropiado en **SourceCodeFile**. Aquí, sin embargo, la ruta raíz es pasada en **SourceCodeFile.writeFile**() y cuando todos los archivos han sido escritos el archivo adicional con el nombre conteniendo el número de archivos es también escrito.

El programa principal

Las clases descritas anteriormente son usadas dentro de **CodePackager**. Primero ves el uso string que queda impreso cada vez que el usuario final invoca el programa incorrectamente, junto con el método **usage**() que lo llama y sale del programa. Todo lo que **main**() hace es determinar si quieres crear un extracto o archivo empaquetado de uno, luego asegura que los argumentos son correctos y llama el método apropiado.

Cuando un archivo empaquetado es creado, - se asume - se hace en el directorio actual, así es que el **DirMap** es creado usando al constructor predeterminado. Después de que el archivo es abierto cada línea es leída y examinada para condiciones particulares:

1. Si la línea comienza con el marcador que empieza para un listado de código fuente, un objeto nuevo **SourceCodeFile** es creado. El constructor lee en el resto de

listado de programa. El manipulador que resulte es directamente añadida al **DirMap**.

2. Si la línea comienza con el marcador de fin para un listado de código fuente, algo ha salido mal, ya que los marcadores de fin deberían ser encontrados sólo por el constructor **SourceCodeFile**.

Al extraer un archivo empaquetado, la extracción puede estar en el directorio actual o en un directorio alterno, así es que el objeto **DirMap** es creado consecuentemente. El archivo es abierto y la primera línea es leída. El separador viejo de información de la ruta del archivo es extraído de esta línea. Luego la entrada se usa para crear el primer objeto **SourceCodeFile** , el cual es añadido al **DirMap**. Los objetos nuevos **SourceCodeFile** son creados y añadidos con tal de que contengan un archivo. (Lo último creado simplemente retornará cuando se queda sin entrada y luego **hasFile**() retornará **false**.)

Comprobando estilo de capitalización

Aunque el ejemplo previo puede venir bien como un guía para algún proyecto tuyo que involucra proceso de texto, este proyecto será directamente útil porque realiza una comprobación de estilo para asegurarse de que tu capitalización se conforma al estilo Java efectiva. Abre cada archivo **.java** en el directorio actual y extrae todos los nombres de clase e identificadores, luego te muestra si cualquiera de ellos no encuentra el estilo Java.

Para que el programa maneje correctamente, primero debes fortalecer un depositario de nombre de clase para contener todos los nombres de clase en la biblioteca estándar Java. Haces esto moviéndose hacia todos los subdirectorios de código fuente para la biblioteca estándar Java y corriendo a **ClassScanner** en cada subdirectorio. Provee como los argumentos el nombre del archivo del depositario (usando la misma ruta y nombre cada vez) y la opción de línea de comando **-a** señala que los nombres de clase deberían ser añadidos al depositario.

Para usar el programa para comprobar tu código, lo corres y le das la ruta y nombre del depositario a usar. Comprobará todas las clases e identificadores en el directorio actual y te dirá cuáles no siguen el estilo típico de capitalización Java.

Deberías ser consciente de que el programa no sea perfecto; Allí pocas veces señalará cuando se piensa que hay un problema pero al mirar el código verás que nada necesita variarse. Esto es un poco molesto, pero es todavía mucho más fácil tratando de encontrar todos estos casos mirando tu código.

La explicación inmediatamente sigue el listado:

```
//: ClassScanner.java
// Scans all files in directory for classes
// and identifiers, to check capitalization.
// Assumes properly compiling code listings.
// Doesn't do everything right, but is a very
// useful aid.
import java.io.*;
```

```
import java.util.*;

class MultiStringMap extends Hashtable {
  public void add(String key, String value) {
    if(!containsKey(key))
      put(key, new Vector());
    ((Vector)get(key)).addElement(value);
  }
  public Vector getVector(String key) {
    if(!containsKey(key)) {
      System.err.println(
        "ERROR: can't find key: " + key);
      System.exit(1);
    }
    return (Vector)get(key);
  }
  public void printValues(PrintStream p) {
    Enumeration k = keys();
    while(k.hasMoreElements()) {
      String oneKey = (String)k.nextElement();
      Vector val = getVector(oneKey);
      for(int i = 0; i < val.size(); i++)
        p.println((String)val.elementAt(i));
    }
  }
}

public class ClassScanner {
  private File path;
  private String[] fileList;
  private Properties classes = new Properties();
  private MultiStringMap
    classMap = new MultiStringMap(),
    identMap = new MultiStringMap();
  private StreamTokenizer in;
  public ClassScanner() {
    path = new File(".");
    fileList = path.list(new JavaFilter());
    for(int i = 0; i < fileList.length; i++) {
      System.out.println(fileList[i]);
      scanListing(fileList[i]);
    }
  }
  void scanListing(String fname) {
    try {
      in = new StreamTokenizer(
          new BufferedReader(
            new FileReader(fname)));
      // Doesn't seem to work:
      // in.slashStarComments(true);
      // in.slashSlashComments(true);
      in.ordinaryChar('/');
      in.ordinaryChar('.');
```

```
    in.wordChars('_', '_');
    in.eolIsSignificant(true);
    while(in.nextToken() !=
          StreamTokenizer.TT_EOF) {
      if(in.ttype == '/')
        eatComments();
      else if(in.ttype ==
              StreamTokenizer.TT_WORD) {
        if(in.sval.equals("class") ||
           in.sval.equals("interface")) {
          // Get class name:
           while(in.nextToken() !=
                 StreamTokenizer.TT_EOF
                 && in.ttype !=
                 StreamTokenizer.TT_WORD)
             ;
           classes.put(in.sval, in.sval);
           classMap.add(fname, in.sval);
        }
        if(in.sval.equals("import") ||
           in.sval.equals("package"))
          discardLine();
        else // It's an identifier or keyword
          identMap.add(fname, in.sval);
      }
    }
  } catch(IOException e) {
    e.printStackTrace();
  }
}
void discardLine() {
  try {
    while(in.nextToken() !=
          StreamTokenizer.TT_EOF
          && in.ttype !=
          StreamTokenizer.TT_EOL)
      ; // Throw away tokens to end of line
  } catch(IOException e) {
    e.printStackTrace();
  }
}
// StreamTokenizer's comment removal seemed
// to be broken. This extracts them:
void eatComments() {
  try {
    if(in.nextToken() !=
       StreamTokenizer.TT_EOF) {
      if(in.ttype == '/')
        discardLine();
      else if(in.ttype != '*')
        in.pushBack();
      else
        while(true) {
```

```
          if(in.nextToken() ==
            StreamTokenizer.TT_EOF)
            break;
          if(in.ttype == '*')
            if(in.nextToken() !=
              StreamTokenizer.TT_EOF
              && in.ttype == '/')
              break;
        }
    }
  } catch(IOException e) {
    e.printStackTrace();
  }
}
public String[] classNames() {
  String[] result = new String[classes.size()];
  Enumeration e = classes.keys();
  int i = 0;
  while(e.hasMoreElements())
    result[i++] = (String)e.nextElement();
  return result;
}
public void checkClassNames() {
  Enumeration files = classMap.keys();
  while(files.hasMoreElements()) {
    String file = (String)files.nextElement();
    Vector cls = classMap.getVector(file);
    for(int i = 0; i < cls.size(); i++) {
      String className =
        (String)cls.elementAt(i);
      if(Character.isLowerCase(
           className.charAt(0)))
        System.out.println(
          "class capitalization error, file: "
          + file + ", class: "
          + className);
    }
  }
}
public void checkIdentNames() {
  Enumeration files = identMap.keys();
  Vector reportSet = new Vector();
  while(files.hasMoreElements()) {
    String file = (String)files.nextElement();
    Vector ids = identMap.getVector(file);
    for(int i = 0; i < ids.size(); i++) {
      String id =
        (String)ids.elementAt(i);
      if(!classes.contains(id)) {
        // Ignore identifiers of length 3 or
        // longer that are all uppercase
        // (probably static final values):
        if(id.length() >= 3 &&
```

```
            id.equals(
              id.toUpperCase()))
            continue;
          // Check to see if first char is upper:
          if(Character.isUpperCase(id.charAt(0))){
            if(reportSet.indexOf(file + id)
                == -1){ // Not reported yet
              reportSet.addElement(file + id);
              System.out.println(
                "Ident capitalization error in:"
                + file + ", ident: " + id);
            }
          }
        }
      }
    }
  }
  static final String usage =
    "Usage: \n" +
    "ClassScanner classnames -a\n" +
    "\tAdds all the class names in this \n" +
    "\tdirectory to the repository file \n" +
    "\tcalled 'classnames'\n" +
    "ClassScanner classnames\n" +
    "\tChecks all the java files in this \n" +
    "\tdirectory for capitalization errors, \n" +
    "\tusing the repository file 'classnames'";
  private static void usage() {
    System.err.println(usage);
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length < 1 || args.length > 2)
      usage();
    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
      try {
        // Try to open an existing
        // properties file:
        InputStream oldlist =
          new BufferedInputStream(
            new FileInputStream(old));
        c.classes.load(oldlist);
        oldlist.close();
      } catch(IOException e) {
        System.err.println("Could not open "
          + old + " for reading");
        System.exit(1);
      }
    }
    if(args.length == 1) {
      c.checkClassNames();
```

```
      c.checkIdentNames();
    }
    // Write the class names to a repository:
    if(args.length == 2) {
      if(!args[1].equals("-a"))
        usage();
      try {
        BufferedOutputStream out =
          new BufferedOutputStream(
            new FileOutputStream(args[0]));
        c.classes.save(out,
          "Classes found by ClassScanner.java");
        out.close();
      } catch(IOException e) {
        System.err.println(
          "Could not write " + args[0]);
        System.exit(1);
      }
    }
  }
}

class JavaFilter implements FilenameFilter {
  public boolean accept(File dir, String name) {
    // Strip path information:
    String f = new File(name).getName();
    return f.trim().endsWith(".java");
  }
} ///:~
```

La clase **MultiStringMap** es una herramienta que te permite asociar un grupo de string encima de cada entrada crucial. Como en el ejemplo previo, usa a un **Hashtable** (esta vez con herencia) con la clave como el sencillo string que se asoció encima del valor Vector. El método **add**() simplemente inspecciona para ver si hay ya una clave en el **Hashtable**, y en caso de que no coloca uno allí. El método **getVector**() produce a un Vector para una clave particular, y **printValues**(), lo cual es muy apropiado para corrección de errores, imprime todo el valor Vector por Vector.

Para conservar la vigencia básica, los nombres de clase de las bibliotecas estándar Java son todo puestos en un objeto **Properties** (de la biblioteca estándar Java). Recuerde que un objeto **Properties** es un **Hashtable** que contiene sólo objetos **String** para ambos la clave y entradas de valor. Sin embargo, puede ser salvado a disco y recuperado de disco en una llamada de método, así es que es ideal para el depositario de nombres. Realmente, necesitamos sólo una lista de nombres, y un **Hashtable** no puede aceptar nulo para ya sea su clave o su entrada de valor. Así es que el mismo objeto servirá para ambos la clave y el valor.

Para las clases e identificadores que son descubiertos para los archivos en un directorio particular, dos **MultiStringMaps** son usados: **ClassMap** e **identMap**. También, cuando el programa empieza le carga el depositario estándar de nombre de clase en el objeto **Properties** llamadas **classes**, y

cuando un nombre nuevo de clase es encontrado en el directorio local que es también añadido a las clases así como también para **classMap**. Así, **classMap** puede estar acostumbrado a dar un paso hasta el final todas las clases en el directorio local, y las clases pueden usarse para ver si la muestra actual es un nombre de clase (que indica que una definición de un objeto o método empieza, así es que agarra las siguientes muestras – hasta un punto y coma – y las introduce en **identMap**).

El constructor predeterminado para **ClassScanner** crea una lista de nombres de archivo (usando la implementación **JavaFilter** de **FilenameFilter**, como se ha descrito en el Capítulo 12). Luego llama a **scanListing**() para cada nombre de archivo.

Dentro de **scanListing**() el archivo del código fuente es abierto y revuelto en un **StreamTokenizer**. En la documentación, pasando **true** a **slashStarComments**() y **slashSlashComments**() - se supone - devasta esos comentarios fuera, pero esto parece ser un poco defectuoso (realmente no trabaja en Java 1.0). En lugar de eso, esas líneas son comentadas y los comentarios son extraídos por otro método. Para hacer esto, el debe ser capturado como un carácter común en vez de dejar a **StreamTokenizer** absorberlo como parte de un comentario, y el método **ordinaryChar**() dice al **StreamTokenizer** que haga esto. Esto es también cierto por puntos (**'.'**), Ya que queremos tener las llamadas de método jaladas aparte en identificadores individuales. Sin embargo, la línea subrayada, el cual es ordinariamente tratado por **StreamTokenizer** como un carácter individual, debería quedar como parte de identificadores desde que aparezca en tales valores finales estáticos como **TT_EOF** etcétera., Usado en este mismo programa. El método **wordChars**() toma un rango de personajes que quieres añadirle a esos que se quedan dentro de una muestra que está siendo analizada gramaticalmente como una palabra. Finalmente, al analizar gramaticalmente por comentarios de un línea o descartar una línea necesitamos conocer cuándo ocurre un fin de línea, así llamando a **eolIsSignificant**(**true**) el **eol** aparecerá en vez de siendo absorbido por el **StreamTokenizer**.

El resto de **scanListing**() lee y reacciona a los valores simbólicos hasta que el final del archivo, significado cuando **nextToken**() devuelva el valor estático final **StreamTokenizer.TT_EOF**.

Si la muestra es uno que es potencialmente un comentario, así **eatComments**() son llamados para ocuparse de ella. La única otra situación que estamos interesados aquí dentro es que si es una palabra, del cual hay algunas causas especiales.

Si la palabra es clase o interfaz entonces la siguiente muestra representa una clase o un nombre de la interfaz, y es introducida en **classs** y **classMap**. Si la palabra es **import** o **package**, entonces no queremos el resto de la línea. Cualquier otra cosa debe ser un identificador o una palabra clave (en el cual estamos interesados) (los cuales no son, pero son toda letra minúscula de cualquier manera así es que no echará a perder cosas para echar esos). Estos son añadidos a **identMap**.

El método **discardLine**() es una herramienta simple que busca el fin de una línea. Nota que siempre que obtienes una muestra nueva, debes revisar en

busca del fin del archivo.

El método **eatComments**() es llamado cada vez que un slash adelantado es encontrado en el bucle principal de análisis gramatical. Sin embargo, eso necesariamente no quiere decir que un comentario haya sido encontrado, así es que la siguiente muestra debe ser extraída para ver si es otro slash adelantado (en cuyo caso la línea es descartada) o un asterisco. ¡Pero si son ningún de esos, quiere decir que la muestra que justamente has arrancado es necesaria retornarlo en el bucle general de análisis sintáctico! Afortunadamente, el método **pushBack**() te permite retroceder la muestra actual encima del flujo de entrada a fin de que cuando el bucle principal de análisis gramatical llama a **nextToken**() traerá el mismo que justamente retrocediste.

Por conveniencia, el método **classNames**() produce un arreglo de todos los nombres en la colección de clases. Este método no es usado en el programa pero es de ayuda para depurar.

Los siguientes dos métodos son los únicos en los cuales la comprobación real tiene lugar. En **checkClassNames**(), los nombres de clase son extraídos del **classMap** (el cual, recuerda que, contiene sólo los nombres en este directorio, organizado por el nombre de archivo así es que el nombre de archivo puede ser impreso junto con el nombre errante de clase). Esto está consumado atrayendo a cada Vector asociado y dando un paso a través de eso, viendo si el primer carácter es letra minúscula. Si es así, el mensaje apropiado de error es impreso.

En **checkIdentNames**(), un acercamiento similar es tomado: Cada nombre de identificador es extraído de **identMap**. Si el nombre no está en la lista de clases, - se asume - es un identificador o palabra clave. Una causa especial es comprobada: Si la longitud de identificador es 3 o más y todos los caracteres son mayúsculas, este identificador es ignorado porque es probablemente un valor final estático como **TT_EOF**. Por supuesto, éste no es un algoritmo perfecto, pero da por supuesto que eventualmente notarás cualquiera de todos los identificadores mayúsculas que están fuera de lugar.

En lugar de reportar cada identificador que comienza con un carácter mayúscula, este método le sigue la pista a los cuales ya han sido reportados en un Vector llamado **reportSet**(). Esto trata al Vector como un conjunto que te dice ya sea que un artículo está ya en el conjunto. El artículo se produce concatenando el nombre de archivo e identificador. Si el elemento no está en el conjunto, se agrega y entonces el informe se hace.

El resto de listado está implícito de **main**(), lo cual se ocupa manejando los argumentos de la línea de comando e imaginándose que fuera de si estás fortaleciendo a un depositario de nombres de clase de la biblioteca estándar Java o comprobando la validez de código que has escrito. En ambos casos hace un objeto **ClassScanner**.

Si estás construyendo a un depositario o usando uno, debes tratar de abrir al depositario existente. Haciendo un objeto File y haciendo pruebas para la existencia, puedes decidirte ya sea abrir el archivo y **load**() las clases de la lista de **Properties** dentro de **ClassScanner**. (Las clases del depositario se añaden a, en vez de sobrescribir, las clases encontradas por el constructor

ClassScanner.) Si provees sólo un argumento de línea de comando quiere decir que quieres realizar una comprobación de los nombres de clase y los nombres de identificador, pero si provees dos argumentos (el segundo siendo -a) construyes un depositario de nombre de clase. En este caso, un archivo de salida es abierto y el método **Properties.save**() se usa para escribir la lista en un archivo, junto con uno string que provee información del archivo del encabezado.

Una herramienta de búsqueda de método

El capítulo 10 introdujo a la Java 1.1 concepto de reflexión y se usó esa característica para buscar métodos para una clase particular – ya sea la lista entera de métodos o un subconjunto de esos cuyos nombres corresponden a una palabra clave que provee. La magia de esto es que automáticamente os puede demostrar a todos ustedes los métodos para una clase sin obligarte a caminar arriba de la jerarquía de la herencia examinando las clases base en cada nivel. Así, provee una herramienta ahorradora de tiempo valiosa para programar: Porque los nombres de la mayoría de nombres de método Java son hechos amablemente verboso y descriptivo, puedes ir en busca de los nombres de método que contienen una palabra particular de interés. Cuando encuentras lo que piensas que estás buscando, compruebas la documentación en línea.

Sin embargo, por el Capítulo 10 no habías visto Swing, a fin de que la herramienta fuera desarrollada como una aplicación de línea de comando. Aquí está la versión GUI más útil, lo cual dinámicamente actualiza la salida cuando escribes y también te permite cortar y pegar desde la salida:

```
//: DisplayMethods.java
// Display the methods of any class inside
// a window. Dynamically narrows your search.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.lang.reflect.*;
import java.io.*;

public class DisplayMethods extends Applet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  TextField
    name = new TextField(40),
    searchFor = new TextField(30);
  Checkbox strip =
    new Checkbox("Strip Qualifiers");
  TextArea results = new TextArea(40, 65);
  public void init() {
```

```
    strip.setState(true);
    name.addTextListener(new NameL());
    searchFor.addTextListener(new SearchForL());
    strip.addItemListener(new StripL());
    Panel
      top = new Panel(),
      lower = new Panel(),
      p = new Panel();
    top.add(new Label("Qualified class name:"));
    top.add(name);
    lower.add(
      new Label("String to search for:"));
    lower.add(searchFor);
    lower.add(strip);
    p.setLayout(new BorderLayout());
    p.add(top, BorderLayout.NORTH);
    p.add(lower, BorderLayout.SOUTH);
    setLayout(new BorderLayout());
    add(p, BorderLayout.NORTH);
    add(results, BorderLayout.CENTER);
  }
  class NameL implements TextListener {
    public void textValueChanged(TextEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("No match");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName(nm);
      } catch (ClassNotFoundException ex) {
        results.setText("No match");
        return;
      }
      m = cl.getMethods();
      ctor = cl.getConstructors();
      // Convert to an array of Strings:
      n = new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      for(int i = 0; i < ctor.length; i++)
        n[i + m.length] = ctor[i].toString();
      reDisplay();
    }
  }
  void reDisplay() {
    // Create the result set:
    String[] rs = new String[n.length];
    String find = searchFor.getText();
    int j = 0;
    // Select from the list if find exists:
    for (int i = 0; i < n.length; i++) {
```

```
      if(find == null)
        rs[j++] = n[i];
      else if(n[i].indexOf(find) != -1)
          rs[j++] = n[i];
    }
    results.setText("");
    if(strip.getState() == true)
      for (int i = 0; i < j; i++)
        results.append(
          StripQualifiers.strip(rs[i]) + "\n");
    else // Leave qualifiers on
      for (int i = 0; i < j; i++)
        results.append(rs[i] + "\n");
  }
  class StripL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      reDisplay();
    }
  }
  class SearchForL implements TextListener {
    public void textValueChanged(TextEvent e) {
      reDisplay();
    }
  }
  public static void main(String[] args) {
    DisplayMethods applet = new DisplayMethods();
    Frame aFrame = new Frame("Display Methods");
    aFrame.addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      });
    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(500,750);
    applet.init();
    applet.start();
    aFrame.setVisible(true);
  }
}

class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
      st = new StreamTokenizer(
        new StringReader(qualified));
      st.ordinaryChar(' ');
  }
  public String getNext() {
    String s = null;
    try {
      if(st.nextToken() !=
           StreamTokenizer.TT_EOF) {
```

```
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
            s = new String(st.sval);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
      }
    } catch(IOException e) {
      System.out.println(e);
    }
    return s;
  }
  public static String strip(String qualified) {
    StripQualifiers sq =
      new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
      int lastDot = si.lastIndexOf('.');
      if(lastDot != -1)
        si = si.substring(lastDot + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

Algunas cosas que has visto antes. Al igual que con muchos de los programas GUI en este libro, esto es creado para realizar ambos como una aplicación y como un applet. También, la clase **StripQualifiers** es exactamente igual como estaba en el Capítulo 10.

El GUI contiene un nombre **TextField** en el cual puedes introducir el nombre de clase con creces calificado que quieres buscar, y otro, **searchFor**, en el cual puedes entrar en el texto optativo para ir en busca dentro de la lista de métodos. El **Checkbox** te permite decir si quieres usar los nombres completamente calificados en la salida o si quieres la capacitación quitada. Finalmente, los resultados son desplegados en un **TextArea**.

Notarás que no hay botones u otros componentes por los cuales señala que quieres que la búsqueda empiece. Eso es porque ambos de los **TextFields** y el **Checkbox** son monitoreados por sus objetos del oyente. Cada vez que haces un cambio, la lista está inmediatamente actualizada. Si cambias el texto dentro del campo **name**, el texto nuevo es capturado en la clase **NameL**. Si el texto no está vacío, es usado dentro de **Class.forName**() para tratar de buscar la clase. Como escribe, claro está, el nombre será incompleto y **Class.forName**() fallará, lo cual quiere decir que lanza una excepción. Esto es atrapado y la

TextArea está lista para "No match". Pero tan pronto como introduces un nombre correcto (la capitalización tiene importancia), **Class.forName**() tiene éxito y **getMethods**() y **getConstructors**() devolverán arreglos de objetos **Method** y **Constructor**, respectivamente. Cada uno de los objetos en estos arreglos son convertidos en un **String** por **toString**() (éste produce la firma de método completo o del constructor) y ambas listas están combinados en n, un sencillo arreglo **String**. El arreglo n forma parte de la clase **DisplayMethods** y es usada en actualizar el despliegue cada vez que **reDisplay**() es llamado.

Si cambias al **Checkbox** o componentes **searchFor**, sus oyentes simplemente llaman **reDisplay**(). reDisplay() crea un arreglo temporal **String** llamados **rs** (para el "conjunto de resultado"). El conjunto de resultado es cualquier copiado directamente de n si no esta la palabra **find**, o condicionalmente copiado de los **Strings** en n que contiene la palabra **find**. Finalmente, el strip **Checkbox** es interrogado para ver si el usuario quiere que los nombres fueran devastados (el defecto es **yes**). Si es así, **StripQualifiers**.strip() hace el trabajo; Si no, la lista es simplemente desplegada.

En **init**(), podrías pensar que abunda el trabajo ocupado involucrado en establecer el diseño. De hecho, cabe diseñar los componentes con menos trabajo, pero la ventaja de usar a **BorderLayouts** así es que le permite al usuario ajustar el tamaño de la ventana y hacer – en particular – al **TextArea** más grande, que quiere decir que puedes ajustar el tamaño para permitirte ver nombres más largos sin desplazar.

Podrías encontrarte con que conservarás esta herramienta corriendo mientras programas, ya que provee una de las mejores “primeras líneas de ataque” cuando estás tratando de resolver qué método llamar.

La teoría de complejidad

Este programa fue modificado de código originalmente creado por Larry O'Brien, y se basa en los programas “Boids” creado por Craig Reynolds en 1986 para demostrar un aspecto de teoría de complejidad llamada “emergencia.”

La meta aquí es producir una reproducción razonablemente parecida a la realidad de viajar en tropel o reuniendo en manada el comportamiento en animales estableciendo un conjunto pequeño de reglas simples para cada animal. Cada animal puede mirar la escena entera y todos los demás animales en la escena, pero reacciona sólo a un conjunto cercanos “flockmates.” El animal se mueve según tres comportamientos simples de la dirección:

1. Separación: Evita atestar flockmates locales.
2. Alineación: Sigue el encabezamiento común de flockmates locales.
3. Cohesión: Se Mueve hacia el centro del grupo de flockmates locales.

Los modelos más elaborados pueden incluir obstáculos y la habilidad para los

animales a predecir colisiones y evitarles a ellos, así es que los animales pueden fluir alrededor de objetos fijos en el ambiente. Además, los animales también podrían recibir una meta, lo cual puede causar que el rebaño siga un camino deseado. Para la simplicidad, la evitación de obstáculo y el buscar metas no son incluidos en el modelo presentado aquí.

La emergencia quiere decir que, a pesar de la naturaleza limitada de computadoras y la simplicidad de las reglas de la dirección, el resultado parece realista. Es decir, el comportamiento notablemente parecido a la realidad "emerge" de este modelo simple.

El programa se replantea como una aplicación/applet combinada:

```
//: FieldOBeasts.java
// Demonstration of complexity theory; simulates
// herding behavior in animals. Adapted from
// a program by Larry O'Brien lobrien@msn.com
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;

class Beast {
  int
    x, y,            // Screen position
    currentSpeed;    // Pixels per second
  float currentDirection;  // Radians
  Color color;       // Fill color
  FieldOBeasts field; // Where the Beast roams
  static final int GSIZE = 10; // Graphic size

  public Beast(FieldOBeasts f, int x, int y,
      float cD, int cS, Color c) {
    field = f;
    this.x = x;
    this.y = y;
    currentDirection = cD;
    currentSpeed = cS;
    color = c;
  }
  public void step() {
    // You move based on those within your sight:
    Vector seen = field.beastListInSector(this);
    // If you're not out in front
    if(seen.size() > 0) {
      // Gather data on those you see
      int totalSpeed = 0;
      float totalBearing = 0.0f;
      float distanceToNearest = 100000.0f;
      Beast nearestBeast =
        (Beast)seen.elementAt(0);
      Enumeration e = seen.elements();
      while(e.hasMoreElements()) {
        Beast aBeast = (Beast) e.nextElement();
```

```
        totalSpeed += aBeast.currentSpeed;
        float bearing =
          aBeast.bearingFromPointAlongAxis(
            x, y, currentDirection);
        totalBearing += bearing;
        float distanceToBeast =
          aBeast.distanceFromPoint(x, y);
        if(distanceToBeast < distanceToNearest) {
          nearestBeast = aBeast;
          distanceToNearest = distanceToBeast;
        }
      }
      // Rule 1: Match average speed of those
      // in the list:
      currentSpeed = totalSpeed / seen.size();
      // Rule 2: Move towards the perceived
      // center of gravity of the herd:
      currentDirection =
        totalBearing / seen.size();
      // Rule 3: Maintain a minimum distance
      // from those around you:
      if(distanceToNearest <=
         field.minimumDistance) {
        currentDirection =
          nearestBeast.currentDirection;
        currentSpeed = nearestBeast.currentSpeed;
        if(currentSpeed > field.maxSpeed) {
          currentSpeed = field.maxSpeed;
        }
      }
    }
    else {  // You are in front, so slow down
      currentSpeed =
        (int)(currentSpeed * field.decayRate);
    }
    // Make the beast move:
    x += (int)(Math.cos(currentDirection)
               * currentSpeed);
    y += (int)(Math.sin(currentDirection)
               * currentSpeed);
    x %= field.xExtent;
    y %= field.yExtent;
    if(x < 0)
      x += field.xExtent;
    if(y < 0)
      y += field.yExtent;
  }
  public float bearingFromPointAlongAxis (
      int originX, int originY, float axis) {
    // Returns bearing angle of the current Beast
    // in the world coordiante system
    try {
      double bearingInRadians =
```

```
          Math.atan(
            (this.y - originY) /
            (this.x - originX));
        // Inverse tan has two solutions, so you
        // have to correct for other quarters:
        if(x < originX) {
          if(y < originY) {
            bearingInRadians += - (float)Math.PI;
          }
          else {
            bearingInRadians =
              (float)Math.PI - bearingInRadians;
          }
        }
        // Just subtract the axis (in radians):
        return (float) (axis - bearingInRadians);
      } catch(ArithmeticException aE) {
        // Divide by 0 error possible on this
        if(x > originX) {
            return 0;
        }
        else
          return (float) Math.PI;
      }
    }
    public float distanceFromPoint(int x1, int y1){
      return (float) Math.sqrt(
        Math.pow(x1 - x, 2) +
        Math.pow(y1 - y, 2));
    }
    public Point position() {
      return new Point(x, y);
    }
    // Beasts know how to draw themselves:
    public void draw(Graphics g) {
      g.setColor(color);
      int directionInDegrees = (int)(
        (currentDirection * 360) / (2 * Math.PI));
      int startAngle = directionInDegrees -
        FieldOBeasts.halfFieldOfView;
      int endAngle = 90;
      g.fillArc(x, y, GSIZE, GSIZE,
        startAngle, endAngle);
    }
}

public class FieldOBeasts extends Applet
    implements Runnable {
  private Vector beasts;
  static float
    fieldOfView =
      (float) (Math.PI / 4), // In radians
    // Deceleration % per second:
```

```
  decayRate = 1.0f,
  minimumDistance = 10f; // In pixels
static int
  halfFieldOfView = (int)(
    (fieldOfView * 360) / (2 * Math.PI)),
  xExtent = 0,
  yExtent = 0,
  numBeasts = 50,
  maxSpeed = 20; // Pixels/second
boolean uniqueColors = true;
Thread thisThread;
int delay = 25;
public void init() {
  if (xExtent == 0 && yExtent == 0) {
    xExtent = Integer.parseInt(
      getParameter("xExtent"));
    yExtent = Integer.parseInt(
      getParameter("yExtent"));
  }
  beasts =
    makeBeastVector(numBeasts, uniqueColors);
  // Now start the beasts a-rovin':
  thisThread = new Thread(this);
  thisThread.start();
}
public void run() {
  while(true) {
    for(int i = 0; i < beasts.size(); i++){
      Beast b = (Beast) beasts.elementAt(i);
      b.step();
    }
    try {
      thisThread.sleep(delay);
    } catch(InterruptedException ex){}
    repaint(); // Otherwise it won't update
  }
}
Vector makeBeastVector(
    int quantity, boolean uniqueColors) {
  Vector newBeasts = new Vector();
  Random generator = new Random();
  // Used only if uniqueColors is on:
  double cubeRootOfBeastNumber =
    Math.pow((double)numBeasts, 1.0 / 3.0);
  float colorCubeStepSize =
    (float) (1.0 / cubeRootOfBeastNumber);
  float r = 0.0f;
  float g = 0.0f;
  float b = 0.0f;
  for(int i = 0; i < quantity; i++) {
    int x =
      (int) (generator.nextFloat() * xExtent);
    if(x > xExtent - Beast.GSIZE)
```

```
        x -= Beast.GSIZE;
      int y =
        (int) (generator.nextFloat() * yExtent);
      if(y > yExtent - Beast.GSIZE)
        y -= Beast.GSIZE;
      float direction = (float)(
        generator.nextFloat() * 2 * Math.PI);
      int speed = (int)(
        generator.nextFloat() * (float)maxSpeed);
      if(uniqueColors) {
        r += colorCubeStepSize;
        if(r > 1.0) {
          r -= 1.0f;
          g += colorCubeStepSize;
          if( g > 1.0) {
            g -= 1.0f;
            b += colorCubeStepSize;
            if(b > 1.0)
              b -= 1.0f;
          }
        }
      }
      newBeasts.addElement(
        new Beast(this, x, y, direction, speed,
          new Color(r,g,b)));
    }
    return newBeasts;
  }
  public Vector beastListInSector(Beast viewer) {
    Vector output = new Vector();
    Enumeration e = beasts.elements();
    Beast aBeast = (Beast)beasts.elementAt(0);
    int counter = 0;
    while(e.hasMoreElements()) {
      aBeast = (Beast) e.nextElement();
      if(aBeast != viewer) {
        Point p = aBeast.position();
        Point v = viewer.position();
        float bearing =
          aBeast.bearingFromPointAlongAxis(
            v.x, v.y, viewer.currentDirection);
        if(Math.abs(bearing) < fieldOfView / 2)
         output.addElement(aBeast);
      }
    }
    return output;
  }
  public void paint(Graphics g)  {
    Enumeration e = beasts.elements();
    while(e.hasMoreElements()) {
      ((Beast)e.nextElement()).draw(g);
    }
  }
```

```
  public static void main(String[] args)   {
    FieldOBeasts field = new FieldOBeasts();
    field.xExtent = 640;
    field.yExtent = 480;
    Frame frame = new Frame("Field 'O Beasts");
    // Optionally use a command-line argument
    // for the sleep time:
    if(args.length >= 1)
      field.delay = Integer.parseInt(args[0]);
    frame.addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      });
    frame.add(field, BorderLayout.CENTER);
    frame.setSize(640,480);
    field.init();
    field.start();
    frame.setVisible(true);
  }
} ///:~
```

Aunque ésta no es una reproducción perfecta del comportamiento en “Boids” de Craig Reynold el ejemplo, exhibe sus propias características fascinantes, el cual puedes modificar ajustando los números. Puedes encontrar más acerca del modelado de comportamiento de viajar en tropel y puedes ver una versión espectacular 3-D de Boids en la página de Craig Reynold *http://www.hmt.com/cwr/boids.html*.

Para correr éste programa como un applet, pones la siguiente etiqueta de applet en un archivo de HTML:

```
<applet
code=FieldOBeasts
width=640
height=480>
<param name=xExtent value = "640">
<param name=yExtent value = "480">
</applet>
```

Resumen

Este capítulo demuestra que una cierta cantidad de las cosas más sofisticadas se pueden hacer con Java. También hace el punto que mientras Java ciertamente debe tener sus límites, esos límites son primordialmente relegados para el desempeño. (Cuando a los programas de procesos de texto les fue escrito, por ejemplo, las versiones C++ fueron mucho más rápidas – éste podría deberse en parte a una implementación ineficiente de la biblioteca IO, lo cual debería cambiar con el tiempo.) Los límites de Java no parecen estar en el

área de expresividad. No sólo lo hace parece posible expresar justamente acerca de todo lo que puedes imaginar, pero Java parece orientada a facilitar esa expresión de escribir y leer. Por eso no entras corriendo a la pared de complejidad que a menudo cursa con lenguajes que son más trivial por usar que Java (al menos parecen de ese modo, al principio). Y con la librería JFC/Swing 1.2 de Java, emparejan la expresividad y la facilidad de uso del AWT mejora dramáticamente.

Ejercicios

1. (**Avanzado**) Reescribe **FieldOBeasts.java** a fin de que su estado puede ser persistente. Implementa botones para permitirte salvar y recordar archivos diferentes del estado y continuar corriéndolos donde salió completamente. Usa **CADState.java** desde el Capítulo 12 como un ejemplo de cómo hacer esto.
2. (**Proyecto de período**) Tomando a **FieldOBeasts.java** como un punto de partida, construye un sistema de simulación de tráfico del automóvil.
3. (**Proyecto de período**) Usando a **ClassScanner.java** como un punto de partida, construye una herramienta que señala métodos y campos que están definidos pero nunca usados.
4. (**Proyecto de período**) Usando a JDBC, construye un programa de la gerencia de contacto usando una base de datos de archivo plano conteniendo nombres, direcciones, números de teléfono, direcciones de correo electrónico, etc. Deberías poder fácilmente añadirle los nombres nuevos a la base de datos. Al introducir el nombre para ser buscado, usa terminación automática de nombre como se muestra en **VLookup.java** en el Capítulo 16.

19: Análisis y Diseño

El paradigma de la orientación a objetos es una filosofía de programación nueva y diferente.

Al principio mucha gente tiene problemas para saber cómo aproximarse a los proyectos de POO. Ahora que ya entiendes el concepto de objeto, y según vayas aprendiendo a pensar más y más en un estilo orientado-a-objetos, puedes empezar a crear "buenos" diseños que aprovechen todos los beneficios que la POO puede ofrecer. Este capítulo introduce las ideas de análisis, diseño y algunas formas de aproximación a los problemas que aparecen al desarrollar programas orientados-a-objetos buenos en un período de tiempo razonable.

Metodología

Una *metodología* (llamada a veces simplemente *método*) es un conjunto de procesos y heurísticas que se utilizan para romper la complejidad de un problema de programación. Muchas metodologías de POO han sido formuladas desde el amanecer de la programación

La metodología, especialmente en el caso de la POO, es un campo de múltiple experimentación, así que es importante entender que problema está intentando resolver dicha metodología antes de considerar su adopción. Esto es particularmente cierto en Java, donde el lenguaje de programación pretende reducir la complejidad (comparado con C) que existe al escribir un programa. De hecho, esto puede hacer menos necesario la adopción de metodologías que se hacen cada vez más complejas. En su lugar, puede ser suficiente recurrir a metodologías simples para un rango mucho mayor de problemas del que podrías gestionar utilizando metodologías simples con lenguajes procedurales.

También es importante darse cuenta de que el término "metodología" a menudo es demasiado pretencioso y promete demasiado. Sea lo que sea que haces ahora cuando diseñas y escribes un programa es una metodología. Puede que sea tu propia metodología, y puede que no seas consciente de que la estás utilizando, pero es un proceso que sigues mientras creas. Si es un proceso eficaz, puede que sólo sea necesario un pequeño ajuste para que sirva para Java. Si no estás satisfecho con tu productividad y la forma en la que se desarrollan tus programas puede que quieras considerar la adopción de una metodología formal, o tomar elementos de las varias metodologías formales existentes.

Durante el proceso de desarrollo, la cuestión más importante es la siguiente: No perderse. Es fácil. La mayoría de las metodologías de análisis y diseño pretenden resolver el problema más complejo. Recuerda que la mayoría de los proyectos no encajan en esta categoría, así que normalmente puedes obtener un buen análisis y diseño con un subconjunto relativamente pequeño de las recomendaciones de una metodología. **[1]** Pero por lo general, algún tipo de

proceso, no importa cuan pequeño o limitado, te guiará de una forma mucho mejor que limitarse codificar desde el principio.

[1] Un ejemplo excelente de esto es el libro de Martin Fowler *UML Distilled*, 2nd edition (Addison-Wesley 2000), que reduce el a veces inabarcable proceso UML a un subconjunto manejable.

También es fácil atascarse, caer en la "paralización por la investigación," en la que te sientes como si no pudieras avanzar porque no has asegurado cada pequeño detalle de la fase actual. Recuerda, no importa cuanto análisis hagas, hay elementos del sistema que no se revelarán hasta la fase de diseño, y más cosas aún que no se revelarán hasta que has empezado a codificar, o incluso no lo harán hasta que el programa esté en funcionamiento. Debido a esto, es crucial moverse rápidamente por el análisis y el diseño, e implantar una prueba de concepto del sistema propuesto.

Este punto es digno de mención. Debido a la experiencia que tenemos con los lenguajes procedurales, es encomiable que un equipo quiera proceder con cautela y entender cada detalle por pequeño que sea antes de pasar al diseño y la construcción del sistema. Sin duda, cuando se trata de crear un Sistema de Gestión de Bases de Datos (*SGBD*), merece la pena entender completamente las necesidades del cliente. Pero un SGBD es un tipo de problema que está muy bien planteado y es bien comprendido; en muchos programas semejantes, la estructura de la base de datos *es* el problema a sortear. El tipo de problemas que se discuten en este capítulo es de la variedad "comodín" (expresión mía), en la que la solución no es una simple reformulación de una solución bien conocida, sino que implica a uno o más "factores comodín"—elementos para los que previamente no existe una solución bien comprendida, para los que es necesario trabajo de investigación.**[2]** El intento de analizar completamente un problema comodín antes de pasar al diseño y la construcción resultan en la paralización por investigación porque no se tiene suficiente información para resolver este tipo de problemas durante la fase de análisis. Resolver semejante problema necesita la iteración de todo el ciclo, y requiere de un comportamiento de aceptación de riesgos (que tiene sentido, porque estás intentando hacer algo nuevo y los beneficios potenciales son mayores). Podría parecer que aceptar el riesgo implica "precipitarse" en una construcción preliminar, pero por contra, éste hecho puede reducir el riesgo que se da en un proyecto comodín porque de esta manera se puede averiguar en una etapa temprana si una aproximación concreta a un problema es viable. El desarrollo de productos es gestión de riesgos.

[2] La regla de estimación a groso modo que utilizo para dichos proyectos es: Si hay más de un comodín, ni siquiera trates de planificar cuánto se va a tardar o cuánto va a costar hasta que hayas creado un prototipo. Hay demasiados grados de libertad.

A menudo se propone "construir uno para tirar." Con la POO podrías seguir tirando *parte* del mismo, pero debido a que el código está encapsulado en

clases, durante la primera iteración inevitablemente producirás algunos diseños de clase que son útiles y desarrollarás algunas ideas valiosas sobre el diseño del sistema que no es necesario tirar. Así, un rápido primer repaso del problema no sólo produce información crítica para los siguientes pasos de análisis, diseño y construcción, también crea una base de código.

Dicho esto, si miras una metodología que contiene mucho detalle y sugiere muchos pasos y documentos, sigue siendo difícil saber cuándo parar. Mantén presente lo que intentas descubrir:

1. ¿Qué objetos hay? (¿Cómo divides tu proyecto en sus partes componentes?)
2. ¿Cuáles son sus inferfaces? (¿Qué mensajes necesitas enviar a cada objeto?)

Aunque no obtengas más que los objetos y sus interfaces ya puedes escribir un programa. Puede que, por diversas razones, necesites más descripciones y documentos, pero no podrás proceder con menos.

El proceso puede realizarse en cinco fases, y una Fase 0 que es simplemente el compromiso inicial para utilizar algún tipo de estructura.

Fase 0: planificar

Primero debes decidir qué pasos vas a tener que dar en tu proceso. Parece sencillo (de hecho *todo* esto parece sencillo), y sin embargo a menudo la gente no toma esta decisión antes de empezar a codificar. Si tu plan es “dejémonos de preliminares y empecemos a codificar,” bien. (A veces, cuando tienes un problema que comprendes bien es lo adecuado.) Al menos admite que este es el plan.

También podrías decidir en esta fase que es necesario tener alguna estructura de proceso, pero no las nueve yardas al completo. Es comprensible que algunos programadores les gusta trabajar en “modo vacacional,” en el que no se impone ninguna estructura en el proceso de desarrollo de su trabajo; “terminaré cuando todo haya terminado.” Esto puede ser atractivo durante un tiempo, pero he llegado a la conclusión de que unos cuantos hitos en el camino ayudan a focalizar y motivar tus esfuerzos sobre esos hitos en lugar de atascarte en la única meta de “terminar el proyecto.” Además, esto divide el proyecto en trozos más manejables y lo hace parecer menos amenazador (sin contar con que los hitos ofrecen más oportunidades para celebrar el progreso).

Cuando comencé a estudiar estructura narrativa (por si quería escribir algún día una novela) al principio era reluctante a la idea de estructura, sentía que podía escribir mejor cuando me limitaba a dejar la narración fluir en la página. Pero luego me di cuenta de que cuando escribo sobre ordenadores la estructura me es lo suficientemente clara como para no tener que pensar en ella mucho. Sigo estructurando mi trabajo, aunque sólo de forma semiconsciente en mi cabeza. Incluso si piensas que tu plan es simplemente

empezar a codificar, todavía tienes que pasar por las fases siguientes haciéndote ciertas preguntas y contestando a las mismas.

La declaración de la misión

Cualquier sistema que construyas, no importa lo complicado que sea, tiene un propósito fundamental—el negocio en el que se sitúa, la necesidad básica que satisface. Si eres capaz de mirar más allá de la interfaz de usuario, los detalles específicos del hardware o del sistema, los algoritmos de codificación y los problemas de eficiencia, con el tiempo encontrarás el corazón de su ser—simple y directo. Es como lo que llaman *concepto principal* de una película de Hollywood, puedes describirlo en una o dos frases. Esta pura descripción es el punto de partida.

El concepto principal es muy importante porque fija la orientación de tu proyecto; es una declaración de la misión. No conseguirás que sea completamente acertada desde el principio (puede que estés en una fase posterior del proyecto antes de que esté completamente clara), pero sigue intentándolo hasta que parezca adecuada. Por ejemplo, en un sistema de control de tráfico aéreo puedes comenzar con un concepto principal enfocado en el sistema que estás construyendo: “El programa de la torre supervisa la posición de las aeronaves.” Pero considera lo que ocurre cuando tu sistema es el de un aeródromo muy pequeño; quizás sólo hay un controlador humano, o ninguno. Un modo más útil no tendrá en cuenta la solución que estás definiendo sino que describirá el problema: “Las aeronaves llegan, descargan, se abastecen y recargan, luego se van.”

Fase 1: ¿Qué estamos haciendo?

En la generación anterior de diseño de programas (llamada *diseño procedural*), esto se denominaba “creación del *análisis de requisitos* y *especificación del sistema*.” Por supuesto, estos eras lugares en los que era fácil perderse; documentos con nombres intimidatorios que podrían llegar a ser grandes proyectos en si mismos. No obstante, su intención era buena. El análisis de requisitos dice “haz una lista de las líneas guía que seguiremos para saber cuando hemos terminado el trabajo y el cliente está satisfecho.” **[3]** La especificación del sistema dice “Aquí tienes una descripción de lo *que* el programa debe hacer (no del *cómo*) para satisfacer los requisitos.” El análisis de requisitos es un verdadero contrato entre tú y el cliente (incluso si el cliente está en tu propia empresa, o si se trata de otro objeto o sistema). La especificación de sistemas es una exploración de alto nivel del problema y en cierto sentido se trata de descubrir si se puede hacer y cuánto tiempo llevará. Dado que ambas cosas requieren de consenso entre las personas (y debido a que éstas habitualmente cambian con el tiempo), creo que lo mejor es mantenerlas tan simples como sea posible—idealmente, reducidas a listas y diagramas básicos—para ahorrar tiempo (esto está en línea con la Programación Extrema, que aboga por una documentación muy reducida, aunque sólo vale para proyectos pequeños a medianos). Podrías tener otras

restricciones que requiriesen una explicación en más detalle en documentos más extensos, pero si mantienes el documento inicial pequeño y conciso, podrás crearlo en unas pocas sesiones de trabajo en grupo utilizando técnicas de tormenta de ideas y un líder que de forma dinámica realiza la descripción. Estas técnicas no sólo piden a todo el mundo ofrezca ideas, también favorece una involucración y un acuerdo iniciales por parte de todos los miembros del grupo. Y quizás lo más importante, puede lanzar un proyecto con mucho entusiasmo.

[3] Un recurso excelente para el análisis de requisitos es el libre de Gaue y Weinberg *Exploring Requirements: Quality Before Design* (Dorset House 1989).

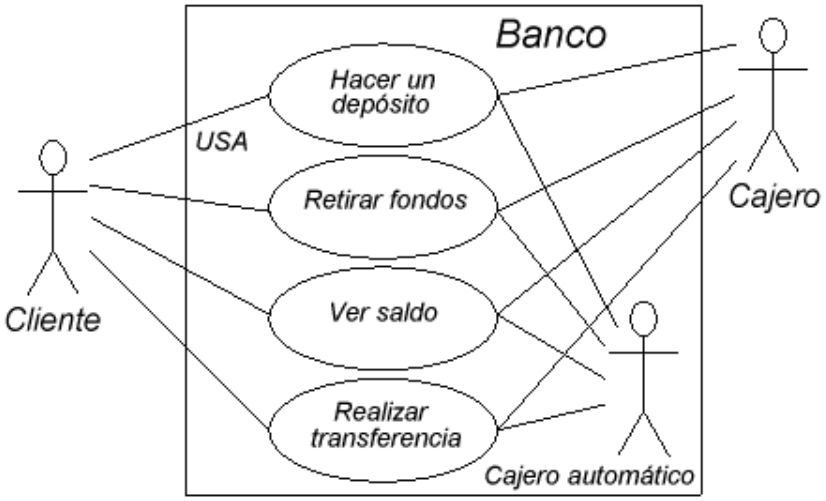
Es necesario mantener el foco en el corazón de lo que se intenta conseguir en esta fase: Determinar lo que se supone que el sistema debe hacer. La herramienta más valiosa para conseguirlo es una colección de lo que se denominan “casos de uso,” o en Programación Extrema, “historias del usuario.” Los casos de uso identifican las características clave del sistema que revelarán algunas de las clases fundamentales que se utilizarán. Se trata esencialmente de respuestas decriptivas a preguntas como: **[4]**

- “¿Quién utilizará el sistema?”
- “¿Qué pueden esos actores hacer son el sistema?”
- “¿Cómo hace *este* actor *eso* con este sistema?”
- “¿De qué otra forma podría funcionar si alguien estuviese haciendo esto otro, o si el mismo actor tuviese un objetivo diferente?” (para descubrir variantes)
- “¿Qué problemas podrían ocurrir mientras se está haciendo esto con el sistema?” (para descubrir excepciones)

[4] Gracias a la ayuda de James H Jarrett.

Si, por ejemplo, estás diseñando un cajero automático, el caso de uso para un aspecto concreto de la funcionalidad del sistema es tratar de describir lo que el cajero automático hace en cada situación posible. Cada una de estas “situaciones” se denomina *escenarios*, y un caso de uso puede ser considerado como una colección de escenarios. Puedes pensar en un escenario como una pregunta que comienza de la siguiente manera: “¿Qué hace el sistema si.....?” Por ejemplo, “¿Qué hace el cajero automático si un cliente acaba de depositar un cheque en las últimas 24 horas y no hay saldo suficiente en la cuenta sin que el cheque haya sido confirmado para realizar una retirada de fondos que el cliente ha solicitado?”

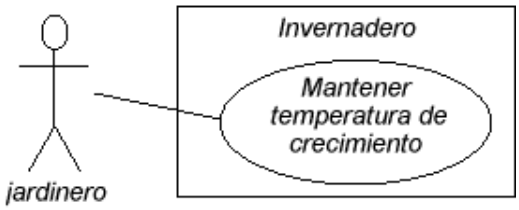
Los diagramas de caso de uso son intencionadamente sencillos para evitar que te enfangues en los detalles de construcción del sistema de forma prematura:



Cada persona palo representa un “actor,” que es típicamente un humano o algún otro tipo de agente de comportamiento libre. (Estos pueden ser incluso otros sistemas informáticos, como es el caso del “ATM.”) La caja representa los límites de tu sistema. Las elipses representan los casos de uso, que son descripciones de trabajo de valor añadido que se puede realizar con el sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

No importa cómo está el sistema realmente construido, siempre que se comporte con el usuario de la manera determinada.

Un caso de uso no tiene que ser necesariamente horriblemente complejo, aunque el sistema subyacente lo sea. La intención es únicamente mostrar el sistema como aparece ante el usuario. Por ejemplo:



Los casos de uso producen las especificaciones de requisitos al determinar todas las interacciones que el usuario puede tener con el sistema. Intenta descubrir un conjunto completo de casos de uso para tu sistema, y una vez que lo hayas hecho tendrás el núcleo de lo que se supone que el sistema debe hacer. Lo bueno de enfocarse en los casos de uso es que siempre te traen de vuelta a lo esencial y evitan que derives hacia asuntos que no son críticos para terminar el trabajo. Esto es, si tienes un conjunto de casos de uso completo, puedes describir tu sistema y pasar a la siguiente fase. Probablemente no conseguirás que todo se perfile perfectamente en el primer intento, pero eso está bien. Todos los detalles aparecerán a su debido momento, si pretendes obtener una especificación perfecta del sistema en este punto, te atascarás.

Si te atascas de verdad, puedes darle un empujón a esta fase utilizando una herramienta de aproximación de alto nivel: Describe el sistema en unos pocos párrafos y busca nombres y verbos. Los nombres pueden sugerir actores, contextos de los casos de uso (p.e., "grupos de interesados", o artefactos que se manipulan en el caso de uso. Los verbos pueden sugerir interacciones entre los actores y el caso de uso, y especificar pasos dentro del caso de uso. También descubrirás que los nombres y los verbos producen objetos y mensajes durante la fase de diseño (y ten en cuenta que los casos de uso describen interacciones entre subsistemas, de forma que la técnica de "los nombres y los verbos" sólo puede ser considerada como una herramienta para las tormentas de ideas porque no generan casos de uso). **[5]**

[5] Se puede encontrar más información sobre los casos de uso en el libro de Rosenberg *Use Case Driven Object Modeling with UML* (Addison-Wesley 1999). Y una buena visión preliminar de las historias de usuario se puede leer en el libro de Beck y Fowler *Planning Extreme Programming* (Addison-Wesley 2001).

La frontera entre un caso de uso y un actor puede indicar la existencia de un interfaz de usuario, pero no define dicha interfaz. Si quieres profundizar en el proceso de definición y creación de interfaces de usuario, mira en el libro de Larry Constantine y Lucy Lockwood *Software for Use*, (Addison-Wesley Longman, 1999) o busca en la web en *www.ForUse.com*.

Aunque se trate de magia negra, en este punto es importante recurrir algún tipo de técnica de planificación de tareas. Ya tienes una percepción de lo que vas a construir, así que probablemente podrás hacerte una idea de cuánto tiempo te llevará hacerlo. Muchos factores entran aquí a jugar un papel. Si estimas un plan largo, puede que la empresa decida no construirlo (y utilizar así sus recursos en algo más razonable—esto es algo *bueno*). También puede que un directivo haya decidido de antemano cuánto debería llevar el proyecto y trate de influir en tu estimación. Pero lo mejor es tener un calendario honesto desde el principio y tratar con las decisiones difíciles al principio. Ha habido muchos intentos de conseguir técnicas de planificación temporal ajustadas (tantas como ha técnicas de predicción en el mercado de valores), pero probablemente la mejor aproximación es confiar en tu experiencia e intuición. Utiliza tus instintos para estimar la duración del proyecto y luego dobla la cantidad y añade un 10 por ciento. Tus instintos probablemente sean correctos; puede que *consigas* algo que funcione en ese tiempo. "Doblar" la cantidad convertirá el producto en algo decente, y el 10 por ciento permitirá el pulido y los detalles finales. **[6]** Por mucho que quieras explicarlo, y a pesar de las quejas y manipulaciones que se dan cuando das un calendario como ese, simplemente parece que la cosa funciona así. **[7]**

[6] Últimamente, mi regla personal para éste tema ha cambiado. Doblar y añadir un 10 por ciento te da una estimación razonablemente ajustada (asumiendo que no haya demasiados factores comodín), pero todavía tendrás que trabajar de forma muy diligente para terminar en ese plazo. Si quieres tener tiempo para hacerlo realmente elegante y para disfrutar durante el proceso, creo que el factor correcto es más del orden de tres o cuatro veces tu

instinto inicial. Si quieres ver un estudio sobre el efecto de los calendarios en la productividad y una refutación de la "Ley de Parkinson" busca en el libro de DeMarco y Lister *PeopleWare* (Dorset House 1999).

[7] *Planning Extreme Programming* (ibid.) tiene algunas percepciones valiosas sobre el tema de la planificación y la estimación temporal.

Fase 2: ¿Cómo lo vamos a construir?

En esta fase debes obtener un diseño que describa qué aspecto tienen las clases y la forma en la que interactúan. Una técnica excelente para determinar las clases e interacciones existentes es la de las tarjetas *Clase-Responsabilidad-Colaboración* (CRC). Parte del valor de esta herramienta es que no tiene asociada una tecnología espacial: Comienzas con una pila de tarjetas 3 x 5 en blanco, y escribes sobre ellas. Cada tarjeta representa una única clase, y en la tarjeta escribes:

1. El nombre de la clase. Es importante de que este nombre captura la esencia de lo que la clase hace, para que tenga sentido a primera vista.
2. Las "responsabilidades" de la clase: que debería hacer. Esto se puede resumir normalmente mediante una lista de los nombres de los métodos (dado que dichos nombres deberían ser descriptivos en un buen diseño), pero no excluye la utilización de otras anotaciones. Si necesitas sembrar el proceso, estudia el problema desde la perspectiva de un programador perezoso: ¿Qué objetos te gustaría que mágicamente aparecieran para resolver tu problema?
3. Las "colaboraciones" de la clase: ¿Con qué otras clases interactúa? "Interactuar" es un término intencionadamente ambiguo; podría indicar una agregación o simplemente la existencia de algún otro objeto que realizase los servicios para un objeto de esa clase. Las colaboraciones deberían también considerar la audiencia de la clase actual. Por ejemplo, si creas una clase **FuegoDeArtificio**, ¿quién va a observarlo, un **Pirotécnico** o un **Expectador**? El primero estará interesado en conocer los componentes químicos de los que está compuesto, el segundo responderá a los colores y la forma que toma cuando explota.

Puede que pienses que las tarjetas deberían ser más grandes debido a la información que te gustaría tener en ellas. Sin embargo, se han dejado así de pequeñas de forma intencionada, no sólo para mantener pequeñas tus clases, sino también para evitar que entres en demasiado detalle demasiado pronto. Si no puedes encajar todo lo que necesitas sabes sobre una clase en un pequeña tarjeta, entonces la clase es demasiado compleja (o estás entrando en demasiado detalle o deberías crear más de una clase). La clase ideal debería ser entendida en un vistazo. La idea de las tarjetas CRC es ayudarte a obtener un primer corte del diseño de forma que puedas ver el dibujo completo y luego refinar tu diseño.

Uno de los mayores beneficios de las tarjetas CRC está en la comunicación. Ésta se realiza mejor en tiempo real, en un grupo, sin ordenadores. Cada persona se hace responsable de varias clases (que al principio no tienen nombres ni ninguna otra información). Realizas una simulación en vivo representando un escenario cada vez, diciendo que mensajes se envían a los diferentes objetos para resolver cada escenario. Según se avanza en este proceso, vas descubriendo las clases que necesitas y sus responsabilidades y colaboraciones, y vas rellenando las tarjetas. Cuando has cubierto todos los casos de uso, deberías tener un primer corte bastante completo de tu diseño.

Antes de empezar a utilizar tarjetas CRC, las experiencias que había tenido con mayor éxito en obtención de un diseño inicial implicaban ponerme delante de un equipo—que no había realizado antes ningún proyecto de POO—y dibujado objetos en una pizarra. Comentábamos cómo se suponía que los objetos debían comunicarse entre sí, borrábamos algunos y los reemplazábamos con otros. En la práctica estaba gestionando todas las "tarjetas CRC" en la pizarra. El equipo (cuyos miembros sabían lo que se suponía que el producto debía hacer) en realidad creaban el diseño, eran sus "propietarios", en lugar de dárselo hecho. Todo lo que yo me limitaba a hacer era guiar el proceso haciendo las preguntas adecuadas, extrayendo las premisas y recibiendo la realimentación del equipo para modificar dichas premisas. Lo verdaderamente bello del proceso fue que el equipo no aprendió como conseguir un diseño orientado a objetos mediante la revisión de ejemplos abstractos, sino trabajando en el diseño en el que más interés tenían en ese momento: el suyo.

Una vez que has obtenido un conjunto de tarjetas CRC, puede que quieras crear una descripción más formal de tu diseño utilizando UML. **[8]** No necesitas utilizar UML, pero puede ser útil, especialmente si quieres colocar un diagrama en la pared para que todo el mundo medite sobre él, lo que es una buena idea (hay una plétora de herramientas de diagramación para UML disponibles). Una alternativa a UML es la descripción textual de los objetos y sus interfaces o, dependiendo de tu lenguaje de programación, el propio código. **[9]**

[8] Para los principiantes recomiendo el anteriormente mencionado *UML Distilled*, 2nd edición.

[9] Python (www.Python.org) se utiliza a menudo como "pseudocódigo ejecutable."

UML también ofrece una notación de diagramas adicional que permite describir el modelo dinámico de tu sistema. Esto es útil en aquellas situaciones en las que la transición entre estado de un sistema o subsistema son lo suficientemente importantes como para necesitar sus propios diagramas (como por ejemplo en un sistema de control). Puede que también necesites describir las estructuras de datos para los sistemas o subsistemas en los que los datos son el factor dominante (como los de una base de datos).

Sabrás que has terminado con la Fase 2 cuando hayas descritos los objetos y sus interfaces. Bueno, la mayoría de ellos—suele haber unos cuantos que se deslizan por las grietas y no se dan a conocer hasta la Fase 3. Pero eso está bien. Lo que es importante es que con el tiempo descubrirás todos tus objetos. Es agradable descubrirlos todos en una etapa temprana del proceso, pero la POO ofrece suficiente estructura como para que no sea tan negativo si los descubres más tarde. De hecho, el diseño de un objeto tiende a ocurrir en cinco etapas, a lo largo del proceso de desarrollo del programa.

Las cinco etapas del diseño de objetos

El ciclo de vida del diseño de un objeto no está limitado al período en el que escribes el programa. En vez de esto, el diseño de un objeto aparece durante una secuencia de etapas. Es útil tener esta perspectiva porque dejas de esperar la perfección desde el principio; en lugar de ello, te das cuenta de que comprender lo que hace un objeto y cómo debería evolucionar con el tiempo. Esta visión también se aplica al diseño de diversos tipos de programa; el patrón para un tipo particular de programa emerge de la lucha continua con ese problema (cuya crónica es relatada en el libro *Pensando sobre Patrones (con Java)* disponible en *www.BruceEckel.com*). Los objetos, también, tienen sus patrones y emergen mediante la comprensión, la utilización y la reutilización.

1. Descubrimiento de los objetos. Esta etapa ocurre durante el análisis inicial de un programa. Se puede descubrir a los objetos mirando los factores externos y los límites, la duplicación de elementos en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si ya tienes un conjunto de bibliotecas de clases. Los puntos en común entre las clases que sugieren las clases básicas y la herencia pueden aparecer desde el principio, o más avanzado el proceso de diseño.

2. Ensamblado de objetos. Según vas construyendo un objeto descubrirás la necesidad de nuevos miembros que no aparecieron durante la fase de descubrimiento. Las necesidades internas del objeto pueden hacer necesarias otras clases que den soporte a dicho objeto.

3. Construcción del sistema. Una vez más, puede que en esta fase aparezcan más requisitos del objeto. Según vas aprendiendo, tus objetos evolucionan. La necesidad de comunicación e interconexión con otros objetos del sistema puede hacer cambiar las tus clases o hacer necesarias nuevas clases. Por ejemplo, puede que descubras la necesidad de clases de facilidades o apoyo, tales como una lista enlazada, que contienen poca o ninguna información y sirven sólo para dar servicio a la funcionalidad de otras clases.

4. Extensión del sistema. Según vas añadiendo nuevas características al sistema puede que descubras que tu diseño anterior no facilita una extensión fácil del sistema. Con esta nueva información, puedes reestructurar partes del sistema, posiblemente añadiendo nuevas clases o nuevas jerarquías de clase.

Este es también un buen momento para considerar la *retirada* de funcionalidades del proyecto.

5. Reutilización de objetos. Esta es la verdadera prueba de fuego de una clase. Si alguien intenta reutilizar la clase en una situación completamente nueva, probablemente encontrará defectos. Según la vayas adaptando a más programas nuevos, los principios generales de la clase se harán más claros, y así será hasta que tengas un tipo verdaderamente reutilizable. Sin embargo, no esperes que la mayoría de los objetos del sistema sean reutilizables—es perfectamente normal que el grueso de tus objetos sean específicos del sistema. Los tipos reutilizables tienden a ser menos habituales y deben resolver problemas más generales si quieren ser reutilizables.

Guías para el desarrollo de objetos

Estas etapas sugieren algunas líneas de guía cuando se trata de pensar en las clases que estás desarrollando:

1. Crea una clase para un problema específico y luego deja que la clase madure y se desarrolle para solucionar otros problemas.
2. Recuerda que descubrir las clases que necesitas (y sus interfaces) supone la mayor parte del diseño del sistema. Si ya tenías dichas clases, éste debería ser un proyecto fácil.
3. No te esfuerces por conocerlo todo desde el principio. Aprende según vayas avanzando. De todos modos será así.
4. Empieza a programar. Consigue algo que funcione de forma que puedas aprobar o rechazar tu diseño. No temas terminar con un código espagueti de tipo procedural—las clases dividen el problema y ayudan a controlar la anarquía y la entropía. Las clases problemáticas no afectarán a las buenas.
5. Mantén la simplicidad. Los objetos pequeños y limpios con una utilidad obvia son mejores que las interfaces grandes y complicadas. Cuando llegue el momento de decidir, utiliza una aproximación del tipo Navaja de Ocán **[10]**: Considera las alternativas y selecciona la que sea más sencilla, porque las clases simples son casi siempre las mejores. Comienza haciéndolo pequeño y sencillo, y podrás ampliar el interfaz de la clase cuando lo entiendas mejor. Es fácil añadir métodos, pero según pasa el tiempo, es difícil eliminar métodos de una clase.

[10] “Lo que puede hacerse con menos... en vano se hace con más... la mente no debería multiplicar las cosas sin necesidad.” Guillermo de Ockham, 1290-1349.

Fase 3: Construir el núcleo

Esta fase supone la conversión inicial del diseño aproximado en un cuerpo de código compilable y ejecutable que puede ser probado y, especialmente, te servirá para aprobar o refutar tu arquitectura. Este no es un proceso de un

sólo paso, sino el comienzo de una serie de pasos que construirán el sistema de forma iterativa, como verás en la Fase 4.

Tu meta es encontrar el núcleo de la arquitectura de tu sistema, el núcleo que necesita ser construido para generar un sistema que funcione, sin importar lo incompleto que sea ese sistema en su primera pasada. Estás creando una infraestructura sobre la que puedes construir en iteraciones posteriores. Además estás realizando la primera de las muchas integraciones y pruebas que harás, y ofreciendo a los accionistas una idea de cómo va a ser su sistema y cómo va progresando. De manera ideal revelarás algunos de los riesgos más críticos. Probablemente descubrirás los cambios y mejoras que puedes hacerle a la arquitectura original—cosas que no habrías descubierto sin construir el sistema.

Parte de la construcción del sistema es el contraste con la realidad que se obtiene al comprobar con el análisis de requisitos y especificación del sistema (sea la que sea la forma que tenga). Asegúrate de que las pruebas verifican los requisitos y los casos de uso. Cuando el núcleo del sistema sea estable, estás listo para progresar y añadir más funcionalidad.

Fase 4: Iterar para cada caso de uso

Una vez que la infraestructura básica está funcionando, cada conjunto de características que añadas es en si misma un pequeño proyecto. Añades un conjunto de características durante cada *iteración*, un período de tiempo de desarrollo razonablemente corto.

¿Cómo de larga es una iteración? De forma ideal, cada iteración dura entre una a tres semanas (esta duración puede variar dependiendo del lenguaje de programación). Al final de ese período, tienes un sistema integrado y probado con más funcionalidad de la que tenía antes. Pero lo que es particularmente interesante es la base de la iteración: un único caso de uso. Cada caso de uso es un paquete de funcionalidad relacionada que incluyes en el sistema de una vez, durante el curso de una iteración. Esto, no sólo te da una mejor idea del ámbito que el caso de uso debería tener, sino que también ofrece mayor validación a la idea del caso de uso, puesto que el concepto no lo descartamos después del análisis y el diseño, es una unidad fundamental de desarrollo a lo largo de todo el proceso de construcción del software.

Terminas las iteraciones cuando alcanzas la funcionalidad objetivo o se alcanza una fecha límite impuesta desde el exterior y el cliente se considera satisfecho con la versión actual. (Recuerda que el software es un negocio de suscripción.) Dado que el proceso es iterativo, tienes muchas oportunidades de entregar un producto en lugar de tener un único punto final; los proyectos de fuente abierta funcionan exclusivamente en un entorno iterativo, de mucha realimentación y es eso precisamente lo que los convierte en un éxito.

Un proceso de desarrollo iterativo es valioso por muchas razones. Puedes descubrir y resolver los riesgos críticos antes, los clientes tienen muchas

oportunidades de cambiar de idea, la satisfacción del programador es mayor, y el proyecto puede dirigirse con mayor precisión. Pero hay un beneficio adicional importante y es la realimentación que llega a los accionistas, los cuales pueden ver cómo van las cosas por el estado actual del producto. Esto puede reducir o eliminar la necesidad de las embotadoras reuniones de estado e incrementar la confianza y el apoyo de los accionistas.

Fase 5: Evolución

Este es el punto del ciclo de desarrollo que tradicionalmente se denomina "mantenimiento," un término globalizador que puede significar cualquier cosa desde "hacer que funcione realmente de la forma que se suponía debía hacerlo desde el principio" a "añadir características que el cliente olvidó mencionar" a la más tradicional "arreglar los errores que aparezcan" y "añadir nuevas características según vaya siendo necesario." Se ha utilizado tantas veces el término "mantenimiento" de forma equivocada que lo que conlleva es cierto halo de calidad decepcionante, en parte porque sugiere que tu objetivo era construir un programa perfecto y que todo tenías que hacer en adelante era cambiar partes del mismo, engrasarlo y evitar que se oxidara. Quizás existe una palabra mejor para describir lo que realmente ocurre.

Yo utilizaré la palabra *evolución.* **[11]** Esto es, "No lo vas a hacer bien la primera vez, así que déjate a ti mismo el espacio necesario para aprender y volver atrás para hacer cambios." Puede que necesites hacer muchos cambios según vayas aprendiendo y entendiendo el problema mejor. La elegancia que obtendrás si evolucionas hasta conseguirlo compensará tanto a corto como a largo plazo. La evolución es la forma en la que tu programa pasa de ser bueno a ser mejor, y es la forma en la que las cuestiones que no comprendiste bien la primera vez se clarifican. También es la forma en las que tus clases pueden pasar de ser utilizadas en un único proyecto a ser recursos reutilizables.

[11] En el libro de Martin Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999), que sólo utiliza ejemplos en Java, se cubre al menos un aspecto de la evolución de la que estamos hablando.

Lo que implica "hacerlo bien" no es sólo que el programa funcione según los requisitos y los casos de uso. También quiere decir que la estructura interna del código tiene sentido para ti y que parece encajar en su conjunto, sin sintaxis extraña, objetos demasiado grandes, o trozos de código expuestos torpemente. Además, debes tener la sensación de que la estructura del programa sobrevivirá a los cambios que inevitablemente tendrá que sufrir a lo largo de su vida, y que dichos cambios se podrán realizar de forma sencilla y limpia. Este no es un objetivo pequeño. No sólo debes entender lo que estás construyendo, sino que también deberás entender cómo evoluciona el programa (lo que yo denomino el *vector de cambio*). Afortunadamente, los lenguajes de programación orientados a objetos permiten particularmente bien este tipo de modificación continua—los límites que los objetos definen son los que tienden a proteger del colapso a la estructura. También te permiten hacer

cambios—ésos que parecerían drásticos en un programa procedural—sin provocar terremotos por todo tu código. De hecho, las facilidades que ofrece para la evolución del sistema podría ser el beneficio más importante de la POO.

Con el mecanismo de la evolución, creas algo que más o menos se aproxima a lo que piensas que estás construyendo, y luego frenas, lo comparas con tus requisitos y ves en qué se queda corto. Luego puedes volver atrás y arreglarlo, rediseñando y rehaciendo las partes de tu programa que no funcionaron correctamente. **[12]** En realidad, puede que hayas resuelto el problema, o un aspecto del problema, muchas veces antes de que des con la solución adecuada. (Aquí te puede ser de ayuda estudiar *Patrones de diseño*. Puedes encontrar información sobre éste tema en *Pensando sobre patrones (con Java)* disponible en *www.BruceEckel.com*.)

[12] Esto es parecido al "prototipado rápido," en el cúal se suponía que debías realizar una versión rápida y sucia que te permitiera aprender sobre el sistema, para luego tirar el prototipo y hacerlo bien. El problema con el prototipado rápido es que la gente no tiraba el prototipo, sino que construía sobre él. Esto, combinado con la falta de estructura de la programación procedural, conducía a menudo a sistemas enrevesados que eran costosos de mantener.

También se observa evolución cuando tras la realización del sistema, observas si se amolda a tus requisitos, y descubres que en realidad no es lo que querías. Cuando ves un sistema funcionando, puedes darte cuenta de que en realidad querías resolver un problema diferente. Si crees que este tipo de evolución puede darse, entonces deberías construir tu primera versión tan rápido como sea posible para averiguar si realmente es lo que quieres.

Quizás, lo más importante que has de recordar es que por defecto—por definición, en realidad—si modificas una clase, sus superclases y subclases seguirán funcionando. No tienes por qué temer la modificación (especialmente si tienes un conjunto integrado de unidades de prueba que verifiquen la corrección de tus modificaciones). Una modificación no tiene necesariamente que estropear el programa, y cualquier cambio en el resultado se limitará a las subclases y/o colaboradores específicos de la clase que has cambiado.

Los planos son valiosos

Por supuesto que no construirías una casa sin unos planos cuidadosamente delineados. Si estás construyendo una cubierta o una caseta para el perro los planos no serán tan elaborados, pero probablemente no empezarías sin algún tipo de diagrama que te sirva de guía. El desarrollo de software ha llevado esto al extremo. Durante mucho tiempo, la gente no incorporaba mucha estructura en su desarrollo, pero entonces comenzaron a fallar los grandes proyectos. Debido a esto, terminamos con metodologías que tenían una cantidad intimidatorio de estructura y detalle, fundamentalmente dirigida a esos grandes proyectos. El uso de estas metodologías era tan espeluznante—

parecía como si tuvieras que emplear todo tu tiempo en escribir documentos y no dedicarle nada a la programación. (Este era a menudo el caso.) Espero que lo que te he enseñado hasta el momento te sugiera que hay un término medio—una escala con situaciones intermedias. Utiliza una aproximación que se ajuste a tus necesidades (y a tu personalidad). No importa lo mínimo que sea el plan que escojas, un plan de *algún* tipo supondrá una gran mejora en tu proyecto, si se contrapone a no tener ningún plan en absoluto. Recuerda que, la mayoría de las estimaciones dicen que ás del 50 por ciento de los proyectos terminan mal (algunas estimaciones elevan esa cantidad al ¡70 por ciento!).

Al seguir un plan—preferiblemente uno que sea sencillo y breve—y obtener una estructura de diseño antes de empezar a codificar, descubrirás que las cosas encajan de forma mucho más fácil de la que lo harían si te lanzas de cabeza a machete. También conseguirás mayor satisfacción. Mi experiencia es que obtener una solución elegante es satisfactorio a un nivel completamente diferente; te sientes más cerca del arte que de la tecnología. Además, la elegancia siempre merece la pena; no es un objetivo frívolo. No sólo conseguirás un programa más fácil de realizar y depurar, también será más fácil de entender y mantener, y es ahí donde reside el valor financiero del conjunto.

Programación Extrema

He estudiado técnicas de análisis y diseño, una y otra vez, desde que estaba en el instituto. Y el concepto de *Programación Extrema* (XP) es el más radical y delicioso que he visto hasta ahora. puedes leer su crónica en el libro de Kent Beck *Extreme Programming Explained* (Addison-Wesley, 2000) y en la Web en *www.xprogramming.com*. También parece ser que Addison-Wesley va a sacar un nuevo libro de la serie XP cada uno o dos meses; el objetivo parece ser mover a la conversión casi religiosa de todo el mundo por el sólo peso de los libros (aunque generalmente estos libros son pequeños y agradables de leer).

XP es tanto una filosofía sobre la programación como un conjunto de guías para programar. Algunas de estas líneas de guía están presentes en otras metodologías recientes, pero, en mi opinión, las dos contribuciones más importantes y singulares, son la de “escribir primero las pruebas” y la “programación en parejas.” Aunque Beck defiende con ahínco el uso del proceso completo, también señala que si sólo adoptases estas dos prácticas mejorarías enormemente tu productividad y fiabilidad.

Escribe primero las pruebas

Tradicionalmente las pruebas se relegan a la última etapa de un proyecto, después de que “todo funciona, y sólo para asegurarse de ello.” Implícitamente tienen una prioridad baja, y a la gente que se especializa en ellas no se les da mucho estatus y a menudo se les retiene en un sótano, lejos de los “verdaderos programadores.” Los equipos de pruebas han respondido en consonancia, llegando a llevar ropaje negro y carcajeándose con júbilo cuando

rompen algo (para ser honesto, yo mismo me he sentido así cuando he roto algún compilador).

XP ha revolucionado completamente el concepto de la prueba al darle una prioridad equivalente a (o incluso mayor que) la codificación. De hecho, debes escribir las pruebas *antes* de escribir el código que deberá ser probado, y las pruebas permanecen con el código para siempre. Las pruebas deben ejecutarse con éxito cada vez que recompilas y montas el proyecto (lo que se hace a menudo, a veces más de una vez al día).

Escribir primero las pruebas tiene dos efectos extremadamente importantes.

Primero, obliga a tener una definición clara de la interfaz de la clase. A menudo sugiero que la gente "imagine una clase perfecta para resolver un problema particular" como herramienta cuando tratan de diseñar el sistema. La estrategia de pruebas de XP va más allá—especifica exactamente cuál debe ser el aspecto de la clase para el consumidor de dicha clase y exactamente cómo debe comportarse. Sin términos inciertos. Puedes escribir toda la prosa o crear todos los diagramas que quieras que describan cómo se debería comportar una clase y qué aspecto debe tener, pero no hay nada como un conjunto de pruebas reales. Lo primero es una carta a los reyes magos, pero las pruebas son un contrato que se mantiene en vigor gracias al compilador y la infraestructura de pruebas. Es difícil imaginar una descripción más concreta de una clase que las pruebas.

Mientras creas las pruebas, te fuerzas as repensar completamente sobre la clase y a menudo descubrirás funcionalidad necesaria que había pasado desapercibida a pesar de los experimentos mentales que conllevan los diagramas UML, las tarjetas CRC, los casos de uso, etc.

El segundo efecto importante de escribir primero las pruebas viene del hecho de que las pruebas se ejecutan cada vez que se reconstruye el software. Esta actividad te ofrece la parte del testeo que no te da el compilador. Si observas la evolución de los lenguajes de programación desde estas perspectivas, verás que los verdaderos avances tecnológicos en realidad han venido del lado de la prueba. El lenguaje ensamblador sólo comprobaba la sintaxis, pero C impuso algunas restricciones semánticas, y dichas restricciones evitaban que cometieras ciertos tipos de error. Los lenguajes de POO impusieron aún más restricciones semánticas, lo que, si lo piensas, en realidad son comprobaciones. "¿Se utiliza este tipo de datos de la forma adecuada?" y "¿Se invoca a este método de la forma adecuada?" son los tipos de comprobaciones que o el compilador o el sistema de ejecución realizan. Hemos visto los resultados de tener incorporados estas comprobaciones al lenguaje: la gente es capaz de escribir sistemas más complejos, y hacerlos funcionar, en mucho menos tiempo y con mucho menos esfuerzo. Me he devanado los sesos pensando en el porqué de esto, pero ahora me doy cuenta de que son las pruebas: haces algo mal, y la red de seguridad que te ofrecen las comprobaciones integradas te dicen que hay un problema y te señalan dónde está.

Pero las comprobaciones integradas que el diseño del lenguaje permite no pueden ir más allá. En un cierto punto, *tú* debes incorporarte al proceso y añadir el resto de las pruebas que producen un conjunto completo (en cooperación con el compilador y el sistema de ejecución) que verifique todo tu programa. Y, así como tienes un compilador que vigila por encima de tu hombro, ¿no querrías que estas pruebas te ayudasen desde el principio? Esa es la razón por la que escribes las pruebas primero, y las ejecutas automáticamente con cada construcción del sistema. Tus pruebas constituyen una extensión de la red de seguridad que ofrece el lenguaje.

Una de las cosas que he descubierto con el uso de lenguajes de programación cada vez más potentes es que me atrevo a probar con experimentos más descarados, porque se que el lenguaje evita que pierda tiempo buscando errores. El esquema de pruebas de XP hace esto mismo por tu proyecto al completo. Porque sabes que tus pruebas interceptarán los problemas que introduzcas (y regularmente añades nuevas pruebas según vayas pensando en ellas), puedes hacer grandes cambios cuando lo necesites sin preocuparte de sacar todo el proyecto de su línea. Esto es increíblemente potente.

En esta tercera edición de éste libro, me he dado cuenta de que las pruebas son tan importantes que deberían aplicarse también a los ejemplos del libro. Con ayuda de los Internos del Verano 2002 de Crested Butte, desarrollamos el sistema de pruebas que verás que hemos utilizado a lo largo de este libro. El código y la descripción están en el Capítulo 15. Este sistema ha mejorada la robustez de los ejemplos de código de este libro de forma inconmensurable.

Programación en parejas

La programación en parejas pretende luchar contra el exacerbado individualismo en el que hemos sido adoctrinados desde el principio, desde la escuela (donde aprobamos o suspendemos por nosotros mismos, y el trabajo con nuestros vecinos se considera “hacer trampa”), hasta los medios de comunicación, especialmente en las películas de Hollywood en las que el héroe habitualmente lucha contra la conformidad sin sentido. **[13]** También los programadores son considerados ejemplos de individualidad—los “codificadores vaqueros,” como Larry Constantine suele llamarlos. XP por contra, lo cual supone una batalla contra el pensamiento convencional, dice que el código debería escribirse con dos personas por estación de trabajo. Y que debería hacerse en un área con un grupo de estaciones de trabajo que no tenga las barreras a las que la gente que diseña las instalaciones es tan aficionada. De hecho, Beck dice que la primera tarea para convertirse al XP es llegar al trabajo con destornilladores y llaves Allen y apartar todo lo que esté en medio. **[14]** (Esto requiere un jefe que pueda desviar las iras del departamento de instalaciones.)

[13] Aunque puede que sea una perspectiva norteamericana no compartida, las historias de Hollywood llegan a todas partes.

[14] Esto incluye (especialmente) el sistema megafonía. En una ocasión trabajé en una empresa que insistía en emitir por megafonía cada llamada entrante que recibía cada ejecutivo, lo que constantemente interrumpía nuestra concentración (pero los responsables no podían ni siquiera concebir la supresión de un servicio tan importante como la megafonía). Al final, cuando nadie miraba empecé a desconectar a escondidas los cables de los altavoces.

El valor de la programación en parejas reside en que una persona es la que en realidad realiza la codificación mientras que la otra está pensando sobre ello. La que piensa mantiene en su mente la imagen de la totalidad de lo que se hace—no sólo la imagen del problema que tienen entre manos, sino las líneas guía de XP. Por ejemplo, si dos personas están trabajando juntas, es menos probable que una de ellas se desvíe del proceso y diga, "no quiero escribir primero las pruebas." Y si el codificador se atasca, pueden intercambiarse el puesto. Si los dos se atascan, puede que alguien más del área de trabajo escuche sus dudas pueden, alguien que pueda hacer alguna contribución. La programación en parejas mantiene las cosas fluidas y bajo control. Lo que es probablemente más importante, hace la programación más social y divertida.

He empezado a usar la programación en parejas durante los ejercicios de algunos de mis seminarios y parece mejorar significativamente la experiencia de todo el mundo.

Estrategias para el cambio

Si ya te has decidido por la POO, tu próxima pregunta probablemente será, "¿Cómo puede hacer que mi jefe/mi equipo/mi departamento/mis compañeros empiecen a utilizar objetos?" Piensa en cómo tú—un programador independiente—empezaría a aprender a utilizar un nuevo lenguaje y un nuevo paradigma de programación. Ya lo has hecho antes. Primero viene la formación y los ejemplos; luego viene un proyecto de prueba que te permita entender lo básico sin confundirte demasiado. Luego viene un proyecto "del mundo real" que haga algo realmente útil. Durante tus primeros proyectos sigues educándote por la lectura, haciendo preguntas a los expertos e intercambiando ideas con los amigos. Esta es la aproximación que sugieren muchos programadores experimentados para cambiar a Java. Por supuesto que cambiar a toda una empresa introduce cierta dinámica de grupos, pero en cada paso ayudará recordar cómo lo haría una sola persona.

Guías

A continuación se dan algunas líneas de guía que conviene considerar cuando se hace la transición a la POO y a Java:

1. Formación

El primer paso es algún tipo de formación. Recuerda la inversión que la empresa ha hecho en código, y no intentes desarreglar las cosas durante seis o

nueve meses, el tiempo que tarda la gente en resolver los problemas que aparecen con las características menos familiares. Escoge a un pequeño grupo para adoctrinarlos, preferiblemente uno compuesto por personas que sean curiosas, trabajen bien juntas y puedan funcionar como una red de autosoporte mientras aprenden Java.

Una aproximación alternativa es la educación de todos los niveles de la empresa a la vez, incluyendo cursos de introducción para los niveles de jefatura estratégica y cursos de diseño y programación para los equipos de proyectos. Esta aproximación es especialmente buena para las empresas más pequeñas que van a cambiar de manera fundamental la forma en la que hacen las cosas, o en un nivel de división para las empresas más grandes. Sin embargo, y dado que el coste es mayor, algunos pueden que empiecen eligiendo formar a un equipo de proyecto, hacer un proyecto piloto (posiblemente con un mentor externo), y dejar que el equipo se convierta en los que formen al resto de la empresa.

2. Un proyecto de bajo riesgo

Intenta primero abordar un proyecto de bajo riesgo y deja que aparezcan los errores. Una vez que hayas ganado alguna experiencia, puede utilizar los miembros del equipo como semillas para otros proyectos o utilizarlos como personal de soporte técnico de POO. Este primer proyecto puede que no funcione bien la primera vez, así que no debería ser un proyecto crítico para la misión de la empresa. Debería ser simple, sin relación con otros proyectos e instructivo; esto implica que debería implicar la creación de clases que tengan sentido para el resto de programadores de la empresa cuando les llegue el turno de aprender Java.

3. Modela desde el éxito

Busca ejemplos de buen diseño orientado a objetos en vez de empezar desde cero. Hay buenas probabilidades de que alguien más haya resuelto ya tu problema, y si no lo han resuelto exactamente puedes probablemente aplicar lo que has aprendido sobre la abstracción para modificar un diseño existente para que se ajuste a tus necesidades. Este el concepto general que está detrás de los *patrones de diseño,* que se cubre en *Pensando sobre patrones (con Java)* disponible en *www.BruceEckel.com*.

4. Utiliza las bibliotecas de clase existentes

Una motivación económica importante para cambiar a la POO es la facilidad con la que se puede utilizar el código existente en forma de bibliotecas de clase (en particular, las bibliotecas estándar de Java que se cubren en este libro). Obtendrás el más corto de los ciclos de desarrollo sólo cuando puedas crear y utilizar objetos de biblioteca no adaptados. Sin embargo, algunos programadores nuevos no entienden esto, no son conscientes de las bibliotecas de clase existentes o, debido a la fascinación que ejerce el lenguaje sobre ellos, desean escribir clases que muy bien pueden ya existir. Tu éxito

con la POO y Java será óptimo si puedes hacer el esfuerzo de buscar y reutilizar el código de otros que hayan realizado antes el proceso de transición.

5. No reescribas código existente en Java

Coger código existente y funcional y reescribirlo en Java no suele ser la mejor forma de emplear tu tiempo. Si vas a cambiar a la orientación a objetos puedes acceder al código C o C++ utilizando el interfaz de Java para Código Nativo o el *Lenguaje de Marcado Extensible* (XML). Hay en ello beneficios marginales, especialmente si se ha anunciado que el código será reutilizable. Pero hay pocas oportunidades de que vayas a ver mejoras dramáticas de la productividad que esperas de tus primeros proyectos a menos que sean completamente nuevos. Java y la POO brillan mejor cuando se lleva un proyecto al completo, desde su concepción hasta la realidad.

Problemas de gestión

Si eres un gestor, tu trabajo es conseguir recursos para tu equipo, quitar barreras que están en el camino del éxito para tu equipo, y en general tratar de ofrecer el entorno más productivo y divertido, de forma que tu equipo tenga más oportunidades de realizar los milagros que siempre os piden. El cambio a Java cae en esas tres categorías, y además podría ser maravilloso si no te costase nada. Aunque cambiar a Java puede ser más barato —dependiendo de tus restricciones—que las alternativas en POO para un equipo de programadores C (y probablemente para programadores de otros lenguajes procedurales), no será gratis, y hay obstáculos de los que deberías ser consciente antes de intentar vender el cambio a Java en tu empresa y de embarcarte en el mismo.

Costes de establecimiento

El coste de la transición a Java no se sólo el de la adquisición de los compiladores para Java (el compilador para Java de Sun Java es gratuito, así que este apenas supone un obstáculo). Los costes a medio y largo plazo se minimizarán si inviertes en formación (y si es posible un mentor par tu primer proyecto) y también si identificas y compras una biblioteca de clases que resuelva tu problema en vez de tratar de construir dichas bibliotecas por ti mismo. Esto tiene un coste alto que debe factorizarse en una propuesta realista. Además, existen costes ocultos por la pérdida de productividad que se produce mientras se aprende un nuevo lenguaje y posiblemente un nuevo entorno de programación. La formación y la presencia de un mentor seguramente pueden minimizarlos, pero los miembros del equipo deben superar sus propios problemas para entender la nueva tecnología. Durante este proceso cometerán más errores (esta es algo positivo, ya que los errores que se reconocen son el camino más rápido con el que aprender) y ser menos productivos. Incluso entonces, con algunos problemas de programación típicos, las clases adecuadas y el entorno de desarrollo adecuado, es posible ser más productivos durante el aprendizaje de Java (incluso aunque se considere que

cometes más errores y escribes menos líneas de código al día) que si hubieses seguido con C.

Consideraciones de rendimiento

Una pregunta habitual es, "¿No hará la POO automáticamente a mis programas más voluminosos y lentos?" La respuesta es, "Depende." Las características adicionales de seguridad de Java tradicionalmente han supuesto una penalización frente a los que usan un lenguaje com C++. Las tecnologías como "hotspot" y otras tecnologías de compilación han mejorado significativamente la velocidad de ejecución en la mayoría de los casos, y se siguen haciendo esfuerzos para mejorar el rendimiento.

Cuando tu foco es el prototipado rápido, puedes unir componentes tan rápido como puedas ignorando cuestiones de eficiencia. Si utilizas bibliotecas de otros proveedores, generalmente están optimizadas por sus fabricantes; en cualquier caso no es para ti un punto a tratar mientras estés en el modo de prototipado rápido. Cuando tienes el sistema que quieres, si es suficientemente pequeño y rápido, ya has terminado. En caso contrario, empiezas a ajustarlo con un perfilador, buscando primero las mejoras de rendimiento que se pueden obtener reescribiendo pequeñas partes del código. Si eso no ayuda, busca modificaciones que se puedan hacer en la parte oculta de la implementación de forma que el código que utiliza una clase particular deba modificarse. Si el rendimiento es tan crítico en esa parte del diseño, debe ser parte de tu criterio de diseño. Así que te has beneficiado al averiguar este hecho en una etapa temprana al utilizar técnicas de desarrollo rápido.

El capítulo 15 introduce el tema de los *perfiladores*, los cuales pueden ayudarte a descubrir cuellos de botella en tus sistema de forma que puedas optimizar esa parte de tu código (desde el lanzamiento de las tecnologías hotspot, Sun ha dejado de recomendar el uso de métodos nativos para optimizar el rendimiento). También hay disponibles otras herramientas de optimización.

Errores de diseño típicos

Cuando tu equipo empieza a utiliza la POO y Java, los programadores suelen cometer una serie de errores típicos. A menudo, esto ocurre debido a una realimentación insuficiente por parte de los expertos durante las etapas de diseño y construcción de los primeros proyectos, porque no se han desarrollado expertos en el seno de la empresa y porque puede haber resistencia a mantener consultores por mucho tiempo. Es fácil creerse que entiendes la POO demasiado pronto en el ciclo y salirse por la tangente. Algo que es obvio para alguien experto en el lenguaje puede ser una cuestión de debate para un novato. Gran parte de estos traumas pueden evitarse utilizando un experto externo con suficiente experiencia en la formación como mentor.

Resumen

La intención de este capítulo sólo era ofrecerte conceptos de las metodologías de POO y las clases de preguntas que te encontrarás cuando lleves tu empresa hacia la POO y Java. Puedes encontrar más información sobre el diseño de Objetos en el seminario de MindView “Diseño de Objetos y Sistemas” (busca en “Seminars” en el sitio *www.MindView.net*).

A: Pasando y Retornando Objetos

A estas alturas Usted debería estar razonablemente cómodo con la idea de que cuando usted " pasa " un objeto, usted pasa realmente una referencia.

En muchos lenguajes de programación usted puede usar la forma "regular" de estos para pasar objetos, y la mayoría de las veces todas funciona bien. Pero siempre llega el momento en el cual usted debe hacer algo fuera de lo común, y repentinamente las cosas se ponen un poco más complicadas (o en caso de C ++, muy complicado). Java no es una excepción, y es importante que usted entienda exactamente qué es lo que sucede cuando desplaza objetos de un lado a otro y los manipula. Este apéndice proveerá esa visión.

Otra forma para plantear la pregunta de este apéndice, ¿si usted viene de un lenguaje de programación equipado con punteros, es "Los tiene Java?". Algunos afirman que los punteros son difíciles y peligrosos y por consiguiente malos, y ya que Java es todo bondad y luz y le liberará de sus cargas terrenales de programación, no es posible que tenga tales cosas. Sin embargo, es más preciso decir que Java tiene punteros; ciertamente, cada identificador de un objeto en Java (excepto para los tipos primitivos de datos) es uno de estos punteros, pero su uso está restringido y vigilado no sólo por el compilador sino también por el sistema de tiempo de ejecución. O para ponerlo en otra forma, Java tiene punteros, pero no tiene aritmética de punteros. Estos son lo que he estado llamando "referencias", y usted puede pensar en ellos como "punteros seguros", no muy diferentes de las tijeras de seguridad de la escuela elemental - que no son afiladas, para que Usted no pueda lastimarse sin gran esfuerzo, pero algunas veces pueden ser lentos y tediosos.

Pasando referencias

Cuando usted pasa una referencia a un método, usted está todavía apuntando hacia el mismo objeto. Un experimento simple demuestra esto:

```
//: apendicea:PassReferences.java
// Pasando referencias.
import com.bruceeckel.simpletest.*;

public class PassReferences {
  private static Test monitor = new Test();
  public static void f(PassReferences h) {
    System.out.println("h inside f(): " + h);
  }
  public static void main(String[] args) {
    PassReferences p = new PassReferences();
    System.out.println("p inside main(): " + p);
```

```
    f(p);
    monitor.expect(new String[] {
      "%% p inside main\\(\\): PassReferences@[a-z0-9]+",
      "%% h inside f\\(\\): PassReferences@[a-z0-9]+"
    });
  }
} ///:~
```

El método **toString ()** es automáticamente invocado en las instrucciones de impresión, y **PassReferences** hereda directamente de **Object** sin redefinición de **toString ().** Así, la versión de **toString () de Object** es usada, la cual escribe la clase del objeto seguida por la dirección donde ese objeto está ubicado (no la referencia, sino el almacenamiento real del objeto). La salida de impresión tiene el siguiente aspecto:

```
p adentro de main() PassReferences@ad3ba4
h adentro de f(): PassReferences@ad3ba4
```

Usted puede ver que **p** y **h** se refieren al mismo objeto. Esto es mucho más eficiente que duplicar un objeto nuevo **PassReferences** solo para que Usted pueda enviar un argumento a un método. Pero trae a colación un asunto importante.

Aliasing

Aliasing quiere decir que más de una referencia está atada al mismo objeto, como en el ejemplo precedente. El problema con el aliasing ocurre cuando alguien *escribe* a ese objeto. Si los dueños de las otras referencias no esperan que ese objeto cambie, se llevarán entonces una sorpresa. Esto puede ser demostrado con un ejemplo simple:

```
//: appendixa:Alias1.java
// Aliando dos referencias a un objeto.
import com.bruceeckel.simpletest.*;

public class Alias1 {
  private static Test monitor = new Test();
  private int i;
  public Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Asigne la referencia
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementing x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    monitor.expect(new String[] {
      "x: 7",
      "y: 7",
```

```
      "Incrementing x",
      "x: 8",
      "y: 8"
    });
  }
} ///:~
```

En la línea:

```
Alias1 y = x; // Asigne la referencia
```

una referencia nueva **Alias1** es creada, pero en lugar de ser asignada a un objeto nuevo creado con **new,** es asignada a una referencia existente. De esta manera los contenidos de la referencia **x**, la cual es la dirección a la que el objeto **x** está apuntando, es asignada a **y,** y así tanto **x** como **y** están pegadas al mismo objeto. Por ello, cuando la variable i de x es incrementada en la instrucción:

```
x.i++;
```

la variable **i** de **y** será igualmente afectada. Esto puede verse en la salida:

```
x: 7
y: 7
Incrementando x
X: 8
y: 8
```

Una buena solución en este caso es simplemente no hacerlo. No direccione a propósito más de una referencia a un objeto en el mismo ámbito. Su código será mucho más fácil de entender y depurar. Sin embargo, cuando usted está pasando una referencia como un argumento - la que se supone es la forma como Java trabaja- Usted automáticamente está llevando a cabo el aliasing, porque la referencia local que es creada puede modificar el " objeto exterior " (el objeto que fue creado fuera del ámbito del método). Aquí hay un ejemplo:

```
//: apéndicea:Alias2.java

// Las llamadas a métodos pueden hacer aliasing implícitamente
sobre sus argumentos.

import com.bruceeckel.simpletest.*;
public class Alias2 {
  private static Test monitor = new Test();
  private int i;
  public Alias2(int ii) { i = ii; }
  public static void f(Alias2 reference) { reference.i++; }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
```

```
    f(x);
    System.out.println("x: " + x.i);
    monitor.expect(new String[] {
      "x: 7",
      "Calling f(x)",
      "x: 8"
    });
  }
} ///:~
```

El método está cambiando su argumento, el objeto exterior. Cuando este tipo de situación se presenta, usted debe decidir si tiene sentido, si el usuario lo espera, y si va a causar problemas.

En general, usted llama a un método para producir un valor de retorno y / o un cambio de estado en el objeto *para el cual el método ha sido llamado.* Es mucho menos común llamar a un método para manipular sus argumentos; esto se denomina "llamando un método por sus *efectos secundarios*." Así, cuando usted crea un método que modifica sus argumentos, el usuario debe ser claramente instruido y advertido acerca del uso de ese método y de sus sorpresas potenciales. Por la confusión y escollos que se pueden generar, es mucho mejor evitar afectar el argumento.

Si Usted necesita modificar un argumento durante una llamada a un método y no tiene la intención de modificar el argumento externo, lo que debe hacer es proteger éste último haciendo una copia de él dentro de su método. Ese es el tema de buena parte de este apéndice.

Haciendo copias locales

Para repasar: Todo paso de argumentos en Java es llevado a cabo a través de referencias. Esto es, cuando usted pasa " un objeto, " usted está realmente pasando solo una referencia a un objeto que vive fuera del método, así es que si usted realiza cualquier modificación a esa referencia, entonces usted está modificando el objeto exterior. Además:

- El aliasing ocurre automáticamente durante el paso de argumentos.
- No hay objetos locales, sólo referencias locales.
- Las referencias tienen ámbito, los objetos no.
- El tiempo de vida de un objeto no es nunca un asunto en Java.
- No hay soporte de lenguaje (e.g., "const ") para impedir la modificación de objetos y detener los efectos negativos del aliasing. Usted simplemente no puede usar la palabra clave **final** en la lista de argumentos; eso simplemente impide que vuelva a atar la referencia a un objeto diferente.

Si usted sólo está leyendo información de un objeto y no modificándola, el paso de una referencia es la forma más eficiente de paso de argumentos. Esto es bueno; el modo por defecto de hacer las cosas es también el más eficiente. Sin embargo, algunas veces hay que poder tratar el objeto como si fuera

"local" a fin de que los cambios que usted haga afecten sólo la copia local y no modifiquen el objeto exterior. Muchos lenguajes de programación dan soporte a la posibilidad de hacer automáticamente dentro del método, una copia local del objeto externo.[116] Java no lo hace, pero le permite producir este efecto.

Paso por valor

Esto trae a colación el asunto de la terminología, lo cual siempre parece bueno para una discusión. El término es " paso por valor, " y el significado depende de cómo percibe usted la operación del programa. El significado general es que usted obtiene una copia local de lo que sea que usted está pasando, pero la pregunta real es cómo piensa usted sobre lo que está pasando. En lo concerniente al significado de "paso por valor," hay dos campos relativamente distintos:

1. Java pasa todo por valor. Cuando usted está pasando valores primitivos a un método, usted obtiene una copia distinta del valor primitivo. Cuando usted pasando una referencia a un método, usted obtiene una copia de la referencia. Por ende, todo es pasado por valor. Por supuesto, la suposición es que usted siempre está pensando (y preocupándose de) que son referencias las que están siendo pasadas, pero parece que el diseño de Java ha hecho todo lo posible para permitirle ignorar (la mayoría de las veces) que usted está trabajando con una referencia. Esto es, parece darle permiso de pensar en la referencia como " el objeto, " ya que implícitamente dereferencia a este cuandoquiera que usted hace una llamada a un método.

2. Java pasa los valores primitivos por valor (no hay argumento alguno allí), pero los objetos son pasados por referencia. Ésta es la visión mundial de que la referencia es un alias para el objeto, así que usted *no piensa en* pasar referencias, pero en lugar de eso dice " Estoy pasando el objeto." Ya que usted no obtiene una copia local del objeto cuando lo pasa a un método, los objetos claramente no son pasados por valor. Parece haber algo de soporte para este punto de vista dentro de Sun, ya que en algún momento, uno de las palabras claves "reservadas pero no implementadas" fue **byvalue** (la cual probablemente nunca será implementada).

Habiendo dado a ambos campos una buena exposición, y después de decir "Depende de cómo piense usted de una referencia," intentaré obviar el asunto. Al fin y al cabo, no es *tan* importante - lo que es importante es que usted entienda que pasar una referencia permite que el objeto llamante sea cambiado inesperadamente.

---

[1] *En C, el cual generalmente maneja pequeños bits de información, el paso por valor es la forma predefinida de hacerlo. C++ tenía que seguir esta forma, pero el paso por valor de objetos no es usualmente el camino más eficiente. Adicionalmente, la codificación de las clases para que soporten el paso por valor es un gran dolor de cabeza en C++.*

Clonación de objetos

La razón más probable para hacer una copia local de un objeto es si usted va a modificar el objeto y no quiere modificar el objeto llamante. Si usted decide que quiere hacer una copia local, una alternativa es usar el método **clone ()** para realizar la operación. Éste es un método que es definido como **protected** en la clase base **Object,** y que usted debe redefinir como **public** en cualquier clase derivada que quiera clonar. Por ejemplo, la clase **ArrayList** de la librería estándar redefine **clone (),** así es que podemos llamar **clone ()** para **ArrayList:**

```
//: apéndicea:Cloning.java
// La operación de clone() trabaja solo para algunos pocos
// items en la librería estándar de Java.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() { return Integer.toString(i); }
}

public class Cloning {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Incremente todos los elementos de v2:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int)e.next()).increment();
    // Vea si cambió los elementos de v:
    System.out.println("v: " + v);
    monitor.expect(new String[] {
      "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
      "v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
    });
  }
} ///:~
```

El método **clone ()** produce un **Objeto,** al cual se le debe hacer un cast al tipo correcto. Este ejemplo muestra cómo el método **clone() de ArrayList** *no trata* automáticamente de clonar cada uno de los objetos que el **ArrayList** contiene - el viejo **ArrayList** y el **ArrayList** clonado son aliased a los mismos objetos. Esto es a menudo llamado una *copia superficial,* ya que es copiar solo

la porción "superficial" de un objeto. El objeto real consta de esta " superficie, " más todos los objetos a los que las referencias apuntan, más todos los objetos a los que *esos* objetos apuntan, etc. Esto es a menudo llamado " la red de objetos." Copiar toda la malla es llamado una *copia profunda.*

Usted puede ver el efecto de la copia superficial en la salida, donde las acciones realizadas sobre **v2** afectan a **v:**

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

No tratar de aplicar **clone()** a los objetos contenidos en el **ArrayList** es probablemente una suposición justa, porque no hay seguridad de que esos objetos *sean* clonables.[117]

Añadiendo clonabilidad a una clase

Si bien el método de clonación está definido en **Object,** la-base-de-todas-las-clases**,** la clonación no está automáticamente disponible en cada clase. [118] Esto parecería ser contra-intuitivo a la idea de que los métodos de la clase base están siempre disponibles en las clases derivadas. La clonación en Java ciertamente va en contra de esta idea; si quiere que exista en una clase, usted tiene que específicamente añadir código para hacer que la clonación funcione.

Usando un truco con protected

Para impedir la clonabilidad por defecto en cada clase que usted cree, el método **clone ()** es **protected** en la clase base **Object.** Esto no solo significa que la clonabilidad no está disponible por defecto para el programador cliente que simplemente usa la clase (no generando clases derivadas), pero también

---

[2] *Este no es el deletreo que se encuentra en el diccionario para esta palabra, pero es el utilizado en la biblioteca de Java, por lo cual yo lo he usado aquí también, esperando reducir la confusión.*

[3] *Usted puede aparentemente crear un contra-ejemplo simple contra esta afirmación, como este:*

```
public class Cloneit implements Cloneable {
  public static void main (String[] args)
  throws CloneNotSupportedException {
    Cloneit a = new Cloneit();
    Cloneit b = (Cloneit)a.clone();
  }
}
```

Sin embargo, esto solo funciona porque ***main( )*** *es un método de* ***Cloneit*** *y así tiene permiso para llamar el método* ***protected clone( )*** *de la clase base. Si lo llama desde una clase diferente, no compilará.*

quiere decir que usted no puede llamar **clone ()** por medio de una referencia a la clase base (aunque eso podría parecer ser útil en algunas situaciones, como para clonar polimórficamente un montón de **Objetos.**) Es, en efecto, una forma para darle a usted, en el tiempo de compilación, la información de que su objeto no es clonable - y por raro que parezca, la mayoría de las clases en la biblioteca estándar de Java no lo son. Así, si Usted dice:

```
Integer x = new Integer(1);
x = x.clone();
```

Usted obtendrá, en la fase de compilación, un mensaje de error que dice que **clone ()** no es accesible (ya que **Integer** no le invalida y y este revierte a la versión **protected**).

Sin embargo, si usted está en un método de una clase *derivada* de **Object** (como lo son todas las clases), entonces usted está autorizado para llamar a **Object.clone ()** porque es **protected** y usted está heredando. El método **clone()** de la clase base tiene una útil funcionalidad; lleva a cabo la duplicación real bit a bit *del objeto de la clase derivada,* actuando así como la operación común de clonación. Sin embargo, usted luego necesita convertir en **public** *su* operación de clonación para que sea accesible. En consecuencia, dos asuntos claves cuando usted lleva a cabo una clonación son:

- Llamar a **super.clone ()**
- Convertir su clon en **public**

Usted probablemente querrá anular **clone ()** en cualquier clase derivada adicional; de otra manera, su **clone (),**(ahora **public**) será usado, y eso podría no hacer lo correcto (aunque, ya que **Object.clone ()** hace una copia del objeto mismo, también podría ser que sí). El truco **con protected** surte efecto sólo una vez: la primera vez que usted hereda de una clase que no tiene clonabilidad y usted quiere hacer una clase que es clonable. En cualquier clase derivada de la suya, el método **clone()** está disponible ya que durante la derivación no es posible limitar en Java el acceso a un método. Es decir, una vez que una clase es clonable, cualquier cosa derivada de ella lo es también a menos que usted use los mecanismos provistos para "desactivar" la clonación (los cuales se describen posteriormente).

Implementando la interfaz Cloneable

Hay una cosa más que usted necesita hacer para completar la clonabilidad de un objeto: Implementar la **interfaz Cloneable.** Esta **interfaz** es un poco extraña, porque está vacía!

```
interface Cloneable {}
```

La razón para implementar esta **interfaz** vacía es obviamente no porque usted vaya a hacer casting hacia arriba a **Cloneable** y vaya a llamar a uno de sus

métodos. El uso de **interface** de este modo es llamado una *interfaz de etiqueta* porque actúa como un tipo de bandera, embebido en el tipo de la clase.

Hay dos razones para la existencia de la **interfaz Cloneable.** Primero, usted podría tener una referencia a la que se le ha hecho casting hacia arriba a un tipo base y no sabe si es posible clonar ese objeto. En este caso, usted puede usar la palabra clave **instanceof** (descrita en el Capítulo 10) para averiguar si la referencia está conectada a un objeto que puede ser clonado:

```
if(myReference instanceof Cloneable) // ...
```

La segunda razón es que mezclado en este diseño sobre clonabilidad estaba el pensamiento de que tal vez usted no quería que todos los tipos de objetos fueran clonables. Así, **Object.clone ()** comprueba que una clase implementa la interfaz **Cloneable.** En caso de que no, lanza una excepción **CloneNotSupportedException.** En general entonces, usted se ve forzado a **implementar a Cloneable** como parte del soporte para la clonación.

La clonación exitosa

Una vez que usted entiende los detalles de implementar el método **clone (),** usted puede crear clases que pueden ser fácilmente duplicadas para proveer una copia local:

```
//: appendixa:LocalCopy.java
// Creando copias locales con clone().
import com.bruceeckel.simpletest.*;
import java.util.*;

class MyObject implements Cloneable {
  private int n;
  public MyObject(int n) { this.n = n; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
  public int getValue() { return n; }
  public void setValue(int n) { this.n = n; }
  public void increment() { n++; }
  public String toString() { return Integer.toString(n); }
}

public class LocalCopy {
  private static Test monitor = new Test();
  public static MyObject g(MyObject v) {
    // Pasando una referencia, modifica un objeto externo:
```

```
    v.increment();
    return v;
  }
  public static MyObject f(MyObject v) {
    v = (MyObject)v.clone(); // Copia local
    v.increment();
    return v;
  }
  public static void main(String[] args) {
    MyObject a = new MyObject(11);
    MyObject b = g(a);
    // Equivalencia de referencias, no de objetos:
    System.out.println("a == b: " + (a == b) +
      "\na = " + a + "\nb = " + b);
    MyObject c = new MyObject(47);
    MyObject d = f(c);
    System.out.println("c == d: " + (c == d) +
      "\nc = " + c + "\nd = " + d);
    monitor.expect(new String[] {
      "a == b: true",
      "a = 12",
      "b = 12",
      "c == d: false",
      "c = 47",
      "d = 48"
    });
  }
} ///:~
```

Primero que todo, para que **clone ()** pueda ser accesible, lo tiene que hacer **public**. En segundo lugar, para la parte inicial de la operación de su **clone(),** usted debería llamar la versión de **clone()** de la clase base **.** El **clone()** que se llama aquí es el que está predefinido dentro de **Object,** y usted lo puede llamar porque es **protected** y por consiguiente accesible en las clases derivadas.

Object.clone () calcula qué tan grande es el objeto, crea suficiente memoria para una nuevo, y copia todos los bits del viejo al nuevo. Ésta es llamado una *copia bit a bit,* y es típicamente lo que usted esperaría que haga un método **clone()**. Pero antes de que **Object.clone ()** realice sus operaciones, primero inspecciona para ver si una clase es **Cloneable -**es decir, si implementa la interfaz **Cloneable.** Si no lo hace, **Object.clone ()** lanza una excepción **CloneNotSupportedException** para señalar que usted no le puede clonar. Así, usted tiene que envolver su llamada a **super.clone ()** con un bloque **try** para capturar una excepción que nunca debería ocurrir (porque usted ha implementado la interfaz **Cloneable**).

En **LocalCopy,** los dos métodos **g()** y **f ()** demuestran la diferencia entre las dos maneras de pasar argumentos. El método **g()** muestra el paso por referencia en el cual el método modifica el objeto exterior y devuelve una referencia a ese objeto exterior, mientras que **f()** clona el argumento, por consiguiente desacoplándolo y dejando sin tocar el objeto original. Luego

puede proceder a hacer lo que quiera -aun retornar una referencia a este objeto nuevo sin afectar para nada el original. Note la declaración de aspecto algo curioso:

```
v = (MyObject)v.clone();
```

Aquí es donde se crea la copia local. Para prevenir la confusión que tal declaración podría causar, recuerde que esta más bien extraña forma de codificar es perfectamente factible en Java porque cada identificador de un objeto es de hecho una referencia. Así es que la referencia **v** se usa para **clone()** una copia de a lo que ella se está refiriendo, y esta devuelve una referencia al tipo base **Object** (porque está definido de ese modo en **Object.clone ()**) al que luego hay que hacerle un casting al tipo correcto.

En **main()** se prueba la diferencia entre los efectos de los dos diferentes formas de paso de argumentos. Es importante tener en cuenta que las pruebas de equivalencia en Java no miran en el interior de los objetos que están siendo comparados para ver si sus valores son lo mismos. Los operadores = = y ! = simplemente comparan las *referencias.* Si las direcciones dentro de las referencias son las mismas, ellas apuntan hacia el mismo objeto y por consiguiente son “iguales.” Por ende, lo que los operadores realmente prueban es si las referencias están aliased al mismo objeto.

El efecto de Object.clone ()

¿Qué ocurre realmente cuando **Object.clone ()** es llamado que hace tan esencial llamar a **super.clone ()** cuando usted anula clone **()** en su clase? El método **clone()** en la clase raíz es responsable de crear la cantidad correcta de almacenamiento y hacer la copia bit a bit de los bits del objeto original en el espacio de almacenamiento del objeto nuevo. Esto es,no hace solo almacenamiento y copia un **Object sino** que realmente calcula el tamaño del objeto *real* (no solo el objeto de clase base, sino el objeto derivado) que está siendo copiado y duplica eso. Ya todo esto está ocurriendo desde el código en el método **clone()** definido en la clase raíz (la cual no tiene idea de qué está siendo heredado desde ella), usted puede adivinar que el proceso exige a RTTI determinar el objeto que está siendo realmente clonado. Así, el método **clone()** puede crear la cantidad correcta de almacenamiento y puede hacer la copia correcta bit a bit para ese tipo.

No importando lo que usted haga, la primera parte del proceso de clonación normalmente debería ser una llamada a **super.clone ().** Esto establece el trabajo de base para la operación de clonación mediante la elaboración de un duplicado exacto. En este punto usted puede realizar otras operaciones necesarias para completar la clonación.

Para saber con seguridad cuáles son esas otras operaciones, usted necesidad entender exactamente qué hace exactamente **Object.clone ()**. ¿En particular, clona automáticamente el destino de todas las referencias? El siguiente ejemplo prueba esto:

```
//: apéndicea:Snake.java
// Prueba la clonación para ver si el destino
// de las referencias también es clonado.
import com.bruceeckel.simpletest.*;

public class Snake implements Cloneable {
  private static Test monitor = new Test();
  private Snake next;
  private char c;
  // Valor de i == número de segmentos
  public Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  public void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  public String toString() {
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Snake can't clone");
    }
    return o;
  }
  public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println("after s.increment, s2 = " + s2);
    monitor.expect(new String[] {
      "s = :a:b:c:d:e",
      "s2 = :a:b:c:d:e",
      "after s.increment, s2 = :a:c:d:e:f"
    });
  }
} ///:~
```

Una **Snake** está hecha de un montón de segmentos, cada uno de tipo **Snake.** Así, es una lista enlazada simple. Los segmentos son creados recursivamente,

decrementando el argumento del primer constructor para cada segmento hasta que se alcanza el cero. Para darle a cada segmento una etiqueta única, el segundo argumento, un **char,** es incrementado en cada llamada recursiva del constructor.

El método **increment()** aumenta recursivamente cada etiqueta para que usted pueda ver el cambio, y el método **toString ()** imprime recursivamente cada etiqueta. De la salida, usted puede ver que sólo el primer segmento es duplicado por **Object.clone (),** por consiguiente hace una copia superficial. Si usted quiere que la serpiente entera sea duplicado - una copia a fondo - usted debe realizar las operaciones adicionales dentro de su método **clone()** anulado**.**

Típicamente usted llamará a **super.clone ()** en cualquier clase derivada de una clase clonable para asegurarse de que todas las operaciones de la clase base (incluyendo a **Object.clone ()**) tengan lugar. Esto es seguido por una llamada explícita a **clone()** para cada referencia en su objeto; de otra manera esas referencias serán aliased a aquellas del objeto original. Es análogo a la forma cómo los constructores son llamados: primero el constructor de la clase base, luego el siguiente constructor derivado, y así sucesivamente, hasta el último constructor derivado. La diferencia es que **clone()** no es un constructor, así que no hay nada para que suceda automáticamente. Usted debe asegurarse de hacerlo usted mismo.

Clonando un objeto compuesto

Hay un problema que usted encontrará cuándo intente copiar a fondo un objeto compuesto. Usted debe asumir que el método **clone()** en los objetos miembro a su vez realizarán una copia a fondo sobre *sus* referencias, y así sucesivamente. Esto es una gran obligación. Significa efectivamente que para que una copia a fondo funcione, usted debe ya sea controlar todo el código en todas las clases, o al menos tener suficiente conocimiento de todas las clases involucradas en la copia a fondo para saber que están llevando a cabo correctamente su propia copia a fondo.

Este ejemplo muestra lo que usted debe hacer para lograr una copia a fondo cuando esté trabajando con un objeto compuesto:

```
//: apéndicea:DeepCopy.java
// Clonando un objeto compuesto.
// {Depende de: junit.jar}
import junit.framework.*;

class DepthReading implements Cloneable {
  private double depth;
  public DepthReading(double depth) { this.depth = depth; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
```

```
      e.printStackTrace();
    }
    return o;
  }
  public double getDepth() { return depth; }
  public void setDepth(double depth){ this.depth = depth; }
  public String toString() { return String.valueOf(depth);}
}

class TemperatureReading implements Cloneable {
  private long time;
  private double temperature;
  public TemperatureReading(double temperature) {
    time = System.currentTimeMillis();
    this.temperature = temperature;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace();
    }
    return o;
  }
  public double getTemperature() { return temperature; }
  public void setTemperature(double temperature) {
    this.temperature = temperature;
  }
  public String toString() {
    return String.valueOf(temperature);
  }
}

class OceanReading implements Cloneable {
  private DepthReading depth;
  private TemperatureReading temperature;
  public OceanReading(double tdata, double ddata) {
    temperature = new TemperatureReading(tdata);
    depth = new DepthReading(ddata);
  }
  public Object clone() {
    OceanReading o = null;
    try {
      o = (OceanReading)super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace();
    }
    // Debe clonar las referencias:
    o.depth = (DepthReading)o.depth.clone();
    o.temperature =
      (TemperatureReading)o.temperature.clone();
    return o; // Casting inverso a Object
```

```
  }
  public TemperatureReading getTemperatureReading() {
    return temperature;
  }
  public void setTemperatureReading(TemperatureReading tr){
    temperature = tr;
  }
  public DepthReading getDepthReading() { return depth; }
  public void setDepthReading(DepthReading dr) {
    this.depth = dr;
  }
  public String toString() {
    return "temperature: " + temperature +
      ", depth: " + depth;
  }
}

public class DeepCopy extends TestCase {
  public DeepCopy(String name) { super(name); }
  public void testClone() {
    OceanReading reading = new OceanReading(33.9, 100.5);
    // Ahora, clónelo:
    OceanReading clone = (OceanReading)reading.clone();
    TemperatureReading tr = clone.getTemperatureReading();
    tr.setTemperature(tr.getTemperature() + 1);
    clone.setTemperatureReading(tr);
    DepthReading dr = clone.getDepthReading();
    dr.setDepth(dr.getDepth() + 1);
    clone.setDepthReading(dr);
    assertEquals(reading.toString(),
      "temperature: 33.9, depth: 100.5");
    assertEquals(clone.toString(),
      "temperature: 34.9, depth: 101.5");
  }
  public static void main(String[] args) {
    junit.textui.TestRunner.run(DeepCopy.class);
  }
} ///:~
```

DepthReading y **TemperatureReading** son muy similares; Ambos contienen sólo primitivas. Por consiguiente, el método **clone()** puede ser muy simple: llama a **super.clone ()** y devuelve el resultado. Note que el código de **clone()** para ambas clases es idéntico.

OceanReading está compuesto de los objetos **DepthReading** y **TemperatureReading** y por consiguiente, para producir una copia a fondo, su método **clone()** deba clonar las referencias dentro de **OceanReading.** Para lograrlo, al resultado de **super.clone ()** se le debe hacer casting a un objeto **OceanReading** (para que usted puede ganar acceso a las referencias **depth** y **temperature**).

Una copia a fondo con ArrayList

Volvamos a visitar a **Cloning.java** que vimos antes en esta apéndice. Esta vez la clase **Int2** es clonable, para que se pueda hacer una copia a fondo de **ArrayList**:

```
//: apèndicea:AddingClone.java
// Usted debe pasar por una serie de giros
// para añadir la clonación a su propia clase.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() { return Integer.toString(i); }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Int2 can't clone");
    }
    return o;
  }
}

// La herencia no remueve la clonabilidad:
class Int3 extends Int2 {
  private int j; // Duplicado automáticamente
  public Int3(int i) { super(i); }
}

public class AddingClone {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.increment();
    System.out.println("x = " + x + ", x2 = " + x2);
    // Cualquier cosa heredada también es clonable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int2(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Ahora clone cada elemento:
    for(int i = 0; i < v.size(); i++)
      v2.set(i, ((Int2)v2.get(i)).clone());
    // Incremente todos loe elementos de v2:
```

```
    for(Iterator e = v2.iterator(); e.hasNext(); )
      ((Int2)e.next()).increment();
    System.out.println("v2: " + v2);
    // Vea si cambió los elementos de v:
    System.out.println("v: " + v);
    monitor.expect(new String[] {
      "x = 10, x2 = 11",
      "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
      "v2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]",
      "v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]"
    });
  }
} ///:~
```

Int3 es heredado de **Int2,** y se agrega un miembro primitivo nuevo, **int j,** . Usted podría pensar que usted necesitaría anular **clone()** otra vez para asegurar que **j** sea copiado, pero ese no es el caso. Cuando se llama a **clone() de Int2()** como **el clone() de Int3(),** aquel llama a **Object.clone(),** lo cual determina que está trabajando con un **Int3** y duplica todos los bits en **Int3.** Mientras que usted no agrege referencias que necesiten ser clonadas, la llamada a **Object.clone ()** realiza todas la duplicación necesaria no importa qué tan profundo en la jerarquía esté definido **clone()**.

Usted puede ver qué es lo que se necesita para hacer una copia a fondo de un **ArrayList:** Después de que el **ArrayList** es clonado, usted tiene que avanzar y clonar cada uno de los objetos a los que **ArrayList** apunta. Usted tendría que hacer algo parecido a esto para hacer una copia a fondo de un **HashMap.**

El resto del ejemplo comprueba que la clonación ocurrió mostrando que, una vez que un objeto es clonado, usted lo puede cambiar y sin embargo, el objeto original no sufre modificación alguna.

Copia a fondo mediante la serialización

Cuando usted considera la serialización de objetos de Java (presentado en el Capítulo 12), usted podría observar que un objeto que es serializado y luego deserializado, efectivamente ha sido clonado.

¿Así es que por qué no usar la serialización para llevar a cabo la copia a fondo? Aquí hay un ejemplo que compara los dos métodos cronometrándolos:

```
//: apéndicea:Compete.java
import java.io.*;

class Thing1 implements Serializable {}
class Thing2 implements Serializable {
  Thing1 o1 = new Thing1();
}
```

```
class Thing3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing3 can't clone");
    }
    return o;
  }
}

class Thing4 implements Cloneable {
  private Thing3 o3 = new Thing3();
  public Object clone() {
    Thing4 o = null;
    try {
      o = (Thing4)super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing4 can't clone");
    }
    // Clone el campo también:
    o.o3 = (Thing3)o3.clone();
    return o;
  }
}

public class Compete {
  public static final int SIZE = 25000;
  public static void main(String[] args) throws Exception {
    Thing2[] a = new Thing2[SIZE];
    for(int i = 0; i < a.length; i++)
      a[i] = new Thing2();
    Thing4[] b = new Thing4[SIZE];
    for(int i = 0; i < b.length; i++)
      b[i] = new Thing4();
    long t1 = System.currentTimeMillis();
    ByteArrayOutputStream buf= new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
      o.writeObject(a[i]);
    // Ahora obtenga copias:
    ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(buf.toByteArray()));
    Thing2[] c = new Thing2[SIZE];
    for(int i = 0; i < c.length; i++)
      c[i] = (Thing2)in.readObject();
    long t2 = System.currentTimeMillis();
    System.out.println("Duplication via serialization: " +
      (t2 - t1) + " Milliseconds");
    // Ahora intente la clonación:
    t1 = System.currentTimeMillis();
    Thing4[] d = new Thing4[SIZE];
```

```
    for(int i = 0; i < d.length; i++)
      d[i] = (Thing4)b[i].clone();
    t2 = System.currentTimeMillis();
    System.out.println("Duplication via cloning: " +
      (t2 - t1) + " Milliseconds");
  }
} ///:~
```

Thing2 y **Thing4** contienen objetos miembro de tal manera que hay alguna copia a fondo tomando lugar. Es interesante notar que mientras es fácil armar clases **Serializable**, es mucho más laborioso duplicarlas. La clonación implica un montón de trabajo para armar la clase, pero la duplicación en sí de los objetos es relativamente simple. Los resultados son interesantes. Aquí está la salida de tres corridas diferentes:

```
Duplicación mediante serialización: 547 Milisegundos
Duplicación mediante clonación: 110 Milisegundos

Duplicación mediante serialización: 547 Milisegundos
Duplicación mediante clonación: 109 Milisegundos

Duplicación mediante serialización: 547 Milisegundos
Duplicación mediante clonación: 125 Milisegundos
```

En versiones previas del JDK, el tiempo requerido para la serialización era mucho más largo que para la clonación (aproximadamente 15 veces más lento), y el tiempo de serialización tendía a variar bastante. Versiones más recientes han acelerado la serialización y aparentemente también han hecho el tiempo más consistente. Aquí, es aproximadamente cuatro veces más lento, lo que lo hace razonable para usar como una alternativa de clonación.

Añadiendo clonabilidad más abajo en la jerarquía

Si usted crea una clase nueva, su clase base se revierte a **Object,** y por consiguiente a la no clonabilidad (como se verá en la siguiente sección). Mientras usted explícitamente no adicione clonabilidad, usted no la tendrá. Pero la puede añadir en cualquier nivel y entonces será clonable de ese nivel hacia abajo, como esto:

```
//: apéndicea:HorrorFlick.java
// Usted puede insertar clonabilidad en cualquier nivel de herencia
package appendixa;
import java.util.*;

class Person {}
```

```
class Hero extends Person {}
class Scientist extends Person implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch(CloneNotSupportedException e) {
      // Esto nunca debería suceder.  Ya es clonable !
      throw new RuntimeException(e);
    }
  }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
  public static void main(String[] args) {
    Person p = new Person();
    Hero h = new Hero();
    Scientist s = new Scientist();
    MadScientist m = new MadScientist();
    //! p = (Person)p.clone(); // Error de compilación
    //! h = (Hero)h.clone(); // Error de compilación
    s = (Scientist)s.clone();
    m = (MadScientist)m.clone();
  }
} ///:~
```

Antes de que se agregase la clonabilidad en la jerarquía, el compilador le impidió intentar copiar cosas. Cuando se añade la clonabilidad en **Scientist,** entonces éste y todos sus descendientes son clonables

¿Por qué este diseño extraño?

Si todo esto parece ser un esquema extraño, es que efectivamente lo es. Usted podría preguntarse por qué resultó así. ¿Cuál es la idea detrás de este diseño?

Originalmente, Java fue diseñado como un lenguaje para monitorear elementos de hardware, y definitivamente no con Internet en mente. En un lenguaje de propósito general como éste, tiene sentido que el programador pueda clonar cualquier objeto. Por ello, **clone()** fue colocado en la clase raíz **Object,** *pero* era un método **public** así es que usted siempre podría clonar cualquier objeto. Éste parecía ser el acercamiento más flexible, y después de todo, ¿ qué daño podría hacer?

Bien, cuando se vio a Java como el lenguaje de programación más adecuado para Internet, las cosas cambiaron. Repentinamente, hay preocupaciones de seguridad, y por supuesto, estos asuntos se manejan usando objetos, y usted necesariamente no quiere que cualquiera pueda clonar sus objetos que manejan la seguridad. Por tanto, lo que usted está viendo es una cantidad de parches aplicados sobre el original esquema simple y franco: **clone()** es ahora

protected en **Object.** Usted debe anularlo *e* **implementar Cloneable** *y,* debe ocuparse de las excepciones.

Vale notar que usted debe implementar la interfaz **Cloneable** *sólo* si usted va a llamar el método **clone()** de **Object,** ya que este método inspecciona durante el tiempo de ejecución si su clase implementa **Cloneable.** Pero por consistencia (y ya que **Cloneable** de cualquier manera está vacío), usted lo debería implementar.

Monitoreando la clonabilidad

Usted podría sugerir que para quitar la clonabilidad, el método **clone()** simplemente debería hacerse **private**, pero esto no funcionará, porque no puede tomar un método de una clase base y hacerlo menos accesible en una clase derivada. Y sin embargo, es necesario poder controlar si un objeto puede ser donado. Hay un número de actitudes que usted puede tomar para lograr esto en sus clases:

1. La indiferencia. Usted no hace nada sobre la clonación, lo cual quiere decir que su clase no puede ser clonada, pero una clase que herede de usted puede añadir clonación si lo quiere. Esto trabaja únicamente si el método **Object.clone()** predeterminado hace algo razonable con todos los campos en su clase.
2. Soporte **clone().** Siga la práctica estándar de implementar **Cloneable** y sobrescribir **clone().** En el **clone()** sobrescrito**,** usted llama a **super.clone ()** y captura todas las excepciones (para que su **clone()** sobrescrito no lance excepción alguna).
3. Soporte condicionalmente la clonación. Si su clase tiene referencias a otros objetos que podrían o no ser clonables (una clase contenedora, por ejemplo), su **clone()** puede tratar de clonar todos los objetos para los cuales usted tiene referencias, y si lanzan excepciones, simplemente pasarlas al programador. Por ejemplo, considere un tipo especial de **ArrayList** que trata clonar todos los objetos que tiene. Cuando usted escribe un **ArrayList** de este tipo**,** usted no conoce qué tipo de objetos el programador cliente podría poner en su **ArrayList,** así que usted no sabe si pueden o no ser clonados.
4. No implemente a **Cloneable** pero sobrescriba **clone()** como **protected,** produciendo así la conducta correcta de copiado para cualquiera de los campos. De esta forma, cualquiera que herede de esta clase puede sobrescribir **clone()** y llamar a **super.clone ()** para producir la conducta de copiado correcta. Note que su implementación puede y debería invocar a **super.clone ()** si bien ese método espera un objeto **Cloneable** (de otra manera lanzará una excepción), porque nadie lo invocará directamente sobre un objeto de su tipo. Será invocado sólo a través de una clase derivada, la cual, si se quiere que trabaje exitosamente, implementa a **Cloneable.**
5. Trate de impedir la clonación no implementando a **Cloneable** y sobrescribiendo **clone()** para lanzar una excepción. Esto tiene éxito sólo si cualquier clase derivada llama a **super.clone ()** en su redefinición de **clone().** De otra manera, un programador puede encontrar la manera de eludir esta situación.

6. Impida la clonación haciendo que su clase sea **final.** Si **clone()** no ha sido sobrescrito por alguna de sus clases ancestrales, ya no lo puede ser. Si lo ha sido, entonces sobrescríbalo otra vez y lance una excepción **CloneNotSupportedException.** Hacer la clase **final** es la única forma para garantizar que se impida la clonación. Además, cuando esté manejando objetos de seguridad u otras situaciones en las cuales usted quiere controlar el número de objetos creados, usted debería hacer a todos los constructores **private** y debería proveer uno o más métodos especiales para la creación de objetos. De ese modo, estos métodos pueden restringir el número de objetos creados y las condiciones en las cuales son creados. (Un caso particular de esto es el patrón *singleton* mostrado en *Thinking in Patterns (with Java)* en *www.BruceEckel.com*.)

Aquí hay un ejemplo que muestra las formas diversas como se puede implementar la clonación y luego, más adentro en la jerarquía, cómo "cancelarla":

```
//: apéndicea:CheckCloneable.java
// Verificando para ver si una referencia puede ser clonada.
import com.bruceeckel.simpletest.*;

// Esto no se puede clonar porque no anula a clone():
class Ordinary {}

// Sobreescribe clone, pero no implementa Cloneable:
class WrongClone extends Ordinary {
  public Object clone() throws CloneNotSupportedException {
    return super.clone(); // Lanza una excepción
  }
}

// Hace todo correcto para la clonación:
class IsCloneable extends Ordinary implements Cloneable {
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}

// Cancela la clonación mediante el lanzamiento de la excepción:
class NoMore extends IsCloneable {
  public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class TryMore extends NoMore {
  public Object clone() throws CloneNotSupportedException {
    // Llama a NoMore.clone(), lanza una excepción:
    return super.clone();
  }
}
```

```
class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // De alguna manera haga una copia de b y retórnela
    // Esta es una copia falsa, solo para marcar el punto:
    return new BackOn();
  }
  public Object clone() {
    // No llama a NoMore.clone():
    return duplicate(this);
  }
}

// Usted no puede heredar de esto, por lo tanto no puede
sobreescribir
// el método clone como sí lo puede hacer en BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  private static Test monitor = new Test();
  public static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    System.out.println("Attempting " + id);
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("Could not clone " + id);
      }
    } else {
      System.out.println("Doesn't implement Cloneable");
    }
    return x;
  }
  public static void main(String[] args) {
    // Casting inverso:
    Ordinary[] ord = {
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
    // Esto no compilará; clone() es protected en Object:
    //! x = (Ordinary)x.clone();
    // Primero verifica si una clase implementa Cloneable:
    for(int i = 0; i < ord.length; i++)
      tryToClone(ord[i]);
    monitor.expect(new String[] {
      "Attempting IsCloneable",
```

```
      "Cloned IsCloneable",
      "Attempting WrongClone",
      "Doesn't implement Cloneable",
      "Attempting NoMore",
      "Could not clone NoMore",
      "Attempting TryMore",
      "Could not clone TryMore",
      "Attempting BackOn",
      "Cloned BackOn",
      "Attempting ReallyNoMore",
      "Could not clone ReallyNoMore"
    });
  }
} ///:~
```

La primera clase, **Ordinary,** representa los tipos de clases que hemos visto a todo lo largo de este libro: ningún soporte para la clonación, pero al mismo tiempo, tampoco ninguna prevención de ella. Pero si usted tiene una referencia a un objeto **Ordinary** con casting inverso desde una clase más derivada, usted no puede establecer si puede o no ser clonado.

La clase **WrongClone** muestra una forma incorrecta de implementar la clonación. Sobreescribe **Object.clone ()** y convierte a ese método en **public**, pero no implementa **Cloneable,** así que cuando **super.clone ()** es llamado (lo que redunda en una llamada a **Object.clone ()**), se lanza la excepción **CloneNotSupportedException**, con lo que la clonación no surte efecto.

IsCloneable realiza todas las acciones correctas para llevar a cabo la clonación; se sobrescribe **clone()** y se implementa Cloneable. Sin embargo, este método **clone()** y varios otros que siguen en este ejemplo *no captura la excepción* **CloneNotSupportedException.** En lugar de eso lo pasa a quien llama, quien tiene entonces que colocar un bloque **try-catch** alrededor de él. En sus propios métodos **clone()** Usted típicamente capturará la excepción **CloneNotSupportedException()** en el *interior* de **clone**() en vez de pasarla. Como usted verá, en este ejemplo es más informativo pasar las excepciones hasta el final.

La clase **NoMore** intenta " poner fuera de servicio" la clonación de la manera que los diseñadores Java originalmente pensaron: en la clase derivada **clone()**, usted lanza la excepción **CloneNotSupportedException.** El método **clone()** en la clase **TryMore** llama correctamente a **super.clone (),** y este recurre a **NoMore.clone (),** el cual lanza una excepción e impide la clonación.

¿Pero qué ocurre si el programador no sigue el camino " correcto " de llamar a **super.clone ()** dentro del método **clone()** sobrescrito? En **BackOn,** usted puede ver cómo puede ocurrir esto. Esta clase usa un método **duplicate()** separado para hacer una copia del objeto actual y llama este método dentro de **clone()** *en lugar de* llamar a **super.clone ().** La excepción nunca se lanza y la clase nueva es clonable. Usted no puede confiar en lanzar una excepción para impedir hacer una clase clonable. La única solución de éxito asegurado se ejemplariza en **ReallyNoMore,** la cual es **final** y por ende no puede ser

hereradal. Eso significa que si **clone()** lanza una excepción en la clase **final,** esta no puede ser modificado con herencia, y la prevención de la clonación es asegurada. (Usted no puede llamar explícitamente a **Object.clone ()** desde una clase que tiene un nivel arbitrario de herencia; usted está limitado a llamar a **super.clone (),** el cual tiene acceso sólo a la clase base directa.) Así, si usted hace cualesquiera objetos que involucren asuntos de seguridad, usted querrá hacer esas clases **final.**

El primer método que usted ve en la clase **CheckCloneable** es **tryToClone (),** el cual toma cualquier objeto **Ordinary** y con **instanceof** verifica si es o no clonable**.** Si es así, le hace un casting a un **IsCloneable,** llama a **clone(),** y el resultado lo devuelve mediante casting a **Ordinary,** capturando cualesquiera excepciones que se lancen. Note el uso de identificación de tipo en tiempo de ejecución (RTTI; vea el Capítulo 10) para imprimir el nombre de clase con el fin de que pueda ver qué está sucediendo.

En **main()** se crean diferentes tipos de objetos **Ordinary** y en la definición del arreglo se les hace casting a **Ordinary**. Las primeras dos líneas de código después de eso crean un objeto **Ordinary** simple e intentan clonarlo. Sin embargo, este código no compilará porque **clone()** es un método **protected** en **Object.** El resto del código procesa el array y trata de clonar cada objeto, dando cuenta del éxito o el fracaso en cada uno.

Así para resumir, si usted quiere que una clase sea clonable:

1. Implemente la interfaz **Cloneable.**
2. Sobreescriba **clone().**
3. Llame a **super.clone()** dentro de su **clone().**
4. Capture las excepciones dentro de su **clone().**

Esto producirá los efectos más convenientes.

El constructor de la copia

Puede parecer que organizar la clonación sea un proceso complicado. Podría parecer que debería haber una alternativa. Una posibilidad es usar serialización, como se mostró anteriormente. Otro posibilidad que se le podría ocurrir a usted (especialmente si es un programador de C++) es hacer un constructor especial cuyo trabajo sea duplicar un objeto. En C++, esto se llama el constructor copia. Al principio, esto parece la solución obvia, pero en realidad no funciona. Aquí hay un ejemplo:

```
//: apéndicea:CopyConstructor.java
// Un constructor para copiar un objeto del mismo
// tipo, como un intento de crear una copia local.
```

```
import com.bruceeckel.simpletest.*;
import java.lang.reflect.*;

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc.
  public FruitQualities() { // Constructor por defecto
    // Haga algo significativo...
  }
  // Otros constructores:
  // ...
  // Constructor Copia:
  public FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc.
  }
}

class Seed {
  // Miembros...
  public Seed() { /* Constructor por defecto */ }
  public Seed(Seed s) { /* Constructor Copia*/ }
}

class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  public Fruit(FruitQualities q, int seedCount) {
    fq = q;
    seeds = seedCount;
    s = new Seed[seeds];
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed();
  }
  // Otros constructores:
  // ...
  // Constructor Copia:
  public Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    seeds = f.seeds;
    s = new Seed[seeds];
    // Llame a todos los constructores copia Semilla:
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed(f.s[i]);
```

```
    // Otras actividades de construcción de copias...
  }
  // Para permitir a los constructores derivados (o a otros
  // métodos) establecer cualidades diferentes:
  protected void addQualities(FruitQualities q) {
    fq = q;
  }
  protected FruitQualities getQualities() {
    return fq;
  }
}

class Tomato extends Fruit {
  public Tomato() {
    super(new FruitQualities(), 100);
  }
  public Tomato(Tomato t) { // Constructor Copia
    super(t); // Casting al constructor copia base
    // Otras actividades de construcción de copias...
  }
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  public ZebraQualities() { // Constructor por defecto
    super();
    // haga algo significativo...
  }
  public ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  public GreenZebra() {
    addQualities(new ZebraQualities());
  }
  public GreenZebra(GreenZebra g) {
    super(g); // Llama a Tomato(Tomato)
    // Reinstale las cualidades correctas:
    addQualities(new ZebraQualities());
  }
  public void evaluate() {
    ZebraQualities zq = (ZebraQualities)getQualities();
    // Haga algo con las cualidades
    // ...
  }
}

public class CopyConstructor {
  private static Test monitor = new Test();
  public static void ripen(Tomato t) {
```

```
    // Utilice el "Constructor Copia":
    t = new Tomato(t);
    System.out.println("In ripen, t is a " +
      t.getClass().getName());
  }
  public static void slice(Fruit f) {
    f = new Fruit(f); // Hmmm... funcionará esto?
    System.out.println("In slice, f is a " +
      f.getClass().getName());
  }
  public static void ripen2(Tomato t) {
    try {
      Class c = t.getClass();
      // Utilice el  "constructor-copia":
      Constructor ct = c.getConstructor(new Class[] { c });
      Object obj = ct.newInstance(new Object[] { t });
      System.out.println("In ripen2, t is a " +
        obj.getClass().getName());
    }
    catch(Exception e) { System.out.println(e); }
  }
  public static void slice2(Fruit f) {
    try {
      Class c = f.getClass();
      Constructor ct = c.getConstructor(new Class[] { c });
      Object obj = ct.newInstance(new Object[] { f });
      System.out.println("In slice2, f is a " +
        obj.getClass().getName());
    }
    catch(Exception e) { System.out.println(e); }
  }
  public static void main(String[] args) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    ripen2(tomato); // OK
    slice2(tomato); // OK
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    ripen2(g); // OK
    slice2(g); // OK
    g.evaluate();
    monitor.expect(new String[] {
      "In ripen, t is a Tomato",
      "In slice, f is a Fruit",
      "In ripen2, t is a Tomato",
      "In slice2, f is a Tomato",
      "In ripen, t is a Tomato",
      "In slice, f is a Fruit",
      "In ripen2, t is a GreenZebra",
      "In slice2, f is a GreenZebra"
    });
```

```
  }
} ///:~
```

Esto parece un poco extraño al principio. ¿Seguro, la fruta tiene calidades, pero por qué no simplemente colocar campos representando esas calidades directamente en la clase Fruit? Hay dos razones potenciales.

La primera es que Usted podría querer insertar o cambiar las calidades fácilmente. Note que **Fruit** tiene un método **protected addQualities( )** para permitir a clases derivadas hacer esto. (Usted podría pensar que lo lógico a hacer es tener un constructor **protected** en **Fruit** que tome un argumento **FruitQualities**, pero los constructores no heredan, por lo que no estaría disponible en clases de segundo nivel o de niveles más altos.) Al tener las calidades de la fruta en una clase separada y usar composición, Usted tiene mayor flexibilidad, incluyendo la habilidad de cambiar las calidades en la mitad del tiempo de vida de un objeto **Fruit** particular.

La segunda razón para hacer **FruitQualities** un objeto separado es para el caso en que Usted quiera añadir nuevas calidades o cambiar el comportamiento usando herencia y polimorfismo. Note que para **GreenZebra** (el cual *realmente* es un tipo de tomate – yo los he cultivado y son fabulosos), el constructor llama a **addQualities( )** y le pasa un objeto **ZebraQualities**, el cual es derivado de **FruitQualities**, para que pueda ser fijado a la referencia **FruitQualities** en la clase base. Naturalmente, cuando **GreenZebra** usa **FruitQualities**, tiene que hacerle casting hacia abajo al tipo correcto (como se vió en **evaluate( )**), pero siempre sabe que el tipo es **ZebraQualities**.

También verá Usted que hay una clase **Seed**, y que **Fruit** (que por definición tiene sus propias semillas)[119] contiene un arreglo de **Seed**s. [119]

Finalmente, note que cada clase tiene un constructor copia, y que cada uno debe llamar correctamente los constructores copia para la clase base y los objetos miembro con el fin de lograr una copia profunda. El constructor copia es evaluado dentro de la clase **CopyConstructor**. El método **ripen( )** toma un argumento **Tomato** y realiza una construcción-copia sobre éste con el fin de duplicar el objeto:

```
t = new Tomato(t);
```

mientras **slice( )** toma un objeto **Fruit** más genérico y también lo duplica:

```
f = new Fruit(f);
```

Estos son evaluados con diferentes clases de **Fruit** en **main( )**. Examinando el resultado, Usted puede ver el problema. Después de la construcción-copia que le ocurre a **Tomato** dentro de **slice( )**, el resultado ya no es más un objeto **Tomato**, es solo un **Fruit**. Ha perdido todas sus características de tomate**.** Adicionalmente, cuando toma un **GreenZebra**, ambos métodos **ripen( )** y **slice( )** lo convierten en un **Tomato** y un **Fruit**, respectivamente.

---

[3] *Excepto por el pobre aguacate, el cual ha sido reclasificado a simplemente "grasa."*

Así, desafortunadamente, la estrategia del constructor copia no nos es útil en Java cuando queremos hacer una copia local de un objeto.

Por qué sí funciona en C++ y no en Java?

El constructor copia es una parte fundamental de C++ ya que este hace automáticamente una copia local de un objeto. Sin embargo, el ejemplo precedente prueba que no funciona para Java. Por qué?. En Java todo lo que manipulamos es una referencia, pero en C++, Usted puede tener entidades tipo referencia y Usted *también* puede mover los objetos directamente. Para eso es para lo que el constructor copia sirve: cuando Usted quiere tomar un objeto y pasarlo por valor, duplicando así el objeto. Así que trabaja bien en C++, pero no debe olvidar que esta estrategia falla en Java, por lo tanto, no la use.

Clases de solo-lectura

Aunque la copia local producida por **clone( )** da los resultados deseados en los casos apropiados, es un ejemplo de querer forzar al programador (al autor del método) a ser responsable de prevenir los efectos no deseados del aliasing. Qué pasaría si Usted estuviera haciendo una biblioteca que es de propósito tan general y tan comúnmente usada que Usted no puede asumir que siempre será clonada en los lugares apropiados? O más factiblemente, qué pasaría si Usted quiere permitir aliasing por cuestiones de eficiencia – para prevenir la innecesaria duplicación de objetos – pero Usted no desea sus efectos colaterales negativos?

Una solución es crear objetos inmutables que pertenezcan a clases de solo-lectura. Usted puede definir una clase de tal manera que ningún método en la clase ocasione cambios al estado interno del objeto. En tal clase, el aliasing no tiene impacto alguno ya que Usted puede leer solo el estado interno, con lo que si muchas secciones de código están leyendo el mismo objeto, no se presenta problema alguno.

Como un ejemplo sencillo de objetos inmutables, la biblioteca estándar de Java contiene clases “envoltorio” para todos los tipos primitivos. Usted puede haber descubierto ya que, si quiere almacenar un **int** en un contenedor como por ejemplo un **ArrayList** (que solo toma **referencias de Objetos**), Usted puede envolver su **int** dentro de la clase **Integer** de la biblioteca estándar:

```
//: apéndicea:ImmutableInteger.java
// La clase Integer no puede ser cambiada.
import java.util.*;

public class ImmutableInteger {
  public static void main(String[] args) {
    List v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new Integer(i));
    // Pero, cómo cambia Usted el int dentro de Integer?
```

```
  }
} ///:~
```

La clase **Integer** (así como todas las clases "envoltorio" primitivas) implementan la inmutabilidad de una forma sencilla: No tiene método alguno que le permitan cambiar el objeto.

Si Usted necesita un objeto que tenga un tipo primitivo que pueda ser modificado, debe crearlo Usted mismo. Afortunadamente, esto es trivial. La siguiente clase usa las convenciones de nombres de los JavaBeans:

```
//: apéndicea:MutableInteger.java
// Una clase envoltorio modificable.
import com.bruceeckel.simpletest.*;
import java.util.*;

class IntValue {
  private int n;
  public IntValue(int x) { n = x; }
  public int getValue() { return n; }
  public void setValue(int n) { this.n = n; }
  public void increment() { n++; }
  public String toString() { return Integer.toString(n); }
}

public class MutableInteger {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    List v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new IntValue(i));
    System.out.println(v);
    for(int i = 0; i < v.size(); i++)
      ((IntValue)v.get(i)).increment();
    System.out.println(v);
    monitor.expect(new String[] {
      "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]",
      "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
    });
  }
} ///:~
```

IntValue puede ser aún más simple si no hay problemas de privacidad, la inicialización por defecto a cero es adecuada (con lo que no necesita entonces el constructor), y no está interesado en imprimirlo (con lo que no necesita **toString( )**):

```
class IntValue { int n; }
```

Extraer el elemento y hacerle casting es un poco torpe, pero eso es una característica de **ArrayList**, no de **IntValue**.

Creación de clases de solo lectura

Es posible crear sus propias clases de solo lectura. A continuación un ejemplo:

```
//: apéndicea:Immutable1.java
// Objetos que no pueden ser modificados son inmunes al aliasing.
import com.bruceeckel.simpletest.*;

public class Immutable1 {
  private static Test monitor = new Test();
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 multiply(int multiplier) {
    return new Immutable1(data * multiplier);
  }
  public static void f(Immutable1 i1) {
    Immutable1 quad = i1.multiply(4);
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
  }
  public static void main(String[] args) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
    monitor.expect(new String[] {
      "x = 47",
      "i1 = 47",
      "quad = 188",
      "x = 47"
    });
  }
} ///:~
```

Toda la información es **private**, y Usted verá que ninguno de los métodos **public** modifican la información. De hecho, el método que sí parece modificar un objeto es **mulitply( )**, pero este crea un nuevo objeto **Immutable1** y deja el original incolumne.

El método **f( )** toma un objeto **Immutable1** y lleva a cabo varias operaciones sobre el, y l salida de **main( )** demuestra que no hay cambio alguno en **x**. Así, el objeto de **x** podría ser aliased muchas veces sin peligro ya que la clase **Immutable1** está diseñada para garantizar que los objetos no puedan ser cambiados.

Los inconvenientes de la inmutabilidad

A primera vista crear una clase inmutable parece ser una solución elegante. Sin embargo, cuando quiera que necesite un objeto modificado de ese nuevo tipo, tiene que aguantarse la carga de la creación de un nuevo objeto así como generar potencialmente más frecuentes corridas de recolección de basura. Esto no es problema para algunas clases, pero para otras (tales como la clase **String**), esto es prohibitivamente costoso.

La solución es crear una clase compañera que *pueda* ser modificada. Luego, cuando Usted esté haciendo una gran cantidad de cambios, puede cambiarse a usar la clase compañera modificable y regresar a la inmodificable cuanto haya terminado.

El ejemplo previo puede ser cambiado para ejemplarizar esto:

```
//: apéndicea:Immutable2.java
// Una clase compañera para modificar objetos inmutables.
import com.bruceeckel.simpletest.*;

class Mutable {
  private int data;
  public Mutable(int initVal) { data = initVal; }
  public Mutable add(int x) {
    data += x;
    return this;
  }
  public Mutable multiply(int x) {
    data *= x;
    return this;
  }
  public Immutable2 makeImmutable2() {
    return new Immutable2(data);
  }
}

public class Immutable2 {
  private static Test monitor = new Test();
  private int data;
  public Immutable2(int initVal) { data = initVal; }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable2 add(int x) {
    return new Immutable2(data + x);
  }
  public Immutable2 multiply(int x) {
    return new Immutable2(data * x);
  }
  public Mutable makeMutable() {
    return new Mutable(data);
  }
  public static Immutable2 modify1(Immutable2 y) {
    Immutable2 val = y.add(12);
```

```
    val = val.multiply(3);
    val = val.add(11);
    val = val.multiply(2);
    return val;
  }
  // Esto produce el mismo resultado:
  public static Immutable2 modify2(Immutable2 y) {
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
  }
  public static void main(String[] args) {
    Immutable2 i2 = new Immutable2(47);
    Immutable2 r1 = modify1(i2);
    Immutable2 r2 = modify2(i2);
    System.out.println("i2 = " + i2.read());
    System.out.println("r1 = " + r1.read());
    System.out.println("r2 = " + r2.read());
    monitor.expect(new String[] {
      "i2 = 47",
      "r1 = 376",
      "r2 = 376"
    });
  }
} ///:~
```

Immutable2 contiene métodos que, como antes, preservan la inmutabilidad de los objetos al producir nuevos cada vez que se desee una modificación. Estos son los métodos **add( )** y **mulitply( )**. La clase compañera se llama **Mutable**, y tiene también métodos **add( )** y **multiply( )**, pero estos modifican el objeto **Mutable** en vez de crear uno nuevo. Adicionalmente, **Mutable** tiene un método para usar su información para producir un objeto **Immutable2** y viceversa.

Los dos métodos estáticos **modify1( )** y **modify2( )** muestran dos estrategias diferentes para producir el mismo resultado. En **modify1( )**, todo se hace dentro de la clase **Immutable2** y Usted puede ver que en el proceso se crean cuatro nuevos objetos **Immutable2**. (Y en cada ocasión que **val** es reasignada, el objeto previo se convierte en basura.).

En el método **modify2( )**, Usted puede observar que la primera acción es tomar a **y** de **Immutable2** y producir de esta un **Mutable**. (Esto es como llamar a **clone( )** tal como lo vio Usted antes, pero esta vez se crea un diferente tipo de objeto). A continuación el objeto **Mutable** se usa para realizar una gran cantidad de operaciones de cambio *sin* que se requiera la creación de muchos nuevos objetos. Finalmente, se revierte a un **Immutable2**. Aquí se crean dos objetos nuevos (el **Mutable** y el resultado, **Immutable2**) en vez de cuatro.

En consecuencia, esta estrategia tiene sentido cuando:

1. Usted necesita objetos inmutables y

2. Usted necesita a menudo hacer una gran cantidad de modificaciones o
3. Es costosos crear nuevos objetos inmutables.

Cadenas inmutables

Considere el código siguiente:

```
//: apéndicea:Stringer.java
import com.bruceeckel.simpletest.*;

public class Stringer {
  private static Test monitor = new Test();
  public static String upcase(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = new String("howdy");
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
    monitor.expect(new String[] {
      "howdy",
      "HOWDY",
      "howdy"
    });
  }
} ///:~
```

Cuando **q** se pasa a **upcase( )** en realidad es una copia de la referencia a **q**. El objeto al cual está conectada esta referencia permanece en una única localización física. Las referencias son copiadas a medida que se requieren y se manipulan.

Al mirar la definición de **upcase( )**, Usted puede ver que el nombre de la referencia que se pasa es **s**, y esta existe solo mientras se ejecuta el cuerpo de **upcase( )**. Cuando **upcase( )** termina, la referencia local **s** desaparece. **upcase( )** retorna el resultado el cual consiste en la cadena original con todos los caracteres en mayúsculas. Naturalmente, en realidad retorna una referencia al resultado. Pero resulta que la referencia que retorna es para un nuevo objeto, y el objeto **q** original se deja incolumne. Cómo sucede esto?.

Constantes implícitas

Si Usted dice:

```
String s = "asdf";
String x = Stringer.upcase(s);
```

realmente quiere que el método **upcase( )** *modifique* el argumento? En general, no, ya que un argumento usualmente parece al lector del código como

un pedazo de información proporcionada al método, no algo para ser modificado. Esto es una garantía importante, ya que hace al código más fácil de escribir y entender.

En C++, la disponibilidad de esta garantía fue tan importante que ameritó una palabra clave especial, **const**, que permitiera al programador asegurar que una referencia (puntero o referencia en C++) no podría ser usado para modificar el objeto original. Pero sin embargo, se requería que el programador en C++ fuera diligente y recordara usar **const** en todo lado. Esto puede causar confusión y además ser fácil de olvidar.

Sobrecargando '+' y el StringBuffer

Los objetos de la clase **String** están diseñados para ser inmutables usando la técnica de la clase acompañante mostrada previamente. Si Usted examina la documentación del JDK para la clase **String** (que se encuentra sumarizada un poco más adelante en este apéndice), verá que cada método en la clase que aparentemente modifica un **String** en realidad crea y retorna un objeto **String** completamente nuevo que contiene la modificación. El **String** original no se toca para nada. Así, no hay una característica en Java como la de **const** en C++ para hacer que el compilador soporte la inmutabilidad de sus objetos. Si lo desea, Usted tiene que construirla Usted mismo, como lo hace **String**.

Ya que los objetos **String** son inmutables, Usted puede hacer alias a un **String** particular cuantas veces quiera. Ya que es de solo lectura, no hay posibilidad alguna de que una referencia cambie algo que afecte a otras. Por lo tanto, un objeto de solo lectura resuelve bien el problema de aliasing

También parece posible manejar todos los casos en los que Usted necesite un objeto modificado mediante la creación de una completamente nueva versión del objeto con las modificaciones, tal como String lo hace. Sin embargo, para algunas operaciones esto no es eficiente. Un caso en particular es el operador `+´ que ha sido sobrecargado para los objetos String. Sobrecargar significa que se le ha dado un significado adicional cuando se use con una clase particular. (Los operadores '+' and '+=' para String son los únicos operadores sobrecargados en Java, y Java no le permite al programador sobrecargar ningún otro). 120

Cuando se usa con objetos **String**, el operador `+´ permite concatenar **Strings**:

---

[4] *C++ le permite al programador la sobrecarga de operadores cuando lo desee. Ya que esto puede ser a menudo un proceso complicado (ver el Capítulo 10 de* Thinking in C++, 2a edición, *Prentice Hall, 2000), los creadores de Java consideraron esto como una "mala" característica que no debería ser incluida en Java. No era tan malo sin embargo ya que terminaron haciéndolo ellos mismos, e irónicamente, la sobrecarga de operadores sería mucho más fácil de usar en Java que en C++. Esto se puede apreciar en Python (ver www.Python.org) el cual tiene recolección de basura y una sobrecarga de operadores bastante sencilla..*

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

Usted puede imaginar como *podría* esto suceder. La **String** “abc” podría tener un método **append( )** que crea un nuevo objeto **String** conteniendo a “abc” concatenada con el contenido de **foo**. El nuevo objeto **String** creará entonces un nuevo **String** que añada “def” y así sucesivamente.

Esto ciertamente funcionaría pero requeriría la creación de una gran cantidad de objetos **String** solo para conformar esta nueva **String** y luego Usted tendrá un puñado de objetos **String** intermedios a los que es necesario hacerles el proceso de recolección de basura. Sospecho que los creadores de Java intentaron primero esta estrategia (lo cual es una lección en diseño de software – Usted realmente no sabe nada sobre un sistema hasta que ensaya su código y logra algo que funcione). Sospecho también que descubrieron que su desempeño era inaceptable.

La solución es una clase compañera mutable similar a la mostrada previamente. Para **String**, esta clase compañera se llama **StringBuffer**, y el compilador automáticamente crea un **StringBuffer** para evaluar ciertas expresiones, en particular cuando se usan los operadores sobrecargados ‘+’ and ‘+=’ con objetos **String**. El siguiente ejemplo muestra lo que sucede:

```
//: apéndicea:ImmutableStrings.java
// Demostración de StringBuffer.
import com.bruceeckel.simpletest.*;

public class ImmutableStrings {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    String foo = "foo";
    String s = "abc" + foo + "def" + Integer.toString(47);
    System.out.println(s);
    // El "equivalente" usando StringBuffer:
    StringBuffer sb =
      new StringBuffer("abc"); // Crea el String!
    sb.append(foo);
    sb.append("def"); // Crea el String!
    sb.append(Integer.toString(47));
    System.out.println(sb);
    monitor.expect(new String[] {
      "abcfoodef47",
      "abcfoodef47"
    });
  }
} ///:~
```

En la creación del **String s**, el compilador está haciendo el equivalente aproximado del código subsiguiente que usa **sb**: se crea un **StringBuffer**, y se usa **append( )** para añadir nuevos caracteres directamente en el objeto **StringBuffer** (en vez de hacer nuevas copias cada vez). Mientras que esto es

más eficiente, vale la pena anotar que cada vez que Usted crea una cadena de caracteres enmarcada en comillas como **"abc"** y **"def"**, el compilador las convierte en objetos **String**. Por ende puede hacer más objetos creados de los que Usted espera, a pesar de la eficiencia aportada por **StringBuffer**.

Las clases String y StringBuffer

A continuación un vistazo general de los métodos disponibles tanto para **String** como para **StringBuffer**, con el fin de que pueda identificar la manera como interactúan. Estas tablas no contienen todos y cada uno de los métodos disponibles sino aquellos que son importantes para la discusión. Los métodos que están sobrecargados están resumidos en una sola línea.

Primero, la clase **String**:

Método	**Argumentos, Sobrecarga**	**Uso**
Constructor	Sobrecarga: por defecto, **String**, **StringBuffer**, arreglos **char**, arreglos **byte**.	Creación de objetos **String**.
length( )		Número de caracteres en el **String**.
charAt( )	**int** Indice	El caracter en una localización dentro del **String**.
getChars( ), getBytes( )	El principio y final desde donde copiasr, el arreglo dentro del cual copiar, un índice al interior del arreglo destino	Copia de **chars** o **bytes** hacia un arreglo externo.
toCharArray( )		Genera un **char[]** que contiene los caracteres en el **String**.
equals( ), equals-IgnoreCase( )	Una **String** con la cual comparar	Una verificacion de igualidad sobre los contenidos de las dos **Strings**.

Método	**Argumentos, Sobrecarga**	**Uso**
compareTo( )	Una **String** con la cual comparar.	El resultado es negativo, cero o positivo dependiendo del orden lexicográfico del **String** y del argumento. Las letras mayúsculas y minúsculas no son iguales !
regionMatches( )	Desplazamiento dentro de esta **String**, la otra **String** y su desplazamiento y longitud de comparación. La sobrecarga añade “ignore case.”	El resultado **booleano** indica si las regiones coinciden.
startsWith( )	**String** con la que podría empezar. La sobrecarga añade desplazamiento dentro del argumento.	El resultado **booleano** indica si la **String** empieza con el argumento.
endsWith( )	**String** que puede ser un sufijo de esta **String**	El resultado **booleano** indica si el argumento es un sufijo.
indexOf( ), lastIndexOf( )	Sobrecargado: **char**, **char** e índice de inicio, **String**, **String**, e índice de inicio.	Retorna -1 si no se encuentra el argumento dentro de esta **String**, de otra manera retorna el índice donde empieza el argumento. **lastIndexOf( )** busca hacia atrás empezando en el final.

Método	**Argumentos, Sobrecarga**	**Uso**
substring( )	Sobrecargado: índice de inicio, índice de inicio e índice de final.	Retorna un nuevo objeto **String** que contiene el conjunto de caracteres especificado.
concat( )	La **String** a concatenar.	Retorna un nuevo objeto **String** que contiene los caracteres del **String** original seguidos por los caracteres en el argumento.
replace( )	El viejo caracter a buscar y el nuevo con el que se reemplazará	Retorna un nuevo objeto **String** con los reemplazos hechos. Si no se encuentra concordancia alguna, usa el viejo objeto **String**.
toLowerCase( ) toUpperCase( )		Retorna un nuevo objeto **String** con todas las letras cambiadas a mayúsculas o a minúsculas. Si no es necesario hacer cambio alguno, usa el viejo objeto **String**.
trim( )		Retorna un nuevo objeto **String** con los espacios en blanco removidos del inicio y del final. Si no es necesario hacer cambio alguno, usa el viejo objeto **String**.

Método	**Argumentos, Sobrecarga**	**Uso**
valueOf( )	Sobrecargado: **Object**, **char[]**, **char[]** y desplazamiento y conteo, **boolean**, **char**, **int**, **long**, **float**, **double**.	Retorna un **String** conteniendo la representación en caracteres del argumento.
intern( )		Produce una y sola una referencia **String** por cada secuencia de caracteres única.

Como puede ver, cada método en **String** cuidadosamente retorna un nuevo objeto **String** cuando es necesario cambiar los contenidos originales. Note también que si esto no es necesario, el método retornará únicamente una referencia a la **String** original. Esto ahorra espacio de almacenamiento y sobrecarga de trabajo.

Ahora, la clase **StringBuffer**:

Método	**Argumentos, sobrecarga**	**Uso**
Constructor	Sobrecargado: por defecto, longitud del buffer a crear, **String** desde la cual crear.	Crear un nuevo objeto **StringBuffer**.
toString( )		Crear un **String** a partir de este **StringBuffer**.
length( )		Número de caracteres en el **StringBuffer**.
capacity( )		Retorna el número actual de espacios asignados.
ensure-Capacity( )	Entero que indica la capacidad deseada	Hace que el **StringBuffer** tenga por lo menos el número deseado de espacios.
setLength( )	Entero que indica la nueva longitud de la	Trunca o expande la cadena de

	cadena de caracteres en el buffer.	caracters previa. Si expande, completa el tamaño con nulls.
charAt( )	Entero que indica la localización del elemento deseado.	Retorna el **char** en esa localización en el buffer.
setCharAt( )	Entero que indica la localización del elemento deseado y el nuevo valor **char** para el elemento.	Modifica el valor en esa localización.
getChars( )	El inicio y final de donde se va a a copiar, el arreglo donde se va a copiar, un índice dentro del arreglo de destino.	Copiar **char**s en un arreglo externo. No hay **getBytes( )** como en **String**.
append( )	Sobrecargado: **Object**, **String**, **char[]**, **char[]** con desplazamiento y longitud, **boolean**, **char**, **int**, **long**, **float**, **double**.	El argumento es convertido en una cadena y añadido el final del buffer actual, aumentando este si es necesario
insert( )	Sobrecargado, cada uno con un primer argumento del desplazamiento a partir del cual comenzar a insertar: **Object**, **String**, **char[]**, **boolean**, **char**, **int**, **long**, **float**, **double**.	El segundo argumento se convierte en una cadena y se inserta en el buffer actual comenzando en el desplazamiento. Si se requiere, se aumenta el buffer.
reverse( )		Se invierte el orden de los caracteres en el buffer.

El método más comúnmente usado es **append( )**, el cual es utilizado por el compilador al evaluar expresiones de tipo **String** que contengan los operadores '+' and '+='. El método **insert( )** tiene un formato similar y ambos llevan a cabo manipulaciones significativas al buffer en vez de crear nuevos objetos.

Las cadenas son especiales

A este momento Usted ya ha visto que la clase **String** no es solo una clase más en Java. Hay muchos casos especiales en **String**, no siendo el menos

importante el que sea una clase original y fundamental para Java. Luego está el hecho de que una cadena de caracteres enmarcada en comillas es convertida a un objeto **String** por el compilador y por los operadores especiales sobrecargados '+' and '+='. En este apéndice Usted ha conocido los restantes casos especiales: la cuidadosamente construida inmutabilidad usando la clase compañera **StringBuffer** y alguna magia extra en el compilador.

Resúmen

Debido a que en Java todos los identificadores de objetos son referencias y a que cada objeto es creado sobre la marcha y recolectado como basura solo cuando ya no es usado más, la manera de manipular objetos cambia, especialmente al pasarlos y retornarlos. Por ejemplo, en C o C++, si Usted quiere inicializar un trozo de almacenamiento en un método, probablemente solicitaría que el usuario pase al método la dirección de ese trozo de almacenamiento. De otra manera, Usted tendría que preocuparse acerca de quién sería responsable de destruir ese almacenamiento. Así, la interfaz y el entendimiento de tales métodos es más complicado. Pero en Java, Usted nunca tiene que preocuparse sobre la responsabilidad o sí un objeto todavía existirá cuando sea necesitado. Esto ya ha sido resuelto para Usted. Usted puede crear un objeto en el momento en que se necesita (y no antes) y nunca preocuparse sobre la mecánica de pasar la responsabilidad por él; Usted simplemente pasa la referencia. Algunas veces la simplificación que esto da pasa desapercibida. En otras ocasiones, es asombrosa.

Las desventajas de esta magia subyacente son dos:

1. Siempre tiene que aceptar la desmejora en la eficiencia debido a la administración extra de memoria (aunque la desmejora puede ser bastante pequeña), y siempre hay una ligera cantidad de incertidumbre sobre el tiempo que algo puede requerir para correr (ya que el colector de basura puede ser forzado a actuar cuando quiera que Usted esté bajo de memoria). En la mayoría de las aplicaciones, los beneficies superan las desventajas y las tecnologías de mejoramiento en particular han acelerado las cosas hasta el punto de que este asunto ya no es importante.
2. Aliasing: algunas veces Usted puede terminar con dos referencias al mismo objeto, lo cual es un problema solo si las dos referencias se supone que están apuntando a objetos *distintos*. Aquí es cuando Usted debe prestar mayor atención y, si es necesario, **clone( )** o de otra forma duplicar un objeto para prevenir que la otra referencia se vea sorprendida por un cambio inesperado. Alternativamente, Usted puede apoyar al aliasing por asuntos de eficiencia al crear objetos inmutables cuyas operaciones pueden retornar un nuevo objeto del mismo u otro tipo diferente, pero nunca cambiar el objeto original de tal manera que cualquiera que esté aliased a ese objeto no vea cambio alguno.

Algunas personas dicen que la clonación en Java es un diseño chapucero que no debería ser usado, así que implementan su propia versión de clonación[121] y nunca llaman el método **Object.clone( )**, eliminando así la necesidad de implementar **Cloneable** y de capturar la excepción **CloneNotSupportedException**. Esto es ciertamente una estrategia razonable, y ya que **clone( )** es soportado tan raras veces en la librería estándar de Java, aparentemente es también una estrategia segura.

Ejercicios

Las soluciones a ejercicios seleccionados se pueden encontrar en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible por una módica tarifa en *www.BruceEckel.com.*

1. Demostrar un segundo nivel de aliasing. Crear un método que tome una referencia a un objeto pero que no modifique al objeto de esa referencia. Sin embargo, el método debe llamar a un segundo método pasando a éste la referencia, y este segundo método debe modificar el objeto.

2. Crear una clase **MyString** que contenga un objeto **String** el que Usted inicializa en el constructor usando el argumento del constructor. Añada un método **toString( )** y un método **concatenate( )** que adiciones un objeto **String** a su cadena interna. Implemente **clone( )** en **MyString**. Cree dos métodos **static** donde cada uno tome una referencia **MyString x** como argumento y llame a **x.concatenate("test")**, pero en el segundo método llame primero a **clone( )**. Ensaye los dos métodos y muestre los diferentes resultados.

3. Cree una clase llamada **Battery** conteniendo un **int** el cual es un número de batería (como un identificador único). Hagala clonable y déle un método **toString( )**. Ahora cree una clase llamada **Toy** que contenga un arreglo de **Battery** y un **toString( )** que imprima todas las baterías. Escriba un **clone( )** para **Toy** que clone automáticamente todos sus objetos **Battery**. Pruebe esto clonando a **Toy** e imprimiento el resultado.

4. Cambie **CheckCloneable.java** de tal forma que todos los métodos **clone( )** capturen la excepción **CloneNotSupportedException** en vez de pasarla al llamador.

5. Usando la técnica de la clase-acompañante-mutable, haga una clase inmutable que contenga un **int**, un **double**, y un arreglo de **char**.

---

[5] *Doug Lea, quien fue de ayuda resolviendo este asunto, me lo sugirió, diciendo que el simplemente crea en cada clase una función llamada* ***duplicate( )*** *.*

6. Modifique **Compete.java** para añadir más objetos miembro a las clases **Thing2** y **Thing4** y vea si puede determinar cómo cambian los tiempos con la complejidad—si es una simple relación lineal o si parece más complicada.

7. Comenzando con **Snake.java**, cree una versión de copia profunda de la serpiente.

8. Implemente la interfaz **Collection** en una clase llamada **CloningCollection** usando un **private ArrayList** para proveer la funcionalidad del contenedor. Anule el método **clone( )** de tal forma que **CloningCollection** lleve a cabo una "copia profunda condicional"; intenta hacer **clone( )** a todos los elementos que contiene, pero si no puede, deja la(s) referencia(s) aliased.

B: El Interfaz Nativo Java (JNI [1])

El material de este apéndice se incorporó y usó con el permiso de Andrea Provaglio (www.AndreaProvaglio.com).

El lenguaje Java y su API estándar son lo suficientemente ricos como para escribir aplicaciones completas. Pero en ocasiones hay que llamar a código no-Java; por ejemplo, si se desea acceder a aspectos específicos del sistema operativo, interactuar con dispositivos hardware especiales, reutilizar código base preexistente no-Java, o implementar secciones de código críticas en el tiempo.

Interactuar con código no-Java requiere de soporte dedicado por parte del compilador y de la Máquina Virtual, y de herramientas adicionales para establecer correspondencias entre el código Java y el no-Java. La solución estándar para invocar a código no-Java proporcionada por JavaSoft es el ***Interfaz Nativo Java,*** que se presentará en este apéndice. No se trata de ofrecer aquí un tratamiento en profundidad, y en ocasiones se asume que uno tiene un conocimiento parcial de los conceptos y técnicas involucrados.
JNI es una interfaz de programación bastante rica que permite la creación de métodos nativos desde una aplicación Java. Se añadió en Java 1.1, manteniendo cierto nivel de compatibilidad con su equivalente en Java 1.0: el interfaz nativo de métodos (NMI). NMI tiene características de diseño que lo hacen inadecuado para ser adoptado en todas las máquinas virtuales. Por ello, las versiones futuras del lenguaje podrían dejar de soportar NMI, y éste no se cubrirá aquí.
Actualmente, JNI está diseñado para interactuar con métodos nativos escritos únicamente en C o C++. Utilizando JNI, los métodos nativos pueden:

- Crear, inspeccionar y actualizar objetos Java (incluyendo arrays y Strings).
- Invocar a métodos Java
- Capturar y lanzar excepciones
- Cargar clases y obtener información de clases
- Llevar a cabo comprobación de tipos en tiempo de ejecución.

Por tanto, casi todo lo que se puede hacer con las clases y objetos ordinarios de Java también puede lograrse con métodos nativos.

[1] N. del **traductor:** En **inglés:** ***Java Native Interface.***

Invocando a un método nativo

Empezaremos con un ejemplo simple: un programa en Java que invoca a un método nativo, que de hecho llama a la función ejemplo **printf(**) de la biblioteca estándar de C.

El primer paso es escribir el código Java declarando un método nativo y sus argumentos:

```
//: appendixb:ShowMessage.java
public class ShowMessage {
  private native void ShowMessage(String msg);
  static {
    System.loadLibrary("MsgImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
    //  "/home/bruce/tij2/appendixb/MsgImpl.so");
  }
  public static void main(String[] args) {
    ShowMessage app = new ShowMessage();
    app.ShowMessage("Generated with JNI");
  }
} ///:~
```

A la declaración del método nativo sigue un bloque **static** que invoca a **System.loadLibrary(**) (a la que se podría llamar en cualquier momento, pero este estilo es más adecuado). **System.loadLibrary(**) carga una DLL en memoria enlazándose a ella. La DLL debe estar en la ruta de la biblioteca del sistema. La JVM añade la extensión del nombre del archivo automáticamente en función de la plataforma.

En el código de arriba también se puede ver una llamada al método **System.load(**), marcada como comentario. La trayectoria especificada en este caso es absoluta en vez de radicar en una variable de entorno. Naturalmente, usar ésta última aporta una solución más fiable y portable, pero si no se puede averiguar su valor se puede comentar **loadLibrary(**) e invocar a esta línea, ajustando la trayectoria al directorio en el que resida el archivo en cada caso.

El generador de cabeceras de archivo: javah

Ahora, compile el archivo fuente Java y ejecute **javah** sobre el archivo **.class** resultante, especificando la opción **-jni** (el ***makefile*** que acompaña a la

distribución de código fuente de este libro se encarga de hacer esto automáticamente):

```
javah –jni ShowMessage
```

javah lee el archivo de clase Java y genera un prototipo de función en un archivo de cabecera C o C++ por cada declaración de método nativo. He aquí la salida: el archivo fuente **MostrarMensaje.h** (editado de forma que entre en este libro):

```
/* DO NOT EDIT THIS FILE
   - it is machine generated */
#include <jni.h>
/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ShowMessage
 * Method:    ShowMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Como puede verse por la directiva del preprocesador **#ifdef-cplusplus**, este archivo puede compilarse con un compilador de C o de C++. La primera directiva **#include** incluye **jni.h,** un archivo de cabecera que, entre otras cosas, define los tipos que pueden verse utilizados en el resto del archivo. **JNIEXPORT** y **JNICALL** son macros que se expanden para hacer coincidir directivas específicas de la plataforma. **JNIEnv, jobject** y **jstring** son definiciones de tipos de datos JNI, que se explicarán en breve.

Renombrado de nombres y signaturas de funciones

JNI impone una convención de nombres (denominada ***renombrado de nombres)*** a los métodos nativos. Esto es importante, pues es parte del

mecanismo por el que la máquina virtual enlaza las llamadas a Java con métodos nativos. Básicamente, todos los métodos nativos empiezan con la palabra "Java", seguida del nombre de la clase en la que aparece la declaración nativa Java, seguida del nombre del método Java. El carácter "guión bajo" se usa como separador. Si el método nativo Java está sobrecargado, se añade también la signatura de la función al nombre; puede verse la signatura nativa en los comentarios que preceden al prototipo. Para obtener más información sobre ***renombrado de nombres y*** signaturas de métodos nativos, por favor acudir a la documentación JNI.

Implementando la DLL

En este momento, todo lo que hay que hacer es escribir archivos de código fuente en C y C++ que incluye el archivo de cabecera generado por **javah**, que implementa el método nativo, después compilarlo y generar una biblioteca de enlace dinámico. Esta parte es dependiente de la plataforma. El código de debajo se compila y enlaza a un archivo que en Windows se llama **MsgImpl.dl1** y en Unix/Linux **MsgImpl.so** (el *makefile* empaquetado junto con los listados de código contiene los comandos para hacer esto -está disponible en el CD ROM vinculado a este libro, o como descarga gratuita de www.BruceEcKel.com):

```
//: appendixb:MsgImpl.cpp
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
#include <stdio.h>
#include "ShowMessage.h"

extern "C" JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
  printf("Thinking in Java, JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

Los parámetros que se pasan al método nativo son la pasarela que permite la vuelta a Java. El primero, de tipo **JNIEnv**, contiene todos los anzuelos que te permiten volver a llamar a la JVM. (Echaremos un vistazo a esto en la sección siguiente.) El segundo parámetro tiene significado distinto en función del tipo de método. En el caso de métodos no **static** como el del ejemplo de arriba, el segundo parámetro es equivalente al puntero "**this**" de C++ y similar a **this** en Java: es una referencia al objeto que invocó al método nativo. En el caso de

métodos **static**, es una referencia al objeto **Class** en el que está implementado el método.

Los demás parámetros representan los objetos Java que se pasan a la llamada al método nativo.

También se pasan tipos de datos primitivos así, pero vienen por valor.

En las secciones siguientes, explicaremos este código mirando a las formas de acceder y controlar la JVM desde dentro de un método nativo.

Accediendo a funciones JNI: el parámetro JNIEnv

Las funciones JNI son aquéllas que usa el programador para interactuar con la JVM desde dentro de un método nativo. Como puede verse en el ejemplo de arriba, todo método JNI nativo recibe un parámetro especial en primer lugar: el parámetro **JNIEnv**, que es un puntero a una estructura de datos especial de JNI de tipo **JNIENv**. Un elemento de la estructura de datos JNI es un puntero a un array generado por la JVM. Cada elemento de este array es un puntero a una función JNI. Las funciones JNI pueden ser invocadas desde el método nativo desreferenciando estos punteros (es más simple de lo que parece). Toda JVM proporciona su propia implementación de las funciones JNI, pero sus direcciones siempre estarán en desplazamientos predefinidos.

A través del parámetro **JNIEnv_**, el programador tiene acceso a un gran conjunto de funciones. Estas funciones pueden agruparse en las siguientes categorías:

- Obtener información de la versión
- Llevar a cabo operaciones de clase y objetos
- Acceder a campos de instancia y campos estáticos
- Llamar a métodos de instancia y estáticos
- Llevar a cabo operaciones de **Strings** y **arrays**
- Generar y gestionar excepciones Java

El número de funciones JNI es bastante grande y no se cubrirá aquí. Sin embargo, mostraré el razonamiento que hay tras el uso de estas funciones. Para obtener información más detallada, consulte la documentación JNI del compilador.

Si se echa un vistazo al archivo de cabecera **jni.h,** se verá que dentro de la condicional de preprocesador **#ifdef_cplusplus**, se define la estructura **JNIEnv_** como una clase cuando se compile por un compilador de C++. Esta clase contiene

varias funciones que te permiten acceder a las funciones JNI con una sintaxis sencilla y familiar. Por ejemplo, la línea de código C++ del ejemplo anterior:

```
env->ReleaseStringUTFChars(jMsg, msg);
```

También podría haber sido invocada desde C así:

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

Se verá que el estilo de C es (naturalmente) más complicado -se necesita una desreferencia doble del puntero **env**, y se debe pasar también el nombre del puntero como primer parámetro a la llamada a la función JNI. Los ejemplos de este apéndice usan el ejemplo de C++.

Accediendo a Strings Java

Como ejemplo de acceso a una función JNI, considérese el código de **MsgImpl.cpp.** Aquí, se usa el argumento **JNIEnv env** para acceder a un **String** Java. Éstos están en formato Unicode, por lo que si se recibe uno y se desea pasarlo a una función no-Unicode **(printf(**), por ejemplo) primero hay que convertirlo a caracteres ASCII con la función JNI **GetStringUTFChars(**). Esta función toma un **String** Java y lo convierte a UTF de 8 caracteres. (Estos 8 bits son suficientes para almacenar valores ASCII, o 16 bits para almacenar Unicode. Si el contenido de la cadena de caracteres original sólo estaba compuesto de caracteres ASCII, la cadena resultante también estará en ASCII.)

GetStringUTFChars( ) es una de las funciones miembro de **JNIEnv.** Para acceder a la función JNI usamos la sintaxis típica de C++ para llamar a funciones miembro mediante un puntero. La forma de arriba se usa para acceder a todas las funciones JNI.

Pasando y usando objetos Java

En el ejemplo anterior, se pasaba un **String** al método nativo. También se pueden pasar objetos Java de tu creación al método nativo. Dentro de éste, se puede acceder a los campos y métodos del objeto que se recibió.

Para pasar objetos, puede usarse la sintaxis general de Java al declarar el método nativo. En el ejemplo de abajo, **MiClaseJava** tiene un campo **public** y un método **public.** La clase **UsarObjetos** declara un método nativo que toma un objeto de clase **MiClaseJava.** Para ver si el método nativo manipula su parámetro, se pone el campo **public** del parámetro, se invoca al método nativo, y después se imprime el valor del campo **public:**

```
//: appendixb:UseObjects.java
class MyJavaClass
```

```
{
     public int aValue;
     public void divByTwo() { aValue /= 2; }
}

public class UseObjects
{
     private native void
          changeObject(MyJavaClass obj);
     static
     {
          System.loadLibrary("UseObjImpl");
          // Linux hack, if you can't get your library
          // path set in your environment:
          // System.load(
          //"/home/bruce/tij2/appendixb/UseObjImpl.so");
     }
     public static void main(String[] args)
     {
          UseObjects app = new UseObjects();
          MyJavaClass anObj = new MyJavaClass();
          anObj.aValue = 2;
          app.changeObject(anObj);
          System.out.println("Java: " + anObj.aValue);
     }
} ///:~
```

Tras compilar el código y ejecutar **javah,** se puede implementar el método nativo. En el ejemplo de abajo, una vez que se obtienen el campo y el ID del método, se acceden a través de funciones JNI:

```
//: appendixb:UseObjImpl.cpp
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UseObjects_changeObject(
JNIEnv* env, jobject, jobject obj) {
  jclass cls = env->GetObjectClass(obj);
  jfieldID fid = env->GetFieldID(
    cls, "aValue", "I");
  jmethodID mid = env->GetMethodID(
    cls, "divByTwo", "()V");
  int value = env->GetIntField(obj, fid);
  printf("Native: %d\n", value);
  env->SetIntField(obj, fid, 6);
  env->CallVoidMethod(obj, mid);
  value = env->GetIntField(obj, fid);
```

```
  printf("Native: %d\n", value);
} ///:~
```

Ignorando el equivalente "**this**", la función C++ recibe un **jobject**, que es el lado nativo de la referencia al objeto Java que le pasamos desde el código Java. Simplemente leemos un valor, la imprimimos, cambiamos el valor, se llama al método **divByTwo**( ) del objeto, y se imprime de nuevo el valor.

Para acceder a un campo o método Java, primero hay que obtener su identificador usando **GetFieldID**( ) en el caso de los campos y **GetMethodID**( ) en el caso de los métodos. Estas funciones toman el objeto clase, una cadena de caracteres que contiene el nombre del elemento, y otra cadena que proporciona información de tipos: el tipo de datos del campo, o información de signatura en el caso de un método (pueden encontrarse detalles en la documentación de JNI).

Estas funciones devuelven un identificador que se usa para acceder al elemento. Este enfoque podría parecer complicado, pero el método nativo desconoce la disposición interna del objeto Java. En su lugar, debe acceder a los campos y métodos a través de índices devueltos por la JVM. Esto permite que distintas JVMs implementen disposiciones internas de objetos distintas sin que esto afecte a los métodos nativos.

Si se ejecuta el programa Java, se verá que el objeto pasado desde el lado Java es manipulado por el método nativo. Pero ¿qué es exactamente lo que se pasa? ¿Un puntero o una referencia Java? Y ¿qué hace el recolector de basura durante llamadas a métodos nativos?

El recolector de basura sigue operando durante la ejecución de métodos nativos, pero está garantizado que no eliminará ningún objeto durante la llamada a un método nativo. Para asegurar esto, se crean previamente referencias locales, que son destruidas inmediatamente después de la llamada al método nativo. Dado que su tiempo de vida envuelve la llamada, se sabe que los objetos serán válidos durante toda la llamada al método nativo.

Dado que estas referencias son creadas y destruidas subsecuentemente cada vez que se llama a la función, no se pueden hacer copias locales en los métodos nativos, en variables **static.** Si se desea una referencia que perdure a través de invocaciones a funciones, se necesita una referencia global.

Éstas no las crea la JVM, pero el programador puede hacer una referencia global a partir de una local llamando a funciones de JNI específicas. Cuando se crea una referencia global, uno es responsable de la vida del objeto referenciado. La referencia global (y el objeto al que hace referencia) estarán en memoria hasta que el programador libere la referencia explícitamente con la función JNI apropiada. Es semejante al **malloc(**) y **free(**) de C.

JNI y las excepciones Java

Con JNI, pueden lanzarse, capturarse, imprimirse y relanzarse excepciones Java exactamente igual que si se estuviera dentro de un programa Java. Pero depende del programador el invocar a funciones JNI dedicadas a tratar las excepciones. He aquí las funciones JNI para la gestión de excepciones:

- **Throw( )**
  Lanza un objeto excepción existente. Se usa en los métodos nativos para relanzar una excepción.

- **ThrowNew( )**
  Genera un nuevo objeto excepción y lo lanza.

- **ExceptionOccurred( )**
  Determina si se lanzó una excepción aún sin eliminar.

- **ExceptionDescribe( )**
  Imprime una excepción y la traza de la pila.

- **ExceptionClear( )**
  Elimina una excepción pendiente.

- **FatalError( )**
  Lanza un error fatal. No llega a devolver nada.

Entre éstas, no se puede ignorar **ExceptionOcurred**( ) y **ExceptionClear**( ). La mayoría de funciones JNI pueden generar excepciones y no hay ninguna faceta del lenguaje que pueda usarse en el lugar de un bloque *try* de Java, por lo que hay que llamar a **ExceptionOccurred**( ) tras cada llamada a función JNI para ver si se lanzó alguna excepción. Si se detecta una excepción, se puede elegir manejarla (y posiblemente relanzarla). Hay que asegurarse, sin embargo, de que la excepción sea siempre eliminada. Esto puede hacerse dentro de la función **ExceptionClear**( ) o en alguna otra función si se relanza la excepción, pero hay que hacerlo.

Hay que asegurar que se elimine la excepción, pues si no, los resultados serían impredecibles si se llama a una función JNI mientras está pendiente una excepción. Hay pocas funciones JNI que pueden ser invocadas de forma segura durante una excepción; entre éstas, por supuesto, se encuentran las funciones de manejo de excepciones.

JNI y los hilos

Dado que Java es un lenguaje multihilo, varios hilos pueden invocar a métodos nativos de forma concurrente. (El método nativo podría suspenderse en el medio de su operación al ser invocado por un segundo hilo.) Depende enteramente del programador el garantizar que la llamada nativa sea inmune a los hilos; por ejemplo, no modifica datos compartidos de forma no controlada. Básicamente, se tienen dos opciones: declarar el método nativo como **synchronized**, o implementar alguna otra estrategia dentro del método nativo para asegurar una manipulación de datos concurrentes correcta.

También se podría no pasar nunca el puntero **JNIEnv** por los hilos, puesto que la estructura interna a la que apunta está ubicada en una base hilo a hilo y contiene información que sólo tiene sentido en cada hilo en particular.

Usando un código base preexistente

La forma más sencilla de implementar métodos JNI nativos es empezar a escribir prototipos de métodos nativos en una clase Java, compilar la clase y ejecutar el archivo **.class** con **javah**. Pero ¿qué ocurre si se tiene un código base grande preexistente al que se desea invocar desde Java? Renombrar todas las funciones de las DLLs para que coincidan con la convención de nombres de JNI no es una solución viable. El mejor enfoque es escribir una DLL envoltorio "fuera" del código base original. El código Java llamaría a funciones de esta nueva DLL, que a su vez invoca a funciones de la DLL original. Esta solución no es sólo un rodeo; en la mayoría de casos hay que hacerlo así siempre porque hay que invocar a funciones JNI con referencias a objetos antes de su uso.

Información adicional

Puede encontrarse más material introductorio, incluyendo un ejemplo en C (en vez de en C++) y una discusión de aspectos Microsoft en el Apéndice A de la primera edición de este libro, que puede encontrarse en el CD ROM que acompaña a este libro, o como descarga gratuita de *www.BruceEckel.com.* Hay información más extensa disponible en *java.sun.com.* (en el motor de búsqueda, seleccione las palabras clave "native methods" dentro de "training & tutorials"). El Capítulo 11 de *Core Java* 2, *Volume 11,* por Horstmann & Cornell (Prentice-Hall, 2000) da una cobertura excelente a los métodos nativos.

C: Consejos para la programación en Java

Este apéndice contiene sugerencias que te ayudarán y guiarán en la realización de diseños de bajo nivel de programas y en la escritura de código.

Por supuesto, que esto son sólo consejos no reglas. La idea es utilizarlos como inspiración y recordar que en ocasiones hay situaciones donde deben de ser flexibles o incluso donde deben no ser seguidos.

Diseño

1. **La elegancia siempre trae a cuenta**. En el corto plazo podría parecer que lleva mucho más tiempo obtener una solución verdaderamente elegante a un problema, pero cuando funciona la primera vez y se adapta con facilidad a las nuevas situaciones en lugar de ser necesarias horas, días o meses de dura lucha, verás las recompensas (incluso aunque nadie pueda medirlas). No sólo te da un programa que es más sencillo de hacer y depurar, también será más fácil de entender y mantener, y ahí es donde está el valor financiero. Puede que necesites alguna experiencia para entender este punto, porque te parecerá que no eres productivo haciendo una porción de código elegante. Resiste la urgencia de la prisa; sólo te hará ir más lento.
2. **Consigue que vaya primero, haz que vaya rápido luego**. Esto es cierto incluso cuando estés seguro de que una porción del código es realmente importante y que será un cuello de botella importante en tu sistema. No lo hagas. Consigue primero que el sistema funcione con un diseño tan sencillo como sea posible. Entonces, si no va suficientemente rápido, perfílalo. Casi siempre descubrirás que "tu" cuello de botella no es el problema. Ahorra tu tiempo para las cuestiones realmente importantes.
3. **Recuerda el principio "divide y vencerás"**. Si el problema al que te enfrentas es demasiado confuso, intenta imaginar cuál debería ser el funcionamiento básico del sistema suponiendo que existiera una "pieza" que se ocupase de las partes complicadas. Haz de esa "pieza" un objeto—escribe el código que utilice ese objeto, luego estudia el objeto y encapsula *sus* partes complicadas en otros objetos, etc.
4. **Separa la clase creadora de la utilizadora (*el programador cliente*)**. La clase utilizadora es la "cliente" y no necesita ni quiere saber qué ocurre tras las bambalinas. La clase creadora debe ser experta en el diseño de la clase y escribir la clase de forma que pueda ser utilizada por el programador más inexperto que sea posible y, aún así, funcionar en la aplicación de forma robusta. Piensa en la clase como un *proveedor de servicios* de otras clases. La utilización de bibliotecas será sencilla sólo si es transparente.
5. **Cuando creas una clase, intenta hacer que tus nombres sean tan claros que los ios sean innecesarios**. Tu objetivo debería ser hacer la interfaz de programación con el cliente conceptualmente simple. Para ello, utiliza la sobrecarga de métodos cuando sea aconsejable para crear un interfaz intuitivo y fácil de usar.
6. **Tu análisis y diseño deben producir, como mínimo, las clases de tu sistema, sus interfaces públicas y las relaciones que tienen con otras**

clases. Si tu metodología de diseño produce más, pregúntate si todo lo que produce esa metodología tiene valor durante la vida del programa. Si no lo tiene, el mantener esos productos te costará trabajo. Los miembros de un equipo de desarrollo tienden a no mantener nada que no contribuya a aumentar su productividad; es un hecho de la vida que muchos métodos de diseño no tienen en cuenta.

7. **Automatízalo todo**. Escribe primero el código de prueba (antes de escribir la clase), y mantenlo junto a la clase. Automatiza la ejecución de tus pruebas con la herramienta de montaje—probablemente quieras utilizar Ant, la herramienta de montaje que es el estándar de facto para Java. De esta manera, los cambios que hagas serán verificados de forma automática por el código de prueba y descubrirás los errores inmediatamente. Como sabes que tienes la red de seguridad de tu infraestructura de pruebas, serás más atrevido cuando descubras la necesidad de hacer cambios importantes. Recuerda que las mejoras más importantes que se han hecho a los lenguajes de programación vienen de la prueba integrada que ofrece la comprobación de tipos, la gestión de excepciones, etc., pero que esas características no permiten avanzar más allá. Eres tú quien debe hacer el resto del camino creando un sistema robusto añadiendo las pruebas que verifican las características que son específicas a tu clase o programa.
8. **Escribe primero el código de pruebas (antes de escribir la clase) de forma que puedas verificar que el diseño de tu clase está completo**. Si no puedes escribir el código de prueba, no sabes que aspecto tiene tu clase. Además, el acto de escribir el código de pruebas a menudo entresaca características o restricciones de tu clase—estas características o constantes no siempre aparecen durante las fases de análisis y diseño. Las pruebas también te permiten tener código de ejemplo que muestre cómo se puede utilizar tu clase.
9. **Todo problema de diseño se puede simplificar introduciendo un nivel extra de indirección conceptual**. Esta regla fundamental de la ingeniería del software **[122]** es la base de la abstracción, la principal característica de la programación orientada a objetos. En términos de la POO lo podríamos expresar como: "Si tu código es demasiado complicado, crea más objetos."

[122] Esto me lo explicó Andrew Koenig.

10. **Una indirección debería significar algo** (en relación con el consejo número 9). El significado puede ser algo tan simple como "aglutina código que se utiliza a menudo en un único método." Añadir niveles de indirección (abstracción, encapsulación, etc.) que no significan nada puede ser tan negativo como no hacer una indirección adecuada.
11. **Haz clases tan atómicas como sea posible**. Haz que cada clase tenga un propósito sencillo y claro—un servicio cohesivo que ofrecer a otras clases. Si tus clases o tu sistema se hace demasiado complicado, divide las clases complejas en clases más simples. El indicador más obvio de esto es el tamaño bruto; si una clase es grande, tiene todas las papeletas para que esté haciendo demasiado y deba considerarse su descomposición.
    Las pistas que sugieren el rediseño de una clase son las siguientes:
    1) Una instrucción complicada: considera el uso del polimorfismo.
    2) Un gran número de métodos que cubren de forma genérica diversos tipos de operaciones: considera el uso de varias clases.
    3) Un alto número de variables miembro que conciernen de a diversas características forma genérica: considera utilizar varias clases.
    4) Puedes encontrar otras sugerencias sobre el tema en el libro de Martin Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).

12. **Vigila las listas de argumentos largas**. Entonces las llamadas a los métodos pueden hacerse difíciles de escribir, leer y mantener. En vez de ello, intenta mover el método a otra clase donde su uso sea (más) apropiado y/o pásale los objetos como argumentos.
13. **No te repitas**. Si una porción de código aparece de forma repetida en muchos métodos de las clases derivadas, pon ese código en un único método de la clase base y utilízalo desde los métodos de las clases derivadas. No solo ahorrarás espacio, sino que conseguirás una propagación de cambios más sencilla. A veces, el descubrimiento de este código común añadirá una funcionalidad valiosa al interfaz de la clase. Una versión más sencilla de es consejo también se da en ejemplos sin herencia: si una clase tiene métodos que repiten código, factoriza ese código en un método común y utilízalo desde los restantes métodos.
14. **Vigila las sentencias con cláusulas *switch* y *if-else* en cadena**. Suelen ser un indicador típico de *código de comprobación de tipos*, que implica que estás seleccionando que código se debe ejecutar en función de información de algún tipo (el tipo de información exacto puede no ser obvio al principio). Normalmente puedes remplazar este tipo de código mediante el uso de la herencia y el polimorfismo; una llamada a un método polimórfico realizará la selección del tipo por ti y conseguirá un sistema más fiable y más fácil de extender.
15. **Desde la perspectiva del diseño, busca y separa aquellas cosas que cambian de las que permanecen inalterables**. Esto es, busca los elementos del sistema que quisieras cambiar sin forzar un rediseño, luego encapsula dichos elementos en clases. Puedes aprender mucho de este concepto en el libro *Pensando sobre patrones (con Java)* disponible en *www.BruceEckel.com*.
16. **No extiendas funcionalidad esencial mediante subclases**. Si un elemento de una interfaz es esencial para una clase debería estar en la clase básica, no añadirse mediante derivación. Si añades métodos mediante el mecanismo de herencia puede que debas volver a pensar el diseño.
17. **Menos es más**. Comienza con una interfaz mínima para una clase, tan pequeña y sencilla como sea necesaria para resolver el problema que tienes entre manos, pero no intentes anticipar la forma en la que tu clase *debería* usarse. Cuando la clase sea utilizada, descubrirás las formas en las que debes ampliar la interfaz. Sin embargo, una vez que una clase se utiliza, no puedes quitar elementos de la interfaz sin afectar al código cliente. Si necesitas añadir métodos, de acuerdo; no afectará al código. Pero incluso si reemplazas la funcionalidad de métodos antiguos con nuevos métodos, deja la interfaz existente intacta (puedes combinar la funcionalidad en la implantación subyacente si quieres). Si necesitas ampliar la interfaz de un método existente añadiendo nuevos argumentos, sobrecarga el método con los nuevos argumentos; de esta forma, las llamadas al método existente no se verán afectadas.
18. **Lee tus clases en voz alta para convencerte de que son lógicas** Haz referencia a las relaciones entre una clase base y las clases derivadas como “es un/a” y a los objetos miembro como “tiene un/a.”
19. **Cuando decidas entre herencia y composición, pregúntate si necesitarás hacer conversiones al tipo base**. Si no es así, prefiere la composición (objetos miembro) a la herencia. Esto puede eliminar la necesidad que se percibe de tener múltiples tipos base. Si utilizas herencia, los usuarios pensarán que es necesaria la conversión.
20. **Utiliza campos para la variabilidad de valor y sobrecarga métodos para la variabilidad de comportamiento**. Esto es, si encuentras una clase que utiliza variables de estado en los métodos que cambian su comportamiento en función de esas variables, probablemente deberías rediseñarla para expresar los diferentes comportamientos mediante subclases y métodos sobrecargados.

21. **Cuidado con la sobrecarga**. Un método no debería ejecutar código de forma condicional en función del valor de un argumento. En este caso, deberías crear dos o más métodos sobrecargados.
22. **Utiliza una jerarquía de excepciones**—preferiblemente derivada de las clases específicas apropiadas de la jerarquía estándar de excepciones de Java. La persona que captura las excepciones puede entonces escribir manejadores para los tipos específicos de las excepciones, y seguidamente manejadores para el tipo base. Si añades nuevas excepciones derivadas, el código cliente existente seguirá capturando la excepción mediante el tipo básico.
23. **A veces la simple agregación funciona**. Un "sistema de confort para pasajeros" de una línea aérea consiste en una serie de elementos no conexos: asiento, aire acondicionado, video, etc., y sin embargo necesitas muchos de ellos en un avión. ¿Haces que sean miembros privados y construyes un interfaz completamente nuevo? No—en este caso, los componentes también son parte del interfaz público, así que deberías crear objetos miembro públicos. Estos objetos tienen sus propias codificaciones privadas, que siguen siendo seguras. Ten en cuenta que la simple agregación no es una solución que pueda darse con frecuencia, pero darse se da.
24. **Considera la perspectiva del programador cliente y la persona que mantiene el código**. Diseña tu clase de forma que su uso sean tan obvio como sea posible. Anticipa la clase de cambios que se harán y diseña tu clase para que esos cambios sean sencillos.
25. **Cuidado con el "síndrome del objeto gigante."** Esta es con frecuencia una enfermedad de los programadores procedurales que son nuevos en la POO y que terminan escribiendo un programa procedural y pegándolo dentro de uno o dos objetos gigantes. Con la excepción de los entornos de aplicaciones, los objetos representan conceptos de tu aplicación, no a la aplicación en sí.
26. **Si debes hacer algo feo, por lo menos coloca la fealdad dentro de una clase**.
27. **Si debes hacer algo que no sea portable, crea una abstracción para ese servicio y colócalo dentro de una clase**. Este nivel extra de indirección evita que la falta de portabilidad se distribuya por todo tu programa. (Este consejo se enmarca, entre otros, en el patrón *Puente*).
28. **Los objetos no deberían ser simples contenedores de datos**. Deberían también tener comportamientos bien definidos. (De manera ocasional, los "objetos de datos" son apropiados, pero sólo cuando se utilizan de forma expresa para empaquetar y transportar un grupo de elementos y cuando un contenedor genérico no es apropiado.)
29. **Elije primero la composición cuando crees nuevas clases a partir de las existentes**. Sólo deberías utilizar la herencia en tu diseño cuando sea necesario. Si utilizas la herencia donde la composición puede funcionar, tus diseños se harán innecesariamente complicados.
30. **Utiliza la herencia y la sobrecarga de métodos para expresar las diferencias de comportamiento, y los campos para expresar las variaciones de estado**. Un ejemplo extremo de lo que no se debe hacer es heredar diferentes clases para representar los colores en vez de utilizar un campo de "color".
31. **Cuidado con la varianza**. Dos objetos semánticamente diferentes pueden tener acciones, o responsabilidades, idénticas y existe la tentación natural a intentar hacer uno subclase del otro sólo para aprovechar el mecanismo de herencia. Esto se denomina varianza, pero no existe una justificación real para forzar una relación de superclase/subclase donde no existe. Una solución mejor es crear un clase básica que ofrezca un interfaz de ambas como clases derivadas; requiere un poco más de

espacio, pero seguirás aprovechando la herencia y probablemente descubrirás algo importante sobre el diseño.

32. **Cuidado con la *limitación* en la herencia**. El diseño más claro añade nueva capacidad a los diseños heredados. Un diseño sospechoso elimina capacidad mediante la herencia sin añadir nada. Pero las reglas se han hecho para romperse, y si trabajas con una biblioteca de clases antigua, puede ser más eficiente restringir una clase existente en subclases que lo que implicaría reestructurar la jerarquía para hacer que las nuevas clases se coloquen como debería ser por encima de las clases preexistentes.
33. **Utiliza patrones de diseño para eliminar "funcionalidad desnuda."** Esto es, si sólo un objeto de tu clase debe crearse, no te vayas a la aplicación a escribir un comentario del como "crear sólo uno de estos." Envuélvelo en un *singleton*. Si tienes mucho código desaliñado en el programa principal que crea objetos, busca un patrón de creación como un método fábrica en el que puedas encapsular dicha creación. Eliminar la "funcionalidad desnuda" no sólo hará que tu código sea más sencillo de entender y mantener, sino que también le hará más seguro contra los bien intencionados programadores de mantenimiento que te seguirán.
34. **Cuidado con la "parálisis por análisis."** Recuerda que normalmente debes avanzar en un proyecto antes de conocerlo todo, y que con frecuencia la mejor forma y la más rápida de aprender sobre algunos de los factores desconocidos es pasar al siguiente paso en vez de tratar de resolverlo mentalmente. No puedes conocer cuál es la solución hasta que *tienes* la solución. Java tiene mecanismos de protección integrados; deja que trabajen por tí. Los errores que cometas en una clase o un conjunto de clases no destruirán la integridad del sistema en su conjunto.
35. **Cuando creas que tienes un buen análisis, diseño o implementación, revísala de arriba a abajo**. Trae a alguien ajeno a tu equipo—no tiene por qué ser un consultor, sino que puede ser alguien de otro grupo de tu misma empresa. Revisar tu trabajo con un par de ojos frescos puede revelar problemas en una fase donde son mucho más sencillos de arreglar, y compensa con creces el tiempo y el dinero que se "pierden" con el proceso de revisión.

Construcción

1. **En general, sigue las convenciones de codificación de Sun**. Están disponibles en *http://java.sun.com/docs/codeconv/index.html* (he seguido dichas convenciones en el código de este libro tanto como he podido). Se utilizan en lo que posiblemente constituye la mayoría del código al que se expondrán los programadores de Java. Si te obstinas en seguir el estilo de codificación que siempre has seguido, se lo pondrás difícil a tus lectores. Cualesquiera convenciones de codificación que decidas utilizar, asegúrate que son consistentes en todo el proyecto. Hay una herramienta libre que reformatea automáticamente código Java en *http://jalopy.sourceforge.net*. Puedes encontrar un comprobador de estilo libre en *http://jcsc.sourceforge.net*.
2. **Cualquiera que sea el estilo de codificación que sigas, lo que marca la diferencia es si tu equipo (mejor aún, tu empresa) lo utiliza como estándar**. Esto se lleva al extremo de considerar adecuado corregir el estilo de codificación de cualquier otra persona que no sea conforme al estándar. El valor de estandarizar un estilo es que serán necesarios menos ciclos mentales para analizar el código, y así puedes dedicarte más a lo que el código implica.
3. **Sigue las reglas estándar de utilización de mayúsculas**. Utiliza mayúsculas para la primera letra de los nombres de las clases. La primera letra de los campos, métodos y objetos (referencias) deberían ir en minúsculas. Todos los identificadores

deberían tener todas las palabras juntas y utilizar mayúsculas para la primera letra de todas las palabras intermedias. Por ejemplo: **EsteEsUnNombreDeClase**
esteEsUnNombreDeMetodoOCampo
Utiliza mayúsculas para *todas* las letras (y utiliza el carácter de subrayado para separar las palabras) de identificadores primitivos **static final** cuyas definiciones tengan valores iniciales constantes. Esto indica que son constantes cuyo valor se resuelve durante la compilación.
Los paquetes son un caso especial—se utilizan letras minúsculas en todo el nombre, incluso en las palabras intermedias. Las extensiones de dominio (com, org, net, edu, etc.) también deberían ir en minúsculas. (Esto cambió al pasar de Java 1.1 a Java 2.)

4. **No crees tus propios nombres con "marcas especiales" para los campos privados**. Esto suele verse en casos en los que se utilizan caracteres especiales como el de subrayado precediendo al nombre. La notación húngara es el peor ejemplo de lo que estoy diciendo, en ellas se añaden caracteres adicionales que indican el tipo de datos, el uso, el ámbito, etc., como si estuvieses escribiendo código en ensamblador y el compilador no ofreciese ningún apoyo en absoluto. Estas notaciones son confusas, difíciles de leer y es desagradable forzar su cumplimiento y su mantenimiento. Deja que las clases y los paquetes se encarguen del ámbito del nombre por ti. Si crees que debes decorar tus nombres para evitar confusiones, probablemente tu código será confuso de cualquier forma y deberías simplificarlo.
5. **Sigue una "forma canónica"** cuando crees una clase de propósito general. Incluye las definiciones para los métodos **equals( )**, **hashCode( )**, **toString( )**, **clone( )** (utiliza **Cloneable**, o elige alguna otra aproximación para copiar objetos como la serialización), y programa las interfaces **Comparable** y **Serializable**.
6. **Utiliza las convenciones de nombrado de JavaBeans "get," "set," y "is"** para los métodos que leen y cambian los campos **privados**, aunque no creas estar creando un JavaBean en ese momento. No sólo hace que tu clase sea fácil de utilizar como un Bean, sin que es una forma normalizada de referirse a estos tipos de métodos, y así será entendido más fácilmente por el lector.
7. **Para cada clase que crees, incluye las pruebas JUnit de esa clase** (mira en *www.junit.org*, y los ejemplos del capítulo 15). No es necesario que elimines el código de prueba para utilizar la clase en un proyecto, y si no haces cambios, puedes fácilmente volver a ejecutar las pruebas. Este código también te sirve como ejemplos de uso de tu clase.
8. **A veces es necesario heredar para poder acceder a los miembros *protegidos* de la clase básica**. Esto puede conducirte a la sensación de que necesitas varios tipos de clase básica. Si no necesitas realizar conversiones, deriva primero una nueva clase que te permita acceder a las partes protegidas y luego haz que esa nueva clase sea un objeto miembro de la cualquier clase que necesite usarlo, en lugar de utilizar el mecanismo de herencia.
9. **Evita el uso de métodos *final* con el propósito de mejorar la eficiencia**. Usa **final** sólo cuando el programa esté funcionando, pero no sea suficientemente rápido y el perfilador te diga que la invocación a un método es un cuello de botella.
10. **Si hay dos clases asociadas entre sí de alguna forma funcional (como por ejemplo los contenedores y los iteradores), intenta que una de ellas sea una clase interna de la otra**. Esto no sólo enfatiza la asociación entre las clases, sino que permite que el nombre de la clase se pueda reutilizar en un único paquete al anidarlo en otra clase. La biblioteca de contenedores de Java lo hace definiendo una clase interna **Iterator** dentro de cada clase contenedora, y consiguiendo de esa manera que los contenedores tengan un interfaz común. La otra razón por la que querrás utilizar una clase interna será para hacer parte de la

implementación **privada**. En este caso, la clase interna es beneficiosa porque esconde los detalles de la implementación y no tanto porque evite la polución del espacio de nombres en el caso de asociación entre clases que comentaba arriba.

11. **Siempre que te des cuenta de que hay clases que parecen tener un alto acoplamiento entre si, considera las mejoras en la codificación y el mantenimiento que podrías obtener utilizando clases internas**. El uso de clases internas no evitará el acoplamiento, es más, hará dicho acoplamiento explícito y más adecuado.
12. **No caigas en la trampa de la optimización prematura**. Este camino conduce a la locura. En particular, no te preocupes por escribir (o evitar) los métodos nativos, hacer métodos **final**, o retocar el código para que sea eficiente cuando estás empezando a construir el sistema. Tu objetivo prioritario debería ser probar el diseño. Aunque el diseño implique cierta eficiencia, *primero consigue que funcione, luego haz que vaya rápido.*
13. **Mantén el ámbito de las variables tan pequeño como sea posible para que la visibilidad y la vida de tus objetos sea tan corta como sea posible**. Esto reduce la posibilidad de utilizar un objeto en el contexto inadecuado e introducir un error difícil de localizar. Por ejemplo, supón que tienes un contenedor y una parte del código que itera por él. Si copias ese código para utilizarlo en otro contenedor, puede que accidentalmente termines utilizando el tamaño del contenedor antiguo como límite superior del nuevo. Sin embargo, si el antiguo contenedor no está en el ámbito del nuevo código, te darás cuenta del error durante la compilación.
14. **Utiliza los contenedores que hay en la biblioteca estándar de Java**. Hazte competente en su uso y mejorarás tu productividad enormemente. Utiliza preferentemente **ArrayList** para las secuencias, **HashSet** para los conjuntos, **HashMap** para las formaciones asociativas, y **LinkedList** para las pilas (mejor que **Stack**, aunque puede que quieras crear un adaptador para darle un interfaz de pila) y colas (que puede que también quieras utilizar con un adaptador, como se muestra en este libro). Cuando utilices los tres primeros, deberías convertirlos respectivamente en **List**, **Set** y **Map**, de forma que puedas cambiar fácilmente su implementación si fuese necesario.
15. **Para que un programa sea robusto, cada componente debe ser robusto**. Emplea todas las herramientas que ofrece Java—control de acceso, excepciones, comprobación de tipos, sincronización y demás—en cada clase que crees. De esa manera podrás moverte con seguridad al siguiente nivel de abstracción cuando construyas tu sistema.
16. **Es mejor conseguir errores durante la compilación que durante la ejecución**. Procura gestionar un error tan cerca del punto en el que se produce como sea posible. Captura cualquier excepción en el manejador más próximo que tenga suficiente información para gestionarla. Haz lo que puedas con la excepción en el nivel actual; si eso no soluciona el problema, vuelve a lanzar la excepción.
17. **Cuidado con las definiciones de métodos largas**. Los métodos deberían ser unidades breves y funcionales que describan e implementen una parte concreta de una de la interfaz de una clase. Un método largo y complicado es difícil y caro de mantener, y probablemente trata de hacer demasiado. Si ves un método de este tipo, ello indica que, cuanto menos, debería ser dividido en varios métodos. Puede también sugerir que sea necesario crear una nueva clase. Además, los métodos pequeños fomentarán la reutilización de código dentro de la clase. (A veces los métodos tienen que ser largos, pero deberían seguir haciendo una única cosa.)
18. **Mantén las cosas tan *"privadas* como sea posible.*"*** Una vez que haces público un elemento de tu biblioteca (un método, una clase, un campo), puede que nunca lo puedas volver a sacar de la parte pública. Si lo haces, arruinarás el código

de alguien, y le forzarás a volver a diseñarlo y escribirlo de nuevo. Si sólo publicas lo que es necesario, podrás cambiar todo lo demás con impunidad, y dado que los diseños tienden a evolucionar, este es un grado de libertad importante. De esta forma, los cambios en la implementación tendrán un impacto mínimo en las clases derivadas. La privacidad es especialmente importante cuando se trata de la programación con múltiples hebras—sólo los campos **privados** pueden protegerse del uso no-**sincronizado**.
Las clases con acceso a nivel de paquete deberían seguir teniendo campos **privados**, aunque suele tener sentido dar acceso a los métodos del paquete en vez de hacerlos **públicos**.

19. **Utiliza los comentarios generosamente, y haz uso de la sintáxis de comentarios orientados a la documentación de *javadoc* para producir la documentación de tu programa**. No obstante, los comentarios deberían añadir un significado genuino al código; los comentarios que sólo repiten lo que el código claramente expresa son molestos. Ten en cuenta que los nombres típicamente explícitos de las clases y métodos en Java reducen la necesidad de ciertos comentarios.
20. **Evita el uso de "números mágicos"**—que son los números que van grabados directamente en el código. Son una pesadilla si necesitas cambiarlos, ya que nunca sabes si "100" indica "el tamaño de la formación" o "algo completamente distinto." En vez de ello, crea una constante con un nombre descriptivo y utiliza la constante en todo el programa. Esto hace que el programa sea más sencillo de entender y mantener.
21. **Cuando crees constructores, considera las excepciones**. En el mejor de los casos, el constructor no hará nada que produzca una excepción. En el siguiente de los escenarios mejores, la clase estará compuesta sólo por y heredará sólo a partir de clases robustas, así que no será necesario restaurar estados si se produce una excepción. En cualquier otro caso, será necesario que se restauren las clases que la componen dentro de una cláusula **finally**. Si un constructor tiene que fallar, la acción apropiada es lanzar una excepción, de forma que el que lo llama no continúe ciegamente, pensando que el objeto fue creado correctamente.
22. **En el interior de los construcotres, sólo haz lo que sea necesario para situar al objeto en el estado adecuado**. Evita todo lo que puedas llamar a otros métodos (excepto a los métodos de **finalización**), porque dichos métodos puede que sean sobrecargados por alguien y produzcan resultados inesperados durante el proceso de construcción. (Para más detalles, véase el capítulo 7.) Los constructores más pequeños y más simples son menos propensos a lanzar excepciones o causar problemas.
23. **Si tu clase necesita cualquier tipo de restauración cuando el programador cliente ha terminado de utilizar el objeto, coloca el código de limpieza en un único método bien definido**, con un nombre como **dispose( )** que claramente sugiera su intención. Además, crea una señal **boolean** en dicha clase para que indique que se ha invocado a **dispose( )** de forma que **finalize( )** pueda comprobar la "condición de terminación" (véase el capítulo 4).
24. **La responsabilidad de *finalize( )* sólo debe ser verificar la "condición de terminación" de un objeto para su uso en la depuración**. (Véase el capítulo 4.) En algunos casos especiales, podría ser necesario liberar memoria que de otro modo no sería liberada por el recolector de basura. Puesto que el recolector de basura podría no ser invocado para liberar tu objeto, no puedes utilizar **finalize( )** para realizar la limpieza necesaria. En estos casos debes crear tu propio método **dispose( )**. En el método **finalize( )** de la clase, se comprueba que el objeto ha sido liberado y en caso contrario lanzar una clase derivada de **RuntimeException**, para indicar que hay un error de programación. Mejor que recurrir a este esquema,

asegúrate de que **finalize( )** funciona en tu sistema. (Podría ser necesario llamar a **System.gc( )** para asegurarse de ello.)

25. **Si un objeto debe liberarse (de algún modo distinto al recolector de basura) en un ámbito particular, utiliza el siguiente esquema:** inicializa el objeto y, si se consigue con éxito, entra inmediatamente en un bloque **try** con una cláusula **finally** que realice la limpieza.
26. **Cuando sobrecargues *finalize( )* al utilizar el mecanismo de herencia, recuerda llamar a *super.finalize( )***. (Esto no es necesario si **Object** es la superclase inmediata.) En lugar del al principio, deberías llamar a **super.finalize( )** al *final* de tu **finalize( )**, para asegurarte de que si es necesario los componentes de la clase base son válidos.
27. **Cuando creas un contenedor de objetos de tamaño fijo, mételos en una formación**, especialmente si vas a devolver el contenedor desde un método. De esta manera conseguirás las ventajas del chequeo de tipos durante la compilación y el recipiente de la formación puede que no necesite referirse a los objetos de la formación para utilizarlos. Ten en cuenta que la clase base de la biblioteca de contenedores, **java.util.Collection**, tiene dos métodos **toArray( )** que hacen esto.
28. **Elige *interfaces* en vez de clases *abstractas***. Si sabes que algo va a ser una clase base, tu primera opción debería ser convertirla en un **interfaz**, y sólo si te ves forzado a tener definiciones de métodos o variables miembro los debes cambiar para que sean una clase **abstracta**. Una **interfaz** dice lo que el cliente quiere hacer, mientras que una clase tiende a preocuparse por (o permit) detalles de implementación.
29. **Si quieres evitar una experiencia muy frustrante, asegúrate de que sólo hay una clase no empaquetada con un nombre determinado en el conjunto de clases de tu classpath**. De otro modo, el compilador puede que encuentre la otra clase con el mismo nombre primero, y dar mensajes de error que no tengan sentido. Si sospechas que tienes un problema relacionado con el classpath, trata de buscar ficheros **.class** que tengan el mismo nombre en el comienzo de cada una de las rutas del classpath. Idealmente, coloca todas tus clases en paquetes.
30. **Cuidado con la sobrecarga accidental**. Si intentas sobrecargar el método de una clase base y no la escribes completamente bien, terminarás añadiendo un nuevo método en vez de sobrecargar un método existente. Si embargo, esto es perfectamente legal, así que no habrá mensajes de error del compilador o el entorno de ejecución; simplemente tu código no funcionará de forma correcta.
31. **Cudado con la optimización prematura**. Haz que funcione primero, haz que vaya rápido después—pero sólo si es necesario, y sólo si se ha probado que hay un cuello de botella de rendimiento en una sección particular de tu código. A menos que hayas utilizado un perfilador para descubrir un cuello de botella, probablemente perderás el tiempo. El coste adicional de los ajustes de rendimiento es que tu código se hace menos inteligible y no mantenible.
32. **Recuerda que el código se lee muchas más veces de las que se escribe**. Los diseños limpios consiguen programas fáciles de entender, pero los comentarios, las explicaciones detalladas, las pruebas y los ejemplos son inmensamente valiosos. Te ayudarán a ti y a todo el que venga detrás de ti. Si no es por otra cosa, la frustración de tratar de extraer información de la documentación del JDK debería convencerte.

G: Suplementos

Hay varios suplementos para este libro, incluyendo el seminario grabado en el CD que se encuentra en la parte trasera del libro y otros artículos, seminarios y servicios disponibles a través del sitio web de MindView.

Este apéndice describe estos suplementos para que Usted pueda decidir si le serían de utilidad.

Seminario en CD “Bases para Java”

El CD que se encuentra en la parte de atrás de este libro tiene el propósito de darle las bases para prepararse a aprender Java usando este libro o participando en el seminario *Thinking in* Java (“Pensando en Java”, nota del traductor). La mayor parte de los más de 400 MB del CD es un curso llamado *Foundations for Java* (“Bases para Java”) hecho completamente en multimedia. Incluye el seminario *Thinking in C* (“Pensando en C”) le introduce a la sintaxis de C, los operadores y las funciones sobre los cuales se basa la sintaxis de Java. Adicionalmente incluye las primeras siete conferencias de la 2ª. edición del seminario en CD *Hands-On*-Java que yo creé. Aunque históricamente el CD completo de Hands-*on*-Java ha estado disponible solo mediante compra separada (lo mismo ocurre con la 3a. edición del CD de *Hands-on*-Java), decidí incluir las primeras siete conferencias de la 2a. edición ya que lo conceptos en ellas no han cambiado sustancialmente con la 3a. edición del libro. De esta manera no solo le dará (junto con *Thinking in C* -Pensando en C-) una base para este libro y para el seminario *Thinking in Java* (“Pensando en Java”), sino que adicionalmente espero le proporcione una muestra de la calidad y el valor de la 3a. edición del CD *Hands-On* Java.

El CD se describe con mayor detalle en la introducción de este libro.

Seminario “Pensando en Java”

Mi compañía MindView, Inc. ofrece seminarios de cinco días basados en el material de este libro, tanto abiertos al público en general como particulares. Llamado antes el seminario *Hands-On Java*, es nuestro seminario introductorio principal. Provee las bases para nuestros seminarios más avanzados. Cada lección está conformada por material seleccionado de cada capítulo y seguida por un período de ejercicios monitoreados donde cada estudiante recibe atención personalizada. Puede encontrar la información sobre la programación de los seminarios y el lugar donde se llevarán a cabo así como testimonios de antiguos participantes y otros detalles en el sitio Web *www.MindView.net*.

Seminario en CD “Hands-On Java”, 3ª edición

El CD Hands-On Java, 3ª edición, contiene una versión ampliada del material del seminario *Thinking in Java* (“Pensando en Java”) y está basado en este libro. Provee al menos algo de la experiencia que se obtiene en el seminario presencial sin el viaje y los gastos relacionados. Hay una conferencia hablada y diapositivas correspondientes a cada capítulo del libro. Yo creé el seminario (recientemente con contribuciones de Andrea Provaglio, quien dicta la mayoría de las versiones presenciales del mismo) y narro el material que está en el CD. La 3ª edición del CD *Hands-On Java* se encuentra para la venta en el sitio web *www.MindView.net*.

Seminario “Diseñando Objetos & Sistemas”

Este evolucionó del popular seminario *Objects & Patterns*(“Objetos & Patrones”) que Bill Venners y yo hemos dictado juntos en los últimos años. El material en el seminario creció más allá de sus límites originales, razón por la cual lo dividimos en dos: este, y el seminario *Thinking in Patterns* (“Pensando en Patrones”) que se describe posteriormente en este mismo apéndice.

Una parte importante de un buen diseño orientado a objetos son objetos bien diseñados. Una gran porción del seminario (distribuido a lo largo de la semana) es el *Object Design Workshop* (“Taller de Diseño de Objetos”), el cual se enfoca en proporcionar directrices que le ayuden a lograr un buen diseño de objetos. Cada uno de estos será explicado y justificado, y luego discutido por los asistentes. Esta discusión es una parte integral del taller y tiene el propósito de facilitar entre colegas sobre diseño que pueda ayudar a cada uno a aprender de las experiencias y perspectivas de los otros. El Taller de Diseño de Objetos le dará un conjunto específico de directrices prácticas sobre las cuales basarse en su futuro diseño de objetos.

La otra porción de este seminario se enfocará en el proceso de desarrollar y construir un sistema, dando prioridad a los llamados “Métodos Agiles” o “Metodologías Livianas”, especialmente Programación Extrema (XP). Introduciremos metodologías en general, pequeñas herramientas como las técnicas de planeación de “tarjetas-índice” (*index-card*) descritas en *Planning Extreme Programming* (“Planeando la Programación Extrema”) (Beck and Fowler, 2002), tarjetas CRC para diseño de objetos, programación en pareja, planeación iterativa, evaluación de unidad, construcción automatizada (*automated* building), control del código fuente y tópicos similares. El curso incluirá un proyecto XP que será desarrollado a lo largo de la semana.

Visite *www.MindView.net* para conocer la programación, localización, testimonios y otros detalles.

Pensando en Java Empresarial (*Enterprise Java*)

Este es el libro que se ha derivado de algunos de los capítulos más avanzados que se encontraban antes en *Thinking in Java* ("Pensando en Java"). No es un segundo volumen de *Thinking in Java* ("Pensando en Java"), sino un enfoque centrado en los temas más avanzados de la programación empresarial. Actualmente está disponible (en alguna forma), como una descarga gratis en el sitio web *www.BruceEckel.com*. Debido a que es un libro separado, puede expandirse para acomodar los temas necesarios. La meta, tal como en *Thinking in Java*, es producir una muy entendible introducción a los temas básicos de las tecnologías de la programación empresarial de tal manera que el lector esté preparado para un cubrimiento más avanzado de ellos.

La lista de temas incluirá pero no estará limitada a:

- Introducción a la Programación Empresarial
- Programación de redes con Sockets y Canales
- Invocación remota de Métodos (*Remote Method Invocation*) (RMI)
- Conexión a bases de datos
- Servicios de directorios y de nombramiento (*Naming and Directory Services*)
- Servlets
- Java Server Pages
- Etiquetas, Fragmentos JSP y Lenguaje de Expresiones (Expression Language)
- Automatizando la creación de interfaces de usuario
- Java Beans Empresariales (*Enterprise Java Beans*)
- XML
- Servicios Web (*Web Services*)
- Evaluación Automatizada (*Automated Testing*)

Usted puede encontrar el estado actual de *Thinking in Enterprise Java* ("Pensando en Java Empresarial") en el sitio Web *www.BruceEckel.com*.

El seminario de J2EE

Este seminario le introduce al desarrollo práctico con Java de aplicaciones distribuidas reales y basadas en la Web. Cubre J2EE y sus tecnologías claves: JavaBeans Empresariales (*Enterprise Java Beans*), Servlets, Java ServerPages, y los patrones arquitectónicos básicos usados para combinar estas tecnologías en aplicaciones mantenibles.

Usted saldrá de este curso con una amplia comprensión de la arquitectura de J2EE, de los problemas para cuya solución fue diseñado, cómo seleccionar las herramientas más apropiadas y cómo codificar sus soluciones.

Visite *www.MindView.net* para conocer la programación, localización, testimonios y otros detalles.

Pensando en Patrones (con Java)

Uno de los más importantes avances en el diseño orientado a objetos es el movimiento de los "Patrones de Diseño", cuya crónica se encuentra *Design Patterns*, escrito por Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995). Ese libro muestra 23 soluciones diferentes a clases particulares de problemas, escritas principalmene en C++. El libro de Patrones de Diseño (*Design* Patterns) es una fuente de lo que ahora se ha convertido en un vocabulario esencial, casi obligatorio, para los programadores OOP. *Thinking in Patterns* ("Pensando en Patrones") introduce los conceptos básicos de los patrones de diseño junto con ejemplos en java. El libro no pretende ser una simple traducción de *Design Patterns* ("Patrones de Diseño"), sino una nueva perspectiva en un marco de Java. No está limitado a los 23 patrones tradicionales, sino que incluye otras ideas y técnicas de solución de problemas cada vez que se considera apropiado.

Este libro comenzó como el último capítulo de *Thinking in Java*, *1ª Edición*, y a medidas que las ideas se continuaron desarrollando, fue claro que necesitaban su propio libro. En el momento de escribir esto todavía está en desarrollo, pero el material se ha revisado y vuelto a revisar a través de numerosas presentaciones del seminario *Objects & Patterns* ("Objetos & Patrones") (el cual ha sido dividido en dos seminarios: *Designing Objects & Systems* – Diseñando Objetos & Sistemas – y *Thinking in Patterns* – Pensando en Patrones –).

Seminario "Pensando en Patrones"

Este seminario se desarrolló de aquel denominado *Objects & Patterns* ("Objetos & Patrones") el cual hemos dictado Bill Venners y yo los últimos años. Este se volvió tan grande en contenido, que lo partimos en dos: este y el seminario *Designing Objects & Systems* ("Diseñando Objetos & Sistemas") descrito previamente en éste apéndice.

El seminario se adhiere fuertemente al material y a la presentación del libro *Thinking in Patterns* ("Pensando en Patrones"), así que la mejor forma de conocer qué es lo que hay en el, es descargar el libro desde el sitio web *www.MindView.net*.

Gran parte de la presentación es un ejemplo del proceso de evolución del diseño, empezando con una solución inicial y moviéndose a través de la lógica y el proceso de desarrollar la solución hacia diseños más apropiados. El último proyecto que se muestra (una simulación del reciclaje de basura) ha evolucionado con el tiempo. Usted puede ver esa evolución como un prototipo de la forma como su propio diseño puede empezar siendo una solución adecuada a un problema particular, y convertirse luego en un acercamiento flexible a toda una categoría de problemas.

- Aumente dramáticamente la flexibilidad de sus diseños.
- Incluya extensibilidad y reusabilidad.
- Cree comunicaciones más densas sobre diseños usando el lenguaje de patrones.

Después de cada conferencia habrá un conjunto de ejercicios de patrones que Usted debe resolver, a través de los cuales será guiado a escribir código que aplique patrones particulares para solucionar problemas de programación.

Visite *www.MindView.net* para conocer la programación, localización, testimonios y otros detalles.

Consultoría y revisión de diseños

Mi compañía ofrece también consultoría, apadrinamiento, revisiones de diseño y de implementación para ayudar a guiar su proyecto a través de su ciclo de desarrollo — especialmente el primer proyecto Java de su compañía. Visite *www.MindView.net* para conocer sobre disponibilidad y otros detalles.

H: Recursos

Software

El JDK de *java.sun.com.* Aun cuando vaya a utilizar un entorno de desarrollo de terceras personas, siempre es buena idea tener el JDK a mano en caso de encontrarnos con lo que podría ser un error del compilador. El JDK es la piedra de toque, y si hay un fallo en él, es muy probable que sea conocido.

La documentación HTML Java de *java.sun.com.* Nunca he encontrado un libro de referencia sobre las librerías estándar de Java que estuviera al día o que no faltara información. Aunque la documentación del HTML de Sun se publica con pequeños errores y en ocasiones resulta demasiado breve, como mínimo todas las clases y métodos están allí. A veces la gente se siente incómoda al principio usando un recurso en línea en lugar de un libro impreso, ...

Libros

Thinking in Java, primera edición. Disponible en el CD ROM incluido con este libro, o como descarga gratuita en *www.BruceEckel.com* . Incluye material antiguo y material que no se ha considerado interesante trasladar a la segunda edición.

Core Java 2 , por Horstmann & Cornell, Volumen I - Fundamentos (Prentice Hall, 1999). Volumen II - Características avanzadas, 2000. Extenso, comprensivo, y el primero al que acudo en busca de respuestas. Recomiendo el libro una vez haya completado *Pensando en Java* y necesite alcanzar metas superiores.

Java in a Nutshell: A desktop Quick Reference, segunda edición , por David Flanagan (O'Reilly, 1997). Un resumen compacto de la documentación en línea de Java. Personalmente, prefiero explorar los documentos desde *java.sun.com* , especialmente porque cambian frecuentemente. De todas formas, algunos compañeros todavía prefieren la documentación impresa y este libro cumple con los objetivos; proporciona más argumentos que los documentos en línea.

The Java Class Libraries: An Annotated Reference , por Patrick Chan y Rosanna Lee (Addison-Wesley, 1997). Lo que la documentación en línea debería ser: suficientes descripciones para hacerla útil. Uno de los reseñantes técnicos de *Pensando en Java* dijo, "Si yo sólo tuviera un libro de Java, sería este (bien, además del tuyo, claro)." A mi no me entusiasma tanto como a él. Es grande, caro, y la calidad de los ejemplos no me satisface. Pero es un buen libro para consultarlo cuando esté encallado y profundiza más que *Java in a Nutshell.*

Java Network Programming, por Elliotte Rusty Harold (O'Reilly, 1997). No comencé a entender el trabajo en red con Java hasta que encontré este libro. Además, su sitio web, Café au Lait, tiene una estimulante, contrastada y

actualizada perspectiva de los desarrollos Java, libre de alianzas con ningún distribuidor. Sus constantes actualizaciones se mantienen con las news de Java. Visite *metalab.unc.edu/javafaq/* .

JDBC Database Access with Java, por Hamilton, Cattell & Fisher (Addison-Wesley, 1997). Si no tiene conocimiento alguno sobre SQL y bases de datos, esta es una buena introducción. Contiene además algunos detalles y una "referencia comentada" de la API (de nuevo, tal como debería ser la referencia en línea). El inconveniente, como todos los libros en *The Java Series* ("Los ÚNICOS libros autorizados por JavaSoft"), es que ha sido revisado, de forma que sólo dice cosas buenas sobre Java -no encontrará información alguna sobre partes oscuras en estas series.

Java Programming with CORBA, por Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997). Trata de forma seria el tema con ejemplos de código para tres ORBs de Java (Visibroker, Orbix, Joe).

Design Patterns, por Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). El libro seminal que empezó el movimiento de las plantillas en la programación.

Practical Algorithms for Programmers, por Binstock & Rex (Addison-Wesley, 1995). Los algoritmos están en C, por lo que son fácilmente transportables a Java. Cada algoritmo está ampliamente comentado.

Análisis y diseño

Extreme Programming Explained, por Kent Beck (Addison-Wesley, 2000). *Amo* este libro. Sí, tiendo a tomar posturas radicales, pero siempre pensé que debía haber un proceso de desarrollo de programas mucho mejor, y creo que XP se acerca bastante. El único libro que tuvo un impacto similar en mi fue *PeopleWare* (descrito más adelante), que trata principalmente sobre el entorno y el comercio en la cultura corporativa. *Extreme Programming Explained* trata sobre la programación, y recoge muchos aspectos. Llegan tan lejos como para decir que los gráficos están muy bien mientras no pierda demasiado tiempo con ellos. Un libro pequeño, capítulos cortos, agradable de leer.

UML Distilled, segunda edición, por Martin Fowler (Addison-Wesley, 2000). La primera vez que uno se enfrenta al UML es desesperante ya que hay montones de diagramas y detalles. De acuerdo con Fowler, la mayoría de éstos son innecesarios y, por eso, él se limita a los más esenciales. Para la mayoría de proyectos, sólo necesitas conocer unas pocas herramientas de diagramas, y el objetivo de Fowler es conseguir un buen diseño en lugar de preocuparse por todas las herramientas disponibles. Un libro agradable, breve y muy legible; el primero al que debes acudir si necesitas aprender UML.

UML Toolkit, por Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997)- Explica el UML y como utilizarlo, y tiene un caso de estudio en Java. El CD-

ROM que le acompaña contiene el código de Java y una versión resumida de Rational Rose. Una excelente introducción al UML y cómo usarlo para construir un sistema real.

The unified Software Development Process, por Ivar Jacobsen, Grady Booch, y James Rumbaugh (Addison-Wesley, 1999). Estaba preparado para criticar este libro. Parecía tener todos los rasgos de un aburrido texto universitario. Sin embargo, me sorprendió muy gratamente (únicamente algún párrafo del libro da la impresión de que el concepto que se quiere explicar no está claro para los autores). El formato del libro no es únicamente claro, sino que además es agradable. Y lo mejor de todo, la lectura está llena de sentido práctico. Creo que este libro debería ser un modelo de UML, y lo aconsejo después de leer *UML Distilled* de Fowler's para profundizar.

Antes de que escoja ningún método, es aconsejable que observe la perspectiva de aquellos que no quieren venderle uno. Es fácil adoptar un método sin realmente acabar de entender lo que espera de él o cómo puede ayudarle. Otros lo están utilizando, lo que parece una razón suficiente. A veces, cuando las personas quieren creer que algo les va a resolver sus problemas, lo prueban (eso es experimentación, lo que es bueno). Pero si no se los resuelve, multiplican sus esfuerzos y comienzan a anunciar a gritos lo que están descubriendo (eso es no querer ver, y no es bueno). El asunto es que si puede unirse a otra gente en el mismo barco, no se quede sólo, aunque el barco no tenga un rumbo fijo.

Con esto no quiero sugerir que las metodologías no lleven a ninguna parte, sino que debería armarse hasta los dientes con herramientas mentales que le ayuden a permanecer en un modo experimental ("Esto no funciona, probemos con esto otro") y lejos de un modo evasivo ("no, realmente no es un problema. Todo va bien, no es necesario ningún cambio"). Creo que los siguientes libros, pueden proporcionarle estas herramientas.

Software Creativity, por Robert Glass (Prentice-Hall, 1995). Este es el mejor libro que he visto que aborde una perspectiva sobre todas las metodologías. Es una colección de cortos ensayos y artículos que Glass ha escrito y otros que ha adquirido (P.J. Plauger es un colaborador), reflejando años de pensamiento y estudio sobre el tema. Son entretenidos y solamente lo suficientemente largos para decir lo necesario; no divaga ni te aburre. Hay cientos de referencias a otros artículos y estudios. Todos los programadores y managers deberían leer este libro antes de esquivar el atolladero que supone la metodología.

Software Runaways: Monumental Software Disasters, por Robert Glass (Prentice-Hall, 1997). Lo interesante de este libro es que pone en primer plano aquello de lo que no nos gusta hablar: cuántos proyectos no sólo fracasan, sino que lo hacen de forma espectacular. Encuentro que la mayoría de nosotros todavía pensamos "Esto no puede pasarme a mí" (o "Esto no puede volver a pasar"), pero creo que nos equivocamos. Tener en mente todas estos fallos nos permite tomar una mejor posición para en el futuro hacerlas correctamente.

Peopleware, segunda ediicíon, por Tom Demarco y Timothy Lister (Dorset House, 1999). Aunque tiene alguna parte en la que se habla sobre desarrollo de software, este libro trata sobre proyectos y equipos en general. Está enfocado en la gente y sus necesidades más que en las necesidades tecnológicas. Hablan de crear un entorno donde la la gente sea feliz y productiva, en lugar de las reglas que estas personas deben seguir para ser los componentes adecuados de una máquina. Esta última actitud creo que es la mayor contribución para que los progrmadores sonrian y asientan con la cabeza cuando el método XYZ es adoptado y entonces todo el mundo en silencio se pone a hacer lo que ha hecho siempre.

Complexity, por M. Mitchell Waldrop (Simon & Schuster, 1992). Cuenta la unión de un grupo de científicos de diferentes disciplinas en Santa Fe, Nuevo México, para discutir problemas reales que en sus respectivas áreas permanecen sin resolver (el mercado de stocks en economía, la formación inicial de la vida en biología, por qué la gente hace lo que hace en sociología, etc.). Cruzando la física con la economía, la química, las matemáticas, la informática, la sociología y otras, se está desarrollando un acercamiento multidisciplinario a estos problemas. Pero lo más importante, aparece una forma diferente de pensar sobre esos ultracomplejos problemas: alejándose del determinismo matemático y la ilusión de que se puede escribir una ecuación que prediga todos los comportamientos, buscando un patrón y tratando de emularlo de todas las formas posibles. (El libro cuenta por ejemplo el surgimiento de algoritmos genéticos.) Creo que este tipo de pensamiento es útil ya que observamos formas de abordar proyectos de software más y más complejas.

Python

Learning Python, por Mark Lutz y David Ascher (O'Reilly, 1999). Una buena introducción para el programador al que rápidamente se está conviertiendo en mi lenguaje favorito, un excelente compañero de Java. El libro incluye además una introducción a JPython, que permite combinar Java y Python en un único programa (el intérprete de JPython se compila a bytecodes puros de Java, con lo cual no hay nada especial que debas añadir). La unión de estos lenguajes promete grandes posibilidades.

Mi propia lista de libros

Listados por orden de publicación. No todos están actualmente disponibles.

Computer Interfacing with Pascal & C, (propia publicación a través de Eisys, 1988). Sólo disponible desde *www.BruceEckel.com*). Una introducción a la electrónica del pasado, cuado CP/M era todavía el rey y DOS estaba en sus comienzos. Utilicé lenguajes de alto nivel y frecuentemente el puerto paralelo del ordenador para controlar varios proyectos electrónicos. Adaptado de mis artículos en la primera y mejor revista para la que he escrito, *Micro Cornucopia* . (Citando a Larry O'Brien, editor de *Software Development Magazine* durante un largo tiempo:

la mejor revista de ordenadores nunca publicada -¡todavía tienen planes para construir un robot en una maceta!) ¡ay!, Micro C se perdió bastante antes de que internet apareciera. Escribir este libro fue una experiencia muy satisfactoria.

Using C++, (Osborne / McGraw-Hill, 1989). Uno de los primeros libros que salieron sobre C++. Está descatalogado y reemplazado por la segunda edición que se renombró a *C++ Inside & Out.*

C++ Inside & Out, (Osborne / McGraw-Hill, 1993). Como ya he anotado, la segunda edición de **Using C++.** Las explicaciones de C++ en este libro son bastante precisas, sin embargo data de 1992 y *Thinking in C++* pretende sustituirlo. Puedes encontrar más información sobre este libro así como descargar el código fuente desde *www.BruceEckel.com*

Thinking in C++, primera edición, (Prentice-Hall, 1995).

Thinking in C++, segunda edición, volumen 1, (Prentice-Hall, 2000). Descargable desde *www.BruceEckel.com* .

Black Belt C++, the Master's Collection, Bruce Eckel, editor (M&T Books, 1994). Descatalogado. Es una colección de capítulos escritos por varias lumbreras de C++ basados en presentaciones de la Conferencia de Desarrollo de Software, en la que participé. La cubierta de este libro me estimuló a ganar control sobre los diseños de todas las cubiertas posteriores.

Thinking in Java, primera edición, (Prentice-Hall, 1998). La primera edición de este libro ha ganado el Premio de Productividad de *Software Development Magazine* , el Premio *Java Developer's Journal* a la calidad editorial, y el premio *JavaWorld Reader's Choice Award* al mejor libro. Descargable desde *www.BruceEckel.com*