

# Thinking in Java, 2<sup>nd</sup> Edition, Release 11

To be published by Prentice-Hall mid-June, 2000

Bruce Eckel, President,  
MindView, Inc.



Planet PDF brings you the Portable Document Format (PDF) version of Thinking in Java (2nd Edition). Planet PDF is the premier PDF-related site on the web. There is news, software, white papers, interviews, product reviews, Web links, code samples, a forum, and regular articles by many of the most prominent and respected PDF experts in the world. Visit our sites for more detail:

<http://www.planetpdf.com/>  
<http://www.codecuts.com/>  
<http://www.pdfforum.com/>  
<http://www.pdfstore.com/>

# Thinking in Java

Second Edition

Bruce Eckel  
President, MindView, Inc.

ISBN 0-13-027363-5



9780130273635



Traducido del inglés por Esteban Cabezudo ([esteban@cabezudo.com.uy](mailto:esteban@cabezudo.com.uy) - [estebanc@adinet.com.uy](mailto:estebanc@adinet.com.uy)) y Lorena Baldizoni de *Thinking in Java, Second Edition, Bruce Eckel, Prentice Hall.*

2003. Versión 1.00

# Nota del traductor

Inicialmente comencé a leer este libro motivado por mi utópico afán de aprender todo acerca de las computadoras, la programación orientada a objetos y en especial Java. La primera vez que obtuve *Thinking in Java* en la Internet me dio mucha alegría saber que existía un libro de esta calidad al alcance de todos. Comencé sacando algunas notas en español y ya tarde, en el tercer capítulo me di cuenta que tratando de no saltarme las palabras que no conocía estaba traduciendo la totalidad del texto. Tengo que destacar que ni siquiera me acerco a ser un traductor profesional, no estudié formalmente inglés y la mayor parte de este, es producto de la lectura de libros y documentación. Por esta razón se puede notar una sintaxis a veces extraña que trataré de corregir a medida que lo note o me lo hagan notar. También se puede ver un cambio en la traducción a partir del tercer capítulo, tal vez para peor ya que comencé el trabajo ahí y lo terminé en el segundo, espero haber mejorado a medida que avanzaba.

Tengo que agradecer a Bruce Eckel, primero por escribir este libro y segundo por dejarme publicar la traducción en mi sitio sin siquiera conocerme. Esta actitud maravillosa me dio mucha alegría y me hizo sentir muy útil. También quiero agradecer a mi compañera por soportarme estos siete meses sentado durante horas en la madrugada a la luz del monitor y por ayudarme incondicionalmente. Ella es la que realmente estudió inglés y si este trabajo tiene una garantía de calidad, es gracias a su aporte. Lorena aprendí mucho con tu ayuda y seguramente tu aprendiste mucho de Java. Pido perdón por la tortura. A mi hermano Facundo por pedirme la traducción cada vez que me veía, me hacía recordar que hay gente que no lee inglés y que también está interesado en Java. A Gabriel Claramunt por ayudarme con algunos conceptos, por darme algunas ideas de líneas a seguir y por prestarme el notebook para escribir esto.

Quiero que este trabajo pueda ser aprovechado por las personas que no puedan leer inglés y me gustaría que las personas que si lo hacen y tienen comentarios o correcciones me lo hagan saber. Soy consciente de que hay algunas traducciones que tal vez no sean las mas acertadas. Espero que este trabajo les sea útil.

Esteban Cabezudo

# Prefacio

Le he sugerido a mi hermano Todd, que esta haciendo el salto desde el hardware a la programación, que la siguiente gran revolución será en ingeniería genética.

Tendremos microbios diseñados para hacer comida, combustible, y plástico; ellos limpiarán la polución y en general nos permitirán dominar la manipulación del mundo por una fracción de lo que cuesta ahora. Afirmo que harán que la revolución de las computadoras se vea pequeña en comparación.

Entonces afirmo que estaba cometiendo un error común de los escritores de ciencia ficción: perdiéndome en la tecnología (lo cual es fácil de hacer en la ciencia ficción). Un escritor experimentado sabe que la historia nunca es acerca de las cosas; es acerca de las personas. La genética tendrá un gran impacto en nuestras vidas, pero no estoy seguro de que dejara pequeña la revolución de las computadoras (que habilita la revolución genética) -o al menos la revolución de la información. Información es acerca de hablar con cada uno de los otros: es cierto, los autos, zapatos y la cura genética especialmente son importantes, pero en el final estos son solo atavíos. Lo que realmente importa es como reaccionamos a el mundo. Y mucho de esto es acerca de comunicaciones.

Este libro es un ejemplo que hace al caso. La mayor parte de las personas que piensan que es muy atrevido o un poco loco colocar la totalidad de las cosas en la Web. Se preguntan: "Por que alguien querría comprarlo?". Si tuviera una naturaleza mas conservadora nunca lo habría terminado, pero en realidad nunca quise escribir otro libro de computadoras en la manera antigua. No sabía que sucedería, pero lo he convertido en algo mas inteligente que cualquier cosa que haya hecho con un libro.

Por algo, las personas comenzaron a enviar sus correcciones. Esto ha sido un proceso asombroso, porque las personas han observado en cada esquina y grieta para ver errores técnicos y gramaticales, y han sido capaces de eliminar errores de todos los tipos que se que de otra forma los hubiera pasado por alto. Las personas han sido simplemente estupendas con esto, muy a menudo diciendo "Ahora, no trato de decir que es en una forma crítica..." y dan un grupo de errores que estoy seguro nunca hubiera encontrado. Siento como que esto ha sido un tipo de proceso en grupo y realmente han hecho este libro algo especial.

Pero luego que he comenzado a escuchar “OK, bien, es bueno que hayas puesto una versión electrónica, pero quiero una versión impresa y saltar a una copia publicada realmente por una editorial”. He tratado mucho de hacerlo fácil para que cualquiera pueda imprimirla en un formato que se vea bien pero eso no ha disminuido la demanda del libro publicado en una editorial. Muchas personas no quieren leer un libro entero en la pantalla, y tiran con fuerza hacia el fajo de papeles, no importa si están impresos de forma bonita, no les interesa eso (Además, pienso que no es tan económico en términos de tinta para impresoras). Parece que la revolución de las computadoras no han colocados las editoriales fuera de sus negocios después de todo. Sin embargo, un estudiante sugirió que puede ser un modelo para futuras publicaciones: los libros pueden ser publicados en la Web primero, y solo si hay suficientes garantías de interés el libro se colocaría en papel. Actualmente la mayoría de los libros tienen problemas financieros, y tal vez esta nueva aproximación pueda hacer la industria de las editoriales mas provechosa.

Este libro comienza una experiencia iluminadora para mi en otra forma. Originalmente he utilizado la estrategia de que Java es “solo otro lenguaje de programación”, lo que en muchos sentidos es cierto. Pero a medida que el tiempo ha pasado y lo he estudiado mas en profundidad, comienzo a ver que las intenciones fundamentales de este lenguaje es diferente de todos los otros lenguajes que he visto.

Programar es acerca de manejar complejidad: la complejidad del problema que se intenta resolver, trazado sobre la complejidad de la máquina en la cual esta resuelto. Dado esta complejidad, muchos de los proyectos de programación fallan. Y con todo, de todos los lenguajes de programación de los cuales soy conciente, ninguno de ellos se ha escapado totalmente y decidido que su objetivo de diseño principal será conquistar la complejidad del desarrollo y el mantenimiento de programas<sup>1</sup>. Claro, muchas decisiones de diseño de lenguajes fueron echas con la complejidad en mente, pero en algún punto otros temas siempre se consideran esenciales agregarlos dentro de la mezcla. Inevitablemente, estos otros temas son los que causan a los programadores eventualmente “intentarlo” con ese lenguaje. Por ejemplo, C++ cuenta con compatibilidad con sus versiones anteriores de C (para permitirles a los programadores C una fácil migración), al igual que la eficiencia. Estos son objetivos muy útiles y cuentan mucho para el éxito de C++, pero también exponen complejidad extra que impiden que algunos proyectos sean terminados (ciertamente, se puede culpar a los programadores y a la dirección, pero si un lenguaje puede ayudarlo a encontrar sus errores, por que no debería hacerlo?). Otro ejemplo, Visual Basic (VB) fue prendido a BASIC, que no fue diseñado para ser un lenguaje

---

<sup>1</sup> He retomado esto en la 2da edición: Creo que el lenguaje Python de acerca mas ha hacer exactamente eso. Vea [www.Python.org](http://www.Python.org)

extensible, así es que todas las extensiones apiladas sobre VB han producido sintaxis verdaderamente horribles e imposibles de mantener. Perl es compatible con Awk, Sed, Grep y otras herramientas UNIX que intenta reemplazar, y como resultado es a menudo acusado de producir “código de solo escritura” (esto es, luego de un par de meses no se puede leer). Por el otro lado, C++, VB, Perl y otros lenguajes como Smalltalk tienen algo en sus esfuerzos de diseño enfocado en el tema de la complejidad y como resultado son notablemente útiles para solucionar ciertos tipos de problemas.

Lo que mas me ha impresionado mas que comenzar a entender Java es que parece un objetivo resuelto a reducir la complejidad *para el programador*. Como si dijera “no nos importa nada excepto reducir el tiempo y la dificultad de producir un código robusto”. En los primeros días, este objetivo ha resultado en un código que no ejecuta muy rápido (a pesar de que tienen muchas promesas hechas acerca de como correrá de rápido Java algún día) pero ciertamente ha producido reducciones asombrosas en el tiempo de desarrollo; mas o menos el tiempo que toma crear un programa equivalente en C++. Este resultado aislado puede economizar cantidades increíbles de tiempo y dinero, pero Java no se detiene ahí. Continua envolviendo todas las tareas complejas que son importantes, como la multitarea y la programación en redes, en características o librerías que pueden en determinados momentos triviales algunas tareas. Y finalmente, aborda algunos problemas realmente complejos: programas de plataforma múltiples, cambios dinámico de código e inclusive seguridad, cada uno de los cuales pueden encajar dentro de un espectro de complejidad en cualquier parte desde el “impedimento” hasta el “éxito rotundo”. Así es que a pesar de los problemas de rendimiento que se ha visto, la promesa de Java es tremenda: puede hacer a los programadores significativamente mas productivos.

Uno de los lugares en que he visto el gran impacto de esto es en la Web. La programación en la red ha sido siempre dura, y Java lo hace fácil (y los diseñadores de Java están trabajando para hacerlo aún mas fácil). La programación de redes es como hablamos con toros mas eficientemente y mas barato de lo que lo hacemos con teléfonos (el correo electrónico solo ha revolucionado muchos negocios). Como hablamos con otros mas, cosas asombrosas comienzan a suceder, posiblemente mas asombrosas que la promesa de la ingeniería genética.

En todas las formas -la creación de programas, trabajo en equipo para crear los programas, el crear interfaces de usuario así los programas pueden comunicarse con el usuario, ejecutar los programas en distintos tipos de máquinas, y fácilmente escribir programas que se comuniquen a través de la Internet- Java incrementa el ancho de banda de las comunicaciones *entre las personas*. Pienso que quizás los resultados de la revolución en las comunicaciones no serán vistos por los efectos de mover grandes cantidades

de bits alrededor; veremos que la verdadera revolución será porque seremos capaces de hablar con otros mas fácilmente: uno por uno, pero también en grupos y, como un planeta. He escuchado sugerencias que la siguiente revolución es la formación de un tipo de mente global que resulta de suficientes personas y suficientes interconexiones. Java puede o puede no ser la herramienta que fomente esa revolución, pero al menos la posibilidad me ha hecho sentir como que estoy haciendo algo significante tratando de enseñar el lenguaje.

## Prefacio de la 2<sup>da</sup> edición

Las personas han hecho muchos, muchos maravillosos comentarios acerca de la primera edición de este libro, lo que naturalmente ha sido muy placentero para mi. Sin embargo, de vez en cuando alguien tiene quejas, y por alguna razón una queja que comienza a ser periódica es que “el libro es muy grande”. En mi mente los daños son pequeños si su queja es simplemente “son muchas páginas” (Uno es recordando del Emperador de Austria quejándose del trabajo de Mozart: “Demasiadas notas!”). De ninguna forma me estoy tratando de comparar a mi mismo con Mozart). Además, solo puedo asumirlo como una queja que viene de alguien que es todavía no se ha puesto al corriente con la inmensidad del lenguaje Java a si mismo, y no ha visto el resto de los libros acerca del tema -por ejemplo, mi referencia favorita es Cay Horstmann & Gary Cornell’s Core Java (Prentice-Hall), que creció tanto que tienen que dividirlo en dos volúmenes. A pesar de esto, una de las cosas que he intentado hacer en esta edición es cortar un poco las partes que se hacen obsoletas, o al menos no esenciales. Me siento confortable haciendo esto porque el material original queda en el sitio Web y el CD ROM que acompaña este libro, en la forma de descarga libre de la primera edición del libro (en [www.BruceEckel.com](http://www.BruceEckel.com)). Si quiere las cosas viejas, seguirán allí, y esto es un gran alivio para el autor. Por ejemplo, puede verse que el último capítulo original, “Proyectos”, no esta mas aquí; dos de los proyectos han sido integrados en otros capítulos, y el resto no seguían siendo apropiados. También, el capítulo de “Patrones de diseños” se ha hecho muy grande y ha sido movido a un libro acerca de ellos mismos (también en forma de descarga libre en el sitio Web). Así es que, para que estén todos contentos el libro será mas pequeño.

Pero que pena, esto no puede ser.

El gran tema es el continuo desarrollo del propio lenguaje Java, y en particular las APIs expandidas que prometen proporcionar interfaces estándar para a penas todo lo que se quiera hacer (y no me sorprenderé si vemos la API “JTostadora” eventualmente aparecer). Cubrir todas estas APIs es obvio ir mas allá del alcance de este libro y es una tarea delegada a otros autores, pero algunos temas no pueden ser ignorados. El mas grande

de estos incluye Java del lado del servidor (en un inicio Servlets & Java Server pages, o JSPs), lo que es verdaderamente una excelente solución para el problema de la WWW, en donde descubrimos que las diferentes plataformas son solo no suficientemente consistentes para soportar programación. Además, esta es la totalidad del problema de crear fácilmente aplicaciones para interactuar con bases de datos, transacciones, seguridad, y parecidos, lo que le compete a Enterprise Java Beans (EJBs). Estos temas son envueltos en el capítulo antiguamente llamado "Programación de redes" y ahora lo llamamos "Computación distribuida" un tema que comienza a ser esencial para todos. Pero encontraremos también en este capítulo ha sido ampliado para incluir un vistazo de Jini (pronunciado como "genie", y no es tan acrónimo, solo un nombre), que es una tecnología cutting-edge que nos permite cambiar la forma en que pensamos acerca de las aplicaciones interconectadas. Y por su puesto el libro ha sido cambiado para utilizar las librerías GUI Swing en su totalidad. Nuevamente, si se quiere las viejas cosas de Java 1.0 y 1.1 se pueden obtener descargando libremente el libro en [www.BruceEckel.com](http://www.BruceEckel.com) (es también incluido en esta nueva edición del CD ROM, que se encuentra dentro del libro; mas de esto mas tarde).

Dejando de lado las pequeñas características adicionales agregadas en Java 2 y las correcciones hechas en todo este libro, los cambios mayores son en el capítulo de las colecciones (9), que ahora se enfoca en las colecciones de Java 2 utilizadas a través de todo este libro. He mejorado también ese capítulo para ser mas profundo en algo tan importante como las colecciones, en particular como una función hash funciona (así es que se puede saber como crear una propiamente). Hay ahí otros movimientos y cambios, incluyendo el capítulo 1 que fue reescrito, y se ha quitado algunos apéndices y otro material que he considerado que ya no era necesario para la versión impresa del libro, pero aquellos que forman parte de la gran masa de ellos. En general, he experimentado ir por todo, quitando de la 2<sup>da</sup> edición lo que ya no era necesario (pero lo que sigue siendo existiendo en la primera edición electrónica), incluido cambios, y mejoras en todo lo que he podido. Mientras el lenguaje siga cambiando -a pesar de que no sea en el paso precipitado de antes- no hay duda de que habrá futuras ediciones de este libro.

Para aquellos de ustedes que siguen no pudiendo soportar el tamaño de este libro. Me disculpo. Lo crea o no, he trabajado muy duro para mantenerlo pequeño. A pesar del tamaño, siento como que hay suficientes alternativas que pueden satisfacerlo. Por algo el libro esta disponible electrónicamente (desde el sitio Web, y también en el CD ROM que acompaña este libro), así es que si carga con su portátil puede cargar con este libro sin peso extra. Si realmente prefiere las cosas delgadas, actualmente hay versiones del libro para Palm Pilot por ahí (Una persona me dijo que leía el libro en la cama en su Palm con la luz de fondo encendida para no molestar a su esposa. Solo espero que lo ayude a enviarlo a el país de los sueños). Si lo necesita en

papel, conozco personas que imprimen un capítulo por vez y lo cargan en su portafolio para leerlo en el tren.

## Java 2

En el momento que se escribió esto, la versión de Sun de *Java Development Kit* (JDK) 1.3 era inminente, y los cambios propuestos para JDK 1.4 han sido publicitados. A pesar de que los números de versión siguen siendo “unos”, la forma estándar de referirse a cualquier versión del lenguaje que sea mayor que 1.2 es “Java 2”. Esto indica lo significante de los cambios entre “el viejo Java” -que tiene muchas imperfecciones de las cuales me he quejado en la primer edición de este libro- y esta versión mas moderna y mejorada del lenguaje, que tiene menos imperfecciones y muchos agregados y buenos diseños.

Este libro está escrito para Java 2. Tengo el gran placer de librarme de todas las cosas viejas y escribir solo lo nuevo, el lenguaje mejorado porque la vieja información sigue existiendo en la 1<sup>ra</sup> edición electrónica en la Web y en el CD ROM (que es a donde puede ir si esta atascado utilizando una versión previa a la versión 2 del lenguaje). También, dado que cualquiera puede libremente bajar la LDK de [java.sun.com](http://java.sun.com), significa que al escribir para Java 2 no estoy imponiendo una dificultad financiera para alguien forzándolo a realizar una actualización.

Hay aquí un pequeño problema, sin embargo. JDK 1.3 tiene algunas mejoras que me gustaría realmente utilizar, pero la versión de Java que actualmente esta siendo liberada para Linux es JDK 1.2.2. Linux (vea [www.linux.org](http://www.linux.org)) es un desarrollo verdaderamente importante en conjunto con Java, dado que se ha convertido rápidamente en la mas importante plataforma de servidores -rápido, confiable, robusta, segura, bien mantenida, y gratis, una verdadera revolución en la historia de la computación (No pienso que hayamos visto todas estas características en una herramienta antes). Y Java ha encontrado un importante nicho en la programación del lado del servidor en la forma de *Servlets*, una tecnología que es una mejora enorme sobre la programación tradicional CGI (esto es cubierto en el capítulo de “Programación Distribuida”).

Así es que a pesar de que queremos utilizar solo las características mas nuevas, es crítico que todo compile bajo Linux, y de esta manera cuando descomprime el fuente y compila bajo ese SO (con el último JDK) descubrirá que todo compilará. Sin embargo, encontrará que he colocado notas acerca de las características en JDK 1.3 por aquí y allí.

# El CD ROM

Otra bonificación con esta edición es el CD ROM que es empaquetado atrás del libro. Me he resistido a colocar CD ROMs en la parte de atrás de mis libros porque sentido el cargo extra por unos pocos Kbytes de código fuente en ese enorme CD no se justificaba, prefiriendo en lugar de eso permitirle a la gente bajar ese tipo de cosas de mi sitio Web. Sin embargo, vera que este CD ROM es diferente.

El CD no contiene el código fuente del libro, en lugar de eso contiene el libro en su totalidad, en varios formatos. Mi favorito de ellos es el formato HTML, porque es rápido y totalmente indexado -simplemente se hace un clic en una entrada en el índice o en la tabla de contenido y inmediatamente se está en esa parte del libro.

Los 300+ Megabytes del CD, sin embargo, es un curso multimedia completo llamado *Thinking in C: Fundations for C++ & Java*. Originalmente he enviado a CHuck Allison para crear este seminario en CD ROM como un producto por separado , pero he decidido incluirlo con las segundas ediciones de *Thinking in C++* y *Thinking in Java* porque la experiencia consistente de tener personas que van a los seminarios sin una adecuada base de C. El pensamiento aparentemente se dirige a “son un programador inteligente y no *quiero* aprender C, pero en lugar de eso C++ o Java, así es que simplemente me salto C y voy directamente a C++/Java”. Luego llegan al seminario, y lentamente se dan cuenta que el requisito previo de entender la sintaxis de C está ahí por una muy buena razón. Al incluir el CD ROM con el libro, nos aseguramos que todos asisten al seminario con una preparación adecuada.

El CD también permite que el libro le agrade a una audiencia mas amplia. A pesar de que el Capítulo 3 (Controlando el flujo de programa) cubre lo fundamental de las partes de Java que vienen de C, el CD es una introducción amable, y asume cada vez menos acerca de la base de programación del estudiante que tienen el libro. Espero que incluyendo el CD mas personas serán capaces de entrar al grupo de programación Java.

# Introducción

Como un lenguaje humano, Java proporciona una forma de expresar conceptos. Si es exitoso, este medio de expresión será significantemente mas fácil y mas flexible que las alternativas cuando los problemas se hagan mas grandes y complejos.

Se puede ver a Java simplemente como una colección de características - algunas de las características no tienen sentido de forma aislada. Se puede utilizar la suma de las partes solo si se esta pensando en *diseño*, no simplemente en codificación. Y para entender Java de esta forma, se debe entender los problemas con este y con la programación en general. Este libro discute problemas de programación, por que son problemas, y el enfoque que toma Java para solucionarlos. De esta manera, el grupo de características que explicaré en cada capítulo esta basado en la forma en que veo que un tipo particular de problema será solucionado con el lenguaje. De esta forma espero moverlos, de a poco, a el punto donde el pensamiento de Java comience a ser su lengua nativa.

En todas partes, tomaré la actitud de que se quiere crear un modelo en la cabeza que permita desarrollar un profundo entendimiento del lenguaje; si uno se encuentra con un rompecabezas será capaz de proveer a su modelo y deducir la respuesta.

## Requisitos previos

Este libro asume que tiene alguna familiarización con la programación: debe entender que un programa es un grupo de instrucciones, la idea de una subrutina/funcion/macro, instrucciones de control como "if" y constructores de bucles como "while", etc. Sin embargo, se puede aprender esto en muchos sitios, como programar con un lenguaje de macros o trabajar con una herramienta como Perl. Siembre y cuando usted hay a programado hasta el punto don de usted se sienta confortable con las ideas básicas de la programación, se será capaz de ir a través de este libro. Claro, el libro va a ser mas *fácil* para los programadores C y mucho mas para los programadores C++, pero no se sienta fuera si no tiene experiencia con estos lenguajes (pero se debe venir deseoso de trabajar duro; también, el CD multimedia que acompaña este libro dará la velocidad necesaria con la sintaxis básica de C para aprender Java). Voy a introducirlos en los conceptos de la programación orientada a objetos (OOP) y mecanismos de control básicos de Java, así es que seremos expuestos a ellos, y los primeros ejercicios involucran las instrucciones básicas de control de flujo.

A pesar de que se harán a menudo referencias a características de los lenguajes C y C++, no se pretende que sean tomados como comentarios asociados, en lugar de eso ayudará a todos los programadores a colocar a Java en perspectiva con esos lenguajes, desde donde después de todo Java desciende. Intentaré hacer estas referencias simples y explicar todo lo que pienso que un programador C/C++ no este familiarizado.

# Aprendiendo Java

En el mismo momento en que mi primer libro *Using C++* (Osboprne/McGraw-Hill, 1989) se editó, comencé a enseñar el lenguaje. Enseñar lenguajes de programación se ha convertido mi profesión; he visto cabezas caer por el sueño, caras en blanco y expresiones de perplejidad en audiencias en todo el mundo desde 1989. He comenzado a dar entrenamiento en casa con grupos pequeños de personas, he descubierto algo durante los ejercicios. Aún esas personas que sonríen y cabecean cuando están confusas con tantas características. He encontrado, cargando con la pista de C++ en las Conferencia de Desarrollo de Software por muchos años (y luego con la pista de Java), que yo y otros oradores, tienden a dar a la audiencia muchos temas muy rápido. Así es que eventualmente, a través de la variedad de nivel en la audiencia y la forma en que he presentado el material, terminaré perdiendo alguna parte de la audiencia. Tal vez esté preguntando demasiado, pero dado que soy una de esas personas que se resisten a la lectura tradicional (y para muchas personas, creo, esa resistencia es producida por el aburrimiento), voy a tratar de mantener a todos muy acelerados.

Por un tiempo, he creado una gran cantidad de diferentes presentaciones en medianamente poco tiempo. De esta manera, y terminando de aprender por experimentación y repetición (una técnica que trabaja bien también en diseño de programas en Java). Eventualmente he desarrollado un curso utilizando todo lo que he aprendido de mi experiencia enseñando -algo que felizmente he hecho por largo tiempo. Se acomete el problema de aprendizaje en una discreta, pasos fáciles de asimilar, y en un seminario práctico (la situación de aprendizaje ideal) hay ejercicios siguiendo cada uno de las cortas lecciones. Soy ahora este curso en seminarios de Java públicos, que se pueden encontrar en [www.BruceEckel.com](http://www.BruceEckel.com) (El seminario de introducción ésta disponible también en CD ROM. La información esta disponible en el mismo sitio Web).

La realimentación que he obtenido de cada seminario me ha ayudado a modificar y a cambiar el enfoque del material hasta que estoy convencido de que trabaja bien como medio de enseñanza. Pero este libro no es solo notas de seminario -He intentado armar un paquete con toda la información que pudiera colocar dentro de estas páginas, y estructurado y tratando de atraer

al lector al siguiente tema. Mas que todo, el libro esta diseñado para servir a los lectores solitarios que están luchando con un lenguaje de programación nuevo.

# Objetivos

Como mi libro anterior *Thinking in C++*, este libro vuelve a ser estructurado acerca del proceso de aprendizaje del 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 en el libro, pienso en términos de que hace que una lección sea buena en un seminario. Mi meta es obtener pequeñas piezas que puedan ser enseñadas en cantidades razonables de tiempo, seguido por ejercicios que sean posibles realizarlos en una situación de salón de clase.

Mis objetivos en este libro son:

1. Presentar el material de una forma simple así fácilmente se pueda asimilar cada concepto antes de avanzar.
2. Utilizar ejemplos que sean lo mas simples y cortos posibles. Esto a veces me previene de abordar problemas de el “mundo real”, pero he encontrado que las personas que recién comienzan usualmente están felices cuando pueden entender cada detalle de un ejemplo en lugar de impresionarse con el alcance del problema que resuelven. También, hay un límite severo de cantidad de código que pueda ser absorbido en una situación de salón de clase. Por esto no tengo duda de que recibiré críticas por utilizar “ejemplos de juguete”, pero estoy gustoso de aceptar eso en favor de producir algo útil pedagógicamente.
3. Una cuidadosa secuencia de presentación de características así es que no se verán cosas que no se hayan expuesto. Claro que, esto no siempre es posible; en esas situaciones, una breve descripción es dada.
4. Darle al lector lo que pienso que es importante para entender el lenguaje, antes que todo lo que conozco. Creo que es una organización importante de información, y hay algunos cosas que un 95 por ciento de los programadores nunca necesitaran saber y que solo confundirían a las personas y se agregarán a su percepción de la complejidad del lenguaje. Para tomar un ejemplo de C, si se memoriza la tabla de precedencia de operadores (yo nunca lo he hecho), se puede escribir código ingenioso. Pero si se necesita pensar en eso, esto también confundirá la lectura y el mantenimiento del código. Así es que hay que olvidar la precedencia y utilizar paréntesis cuando las cosas no son claras.

5. Mantener cada sección suficientemente enfocada en tiempos de lectura -y el tiempo entre cada período de ejercicios- pequeño. No solo hace la mente de la audiencia mas activa e involucrada durante el seminario, también da al lector un gran sentido de cumplimiento.
6. Proporcionar fundamentos sólidos así se puede entender los temas lo necesario para moverse adelante en trabajos de clase y libros.

## Documentación en línea

El lenguaje Java y las librerías de Sun Microsystems (se pueden bajar libremente) vienen con documentación en forma electrónica, es posible leerla utilizando un navegador, y virtualmente cada tercero que realice una implementación de Java tiene este o un sistema equivalente de documentación. Al menos todos los libros publicados en Java tienen duplicada esta documentación. Así es que se puede tener ya o se puede bajar, y a no ser que sea necesario, este libro no repetirá esta documentación porque es mucho más rápido si se encuentra las descripciones de las clases con su navegador que si se busca adentro de un libro (y es probable que la documentación sea más nueva). Este libro proporcionará descripciones extra de las clases solo cuando sea necesario complementar la documentación así se puede entender un ejemplo en particular.

## Capítulos

Este libro fue diseñando con una sola cosa en mente: la forma en que las personas aprenden el lenguaje Java. La realimentación de la audiencia de los seminarios me ayudaron a entender las partes difíciles que necesitan iluminación. En las áreas donde me pongo ambicioso e incluyo muchas características de una vez, me doy cuenta -a través del proceso de presentar el material- que si se incluye un montón de nuevas características, se necesita explicarlas a todas, y esto fácilmente genera la confusión en el estudiante. Como resultado, he tenido una gran cantidad de problemas para introducir estas características de a pocas cuando el tiempo lo permitía.

El objetivo, entonces, es enseñar en cada capítulo una sola característica, o un pequeño grupo de características asociadas, sin basarme en características adicionales. Esta forma se pude asimilar cada parte en el contexto del propio conocimiento antes de seguir.

He aquí una breve descripción de los capítulos contenidos en este libro, que corresponden a las lecturas y períodos de ejercicio de mis seminarios prácticos.

## **Capítulo 1: Introducción a los objetos**

Este capítulo es una visión general de todo lo referido a programación orientada a objetos, incluyendo la respuesta a la pregunta básica: “¿Qué es un objeto?”, interfase versus implementación, abstracción y encapsulación, mensajes y funciones, herencia y composición y el tan importante polimorfismo. Se tendrá también un vistazo de los temas relacionados con la creación como constructores, donde los objetos existen, donde se colocan una vez creados, y el mágico recolector de basura que limpia los objetos que no se necesitan mas. Otros temas serán introducidos, incluyendo el manejo de error con excepciones, múltiples hilos para interfaces con respuesta y trabajo en redes y en la Internet. Así es que se aprenderá lo que hace a Java especial, por que es tan exitoso, y acerca del análisis y diseño orientado a objetos.

## **Capítulo 2: Todo es un objeto**

Este capítulo se sitúa en el punto en el cual se puede escribir el primer programa Java, así es que da una visión general de lo esencial, incluyendo el concepto de *referencia* a un objeto; como crear un objeto; una introducción a los tipos primitivos y arreglos; alcance y la forma en que los objetos son destruidos por el recolector de basura; como todo en Java es un nuevo tipo de datos (clase) y como crear clases propias; argumentos de funciones, y valores de retorno; visibilidad de nombres y utilización de componentes de otras librerías; la palabra clave static; y los comentarios y la documentación incrustada.

## **Capítulo 3: Controlando el flujo del programa**

Este capítulo comienza con todos los operadores que vienen con Java desde C y C++. Además, se descubrirán dificultades comunes con operadores, conversiones, promociones y precedencia. Luego es seguido, por el control de flujo básico y operaciones de selección que se saben de virtualmente cualquier lenguaje de programación: decisión con if-else; bucles con for y while; salida de un bucle con break y continue de la misma forma que quiebres con etiquetado y continuación con etiquetado (que explica la “perdida del goto” en Java); y selección utilizando switch. A pesar de que mucho de este material tiene cosas en común con el código C y C++, hay algunas diferencias. Además, todos los ejemplos son enteramente en Java así es que el lector se sentirá más confortable con como se ve Java.

## **Capítulo 4: Inicialización y limpieza**

Este capítulo comienza introduciendo el constructor, que garantiza la inicialización apropiada. La definición del constructor cae dentro del concepto de sobrecarga de función (dado que se pueden tener varios constructores). Esto es seguido por una discusión sobre el proceso de limpieza, que no siempre es tan simple como se ve. Normalmente, solo se tira un objeto cuando se ha terminado con él y el recolector de basura eventualmente comienza solo y libera la memoria. Esta porción explora el recolector de basura y alguna de sus idiosincrasia. El capítulo concluye con una mirada cerrada de como las cosas son inicializadas: inicialización automática de miembros, inicialización específica de miembros, el orden de inicialización, inicialización estática e inicialización de arreglos.

### **Capítulo 5: Escondiendo la implementación**

Este capítulo cubre la forma que el código es empaquetado junto, y porque algunas partes de una librería son expuestas cuando otras partes son ocultadas. Esto comienza viendo las palabras clave package e import, que realizan empaquetado a nivel de ficheros y permiten crear librerías de clases. Luego se examina el tema de las rutas a directorios y nombres de ficheros. El resto del capítulo trata de las palabras clave **public**, **private**, y **protected**, el concepto de acceso “amigable”, y de que significan los diferentes niveles de control de acceso cuando se utilizan en varios contextos.

### **Capítulo 6: Reutilizando clases**

El concepto de herencia es estándar en virtualmente todos los lenguajes de POO. Esta es la forma en que se toma una clase existente y se agregan cosas a su funcionalidad (de la misma forma que se puede cambiar, el tema del Capítulo 7). Herencia es a menudo la forma de reutilizar código dejando la misma “clase base”, y se corrigen cosas aquí y allí para producir lo que se desea. Sin embargo, herencia no solo es la única forma de crear nuevas clases de clases existentes. Se puede también incrustar un objeto dentro de una nueva clase mediante *composición*. En este capítulo se aprenderá acerca de estas dos formas de reutilizar código en Java y como aplicarlo.

### **Capítulo 7: Polimorfismo**

Por cuenta propia, toma nueve meses descubrir y entender polimorfismo, un principio básico de la POO. A través de pequeños y simples ejemplos se verá como crear una familia de tipos con herencia y manipulando objetos de esa familia a través de su clase base. El polimorfismo de Java permite tratar todos los objetos de esa familia de forma genérica, lo que significa que el grueso de su código no confía en un tipo específico de información. Esto hace

sus programas extensibles, así es que se pueden crear programas y mantener código de forma mas fácil y barata.

### **Capítulo 8: Interfases y clases internas**

Java proporciona una tercer forma de configurar una relación de reutilización a través de la *interfase*, lo que es una abstracción pura de las interfaces de un objeto. La interfase es más que simplemente una clase abstracta tomada al extremo, dado que permite realizar una variación de “herencia múltiple” de C++, creando una clase a la que se le pueda realizar una conversión ascendente a más de un tipo base.

El inicio, las clases internas se ven simplemente como un mecanismo de ocultar código: se colocan clases dentro de otras clases. Se aprenderá sin embargo, que las clases internas hacen más que eso -ellas saben acerca de la clase que la rodea y pueden comunicarse con ella- y el tipo de código que se puede escribir con una clase interna es más elegante y claro, de la misma forma que es un concepto nuevo para la mayoría y toma algún tiempo sentirse confortable con el diseño utilizando clases interna.

### **Capítulo 9: Conteniendo los objetos**

Es un programa bastante simple el que tiene solo una cantidad fija de objetos con tiempos de vida conocidos. En general, sus programas crearán siempre nuevos objetos en todo momento y estos serán conocidos solamente en tiempo de ejecución o inclusive la cantidad exacta de objetos que se necesitan. Para solucionar este problema general de programación, se necesita crear cualquier número de objetos, en cualquier momento, en cualquier lugar. Este capítulo explora en profundidad la librería de contenedores que Java 2 proporciona para contener objetos mientras se está trabajando con ellos: los arreglos simples y los más sofisticados contenedores (estructuras de datos) como ArrayList y HashMap.

### **Capítulo 10: Manejo de errores con excepciones**

La filosofía básica de Java es que un código mal formado no se ejecutará. Tanto como sea posible, el compilador encuentra problemas, pero algunos de los problemas -errores del programador o condiciones naturales que se suceden como parte de la ejecución normal del programa- pueden ser detectados y tratados solo en tiempo de ejecución. Java tiene *manejo de excepciones* para tratar con cualquier problema que se origine cuando el programa se esté ejecutando. Este capítulo examina como las palabras clave **try**, **catch**, **throw**, **throws**, y **finally** trabajan

en Java; cuando se suele lanzar excepciones y cuando capturarlas. Además se verán las excepciones estándar de Java, como crear excepciones propias, que sucede con las excepciones en el constructor y como los manejadores de excepción son localizados.

### **Capítulo 11: El sistema de E/S de Java**

Teóricamente, se puede dividir cualquier programa en tres partes, entrada, proceso y salida. Esto implica que E/S (entrada/salida) es una parte importante de la ecuación. En este capítulo se aprenderá acerca de las diferentes clases que java proporciona para leer y escribir ficheros, bloques de memoria y la consola. Las diferencias entre el “viejo” y el “nuevo” sistema de E/S de java será mostrado. Además, este capítulo examina el proceso de tomar un objeto, “hacerlo fluir” (para ponerlo en un disco o enviarlo a través de la red) y reconstruirlo, lo que es manejado por la *serialización de objetos* de Java. También las librerías de compresión de Java, que son utilizadas en el formato para archivar ficheros de java (JAR Java ARchive), son examinados)

### **Capítulo 12: Identificación de tipo en tiempo de ejecución**

La identificación de tipos en tiempo de ejecución de Java (RTTI run-time type identification) posibilita encontrar el tipo exacto de un objeto cuando solo se tiene referencia al tipo base.

Normalmente, se quiere intencionalmente ignorar el tipo exacto de un objeto y se deja que el mecanismo de enlace dinámico (polimorfismo) implementen el comportamiento correcto para ese tipo. Pero ocasionalmente ayuda mucho saber el tipo exacto de cada objeto para el cual solo se tiene una referencia a la base. A menudo esta información ayuda a realizar una operación especial de conversión mas eficientemente. Este capítulo explica para que es RTTI, como utilizarla, y como deshacerse de ella cuando no pertenece ahí. Además, este capítulo introduce el mecanismo de *reflexión* de Java.

### **Capítulo 13: Creando Ventanas y Applets**

Java se distribuye con la librería GUI “Swing”, que es un grupo de clases que manejan ventanas de una forma portátil. Estos programas para armar ventanas pueden ser applets o aplicaciones que funcionen solas. Este capítulo es una introducción a Swing y a la creación de applets para la Web. La importante tecnología “JavaBeans” es introducida. Esto es fundamental para la creación de herramientas para crear programas de Desarrollo Rápido de Aplicaciones (RAD Rapid-Application Development).

### **Capítulo 14: Múltiples hilos**

Java proporciona una facilidad incluida para soportar múltiples tareas concurrentes, llamadas *hilos*, que se ejecutan con un solo programa (A no ser que tenga múltiples procesadores en su máquina, esto es solo una *apariencia* de múltiples tareas). A pesar de que esto puede ser utilizado en cualquier parte, los hilos son más visibles cuando se trata de crear una interfase de usuario con respuesta así es que, por ejemplo, un usuario no es impedido de presionar un botón o de entrar datos cuando algún proceso se ejecuta. Este capítulo hace una mirada en la sintaxis y la semántica de la multitarea en Java.

## **Capítulo 15: Computación distribuida**

Todas las características y librerías de Java parecen realmente venir juntas cuando se comienza a escribir a través de redes. Este capítulo explora la comunicación a través de las redes y de la Internet, y las clases que Java proporciona para hacer esto fácil. Este introduce a los muy importantes conceptos de *Servlets* y *JSPs* (Para programación del lado del servidor), mediante *Java Conectividad a bases de datos de Java* (JDBC Java DataBase Connectivity), e *Invocación de Remota de Métodos* (RMI Remote Method Invocation). Finalmente, hay una introducción a las nuevas tecnologías de *JINI*, JavaSpaces, y *Enterprise JavaBeans* (EJBs).

## **Apéndice A: Pasando y retornando objetos**

Dado que la única forma de comunicarse con objetos en Java es a través de las referencias, el concepto de pasar un objeto a una función y retornar un objeto de una función tiene consecuencias interesantes. Este apéndice explica que se necesita conocer para manejar objetos cuando se está moviendo adentro y afuera de funciones, y también mostrar la clase **String**, que utiliza una estrategia diferente para el problema.

## **Apéndice B: La interfase nativa de Java (JNI)**

Un programa totalmente portátil en Java tiene serios inconvenientes: velocidad y la inhabilidad de acceder a servicios específicos de la plataforma. Cuando se sabe la plataforma que se está ejecutando, es posible aumentar dramáticamente la velocidad de ciertas operaciones haciéndolas *métodos nativos* que son funciones que son escritas en otros lenguajes de programación (actualmente, solo C/C++ son soportados). Este apéndice da suficiente introducción para esta característica para ser capaz de crear ejemplos simples de la interfase con código que no sea Java.

## **Apéndice C: Guía de programación Java**

Este apéndice contiene sugerencias que ayudarán cuando se ejecute un programa con diseño y código de bajo nivel.

#### **Apéndice D: Lectura recomendada**

Una lista de los libros de Java que he encontrado particularmente útiles.

## Ejercicios

He descubierto que los ejercicios simples son excepcionalmente útiles para completar el entendimiento de los estudiantes durante un seminario, así es que encontrara un grupo en el final de cada capítulo.

La mayoría de los ejercicios están diseñados para ser lo suficientemente fáciles que ellos puedan ser terminados en un tiempo razonable en una situación de salón de clase mientras el instructor observa, asegurándose que todos los estudiantes se concentran en el material. Algunos ejercicios son mas avanzados para prevenir el aburrimiento para los estudiantes experimentados. La mayoría están diseñados para ser solucionados en un corto tiempo y probados y puliendo su conocimiento. Algunos son mas desafiantes, pero no presenta grandes desafíos (Presumiblemente el lector puede encontrar el suyo propio -o mejor, el problema lo encontrara a el).

La solución de los ejercicios pueden ser encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

## CD ROM multimedia

Hay dos CD ROMs multimedia asociados con este libro. El primero esta asociado con este libro. El primero se encuentra en el mismo libro: *Thinking in C*, descrito en el final del prefacio, el que prepara para el libro dando la velocidad necesaria con la sintaxis C que se necesita para ser capaz de entender Java.

Un segundo CD ROM multimedia esta disponible, el cual esta basado en el contenido de este libro. Este CD ROM es un producto separado y incluye la totalidad de el seminario de entrenamiento “Hands-On Java”. Esto es mas de 15 horas de lecturas que he grabado, sincronizado con cientos de transparencias con información. Dado que el seminario esta basado en este libro, es un acompañamiento ideal.

El CD ROM contiene todas las lecturas (con la importante excepción de la atención personalizada!) de los cinco días totalmente inmerso de

entrenamiento de seminarios. Creo que he ajustado un nuevo estándar de calidad.

El CD ROM con el seminario Hands-On esta disponible solo ordenándolo directamente del sitio Web [www.BruceEckel.com](http://www.BruceEckel.com).

## Código fuente

Todo el código fuente para este libro esta disponible bajo la licencia de libre acceso, distribuido como un paquete por separado, visitando el sitio Web [www.BruceEckel.com](http://www.BruceEckel.com). Para asegurarse de que se tiene la versión mas actual, este el sitio oficial para la distribución del código y la versión electrónica del libro. Se pueden encontrar versiones en servidores espejo versiones electrónicas del libro en otros sitios (algunos de estos sitios se pueden encontrar en [www.BruceEckel.com](http://www.BruceEckel.com)), pero se debería verificar en el sitio oficial para asegurarse que la versión en el espejo es actualmente la edición mas reciente. Se puede distribuir el código en el salón de clase y en otras situaciones educacionales.

La meta primaria de los derechos de autor es asegurar que los fuentes del código es adecuadamente citado, y para prevenir que no se puede publicar el código en un medio impreso sin permiso (siempre y cuando la fuente sea citado, utilizando ejemplos del libro en la mayoría de los medios generalmente no es un problema).

En cada fichero de código fuente encontrará una referencia a la siguiente advertencia de derechos de autor:

```
///:! :CopyRight.txt
Copyright ©2000 Bruce Eckel
Fichero de código fuente de la 2da edición del libro "Thinking en
Java". Todos los derechos reservados EXCEPTO de la forma en
descripta en las siguientes instrucciones:
```

Usted puede libremente utilizar este fichero para su propio trabajo (personal o comercial), incluyendo modificaciones en forma ejecutable solamente. Se otorga permiso para utilizar este fichero en situaciones de salón de clase, incluyendo el uso en materiales de presentación, siempre y cuando el libro "Thinking en Java" sea citado como la fuente.

Exceptuando las situaciones de salón de clase, incluyendo no se puede copiar y distribuir este código; a no ser que el punto de partida de distribución sea <http://www.BruceEckel.com> (y los espejos oficiales) en donde esta disponible libremente. No se puede distribuir versiones del código fuente modificada en este paquete. No se puede utilizar este fichero en medios impresos sin el permiso expreso del autor. Bruce Eckel no realiza representaciones acerca de la aplicabilidad de este software para ningún propósito. Esto es proporcionado "como esta" sin garantías explícitas o implícitas de cualquier tipo, incluyendo y implícitamente garantías

de comerciabilidad, adaptabilidad para cualquier propósito o infracción. La totalidad del riesgo así como la calidad y rendimiento del software es suyo. Bruce Eckel y el editor no serán responsables por cualquier daño sufrido por usted o cualquier tercera parte como resultado de utilizar o distribuir software. En ningún evento Bruce Eckel o el editor son responsables por ninguna pérdida de renta, beneficio, o datos, o por directa, indirecta, especial, consecuente, incidental, o daños punitivos, sin embargo causado e independientemente de la teoría de responsabilidad, se pueden levantar por el uso o la inhabilidad de utilizar el software incluso si Bruce Eckel y el editor tienen un aviso de la posibilidad de estos daños.- Debe el software ser probado de cualquier defecto, usted asume el costo de todos los servicios necesarios, reparaciones, o correcciones. Si piensa que ha encontrado un error, por favor sométalo a consideración utilizando el formulario que encontrará en [www.BruceEckel.com](http://www.BruceEckel.com). (Por favor utilice la misma forma para errores que no sean de código encontrados en el libro).

///:~

Se puede utilizar el código en sus proyectos y en el salón de clase (incluyendo en la presentación de materiales) siempre y cuando el aviso de derecho de autor sea mantenido.

## Estándares de código

En el texto de este libro, los identificadores (funciones, variables, y nombres de clases) son colocados en negrita. La mayoría de las palabras clave son colocadas en negrita, excepto aquellas palabras clave que son utilizadas mucho y la negrita se vuelva tediosa, como lo es “class”.

Utilizo un estilo de código particular en los ejemplos en este libro. Este estilo sigue el estilo que Sun utiliza en virtualmente todo el código que encontrará en este sitio (vea [java.sun.com/doc/codeconv/index.htm](http://java.sun.com/doc/codeconv/index.htm)), y parece ser soportado por la mayoría de los ambientes de desarrollo en Java. Si ha leído los otros trabajos, deberá también advertir de que el estilo de codificación de Sun coincide con el mío -esto me satisface, a pesar de que no tengo nada que ver con esto. El tema del formateo de estilo es bueno para momentos de fuertes debates, así es que solo diré que no estoy tratando de indicar un estilo correcto mediante mis ejemplos; tengo mis propias motivaciones para utilizar el estilo que utilizo. Dado que Java es una forma libre de programación, puede utilizar el estilo que quiera si está confortable con él.

Los programas en este libro son ficheros que están incluidos por el procesador de palabras en el texto, directamente de ficheros compilados. De esta manera, los ficheros de código impreso en este libro pueden todos trabajar sin errores de compilación. Los errores que *suelen* causar mensajes de error en tiempo de compilación son comentados mediante //! de esta forma pueden ser fácilmente descubiertos y probados utilizando medios automáticos. Los errores descubiertos y reportados al autor aparecerán

primero en el código fuente distribuido y luego en las actualizaciones del libro (que aparecerán también en el sitio Web [www.BruceEckel.com](http://www.BruceEckel.com)).

## Versiones de Java

Generalmente confío en la implementación de Java de Sun como una referencia cuando determino si un comportamiento es correcto.

Hasta ahora, Sun ha liberado tres versiones principales de Java: 1.0, 1.1 y 2 (que es llamada versión 2 a pesar de que las versiones de JDK de Sun continúan utilizando el esquema de numeración 1.2, 1.3, 1.4, etc). Parece que la versión 2 finalmente trae a Java a los primeros puestos, en particular en lo que concierne a herramientas de interfase de usuario. Este libro se enfoca en Java 2 y esta probado con Java 2 así es que el código compilará sobre Linux (mediante el Linux JDK que está disponible en el momento en que se escribió esto).

Si necesita aprender acerca de versiones anteriores del lenguaje que no están cubiertas en esta edición, la primera edición de este libro se puede bajar libremente en [www.BruceEckel.com](http://www.BruceEckel.com) y también esta en el CD que se encuentra dentro de este libro.

Una cosa que debe advertir es que, cuando necesito mencionar versiones anteriores del lenguaje, no utilizo los números de revisión secundarios. En este libro me referiré a Java 1.0, 1.1 y 2 solamente, para cuidarme de los errores topográficos producidos por futuras revisiones secundarias de estos productos.

## Seminarios y consejos

Mi empresa proporciona cinco días, de seminarios de entrenamiento en la mano, públicos y para su hogar basados en el material de este libro. El material seleccionado de cada capítulo representa una lección, que es seguida por un período de ejercicio monitoreado así es que cada estudiante recibe atención personal. Las lecturas de audio y transparencias para el seminario de introducción es también capturado en CD ROM para proporcionar al menos alguna de la experiencia del seminario sin viajes ni gastos. Por mas información, vaya a [www.BruceEckel.com](http://www.BruceEckel.com).

Mi empresa también proporciona consultoría, tutores y servicios de chequeos simples de programas para ayudarlo en su proyecto en el ciclo de desarrollo en especial con su primer proyecto Java de su empresa.

# Errores

No importa cuantos trucos utilice un escritor para detectar errores, siempre hay alguno que se arrastra adentro y a menudo brinca en la pagina en busca de un lector tierno.

Hay un enlace para someter a consideración los errores en el comienzo de cada capítulo en la versión HTML de este libro (y en el CD ROM que se encuentra en la parte de atrás de este libro, que además se puede bajar libremente desde [www.BruceEckel.com](http://www.BruceEckel.com)) y también en el sitio Web, en la página de este libro. Si descubre algo que cree que es un error, por favor utilice este formulario para poner a consideración el error con su corrección sugerida. Si es necesario, incluya el fichero fuente original e indique cualquier modificación sugerida. Su ayuda será apreciada.

## Nota en el diseño de la cubierta

La cubierta de *Thinking in Java* esta inspirada en el movimiento Americano Art & Crafts, que comienza cerca de fin siglo y alcanza su punto mas alto entre 1900 y 1920. Comienza en Inglaterra como reacción a la producción mecánica y la Revolución Industrial y el estilo altamente ornamentado de la era Victoriana. Art & Crafts enfatiza el diseño frugal, las formas de la naturaleza son vistas en el movimiento de art nouveau, realizaciones a mano y la importancia de los artesanos, y a pesar de eso sin evadir el uso de herramientas modernas. Hay demasiados ecos con la situación que tenemos ahora: el cambio de siglo, la evolución de los toscos principiantes de la revolución de las computadoras a algo mas refinado y significativo para las personas individuales, y el énfasis en la mano de obra artesanal del software en lugar de solo crear código.

Veo a Java de esta misma forma: como un intento de elevar a el programador fuera de un sistema operativo mecánico y lanzarse hacia un “software artesanal”.

El autor y el diseñador de la cubierta del libro (los cuales son amigos desde pequeños) encontraron inspiración en este movimiento, y sus muebles, lámparas y otras piezas son originales o inspiradas en este período.

El otro tema de esta cubierta sugiere una caja de coleccionista que un naturalista puede utilizar para mostrar los especímenes de insectos que el o ella han preservado. Estos insectos son objetos, que son colocados dentro de la caja de objetos. La caja de objetos es colocada dentro de la “funda”, que

ilustra el concepto fundamental de la acumulación en la programación orientada a objetos. Claro, que un programador no puede ayudar pero hacer la asociación con “insectos”, y aquí los insectos son capturados y presumiblemente matados en una jaula de especímenes, y finalmente confinados dentro de una caja para mostrarlos, de la misma forma alude a la habilidad de Java para encontrar, mostrar y doblegar errores (lo cual es de verdad uno de sus atributos más poderosos).

# Reconocimientos

Primero, gracias a los asociados que han trabajado conmigo dando seminarios, proporcionando consultoría y desarrollando proyectos educativos: Andrea Provaglio, Dave Bartlett (quien contribuyó significativamente a el capítulo 15), Bill Venners, and Larry O’Brien. Aprecio tu paciencia cuando trataba de desarrollar el mejor modelo de compañeros como nosotros para trabajar juntos. Gracias a Rolf André Klaedtke (Suiza); Martin Vlcek, Martin Byer, Vlada & Pavel Lahoda, Martin la carga, y Hanka (Praga); and Marco Cantu (Italia) por hospedarme en el viaje de excursión de mi primer seminario en Europa organizado por mi mismo.

Gracias a la comunidad de la calles Street Cohousing por molestarme los dos años que me insistieron que escribiera la primera edición de este libro (y por involucrarse conmigo con todo). Muchas gracias a Kevin y Sonda Donovan por subarrendarme su gran lugar en el bellísimo Crested Butte, Colorado por el verano mientras trabajaba en la primer edición de este libro. Gracias también a los amigables residentes de Crested Butte y el Laboratorio Biológico de las Montañas Rocosas quienes me hacen sentir tan bienvenido.

Gracias a Claudette Moore y a la agencia literaria Moore por su tremenda paciencia y perseverancia captando exactamente lo que yo quería.

Mis primero dos libros fueron publicados con Jeff Pepper como editor en Osborne/McGraw-Hill. Jeff apareció en el lugar correcto en el momento correcto en Prentice-Hall y ha aclarado el camino y hecho todas las cosas bien para hacer esto una experiencia de publicación placentera. Gracias, Jeff—esto significa mucho para mí.

Estoy especialmente endeudado con Gen Kiyooka y su empresa Digigami, quien con gentileza proporcionó mi servidor Web por los primeros años de mi presencia en la Web. Esto fue una invaluable ayuda externa.

Gracias a Cay Horstmann (co-autor de *Core Java* Prentice-Hall, 2000), D’Arcy Smith (Symantec), y Paul Tyma (co-author de *Java Primer Plus* El grupo blanco, 1996), por ayudarme a aclarar conceptos en el lenguaje.

Gracias a las personas que han hablado en mi área de Java en la Conferencia de Desarrollo de Software, y los estudiantes en mis seminarios, que preguntan las preguntas que necesito escuchar en orden de hacer el material mas claro.

Un especial agradecimiento a Larry y Tina O'Brien, que ayudaron a poner el seminario en el original Hands-On Java CD ROM (Se puede encontrar mas en [www.BruceEckel.com](http://www.BruceEckel.com)).

Muchas personas enviaron correcciones y estoy en deuda con ellos, pero un especial agradecimiento para (por la primera edición): Kevin Raulerson (encontró toneladas de errores grandes), Bob Resendes (simplemente increíble), John Pinto, Joe Dante, Joe Sharp (todos ellos son fabulosos), David Combs (por muchas correcciones gramaticales y correcciones en la claridad), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson, y un huésped de los otros. Prof. Ir. Marc Meurrens quien realizó un gran esfuerzo para publicar y crear la versión electrónica de la primer edición de el libro disponible en Europa.

Ha habido una avalancha de técnicos inteligentes en mi vida que se han convertido en amigos y han tenido también influencia de forma no usual ya que ellos hacen toga y practican otras formas de incremento espiritual, en el cual he encontrado bastante inspiración e instrucción. Ellos son Kraig Brockschmidt, Gen Kiyooka, y Andrea Provaglio (quienes ayudaron en el entendimiento de Java y la programación en general en Italia, y ahora en los Estados Unidos como una asociación en el equipo MindView).

No es tanto la sorpresa para que entender Delphi me ayudó a entender Java, dado que hay muchos conceptos y decisiones de diseño en común. Mis amigos de Delphi proporcionaron asistencia ayudándome a ganar comprensión en ese maravilloso ambiente de trabajo. Ellos son Marco Cantu (otro Italiano -tal vez el estar empapado en Latín le da una aptitud para los lenguajes de programación?), Neil Ruybenking (que ha usado para hacer el yoga, vegetarianismo y Zen hasta que descubrió las computadoras). y por su puesto a Zack Urlocker, un amigo de mucho tiempo con quien he recorrido el mundo,

Mi amigo Richard Hale Shaw's cuyo entendimiento y soporte ha sido muy útil (y a Kim, también). Richars y yo hemos perdido muchos meses dando seminarios juntos y tratando de trabajar con la experiencia del aprendizaje perfecto con los asistentes. Gracias también a KoAnn Vikoren, Eric Faurot, Marco Pardi, y el resto de la gente y tripulación en MFL. Gracias especialmente a Tara Arrowood, quien ha me ha inspirado nuevamente acerca de las posibilidades de las conferencias.

El diseño del libro, el de la cubierta, y la foto de tapa fueron creados por mi amigo Daniel Will-Harris, célebre autor y diseñador ([www.Will-Harris.com](http://www.Will-Harris.com)), quién ha utilizado para jugar con casi escritos en el comienzo del colegio donde esperaba la invención de las computadoras y sus escritorios para publicaciones, y quejándose de mi entre dientes con mis problemas de álgebra. Sin embargo he presentado las páginas listas yo solo, así es que los errores de escritura son míos. Microsoft® Word 97 para Windows fue utilizado para escribir el libro y para crear las páginas listas en Adobe Acrobat; el libro fue creado directamente de los ficheros PDF de Acrobat (como un tributo a la era electrónica, ha sucedido que las dos veces que el libro fue producido fue en el extranjero -la primera edición fue en Capetown, en Sudáfrica y la segunda edición fue en Praga). La fuente del cuerpo de *Georgia* y los títulos son en *Verdana*. El tipo de la cubierta es *ITC Rennie Mackintosh*.

Gracias a los proveedores que crearon los compiladores: Borland, el grupo Blackdown (por Linux), y por su puesto, Sun.

Un agradecimiento especial a todos mis profesores y a mis estudiantes (que también son mis profesores). El profesor de redacción mas divertido fue Gabrielle Rico (autor de *Writing the Natural Way* Putnam, 1983). Siempre atesoré esa fantástica semana en Esalen.

El grupo de amigos de soporte incluye, pero no esta limitado a: Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates en *Midnight Engineering Magazine*, Larry Constantine y Lucy Lockwood, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Andrea Rosenfield, Claire Sawyers, mas italianos (Laura Fallai, Corrado, Ilsa, y Cristina Giustozzi), Chris and Laura Strand, los alquimistas, Brad Jerbic, Marilyn Cvitanic, the Mabrys, los Haflingers, los Pollocks, Peter Vinci, la familia Robbins, la familia Moelter (y la McMillans), Michael Wilk, Dave Stoner, Laurie Adams, the Cranstons, Larry Fogg, Mike y Karen Sequeira, Gary Entsminger y Allison Brody, Kevin Donovan y Sonda Eastlack, Chester y Shannon Andersen, Joe Lordi, Dave y Brenda Bartlett, David Lee, los Rentschlers, los Sudeks, Dick, Patty, y Lee Eckel, Lynn y Todd, y sus familias. Y por su puesto, mamá y papá.

## Contribuyeron a través de la Internet

Gracias a aquellos que me ayudaron a escribir nuevamente los ejemplos para utilizar la librería Swing, y por otras asistencias: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens

Brandt, Nitin Shivaram, Malcolm Davis, y todos los que han expresado apoyo. Esto realmente me ha ayudado lograr el proyecto.

# 1: Introducción a los objetos

La génesis de la revolución de las computadoras fue en una máquina. La génesis de nuestros lenguajes de programación de esta forma tienden a verse como esta máquina.

Pero las computadoras no son solo máquinas tanto como herramientas de amplificación de la mente (“bicicletas para la mente”, como Steve Jobs las llamaba cariñosamente) y un tipo diferente de medio de expresión. Como resultado, las herramientas comienzan a verse menos como máquinas y más como parte de nuestras mentes, y también como otras formas de expresar cosas como escritura, pintura, escultura, animación y producciones de cine. La programación orientada a objetos (POO) es parte de este movimiento hacia la utilización de las computadoras como un medio de expresión.

Este capítulo introduce a el concepto de POO, incluyendo un vistazo general de los métodos de desarrollo. Este capítulo, y este libro, asume que se tiene experiencia en lenguajes de programación procesales, a pesar de que no sea necesario C. Si se piensa que se necesita mas preparación en programación y en la sintaxis de C antes de abordar este libro, se debería trabajar con el CD ROM de entrenamiento *Thinking in C: Fundation for C++ y Java* que se encuentra incluido en este libro y disponible en [www.BruceEckel.com](http://www.BruceEckel.com).

Este capítulo es un material de fondo y suplementario. Muchas personas no se sienten confortables arremetiendo con la programación orientada a objetos sin entender una imagen general primero. De esta forma, hay muchos conceptos que son introducidos aquí para darle un vistazo general sólido de la POO. Sin embargo muchas otras personas no tienen los conceptos generales hasta que hayan visto algunos de los mecanismos primero; estas personas pueden retrasarse y perderse sin que hayan intentado con algo de código. Si se es parte de este último grupo y se está ansioso de obtener cosas específicas del lenguaje, siéntase libre de saltar este capítulo -saltarse el capítulo en este punto no impedirá que se escriban programas o que se aprenda el lenguaje. Sin embargo, se puede querer regresar aquí para completar las el conocimiento y así se puede entender por que los objetos son importantes y como diseñar con ellos.

# El progreso de abstracción

Todos los lenguajes de programación proporcionan abstracciones. Se puede debatir que la complejidad de los problemas que son capaces de resolver esta directamente relacionado con el tipo y la calidad de la abstracción. Por “tipo” quiero decir, “¿Que es lo que se esta abstrayendo?”. El lenguaje ensamblador es una pequeña abstracción de las capas mas bajas de la máquina. Muchos lenguajes llamados “imperativos” que lo siguen (como lo es Fortran,, BASIC, y C) que son abstracciones del lenguaje ensamblador. Estos lenguajes son grandes mejoras sobre el lenguaje ensamblador, pero sus abstracciones primarias siguen requiriendo que se piense en términos de la estructura de la computadora en lugar de la estructura del problema que se está tratando de resolver. El programador debe establecer la asociación entre el modelo de la máquina (el “espacio de solución”, que es el lugar donde se esta modelando ese problema, como una computadora) y el modelo del problema que esta siendo solucionado (en el “espacio de problema”, que es el lugar donde el problema existe). El esfuerzo requerido para realizar este mapeo, y el hecho de que esto es extrínseco a el lenguaje de programación, produce que los programas sean difíciles de escribir y caros de mantener, y como un efecto secundario creado la industria entera de “métodos de programación”.

La alternativa para modelar esta máquina es modelar el problema que esta tratando de resolver. Los primeros lenguajes como lo es LISP y APL, eligen puntos de vista particulares del mundo (“Todos los problemas son finalmente listas” o “Todos los problemas son algoritmos”, respectivamente). PROLOG convierte todos los problemas en cadena de decisiones. Los lenguajes han sido creados basados en restricciones de programación y para programar exclusivamente manipulando símbolos gráficos (Actualmente ha probado ser muy restrictivo). Cada una de estas aproximaciones es una buena solución para una clase de problema particular para el cual esta diseñado, pero cuando se da un paso fuera de los dominios de solución se convierte en torpes.

La estrategia de la programación orientada a objetos da un paso mas adelante proporcionando herramientas para que el programador represente elementos en el espacio del problema. Esta representación es suficientemente general para que el programador no este restringido a un tipo de problema en particular. Nos referimos a los elementos en el espacio del problema y su representación en el espacio de la solución como “objetos” (Claro, necesitará también otros objetos que no tienen problema de espacios análogos). La idea es que al programa se le esta permitido adaptarse solo a el idioma del problema para agregar nuevos tipos de objetos, así es que cuando lea el código describiendo la solución, estará leyendo palabras que también expresan el problema. Esto es la abstracción de un lenguaje mas

flexible y poderosa que hemos tenido. De esta manera, la POO permite describir el problema en términos del problema, en lugar de hacerlo en términos de la computadora donde la solución va a ser ejecutada. Sin embargo sigue habiendo una conexión de fondo con la computadora. Cada objeto se ve bastante como una pequeña computadora; tiene un estado, y operaciones que se pueden pedir que se realicen. Sin embargo, esto no parece una mala analogía para los objetos en el mundo real -ellos tienen características y comportamientos.

Algunos diseñadores de lenguajes han decidido que la programación orientada a objetos por si misma no es adecuada para solucionar fácilmente todos los problemas de programación, y defienden la combinación de varias estrategias en un *multi paradigm* de lenguajes de programación<sup>1</sup>.

Alan Kay resume cinco características de Smalltalk, el primer lenguaje orientado a objetos exitoso y uno de los lenguajes sobre el cual está basado Java. Estas características representan una estrategia pura de programación orientada a objetos.

1. **Todo es un objeto.** Hay que pensar que un objeto como una variable de figurativa; ésta almacena datos, pero se le puede "hacer pedidos" a ese objeto, pidiéndole que realice operaciones él mismo. En teoría, se puede tomar cualquier componente conceptual en el problema que se está tratando de resolver (perros, edificios, servicios, etc.) y representarlos como un objeto en su programa.
2. **Un programa es un conjunto de objetos indicándose entre sí que hacer mediante mensajes.** Para realizar un pedido a un objeto, se "envía un mensaje" a ese objeto. Mas concretamente, se puede pensar en un mensaje como una petición para llamar a una función que pertenece a un objeto particular.
3. **Cada objeto tiene su propia memoria compuesta por otros objetos.** Puesto de otra forma, se puede crear un nuevo tipo de objeto haciendo un paquete que contenga objetos existentes. De esta forma, se puede crear complejidad en un programa ocultándola detrás de la simplicidad de los objetos.
4. **Cada objeto tiene un tipo.** Utilizando la forma de hablar, cada objeto es una *instancia de una clase* en la cual "clase" es sinónimo de "tipo". La más importante característica distintiva de una clase es "¿Qué mensajes se les pueden enviar?".
5. **Todos los objetos de un tipo particular pueden recibir los mismos mensajes.** Esto es actualmente una afirmación falsa, como se verá más adelante. Dado que un objeto del tipo "círculo" es también un objeto

---

<sup>1</sup> Vea *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

del tipo “forma”, esta garantizado que un círculo pueda recibir mensajes del tipo forma. Esto significa que se puede escribir código que se comunique con formas y automáticamente manejar cualquier cosa que encaje en la descripción de forma. Esta *sustituibilidades* uno de los conceptos mas poderosos en la POO.

## Un objeto tiene una interfase

Aristóteles fue probablemente el primero que comenzó a estudiar cuidadosamente el concepto de *tipo*; él hablaba de “la clase de peces y la clase de pájaros”. La idea de todos los objetos, aún siendo únicos, son también parte de una clase de objetos que tienen características y comportamientos en común fue utilizada directamente en el primer lenguaje orientado a objetos, Simula-67, donde es fundamental la palabra clave **class** que introduce un nuevo tipo en un programa.

Simula, como su nombre implica, fue creado para desarrollar simulaciones como el clásico “problema del cajero del banco”. En este, se tiene un grupo de cajeros, clientes, cuentas, transacciones, y unidades de moneda -un montón de objetos”. Objetos que son idénticos excepto por como se manifiestan durante la ejecución de un programa donde son agrupados juntos en “clases de objetos” y ahí es donde la palabra clave **class** aparece. El crear tipos de datos abstractos (clases) es el concepto fundamental en la programación orientada a objetos. Los tipos de datos abstractos trabajan al menos exactamente como los tipos incluidos: Se pueden crear variables de un tipo (llamadas *objetos* o *instancias* en el lenguaje de la orientación a objetos) y manipular esas variables (llamados *mensajes enviadoso peticiones*; se envía un mensaje y el objeto se figura que hacer con el). Los miembros (elementos) de cada clase comparten algunas cosas en común: cada cuenta tiene un balance, cada cajero puede aceptar un depósito, etc. En el mismo momento, cada miembro tiene su propio estado, cada cuenta tiene diferente balance, cada cajero tiene un nombre. De esta manera, los cajeros, clientes, cuentas, transacciones, etc., pueden ser representados mediante una única entidad en el programa de computadoras. Esta entidad es el objeto, y cada objeto pertenece a una clase en particular que define sus características y comportamientos.

Así es que, a pesar de que lo que realmente con la programación orientada a objetos es crear nuevos tipos de datos, virtualmente todos los objetos de los

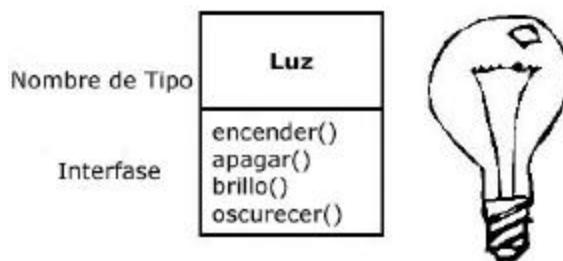
lenguajes de programación orientada a objetos utilizan la palabra clave “class”. Cuando ve la palabra “tipo” se piensa en “clase” y viceversa<sup>2</sup>.

Dado que una clase describe un grupo de objetos en particular que tienen idénticas características (elementos datos) y comportamientos (funcionalidad), una clase es realmente un tipo de datos porque un número de punto flotante, por ejemplo, también tiene un grupo de características y comportamientos. La diferencia es que el programador define una clase para encajar en un problema en lugar de comenzar a forzar el uso de un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Se extiende el lenguaje de programación agregando nuevos tipos de datos específicos a sus necesidades. El sistema de programación le da la bienvenida a la nueva clase y les da toda la atención y chequeo de tipo que le da a los tipos incluidos.

La estrategia de la orientación a objetos es no limitarse a construir simulaciones. Se este o no de acuerdo con que cualquier programa es una simulación del sistema que se está diseñando, el uso de técnicas de POO pueden reducir un gran grupo de problemas a una solución simple.

Una vez que una clase es establecida, se puede crear tantos objetos de esa clase como se quiera, y luego manipular esos objetos como si fueran los elementos que existen en el problema que está tratando de solucionar. Claro que, uno de los retos de la programación orientada a objetos es crear una función uno a uno entre los elementos del espacio del problema y los objetos del espacio solución.

¿Pero, como obtenemos un objetos que haga algo útil? Debe haber por ahí una forma de realizar una petición a un objeto de tal manera que haga algo, como completar una transacción, dibujar algo en la pantalla, o mover una llave. Y cada objeto puede satisfacer solo ciertos pedidos. Los pedidos que se le pueden hacer a un objeto determinan su *interfase* y el tipo es lo que determina la interfase. Un ejemplo simple puede ser la representación de una bombita de luz:



```
| Light lt = new Light();
```

<sup>2</sup> Algunas personas hacen una distinción, declarar un tipo determina la interfase en donde la clase es una implementación particular de esa interfase.

```
| lt.on();
```

La interfase establece *que* peticiones se pueden realizar a un objeto particular. Sin embargo, debe haber código en algún lugar para satisfacer esa petición. Esto, junto con los datos ocultos, comprenden la *implementación*. Desde un punto de vista de programación procesal, no es complicado. Un tipo tiene una función asociada con cada posible petición, y cuando se realiza una petición particular a un objeto, la función es llamada. Este proceso es usualmente resumido diciendo que se “envía un mensaje” (se realiza una petición) a un objeto, y el objeto resuelve que hacer con el mensaje (el ejecuta el código).

Aquí, el nombre del tipo/clase es **Light**, el nombre de este objeto **Light** en particular es **lt**, y las peticiones que se pueden hacer a un objeto **Light** son encender, apagar, darle brillo u oscurecerla. Se crea un objeto **Light** definiendo una “referencia” (**lt**) para un objeto y utilizando **new** para pedir un nuevo objeto de ese tipo. Para enviar un mensaje a el objeto, se establece el nombre del objeto y se conecta con la petición del mensaje mediante un período (punto). Desde el punto de vista del usuario de una clase predefinida, esto es mas o menos todo lo que a programación con objetos se refiere.

El diagrama mostrado mas arriba sigue el formato de *Unified Modeling Language* (UML). Cada clase es representada mediante una caja, con un nombre de tipo en la parte superior de la caja, y miembros datos que se escriben en el medio de la caja, y las *funciones miembro* (las funciones que pertenecen a ese objeto, que reciben los mensajes que se envían al objeto) en la parte inferior de la caja. A menudo, solo el nombre de la clase y las funciones miembro públicas son mostradas en los diagramas de diseño UML, así es que la porción del medio no es mostrada. Si esta interesado solo en el nombre, entonces la porción de abajo no es necesario que se muestre tampoco.

## La implementación oculta

Es útil dividir los campos en juego en *creadores de clases*(aquellos que crean nuevos tipos de datos) y en *programadores clientes*<sup>3</sup> (los consumidores de la clase que utilizan los tipos de datos en sus aplicaciones). La meta del programador cliente es recolectar una caja de herramientas llena de clases para utilizar en un desarrollo rápido de una aplicación. La meta del creador de la clase es armar una clase que solo exponga lo necesario para el cliente programador y mantenga todo lo demás oculto. ¿Por que? Porque si es oculta, el cliente programador no puede utilizarla, lo que significa que el creador de la clase puede cambiar la porción oculta a

---

<sup>3</sup> Estoy en deuda con mi amigo Scott Meyers por este término.

voluntad sin preocuparse acerca del impacto en cualquier otro. La porción oculta usualmente representa la parte delicada dentro de un objeto que puede ser fácilmente corrompido por un descuido o un programador cliente desinformado, así es que ocultar la implementación reduce los errores de programación. Al concepto de implementación oculta no se le puede dar demasiada importancia.

En una relación es importante tener límites que sean respetados por todas las partes involucradas. Cuando se crea una librería, se establece una relación con el cliente programador, quien es también un programador, pero uno que está juntando todo en una aplicación utilizando librerías que no son de él, posiblemente para crear una librería más grande.

Si todos los miembros de una clase están disponibles para todos, entonces el cliente programador puede hacer cualquier cosa con la clase y no hay forma de establecer reglas. Aun cuando realmente se prefiera que el cliente programador no manipule directamente algunos de los miembros de su clase, sin control de acceso no hay forma de prevenirlo. Todo está desnudo para el mundo.

Así es que la primera razón para realizar un control de acceso es para mantener las manos de los clientes programadores fuera de las partes que no deben tocar -partes que son necesarias para las maquinaciones internas de los tipos de datos pero que no son parte de la interfase que el usuario necesita para solucionar su problema en particular. Esto es actualmente un servicio a los usuarios porque ellos pueden fácilmente ver qué es importante para ellos y qué deben ignorar.

La segunda razón para controlar el acceso es permitir a los diseñadores de la librería cambiar el funcionamiento interno de la clase sin preocuparse acerca de cómo afectará a el cliente programador. Por ejemplo, se debe implementar una clase particular en una forma elegante para un desarrollo cómodo, y luego, más tarde descubrir que se necesita volver a escribir para hacerla correr más rápido. Si la interfase y la implementación son claramente separadas y protegidas, se puede lograr esto fácilmente.

Java utiliza tres palabras explícitas para marcar los límites en una clase: **public**, **private** y **protected**. Su uso y significado es muy directo. Estos *especificadores de acceso* determinan quién puede utilizar las definiciones que siguen. **public** significa que las siguientes definiciones están disponibles para cualquiera. La palabra clave **private**, por el otro lado, significa que nadie puede acceder a estas definiciones excepto el creador del tipo, dentro de las funciones miembro de ese tipo, **private** es un muro de ladrillo entre el creador y el programador cliente. Si alguien trata de acceder a un miembro **private**, tendrá un error en tiempo de compilación. **protected** funciona como **private**, con la excepción que las clases heredadas tienen acceso a miembros

**protected**, pero no a miembros **private**. La herencia será introducida en breve.

Java tiene también un acceso por “defecto”, que comienza a jugar si no se utiliza uno de los especificadores mencionados anteriormente. Esto es a veces llamado acceso “amigable” porque las clases pueden acceder a los miembros amigables de otras clases en el mismo paquete, pero fuera del paquete estos miembros aparentan ser **private**.

## Reutilizando la implementación

Una vez que una clase ha sido creada y probada, debería (idealmente) representar una unidad de código útil. La reutilización no es tan fácil de alcanzar como muchos esperarían; toma experiencia y intuición producir un buen diseño. Pero una vez que se tenga el diseño, ruega por ser utilizado. La reutilización de código es una de las mas grandes ventajas que los lenguajes de programación orientada a objetos proporciona. La forma mas simple de reutilizar una clase es utilizando un objeto de esa clase directamente, pero solo se puede colocar un objeto de esa clase dentro de una nueva clase. Llamamos a esto “crear un objeto miembro”. La nueva clase puede ser creada con cualquier cantidad y tipos de otros objetos, en cualquier combinación que se necesite para alcanzar la funcionalidad deseada en la nueva clase. Dado que se esta componiendo una nueva clase de una clase existente, el concepto es llamado *composición* (o mas generalmente, *acumulación*). La composición es a menudo referida como una relación “tiene un”, como en “un auto tiene un motor”.



(El diagrama UML indica composición con el diamante relleno que afirma que hay un auto. Típicamente se utilizará una forma simple: solo una línea, sin el diamante para indicar una asociación<sup>4</sup>).

La composición acarrea una gran ventaja de flexibilidad. Los objetos miembros de su nueva clase son usualmente privados, haciéndolos inaccesibles a el cliente programador que esta utilizando la clase. Esto permite cambiar aquellos miembros si molestar el código cliente existente.

---

<sup>4</sup> Esto es usualmente suficiente detalle para la mayoría de los diagramas, y no se necesita especificar acerca de si se está utilizando acumulación o composición.

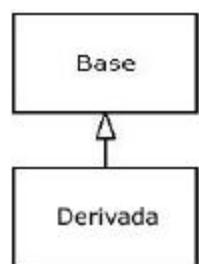
Se puede también cambiar los objetos miembros en tiempo de ejecución, para cambiar dinámicamente el comportamiento del programa. Herencia, que es lo siguiente que se describe, no tiene esta flexibilidad dado que el compilador debe colocar las restricciones en las clases creadas con herencias.

Dado que la herencia es tan importante en la programación orientada a objetos es muchas veces muy enfatizada, y el programador nuevo puede tener la idea que la herencia se debe utilizar en todas partes. Esto puede resultar torpe y en diseños demasiados complicados. En lugar de eso, se debería primero considerar la composición cuando se crean nuevas clases, dado que es muy simple y mas flexible. Si se toma esta estrategia, sus diseños serán mas claros. Una vez que se tenga alguna experiencia, será razonablemente obvio cuando se necesite herencia.

## Herencia: reutilización de la interfase

Por si solo, la idea de un objeto es una herramienta conveniente. Esto permite empaquetar datos y funcionalidad juntos en un *concepto*, así es que se puede representar un problema apropiadamente -el espacio de la idea en lugar de ser forzado a utilizar los idiomas de las capas mas bajas de la máquina. Estos conceptos son expresados como unidades fundamentales en el lenguaje de programación utilizando la palabra clave **class**.

Es una lástima, sin embargo, tener todos los problemas de crear una clase y luego estar forzado a crear una completamente nueva que pueda tener la misma funcionalidad. Sería bueno si pudiéramos tomar la clase existente, clonarla, y luego realizar agregados y modificaciones a el clon. Esto es efectivamente lo que tenemos con *herencia*, con la excepción de que la clase original (llamada la clase *base*, o clase *super*, o clase *padre*) sea cambiada, el “clon” modificado (llamada la clase *derivada* o clase *heredada*, o clase *sub*, o clase *hija*) también refleja estos cambios.

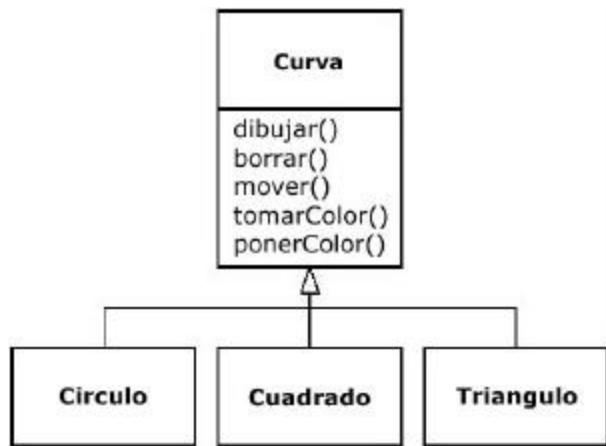


(La flecha en el diagrama UML mas arriba apunta de la clase derivada a la clase base. Como se va a ver mas adelante, pueden haber mas de una clase derivada.)

Un tipo hace mas que describir las restricciones de un grupo de objetos; también su relación con otros tipos de objetos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener mas características que otros y tal vez pueden también manejar mas mensajes (o manejarlos de forma diferentes). La herencia expresa esta similitud entre tipos utilizando conceptos de los tipos base y de los tipos derivados. Un tipo base contiene todas las características y comportamientos que comparte el tipo derivado de este. Si se crea un tipo base para representar el corazón de sus ideas acerca de algunos objetos en su sistema. Del tipo base, se puede derivar otros tipos para expresar las diferentes maneras que el corazón puede ser materializado.

Por ejemplo, una máquina para reciclar basura ordena piezas de basura. El tipo base es “basura” y cada pieza de basura tiene un peso, un valor, y otras cosas, y pueden ser desmenuzadas, fundidas, o descompuestas. De esto, tipos mas específicos de basura son derivados que pueden tener características adicionales (una botella tiene color) o comportamientos (el aluminio puede ser comprimido, el acero es magnético). Además, algunos comportamientos pueden ser diferentes (el valor del papel depende de su tipo y condición). Utilizando herencia, se puede crear un tipo de jerarquía que expresa el problema que se está tratando de resolver en términos de sus tipos.

Un segundo ejemplo es el de la clásico ejemplo de la “forma”, tal vez utilizado en un sistema asistido o en un juego de simulación. El tipo base es una “forma”, y cada forma tiene tamaño, color, posición y otras cosas. Cada forma puede ser dibujada, borrada, movida, coloreada, etc. De esto, tipos específicos de formas son derivados (heredados): círculo, cuadrado, triángulo y otros mas, cada uno de los cuales tiene características y comportamientos adicionales. Ciertas curvas pueden ser invertidas, por ejemplo. Algunos comportamientos pueden ser diferentes, como cuando se quiere calcular el área de una curva. El tipo de jerarquía envuelve las similitudes y diferencias entre las formas.

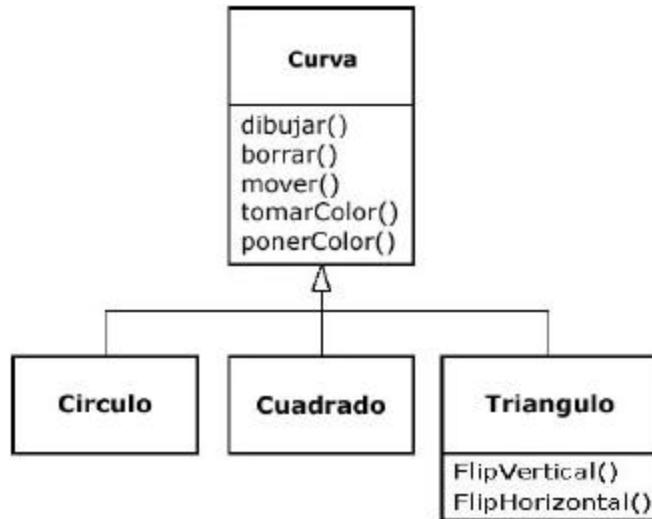


Convertir la solución en los mismos términos del problema es tremadamente beneficioso porque no necesita un montón de modelos intermedios para obtener de una descripción de un problema a la descripción de la solución. Con los objetos, el tipo de jerarquía es el modelo primario, así es que se va directamente de la descripción del sistema en el mundo real a la descripción del sistema en el código. Por supuesto, una de las dificultades que las personas tienen con el diseño orientado a objetos es lo que es muy simple obtener partiendo desde el principio a el fin. Una mente entrenada a ver soluciones complejas queda a menudo perplejo por la simplicidad al comienzo.

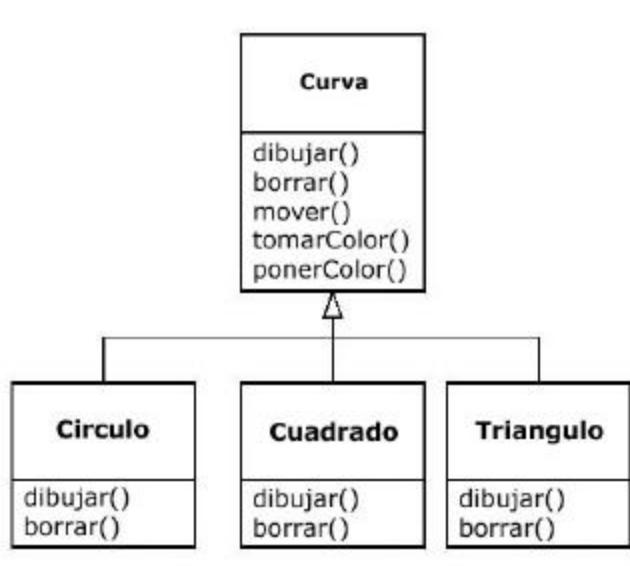
Cuando se hereda de un tipo existente, se crea un nuevo tipo. Este nuevo tipo contiene no solo todos los miembros del tipo existente (a pesar de que los privados son ocultos para los de afuera e inaccesibles), y mas importante, se duplica la interfase de la clase base. Esto es, todos los mensajes que se puede enviar a los objetos de la clase base pueden ser enviados a los objetos de la clase derivada. Dado que conocemos el tipo de la clase por los mensajes que se le pueden enviar, esto significa que la clase derivada *es del mismo tipo que su clase base*. En el ejemplo previo, “un circulo es una forma”. Este tipo de equivalencia mediante herencia es una de los puertas de enlace fundamentales para el entendimiento del significado de la programación orientada a objetos.

Dado que la clase base y la clase derivada tiene la misma interfase, debe haber alguna implementación para ir a través de su interfase. Esto es, debe haber algún código para ejecutar cuando un objeto recibe un mensaje en particular. Si simplemente se hereda una clase y no se hace nada mas, los métodos de la interfase de la clase base estarán presentes también en la derivada. Eso significa que los objetos de la clase derivada no solo tienen el mismo tipo , también tienen el mismo comportamiento, lo que no es particularmente interesante.

Hay dos maneras de diferenciar la clase derivada nueva de la clase base original. La primera es bastante directa: simplemente se agregan funciones completamente nuevas a la clase derivada. Estas funciones nuevas, no son parte de la interfase de la clase base. Esto significa que la clase base simplemente no hace mucho de lo que se quiere que haga, así es que se agregan mas funciones. Este uso simple y primitivo de la herencia es, por momentos, la solución perfecta a su problema. Sin embargo, se debe ver mas de cerca la posibilidad de que su clase base pueda también necesitar de estas funciones adicionales. Este proceso de descubrimiento y iteración de su diseño sucede regularmente en la programación orientada a objetos.



A pesar de que la herencia puede a veces implicar (especialmente en Java, donde la palabra clave que indica herencia es **extends**) que se esta agregando nuevas funciones a la interfase, lo que no es necesariamente cierto. La segunda y mas importante forma de diferenciar la nueva clase es *cambiar el comportamiento* de uno o mas funciones de la clase base. Esto es referido como sobrescribir la función.



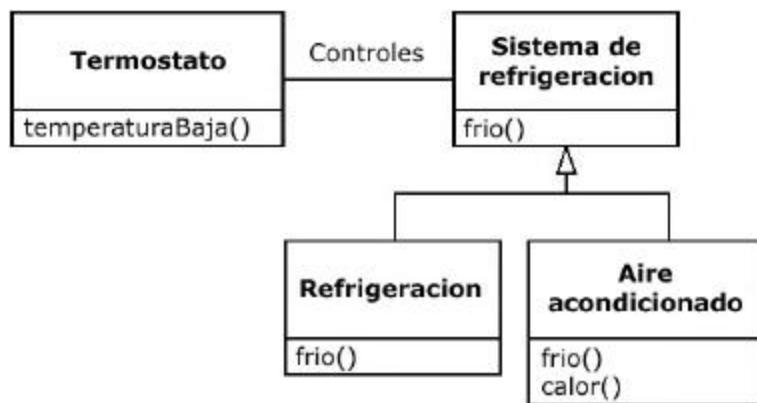
Para sobrescribir una función, simplemente se crea una nueva definición para la función en la clase derivada. Se puede decir, “Estoy utilizando la misma función de interfase aquí, pero quiero hacer algo diferente con mi nuevo tipo”.

## Relación es-una contra es-como-una

Hay un fuerte debate que sucede con la herencia: Debe la herencia sobreescribir *solo* las funciones de la clase base (y no agregar nuevas funciones miembro que no están en la clase base)? Esto significaría que el tipo derivado es *exactamente* el mismo tipo que la clase base dado que es exactamente la misma interfase. Como resultado, se puede sustituir un objeto de la clase derivada por un objeto de la clase base. Esto se puede realizar a través de *sustitución pura*, y a menudo es nombrada como *ley fundamental de sustitución*. En cierto sentido, esta es la forma ideal de tratar la herencia. A menudo nos referimos a la relación entre la clase base y las clases derivadas en este caso como una relación *es-una*, porque se puede decir “un circulo es una forma”. Una prueba para la herencia es determinar cuando se puede afirmar la relación *es una* acerca de las clases y hace que tenga sentido.

Hay veces en que se debe agregar elementos a la nueva interfase para un tipo derivado, de esta manera se extiende la interfase y se crea un nuevo tipo. El nuevo tipo puede seguir siendo sustituido por el tipo base, pero la sustitución no es perfecta porque las nuevas funciones no son accesibles

desde el tipo base. Esto puede ser descrito como una relación *es-como-una*<sup>5</sup>; el nuevo tipo tiene la interfase del viejo pero también contiene otras funciones, así es que se puede realmente decir que es exactamente lo mismo. Por ejemplo, considere un aire acondicionado. Supongamos una casa que está cableada con todos los controles para el aire acondicionado; esto es, tiene una interfase que permite controlarlo y refrigerar cuando es necesario. Imaginemos que el aire acondicionado se rompe y lo reemplaza con un equipo de aire acondicionado que puede calentar y refrigerar. Este es nuevo acondicionador *es-como* el viejo aire acondicionado, pero puede hacer más. Dado que el sistema de control de la casa está diseñado solamente para controlar la refrigeración, esta restringido a comunicar la parte para refrigerar del nuevo objeto. La interfase del nuevo objeto ha sido extendida, y el sistema existente no conoce nada acerca de esta exceptuando la interfase original.



Claro, una vez que se ve que el diseño comienza a aclarar que la clase base “sistema de refrigeración” no es suficientemente general, y debería ser renombrado a “sistema de control de temperatura” así es que se puede incluir el calor -en el punto en que el la ley fundamental de sustitución trabajara. Sin embargo, el diagrama anterior es un ejemplo de que puede suceder en el diseño y en el mundo real.

Cuando se ve la ley fundamental de sustitución es fácil sentir como que esta estrategia (sustitución pura) es la única forma de hacer las cosas, y que de echo *es bonito* si su diseño trabaja de esta forma. Pero encontrará que hay veces donde es igualmente claro que se agreguen nuevas funciones a la interfase de una clase derivada. Con análisis ambas clases suelen ser razonablemente obvias.

---

<sup>5</sup> Mi término

# Objetos intercambiables con polimorfismo

Cuando se trata con jerarquía de tipos, a menudo se quiere tratar un objeto no como el tipo específico que es, en lugar de eso se quiere tratar con su tipo base. Esto permite escribir código que no depende de tipos específicos. En el ejemplo de las formas, teníamos funciones que manipulaban formas genéricas sin considerar si eran círculos, cuadrados, triángulos o alguna forma que no había sido incluso definida todavía. todas las formas podían ser dibujadas, borradas y movidas, así es que esas funciones simplemente enviaban un mensajes a el objeto forma; no importaba acerca de como el objeto hacía frente con el mensaje.

Tal código es no puede ser afectado por los nuevos tipos adicionales, y agregar nuevos tipos es la forma mas común de extender un programa orientado a objetos para manejar nuevas situaciones. Por ejemplo, se puede derivar un nuevo tipo de una forma llamada pentágono sin modificar las funciones que tratan solo con las formas genéricas. Esta habilidad de extender un programa fácilmente derivando nuevos tipos es importante porque mejora enormemente los diseños reduciendo el costo del mantenimiento de software.

Sin embargo, hay un problema aquí, cuando se trata con objetos derivados como tipos base genéricos (círculos como formas, bicicletas como vehículos, cormoranes como pájaros, etc.). si una función va a indicarle a una forma genérica que se dibuje sola, o dirigir un vehículo genérico, o un pájaro genérico que se mueva, el compilador no podrá saber en tiempo de compilación que pieza de código precisamente será ejecutada. Este el punto integral .cuando un mensaje es enviado, el programador no *necesita* saber que pieza de código será ejecutada; la función dibujar puede ser aplicada igualmente a un círculo, a un cuadrado o a un triángulo y el objeto ejecutará el código apropiado dependiendo de su tipo específico. Si no se tiene que saber que pieza de código será ejecutada, entonces cuando se agregue un nuevo tipo, el código que se ejecute puede ser diferente sin requerir cambios en la función de llamada. ¿Por consiguiente, el compilador puede saber precisamente que parte del código es ejecutada, entonces, que es lo que hace? Por ejemplo, en el siguiente diagrama el objeto **ControladorDePajaros** solo trabaja con objetos **Pajaros** genéricos, y no conoce que hace el tipo exactamente. Esto es conveniente de la perspectiva del **ControladorDePajaros** porque no se tiene que escribir código especial para determinar el tipo exacto de **Pajaro** con el cual se esta trabajando, o el comportamiento del **Pajaro**. ¿Así es que como sucede esto, cuando llamamos a **mover()** a pesar de que ignoramos el tipo específico de **Pajaro**, el correcto

comportamiento ocurrirá (Un **Ganzo** corre, vuela o nada y un **Pinguino** corre o nada)?



La respuesta es la peculiaridad primaria de la programación orientada a objetos: el compilador no puede hacer una llamada a una función en el sentido tradicional. La llamada a función generada por un compilador no orientado a objetos produce lo que es llamado *enlazado temprano*, un término que tal vez no se ha escuchado antes porque nunca se ha hecho de otra forma. Esto significa que el compilador genera una llamada a un nombre de función específica, y el enlazador resuelve esta llamada a la dirección absoluta del código que será ejecutado. En la POO, el programa no puede determinar la dirección del código hasta el tiempo de ejecución, así es que otro esquema es necesario cuando un mensaje es enviado a un objeto genérico.

Para solucionar este problema, los lenguajes orientados a objetos utilizan el concepto de *enlazado diferido*. Cuando se envía un mensaje a un objeto, el código que será llamado no se determina hasta el tiempo de ejecución. El compilador no está seguro de que la función exista y realiza un chequeo de tipo con los argumentos y retorna un valor (un lenguaje en el cual esto no es verdad es llamado *weakly typed* que significa que son poco restrictivos con los tipos), y no conoce el código exacto a ejecutar.

Para realizar enlazado diferido, Java utiliza un pequeño trozo de código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo de la función, utilizando información almacenada en el objeto (este proceso está cubierto en mayor detalle en el Capítulo 7). De esta manera, cada objeto puede comportarse diferente de acuerdo a el contenido de esa porción de código. Cuando se envía un mensaje a un objeto, el objeto verdaderamente resuelve qué hacer con ese mensaje.

En algunos lenguajes (C++, en particular) se debe explícitamente indicar que se quiere tener la flexibilidad de las propiedades del enlazado diferido. En estos lenguajes, por defecto, las funciones miembro *no* son saltan de forma dinámica. Esto causó problemas, así es que en Java el enlazado

dinámico se realiza por defecto y no necesita recordar palabras clave extras para obtener polimorfismo.

Considere el ejemplo de la forma. La familia de clases (todas basadas en la misma interfase uniforme) fue diagramada antes en este capítulo. Para demostrar polimorfismo queremos escribir una sola parte de código que ignore los detalles específicos del tipo y se comunique solo con la clase base. Este código es *desacoplado* de la información específica del tipo, y de esta manera es mas simple de escribir y mas fácil de entender. Y, si un nuevo tipo -Un **Hexagono**, por ejemplo- es agregado mediante herencia, el código que se escribirá trabajara exactamente igual de bien para el nuevo tipo de **Curva** como lo hace con los tipos existentes. De esta forma el programa es *extensible*.

Si se escribe un método en Java (como se verá mas adelante que se hace):

```
void hasCosas(Curva s) {  
    s.borrar();  
    // ...  
    s.dibujar();  
}
```

Esta función conversa con cualquier **Curva**, así es que independientemente de el tipo de objeto que se esta dibujando o borrando. Si alguna otra parte del programa utilizará la función **hasCosas()**:

```
Circulo c = new Circulo();  
Triangulo t = new Triangulo();  
Linea l = new Linea();  
hasCosas (c);  
hasCosas (t);  
hasCosas (l);
```

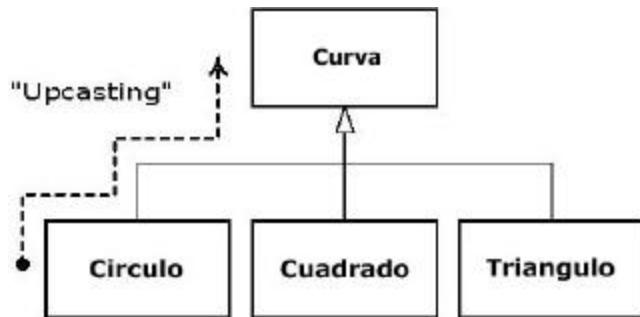
La llamada a **hasCosas()** automáticamente trabaja correctamente, sin importar el tipo exacto del objeto.

Esto es realmente un truco maravillosamente bonito. Considerando la línea:

```
hasCosas(c);
```

Que sucede aquí si es un **Circulo** es pasado en una función que espera una **Curva**. Dado que **Circulo** es una **Curva** puede ser tratado como una por **hasCosas()**. Esto es, cualquier mensaje que **hasCosas()** pueda enviar a una **Curva**, un **Circulo** puede aceptarla. Así es que es algo completamente seguro y lógico de hacer.

Llamamos a este proceso de tratar el tipo derivado como si fuera su tipo base *conversión ascendente o upcasting*. El nombre *cast* es utilizado en el sentido de conversión y el *up* viene de la forma en que el diagrama de herencia es típicamente arreglado, con el tipo base en la parte superior y la clase derivada de acomoda hacia abajo. De esta forma, convertir a un tipo base es moverse arriba en el diagrama de herencia: “upcasting”.



Un programa orientado a objetos contiene alguna conversión ascendente en alguna parte, dado que es como se desacopla uno del conocimiento del tipo exacto con el que se está trabajando. Vea el código en **hasCosas()**:

```

s.erase();
// ...
s.draw();

```

Debe percibirse que no dice “Si eres un **Círculo**, has esto, si eres un **Cuadrado**, has aquello, etc”. Si se está escribiendo ese tipo de código, que verifica todos los tipos que una **Curva** puede actualmente ser, es confuso y se necesitará cambiarlo cada vez que se agrega un nuevo tipo de **Curva**. Aquí, solo se necesita decir “Eres una forma, se que te puedes **borrar()** y **dibujar()** sola, hazlo, y ten cuidado de los detalles correctamente”.

Lo que es impresionante acerca del código de **hasCosas()** es que, de algún modo las cosas correctas se suceden. Llamando a **dibujar()** para un **Círculo** produce que se ejecute un código diferente cuando se llama a **draw()** para un **Cuadrado** o una **Línea**, pero cuando el mensaje **dibujar()** es enviado a una Curva anónima, el comportamiento correcto se sucede basado en el tipo actual de **Curva**. Esto es asombroso porque, como se ha mencionado antes, cuando el compilador de Java está compilando el código para **hasCosas()**, no se puede conocer exactamente qué tipo está manejando. Así es que comúnmente, se esperaría que se terminara llamando a la versión de **borrar()** y **dibujar()** para la clase base **Curva**, y no para la clase **Círculo**, **Cuadrado** o **Línea**. Y todavía las cosas correctas suceden gracias a el polimorfismo. El compilador y el sistema en tiempo de ejecución maneja los detalles; todo lo que se necesita saber es que sucede, y más importante aún es saber cómo diseñar con él. Cuando se envía un mensaje a un objeto, el objeto hará las cosas correctas, aún cuando una conversión ascendente esté involucrada.

## Clases bases abstractas e interfaces

A menudo en un diseño, se quiere que la clase base presente *solo* la interfase para las clases derivadas. Esto es, si no se quiere que cualquiera cree un objeto de la clase base, solo realizando una conversión ascendente de esta se puede utilizar la interfase. Esto se logra haciendo la clase *abstracta* utilizando la palabra clave **abstract**. Si alguien trata de crear un objeto de una clase **abstracta**, el compilador lo previene. Esto es una herramienta para forzar un diseño en particular.

Se puede utilizar también la palabra clave **abstract** para describir un método que no ha sido implementado todavía -como un cabo indicando “ha aquí una función de la interfase para todos los tipos que hereden de esta clase, pero en este punto no tiene ninguna implementación todavía”. Un método abstracto puede ser creado solo dentro de una clase abstracta. Cuando la clase es heredada, este método debe ser implementado, o la clase heredada se convierte también en abstracta. Crear un método abstracto permite colocar un método en una interfase sin ser forzado a proporcionar un cuerpo de código posiblemente sin sentido para ese método.

La palabra clave **interface** toma el concepto de la clase abstracta un paso mas adelante para prevenir del todo cualquier definición de función. La interfase es una herramienta muy conveniente y es comúnmente utilizada, ya que proporciona la perfecta separación entre interfase e implementación. Además, se pueden combinar muchas interfaces juntas, si se quiere, mientras que se heredar de muchas clases regulares o abstractas no es posible.

## Arquitectura de los objetos y tiempos de vida

Técnicamente, POO es solo acerca de clasificar datos abstractos, herencia y polimorfismo, pero otros temas pueden ser al menos tan importantes. El reato de esta sección cubre estos temas.

Uno de los factores mas importantes es la forma en que los objetos son creados y destruidos. ¿Dónde están los datos de un objeto y como es el tiempo de vida del objeto controlado? Hay diferentes filosofías de trabajo aquí. C++ toma la estrategia de que el control y la eficiencia es el tema mas importante, así es que le da al programador la elección. En busca de las máximas velocidades de ejecución, el almacenamiento y el tiempo de vida puede ser determinado cuando el programa es escrito, colocando los objetos en la pila (esto es a veces llamado variables *automáticas o scoped*) o en el

área de almacenamiento estático. Esto coloca una prioridad en la velocidad con que el espacio es asignado y liberado, y un control que puede ser muy valioso en algunas situaciones. Sin embargo, se sacrifica flexibilidad porque se debe saber exactamente la cantidad, el tiempo de vida, y el tipo de los objetos con el que se está escribiendo el programa. Si se está tratando de solucionar un problema más general como un diseño asistido por computadoras, el manejo de un almacén, o el control de tráfico aéreo, esto es muy restrictivo.

La segunda estrategia es crear objetos dinámicamente en un fondo común de memoria llamada el heap. En esta estrategia, no se necesita saber hasta en tiempo de ejecución cuantos objetos se necesitan, cuales son sus tiempos de vida o el tipo exacto que es. Esto es determinado con el estímulo del momento en que el programa se está ejecutando. Si se necesita un nuevo objeto, simplemente se crea uno en el heap en el momento que se necesita. Dado que el almacenamiento es manejado dinámicamente, en tiempo de ejecución, la cantidad de tiempo requerido para asignar el espacio en el heap es significativamente mayor que el tiempo para crear un espacio en la pila (Crear espacio en la pila es a menudo una simple instrucción de Ensamblador para mover el puntero de la pila abajo, y otro para regresarlo). La estrategia dinámica asume lógicamente que los objetos tiende a ser complicados, así es que una sobrecarga extra para encontrar espacio y liberarlo no tendrá un impacto importante en la creación de un objeto. Además, la gran flexibilidad es esencial para solucionar el problema general de programación.

Java utiliza una segunda estrategia, exclusivamente<sup>6</sup>. Cada vez que se quiere crear un objeto, se utiliza la palabra clave **new** para crear una instancia dinámica del objeto.

Hay otro tema, sin embargo, y es el tiempo de vida de un objeto. Con los lenguajes que se permite crear objetos en la pila, el compilador determina cuanto tiempo el objeto dura y puede automáticamente destruirlos. Sin embargo, si se crea en el heap el compilador no tiene conocimiento de este tiempo de vida. En un lenguaje como C++, se debe determinar mediante un programa cuando un objeto es destruido, lo que puede conducir a lagos de memoria si no se hace correctamente (y esto es un problema común en programas C++). Java proporciona una característica llamada recolector de basura que automáticamente descubre cuando un objeto no está mas en uso y lo destruye. Un recolector de basura automáticamente descubre cuando un objeto no se utiliza mas y lo destruye. Un recolector de basura es mucho mas conveniente porque reduce el número de temas que se deben tener presentes y el código que se debe escribir. Mas importante, el recolector de basura proporciona un nivel mucho mas alto de seguridad contra el

---

<sup>6</sup> Los tipos primitivos, como se aprenderá mas adelante, son un caso especial.

problema insidioso de los lagos de memoria (que ha tirado a muchos proyectos de C++ de rodillas).

El resto de esta sección da un vistazo en los factores adicionales concernientes a los tiempos de vida y arquitecturas.

## Colecciones e iteractores

Si no se tiene conocimiento de cuantos objetos se necesitan para resolver un problema en particular, o cuanto tiempo van a durar, tampoco conoce como almacenar esos objetos. ¿Como se puede saber cuanto espacio crear para estos objetos? No se puede, dado que esa información no se conoce hasta la ejecución del programa.

La solución a la mayoría de los problemas en diseños orientados a objetos parecen frívolos: se crea otro tipo de objeto. El nuevo tipo de objeto que soluciona este problema en particular contiene referencias a otros objetos. Claro, se puede hacer la misma cosa con un arreglo, el que está disponible en la mayoría de los lenguajes. Pero hay mas. Este nuevo objeto, generalmente llamado *contenedor* (también llamado una *colección*, pero la librería de Java utiliza el término con un sentido diferente así es que utilizaremos “contenedores”), se expandirá lo que sea necesario para acomodar todo lo que se coloque dentro de el. Así es que no se necesita saber cuantos objetos se van a almacenar en un contenedor. Solo se crea un objeto contenedor y se deja que el se encargue de los detalles.

Afortunadamente, un lenguaje de POO bueno viene con un grupo de contenedores como parte del paquete. En C++, es parte de la librería estándar de C++ y es a veces llamado la Librería Estándar de Plantillas (STL Standar Template Library). Pascal orientado a objetos tiene contenedores en su Librería de Componentes Visuales (VCL Visual Component Library). Smalltalk tiene un grupo completo de contenedores. Java también tiene contenedores en su librería estándar. En algunas librerías, un contenedor genérico es considerado suficiente para todas las necesidades, y en otras (Java, por ejemplo) la librería tiene diferentes tipos de contenedores para diferentes necesidades: se tiene un vector (llamado un **ArrayList** en Java) para acceder continuamente a todos los elementos, y una lista enlazada permite insertar en todos los elementos, por ejemplo, así es que se puede elegir el tipo particular que entre en sus necesidades. Las librerías de contenedores pueden incluir grupos, colas, tablas hash, árboles, pilas, etc.

Todos los contenedores tienen alguna forma de poner y de sacar las cosas; usualmente hay funciones para agregar elementos a un contenedor, y otras para traer estos elementos afuera. Pero traer los elementos puede ser mas problemático, dado que una única función de selección puede ser restrictiva.

¿Que si se quiere manipular o comparar un grupo de elementos en un contenedor en lugar de solo uno?

La solución es un iterator, que es un objetos cuyo trabajo es seleccionar los elementos dentro de un contenedor y presentarlos a el usuario de ese iterator. Como una clase, proporciona un nivel de abstracción. Esta abstracción puede ser utilizada para separar los detalles del contenedor del código que accede a ese contenedor. El contenedor, mediante el iterator, es abstracto para ser simplemente una secuencia. El iterator permite atravesar esa secuencia sin preocuparse acerca de la estructura de las capas mas bajas -esto es, no importa si sea una **ArrayList**, una **LinkedList**, una **Stack** o cualquier otra. Esto da una flexibilidad de cambiar fácilmente las capas mas bajas de la estructura de datos sin molestar el código del programa. Java comienza (en la versión 1.0 y 1.1) con un iterator estándar, llamado **Enumeration**, para todas las clases contenedoras. Java 2 tiene además muchas librerías de contenedores mucho mas completas que contienen un iterator llamado **Iterator** que hace mucho mas que el viejo **Enumeration**.

Desde un punto de vista del diseño, todo lo que se quiere realmente es una secuencia que puede ser manipulada para solucionar su problema. Si un solo tipo de secuencia satisface todas las necesidades, no hay razón para tener diferentes tipos. Hay dos razones para tener la necesidad realizar una elección de contenedores. Primero, los contendores proporcionan distintos tipos de interfaces y de comportamientos externos. Una pila tiene una interfase y un comportamiento distintos a una cola, que es una forma diferente de grupo o lista. Una de estas proporciona una solución mas flexible a el problema que las otras. Segundo, contendores diferentes tienen diferentes eficiencias para ciertas operaciones. El mejor ejemplo es una **ArrayList** y una **LinkedList**. Ambos son simples secuencias que pueden tener interfaces y comportamientos externos idénticos. Pero ciertas operaciones pueden tener costos radicalmente diferentes costos. El acceso aleatorio a elementos en una **ArrayList** es una operación de tiempo constante; toma la misma cantidad de tiempo ver el elemento que se ha seleccionado. Sin embargo, en una **LinkedList** es mucho mas costoso moverse a través de la lista para seleccionar de forma aleatoria un elemento, y toma mucho mas tiempo encontrar un elemento que este mas adelante en la lista que un elemento que se encuentra en el medio de la secuencia. Por el otro lado, si se quiere insertar un elemento en el medio de una secuencia, es mucho mas barato en una **LinkedList** que en un **ArrayList**. Estas y otras operaciones tienen diferentes eficiencias dependiendo de la estructura de capas inferiores de la secuencia. En la fase de diseño, se debe comenzar con una **LinkedList** y cuando se ajusta el rendimiento, cambiar a una **ArrayList**. Dado que la abstracción mediante un iterator, se puede cambiar de uno al otro con un mínimo impacto en el código.

Al final , recuerde que un contenedor es solo un gabinete para almacenar objetos en el. Si ese gabinete soluciona todas sus necesidades, no importa como está implementado (un concepto básico con muchos tipos de objetos). Si se esta trabajando en un ambiente de programación que tiene costos operativos en el armado debido a otros factores, entonces la diferencia de costos entre una **ArrayList** y una **LinkedList** puede no importar. Se puede necesitar solo un tipo de secuencia. Se puede aún imaginar la abstracción del contenedor “perfecto”, que puede automáticamente cambiar su implementación de capas inferiores de acuerdo a la forma en que sea utilizado.

## La jerarquía de raíz única

Uno de los temas de la POO que especialmente sobresale dado la introducción de C++ es si todas las clases deben ser heredadas de una sola clase base. En Java (como en todos los otros lenguajes orientados a objetos) la respuesta es “si” y el nombre de esta última clase base es simplemente **Object**. Esto deja claro que los beneficios de la jerarquías de raíz única son muchas.

Todos los objetos en una jerarquía de raíz única tienen una interfase en común, así es que de última todas tienen el mismo tipo. La alternativa (proporcionada por C++) es que no se necesita saber que todo es del mismo tipo fundamental. Desde un punto de vista de compatibilidad regresiva esto encaja en el modelo de C mejor y puede reflexionarse como menos restrictivo, pero cuando se quiere realizar una programación totalmente orientada a objetos se debe crear su propia jerarquía para proporcionar la misma comodidad que se obtiene con otros lenguajes de POO. Y en cualquier nueva librería que se obtenga, alguna interfase incompatible será utilizada. Esto requiere esfuerzo (y probablemente herencia múltiple) para trabajar con la nueva interfase en su diseño. ¿Vale la pena la “flexibilidad” extra de C++? Si se necesita -si se tiene un gran inversión en C- es bastante valioso. Si se esta comenzando de la nada, otras alternativas como Java a menudo son mas productivas.

Todos los objetos en una jerarquía de raíz única (como la que proporciona Java) pueden garantizarse cierta funcionalidad. Si se conoces que se puede realizar ciertas operaciones básicas en cada objeto de su sistema. Una jerarquía de raíz única, junto con la creación de todos los objetos en el heap, realmente simplifica el pasaje de argumentos (uno de los temas mas complejos de C++).

Una jerarquía de raíz única hace mucho mas fácil implementar un recolector de basura (lo que es convenientemente creado en Java). El soporte necesario puede ser instalado en la clase base, y el recolector de basura puede de esta manera enviar los mensajes apropiados a cada objeto en el sistema. Si una

jerarquía de raíz única y un sistema para manipular un objeto mediante su referencia, es difícil implementar un recolector de basura.

Dado que la información de tipo en tiempo de ejecución está garantizada de estar en todos los objetos, nunca se terminará con un objeto de un tipo que no se pueda determinar. Esto es especialmente importante con operaciones a nivel de sistemas, como lo es el manejo de excepciones, y para permitir una mayor flexibilidad en la programación.

## Librerías de colecciones y soporte para su fácil utilización

Dado que un contenedor es una herramienta que se utilizará frecuentemente, tiene sentido tener una librería de contenedores creadas de una forma reutilizable, así es que se puede tomar una del estante y colocarla en el programa. Java proporciona esta librería, que suele satisfacer la mayoría de las necesidades.

### Conversión descendiente y plantillas genéricas

Para hacer estos contenedores reutilizables, almacenan un tipo universal en Java que ha sido previamente mencionado: **Object**. Esta jerarquía de raíz única significan que todo es un objeto, así es que un contenedor que almacena **Objects** puede almacenar cualquier cosa. Esto hace a los contenedores fáciles de reutilizar.

Para utilizar tal contenedor, simplemente se agrega referencias a objetos a él, y más tarde se pregunta por ellas nuevamente. Pero, dado que el contenedor almacena solo Objetos, cuando se agrega su referencia a objetos en el contenedor se le realiza una conversión ascendente a **Object**, de esta forma pierde su identidad. Cuando se trae de vuelta se obtiene una referencia a un **Object**, y no una referencia al tipo que se colocó. ¿Así es que como se vuelve a algo con una interfase útil para el objeto que se colocó en el contenedor?

Aquí es cuando las conversiones se utilizan nuevamente, pero esta vez no se está convirtiendo hacia arriba en la jerarquía de herencia a un tipo más general, se está realizando una conversión descendente en la jerarquía hacia un tipo más específico. Esta manera de convertir es llamada conversión descendente o downcasting. Con la conversión ascendente, se sabe, por ejemplo, que un **Círculo** es una **Curva** así es que es fácil realizar una conversión ascendente, pero no se sabe que un **Object** es necesariamente un **Círculo** o una **Curva** así es que es difícilmente seguro realizar una conversión descendente a no ser que se conozca exactamente con lo que se está tratando.

No es completamente peligroso, sin embargo, dado que si se realiza una conversión descendente a algo equivocado se obtendrá un error en tiempo de ejecución llamado una *excepción*, el que describirá brevemente. Cuando se obtienen referencias a objetos de un contenedor, sin embargo, se debe tener alguna forma de recordar exactamente que son antes de realizar una conversión descendente apropiada.

La conversión descendente y las pruebas en tiempo de ejecución requiere tiempo extra para ejecutar el programa, y un esfuerzo extra del programador. ¿No tendría sentido para alguien crear un contenedor de tal forma que conozca los tipos que almacena, eliminando la necesidad de realizar una conversión descendente y un posible error? La solución son los tipos parametrizados, que son clases que el compilador puede automáticamente adaptar para trabajar con tipos particulares. Por ejemplo, con un contenedor parametrizado, el compilador puede adaptar ese contenedor para que acepte solo Curvas y traiga solo Curvas.

Los tipos parametrizados son una importante parte de C++, en parte 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 dado que es posible obtenerlo mediante - torpemente sin embargo- utilizando la jerarquía de raíz única. Sin embargo, un propósito actual de los tipos parametrizados utilizan una sintaxis que es sorprendentemente similar a plantillas de C++.

## El dilema del cuidado de la casa: ¿Quien debería limpiar?

Cada objeto requiere recursos para existir, el mas notable es la memoria. Cuando un objeto no se necesita mas debe ser limpiado para que esos recursos sean liberados para su reutilización. Una situación simple en programación es la pregunta de como es limpiado un objeto sin que sea un desafío: se crea un objeto, se utiliza todo lo necesario, y entonces debe ser destruido. Esto no es difícil, sin embargo, se encuentran situaciones en las cuales la situación es mas compleja.

Supongamos, por ejemplo, que se está diseñando un sistema para manejar el tráfico aéreo para un aeropuerto (El mismo modelo puede trabajar también para manejar cajas de mercadería en un almacén, o un sistema de control de rentas o una casa para alojar animales). Al comienzo parece simple: se crea un contenedor para contener aeroplanos, luego se crea un nuevo aeroplano y se coloca en el contenedor para cada aeroplano que ingresa en la zona de control de tráfico aéreo. Para limpiarlo, simplemente se borra el aeroplano adecuado cuando un aeroplano deja la zona.

Pero tal vez se tenga algún otro sistema de almacenar datos acerca de los aeroplano; tal vez lo datos no necesitan atención inmediato como el controlador de la función principal. Tal vez es un registro de los planes de vuelo de todos los pequeños aeroplanos que dejan el aeropuerto. Así es que se tiene un segundo contenedor de pequeños aeroplanos, cuando se crea un objeto aeroplano se coloca en este segundo contenedor se es un aeroplano pequeño. Luego un proceso de fondo realiza operaciones en los objetos en este contenedor durante los momentos de inactividad.

Ahora el problema es más difícil: ¿Como es posible saber cuando destruir los objetos? Cuando se termina con el objeto, alguna otra parte del sistema podría no estar. Este mismo problema se origina en un gran número de otras situaciones, y en sistemas de programación (como es C++) en donde se debe explícitamente borrar un objeto cuando se ha terminado con él puede resultar bastante complejo.

Con Java, el recolector de basura esta diseñado para encargarse del problema de liberar memoria (a pesar de que no incluye otros aspectos de la limpieza de un objeto). El recolector de basura “sabe” cuando un objeto no se encuentra mas en uso, y este libera automáticamente la memoria de ese objeto. Esto (combinado con el hecho de que todos los objetos son heredados de una única clase raíz **Object** y que se crean objetos solamente de una forma, en el heap) hace que el proceso de programación en Java mucho mas simple que la programación en C++. Se tiene pocas decisiones para tomar y cientos para vencer.

## Recolectores de basura versus eficiencia y flexibilidad

¿Si todo esto es una buena idea, por que no si hizo la misma cosa en C++? Ciertamente hay un precio a pagar por toda esta conveniencia en la programación, y este precio es son en costos operativos en tiempo de ejecución. Como se ha mencionado antes, en C++ se pueden crear objetos en la pila, y es este caso son automáticamente limpiados (pero no tiene la flexibilidad de crear tantos como se quiera en tiempo de ejecución). Crear objetos en la pila es la forma mas eficiente de asignar espacio para los objetos y para liberar ese espacio. Crear objetos en el heap puede ser mucho mas costoso. siempre heredando de la clase base y creando todas las llamadas a funciones polimórficas también tiene un pequeño costo. Pero el recolector de basura es un problema particular porque nunca se sabe realmente cuando va a iniciarse o cuanto tiempo tardará en realizar la operación. Esto significa que hay una inconsistencia en la taza de ejecución de un programa en Java, así es que no se puede utilizar en ciertas situaciones, como cuando la taza de ejecución de un programa es uniformemente crítica (Estos son llamados generalmente programas en

tiempo real, a pesar de que no todos los problemas de programación en tiempo real son tan rigurosos).

Los diseñadores del lenguaje C++, trataron de cortejar a los programadores C (y con mucho éxito, en ese punto), no quisieron agregar características a el lenguaje que impactaran en la velocidad o en el uso de C++ en cualquier situación donde de otra forma los programadores puedan elegir C. Esta meta fue alcanzada, pero al precio de una gran complejidad cuando se programa en C++. Java es mas simple que C++, pero a cambio de eficiencia y a veces de aplicabilidad. Para una significante porción de problemas de programación sin embargo, Java es la elección superior.

## Manejo de excepciones: tratando con errores.

Desde el comienzo de los lenguajes de programación, el manejo de errores ha sido uno de los temas mas complicados. Dado que es muy duro diseñar un buen esquema de manejo de errores, muchos lenguajes simplemente ignoran este tema, pasándole el problema a los diseñadores de la librería que toman medidas a medias que funcionan en muchas situaciones pero que pueden ser fácilmente evadidas, generalmente ignorándolas. Un problema mayor con la mayoría de los esquemas de manejo de errores es que confían en que el programador este alerta a seguir una convención acordada que no es forzada por el lenguaje. Si el programador no es despierto -a menudo es el caso si están apurados- estos esquemas pueden ser fácilmente olvidados.

El manejo de excepciones inserta el manejo de excepciones en el lenguaje de programación y a veces incluso en el sistema operativo. Una excepción es un objeto que es “lanzado” desde el sitio del error y puede ser capturado por un manejador de excepciones adecuado diseñado para manejar ese tipo particular de error. Esto es como si el manejador de excepciones fuera un camino diferente, paralelo de ejecución que puede ser tomado cuando las cosas van mal. Y dado que se usa un camino de ejecución separado, no necesita interferir con la ejecución normal de su código. Esto hace la escritura de código simple dado que no esta constantemente forzado a verificar errores. Además, una excepción que es lanzado no es como un valor de error que es retornado de una función o una bandera que es configurada por una función para indicar una condición de error -estas pueden ser ignoradas. Una excepción no puede ser ignorada, así es que es garantido tratarla en algún punto. Finalmente, las excepciones proporcionan una forma de recuperarse confiablemente de una mala situación. En lugar se simplemente salir se es a menudo capaz de hacer las cosas correctas y de

restaurar la ejecución de un programa, lo que produce programas mucho mas robustos.

El manejo de excepciones en Java sobresale de entre medio de los lenguajes de programación, porque en Java, el manejo de excepciones fue introducido desde el comienzo y se esta forzado a utilizarlo. Si no escribe el código para que maneje excepciones correctamente, se obtendrá un mensaje de error de compilación. Esta coherencia garantiza que en el manejo de errores sea mucho mas fácil.

Es de valor notar que el manejo de errores no es una característica de los lenguajes orientados a objetos, a pesar de que en los lenguajes orientados a objetos la excepción es normalmente representada con un objeto. El manejo de excepciones existe antes que los lenguajes orientados a objetos.

## Hilado múltiple

Un concepto fundamental en la programación de computadoras es la idea de manejar mas de una tarea a la vez. Muchos problemas de programación requieren que el programa sea capaz de parar lo que esta haciendo, y tratar con otro problema, y luego retornar al proceso principal. La solución ha sido acometida de distintas maneras. Inicialmente, los programadores con conocimientos de bajo nivel de la máquina escribieron rutinas de servicio de interrupciones y la suspensión del proceso principal fue inicialmente a través de interrupciones hardware. A pesar de que esto trabajaba bien, fue difícil y no era portátil, así es que mover un programa a un nuevo tipo de máquina era lento y caro.

A veces las interrupciones eran necesarias para el manejo de tareas críticas, pero hay una clase grande de problemas en los cuales se trata simplemente de partir en piezas separadas así es que la totalidad del programa puede responder de mejor manera. Dentro de un programa, estas piezas que se ejecutan separadamente son llamadas hilos, y el concepto general es llamado *hilado múltiple*. Un ejemplo común es la interfase de usuario. Utilizando hilos, un usuario puede presionar un botón y obtener una respuesta rápida en lugar de ser forzado a esperar hasta que el programa termine la tarea actual.

Normalmente, los hilos son solo una forma de hacer un uso del tiempo de un procesador solo. Pero si el sistema operativo soporta múltiples procesadores, cada hilo puede ser asignado a un procesador diferentes y ellos pueden verdaderamente correr en paralelo. Una de las características mas convenientes del hilado múltiple a nivel del lenguaje es que el programador no necesita preocuparse acerca de si hay muchos procesadores o solo uno. El programa es lógicamente dividido en hilos y si la máquina

tiene mas de un procesador entonces el programa se ejecuta mas rápidamente, si ajustes especiales.

Todo esto hace que los hilos parezcan bastante simples. He aquí un truco: compartir recursos. Si se tiene mas de un hilo ejecutándose que espera acceder a el mismo recurso se tiene un problema. Por ejemplo, dos procesos pueden simultáneamente enviar información a la impresora. Para solucionar el problema, los recursos pueden ser compartidos, como la impresora, debe ser bloqueado hasta que sea utilizado. Así es que un hilo bloquea un recurso, completa su tarea, y luego libera el bloqueo así es que alguien mas puede utilizar el recurso.

El hilado en Java esta realizado dentro del lenguaje, lo que hace un tema complicado mucho mas simple. El hilado es soportado a nivel de objeto, así es que un hilo de ejecución es representado por un objetos. Java también proporciona bloqueo de recursos limitado. Se puede bloquear la memoria de un objeto (lo que es, después de todo, un recurso compartido) así es que solo un hilo puede utilizarla a la vez. Esto se logra 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 los hilos deben verificar antes de acceder a ese recurso.

## Persistencia

Cuando se crea un objeto, este existe hasta que se lo necesite, pero bajo ninguna circunstancia este existe cuando el programa termina. Esto tiene sentido al principio, hay situaciones en donde puede ser increíblemente útil si un objeto puede existir y mantener su información aún cuando el programa no esta ejecutándose. Entonces la siguiente vez que se ejecuta el programa, el objeto estará ahí y tendrá la misma información que en un momento antes que el programa se ejecutara. Claro se puede obtener un efecto similar escribiendo la información en un fichero o en una base de datos, pero en el espíritu de hacer todo un objetos puede ser bastante conveniente ser capaz de declarar un objeto persistente y todos los detalles cubiertos.

Java proporciona soporte para “persistencia ligera”, que significa que fácilmente se pueden almacenar objetos en el disco y luego recuperarlos. La razón de ser persistencia ligera es que se sigue estando forzado a realizar llamadas explícitas para realizar el almacenamiento y la recuperación. Además, JavaSpaces (descrito en el Capítulo 15) proporciona un tipo de almacenamiento persistente de objetos. En algunas futuras versiones puede aparecer un soporte mas completo para persistencia.

# Java y la Internet

Si Java es, de hecho, solo otra lenguaje de programación de computadora, se debe hacer la pregunta de por que es tan importante y por que es promovido como un paso revolucionario en la programación de computadoras. La respuesta no es inmediatamente obvia si se procede de la perspectiva de la programación tradicional. A pesar de que Java es muy útil para solucionar problemas de programación que trabajan solos, es importante porque soluciona problemas en la Web.

## ¿Que es la Web?

La Web puede parecer un poco como un misterio al principio, con toda esa charla de “navegar”, “presencia” y “paginas de inicio”. Hay incluso crecido un reacción contra la “manía de la Internet”, se cuestionan el valor y el resultado de un movimiento arrollador. Es muy útil dar un paso atrás y ver que es realmente, para hacer esto se debe entender los sistemas cliente/servidor, otro aspecto de la computación que esta lleno de temas confusos.

## Computación cliente/servidor

La idea inicial de un sistema cliente/servidor es que se tiene un depósito central de información -algún tipo de dato, a menudo en una base de datos- que se quiere distribuir en demanda de un grupo de personas o máquinas. Una clave para el concepto cliente/servidor es que el depósito de información está ubicado centralmente así es que se puede cambiar así es que esos cambios se propagarán fuera para informar a los consumidores. Tomándolo junto, el almacén de información, el software que distribuye la información y las máquinas donde la información y el software residen es llamado el servidor. El software que reside en la máquina remota, se comunica con el servidor y toma la información, la procesa, y luego la muestra en la máquina remota es llamada el *cliente*.

El concepto básico de computación cliente/servidor, entonces, no es muy complicado. Los problemas se levantan porque no se tiene un solo servidor tratando de servir muchos clientes de una vez. Generalmente, un manejo de sistemas de bases de datos esta involucrado así es que el diseño “balancea” la capa de datos en las tablas para un uso óptimo. Además, los sistemas a menudo permiten a un cliente insertar información nueva en un servidor. Esto significa que se debe asegurar que un dato nuevo de un cliente no pasa por arriba un dato de otro cliente, o que ese dato no se pierde en el proceso de agregarlo en la base de datos (Esto es llamado proceso de transacción). Cuando el software del cliente cambia, debe ser armado, depurado, e

instalado en la máquina del cliente, lo que puede tornarse mas complicado y caro de lo que se puede pensar. Es especialmente problemático soportar múltiples tipos de computadoras y sistemas operativos. Finalmente, hay un tema importante de rendimiento: se debe tener cientos de clientes realizando peticiones a su servidor a la vez, y cualquier atraso puede ser crucial. Para minimizar la latencia, los programadores trabajan duro para descargar tareas de procesamiento, a menudo en la máquina del cliente, pero a veces en otras máquinas en el sitio del servidor, utilizando el llamado *middleware* (*Middleware* es utilizado también para mejorar el mantenimiento).

La simple idea de distribuir información a personas tiene tantas capas de complejidad en implementación que la totalidad del problema puede ser desesperadamente enigmático. Y aún es crucial: las cuentas de computación cliente/servidor es aproximadamente la mitad de las actividades de programación. Es responsable de todo desde la toma de órdenes y transacciones de tarjetas de crédito a la distribución de algún tipo de datos - stock de un mercado, ciencia, gobierno, su nombre. Lo que hemos hecho en el pasado es armar soluciones individuales para problemas individuales, inventando una nueva solución cada vez. Este es duro de crear y difícil de utilizar, y el usuario tiene que aprender una nueva interfase para cada uno. El problema cliente/servidor entero necesita solucionarse de una gran forma.

## La Web como un servidor gigante

La Web es actualmente un sistema gigante cliente/servidor. Esto es un poco peor que eso, dado que tiene todos los servidores y clientes coexistiendo en una sola red de una sola vez. No necesita saber que, dado que todo lo que le importa acerca de la conexión y la interacción con un servidor a la vez (a no ser que esté saltando alrededor del mundo buscando el servidor correcto).

Inicialmente fue un proceso simple de una vía. Se realiza una petición a un servidor, este pasa un fichero, que el navegador en la máquina (i.e., el cliente) interpretará y le dará formato en la máquina local. Pero en poco tiempo las personas comienzan a esperar mas que solo distribuir páginas desde un servidor. Ellos esperan capacidades totales de cliente/servidor así es que el cliente puede alimentar el servidor, por ejemplo, para hacer que una base de datos busque en el servidor, para agregar mas información al servidor, o para colocar una orden (lo que requiere mucha mas seguridad que la que los sistemas iniciales ofrecen). Estos son los cambios que estamos viendo en los desarrollos en la Web.

El navegador Web fue un gran paso adelante: el concepto de que un pedazo de información pueda ser desplegado en cualquier tipo de computadora sin cambios. Sin embargo, los navegadores a pesar de eso son primitivos y

rápidamente se retrasaron por las demandas efectuadas. Ellos no eran particularmente interactivos, y tendían a atascar el servidor y la Internet a causa de que en cualquier momento se necesitaba hacer algo que requería programación se tenía que enviar información al servidor para ser procesada. Esto podía tomar varios segundos o minutos para encontrar algo mal escrito en su petición. Dado que el navegador era solo un visor no podía realizar inclusive las tareas mas simples de computación (Por el otro lado, esto era seguro, dado que no se podía ejecutar ningún programa en la computadora local que pudiera contener errores o virus).

Para solucionar este problema, diferentes estrategias fueron tomadas. Para comenzar con algo, los estándares gráficos fueron mejorados para permitir mejores animaciones y video dentro de los navegadores. El resto de los problemas pueden ser resueltos solo incorporando la habilidad de ejecutar programas en el cliente, bajo el navegador. Esto es llamado programación del lado del cliente.

## Programación del lado del cliente

El diseño servidor navegador inicial de la Web proporcionaba contenido interactivo, pero la interactividad era completada por el servidor. El servidor producía páginas estáticas para el navegador cliente, que simplemente las interpretaba y desplegaba. El HTML básico contiene mecanismos simples para reunir datos: cajas para la entrada de texto, casillas de verificación, botones de radio, listas y listas desplegables, así como botones que solo pueden ser programados para reiniciar los datos en los formularios o para “someter” los datos del formulario en el servidor. Este sometimiento pasa a través de la Common Gateway Interfase (CGI) proporcionada por todos los servidores. El texto pasado le dice a la CGI que hacer con el. La acción mas común es ejecutar un programa ubicado en el servidor en un directorio comúnmente llamado “cgi-bin” (si se mira la dirección en la barra de título de la ventana de su navegador cuando se presiona el botón en la página Web, se puede a veces ver “cgi-bin” con todas las galimatías ahí). Estos programas pueden ser escritos en muchos lenguajes. Perl es una elección común porque esta diseñado para manipular texto y es interpretado, así es que puede ser instalado en cualquier servidor sin pensar en el procesador o en el sistema operativo.

Muchos sitios Web poderosos hoy están armados estrictamente en CGI, y se puede de hecho hacer casi todo. Sin embargo, los sitios Web creados con programas CGI pueden rápidamente volverse demasiado complicados de mantener, y también esta el problema del tiempo de respuesta. La respuesta de un programa CGI depende de cuantos datos deba enviar, de la misma forma que del tiempo de carga en el servidor y de la Internet (Por encima de esto, iniciar un programa CGI tiende a ser lento). Los diseñadores iniciales

de la Web no visualizaron cuan rápido este ancho de banda puede ser gastado por el tipo de aplicaciones que las personas desarrollan. Por ejemplo, cualquier tipo de gráfico dinámico es prácticamente imposible de realizar con consistencia porque un fichero GIF debe ser creado y movido del servidor al cliente para cada versión del gráfico. Y no se tiene duda de que se ha tenido experiencia directa con algo tan simple como una validación de datos en un formulario de entrada. Se presiona el botón para enviarlo en una página; los datos son enviados de vuelta al servidor; el servidor comienza un programa CGI que descubre un error, arma una página HTML, la página le informa de este error y luego envía la página otra vez hacia usted; se debe entonces guardar la página e intentarlo nuevamente. Esto no es solo lento, es poco elegante.

La solución es la programación del lado del cliente. Muchas máquinas que ejecutan navegadores Web tienen poderosos motores capaces de hacer enormes trabajos, y con la estrategia del HTML original están sentadas ahí, simplemente esperando que el servidor entregue la siguiente página. Programación del lado del cliente significa que el navegador Web es aparejado a hacer cualquier trabajo que pueda, y el resultado para el usuario es mucha más velocidad y una experiencia más interactiva en su sitio Web.

El problema que se discute acerca de la programación del lado del cliente es que no son muy diferentes de las discusiones de la programación en general. Los parámetros son al menos los mismos, pero la plataforma es diferente: un navegador Web es como un sistema operativo limitado. Al final, se debe seguir programando, y estas historias de grupos desequilibrantes de problemas y soluciones producida por la programación del lado del cliente. El resto de esta sección se proporciona una vista general de los temas y estrategias en la programación del lado del cliente.

## Plug-ins

Uno de los pasos adelante más significantes en la programación es el desarrollo de plug-in. Esta es una forma de que un programador agregue nueva funcionalidad a un navegador bajando una pieza de código que coloca solo en una ubicación apropiada en el navegador. Este le indica al navegador "desde ahora puede realizar esta nueva actividad" (Se necesita bajar el plug-in solo una vez). Algunos comportamientos rápidos y poderosos son agregados a los navegadores mediante plug-ins, pero escribir un plug-in no es una tarea trivial, y no es algo que se quiera hacer como parte del proceso de crear un sitio en particular. El valor de los plug-in para la programación del lado del cliente es que permite a un experto programador desarrollar un nuevo lenguaje y agregar ese lenguaje a un navegador sin el permiso del fabricante del navegador. De esta manera, los plug-ins proporciona una "puerta trasera" que permite la creación de nuevos lenguajes de

programación del lado del cliente (a pesar de que no todos los lenguajes son implementados como plug-ins).

## Lenguajes de guiones

Los plug-ins resultaron en una explosión de los lenguajes de guiones. Con lenguajes de guiones se empotra el código fuente del programa del lado del servidor directamente en la página HTML, y el plug-ins que interpreta el lenguaje es automáticamente activado cuando la página HTML se despliega. Los lenguajes de guiones tienden a ser razonablemente fáciles de entender y, dado que son simplemente texto que es parte de la página HTML, se cargan muy rápido como parte del simple envío del servidor requerido para procurar esa página. El problema es que se debe renunciar a que el código sea expuesto a que cualquiera lo vea (y hurte). Generalmente, sin embargo, no se está haciendo alguna cosa asombrosamente sofisticada con lenguajes de guiones así es que esto no es mucho problema.

Esto apunta a que los lenguajes de guiones utilizados dentro de los navegadores Web son realmente un intento de solucionar un tipo específico de problema, inicialmente en la creación de interfaces de usuario (GUI) más ricas e interactivas. Sin embargo, un lenguaje de guiones puede solucionar el 80 por ciento de los problemas encontrados en la programación del lado del cliente. Los problemas pueden encajar muy bien en ese 80 por ciento, y dado que los lenguajes de guiones pueden permitir un desarrollo fácil y rápido, debería probablemente considerar un lenguaje de guiones antes de verse con una solución más compleja como lo es la programación Java o ActiveX.

El lenguaje de guiones más abordado por los navegadores es JavaScript (que no tiene nada que ver con Java; fue nombrado de esta forma solo para arrebatar algo del momento de marketing que tuvo Java), VBScript (que se ve como Visual Basic), y Tcl/Tk, que se origina en la popular lenguaje de plataforma cruzada GUI-building. Hay otros ahí afuera, y no hay duda que hay más en desarrollo.

JavaScript es probablemente el más comúnmente soportado. Viene incluido con el Netscape Navigator y con el Microsoft Internet Explorer (IE). Además, hay probablemente más libros de JavaScript disponibles que para los demás lenguajes de navegadores, y algunas de las herramientas automáticamente crean páginas utilizando JavaScript. Sin embargo, si se es elocuente con Visual Basic o Tcl/Tk, probablemente se será más productivo utilizando estos lenguajes de guiones que aprender uno nuevo (ya podría tener las manos tratando enteramente con los temas de la Web).

## Java

¿Si un lenguaje de guiones puede resolver un 80 por ciento de los problemas de programación del lado del cliente, que sucede con el otro 20 por ciento - las "cosas verdaderamente duras"? La solución mas popular hoy en día es Java. No solo es un poderoso lenguaje de programación creado para ser seguro, multiplataforma e internacional, Java también es continuamente extendido para proporcionar características y librerías que elegantemente maneja problemas que son difíciles en lenguajes de programación, como lo es la multitarea, acceso a bases de datos, programación en redes y computación distribuida. Java permite programación del lado del cliente mediante el *applet*.

Un applet es un pequeño programa que se ejecuta solo en un navegador Web. El applet es bajado automáticamente como parte de la página Web (exactamente como, por ejemplo, un gráfico es automáticamente bajado). Cuando el applet es activado se ejecuta un programa. Esto es parte de su belleza -proporciona una forma de distribuir automáticamente el software cliente del servidor en el momento que el usuario necesita el software, y no antes. El usuario obtiene la última versión del software cliente sin fallar y sin dificultades de reinstalación. Dado por la forma en que Java es diseñado, el programador necesita crear un solo programa, y el programa automáticamente trabaja con todas las computadoras que tienen intérpretes Java insertados (Esta seguramente incluye a mayoría de todas las máquinas). Dado que Java es un lenguaje de programación totalmente maduro, se puede hacer todo el trabajo que el cliente permita antes y después de realizar peticiones al servidor. Por ejemplo, no necesita enviar un formulario de pedido a través de la Internet para descubrir que ha puesto la fecha o algún otro parámetro mal, y la computadora cliente puede rápidamente realizar el trabajo de dibujar los datos en lugar de tener que esperar al servidor para que dibuje los datos y los envíe de vuelta en forma de una imagen gráfica. No solo se obtiene una ganancia inmediata en velocidad y respuesta, en tráfico general de la red y la carga de los servidores puede ser reducida, evitando que la Internet entera se ponga lenta.

Otra ventaja de los applets de Java tienen sobre los lenguajes de script es que está compilado, así es que el código fuente no esta disponible para el cliente. Por el otro lado, un applet Java puede ser descompilado sin mucho trabajo, pero ocultar su código no es un tema muy importante. Otros dos factores pueden ser importantes. Como se verá mas adelante en este libro, un applet Java compilado puede comprender muchos módulos y tomar muchos peticiones del servidor (entradas) a bajar (En Java 1.1 y superiores esto es minimizado por los archivos, llamados ficheros JAR, que permiten que todos los módulos requeridos sean empaquetados juntos y comprimidos para bajara todo de una sola vez). Un programa de guiones solo puede ser integrado en la página Web como parte de su texto (y generalmente es mas

pequeño y reduce los pedidos al servidor). Esto puede ser importante para la reacción de su sitio Web. Otro factor es la curva de aprendizaje.

Independientemente de lo que se ha dicho, Java no es un lenguaje trivial de aprender. Si se es un programador de Visual Basic, moverse a VBScript puede ser la solución mas rápida, y dado que se solucionan los problemas cliente/servidor mas típicos estará duramente presionado a justificar aprender Java. Si tiene experiencia con un lenguaje de guiones estará naturalmente beneficiado al ver JavaScript o VBScript antes de comprometerse con Java, dado que ellos pueden encajar en sus necesidades diestramente y será mas productivo mas adelante.

## ActiveX

En algún grado, el competido de Java es ActiveX de Microsoft, a pesar de que toman una estrategia totalmente distinta. ActiveX fue originalmente una solución solo para Windows, a pesar de que ahora esta siendo desarrollado por un consorcio independiente para convertirlo en multiplataforma. Efectivamente, ActiveX dice “si su programa se conecta a su ambiente exactamente de esta forma, se puede dejar caer en una pagina Web y ejecutar bajo un navegador que soporte ActiveX” (IE directamente soporta ActiveX y Netscape lo hace utilizando un plug-in). De esta forma, ActiveX no se limita a un lenguaje en particular. si, por ejemplo, ya se ha tenido una experiencia programando bajo Windows utilizando un lenguaje como C++, Visual Basic, o Delphi de Borland, se puede crear componentes ActiveX sin cargo para su conocimiento en programación. ActiveX también proporciona un camino para el uso de código legado en las paginas Web.

## Seguridad

Bajar y ejecutar automáticamente programas a través de la Internet puede sonar como un sueño para los creadores de virus. ActiveX especialmente trae el espinoso tema de la seguridad de la programación del lado del cliente. Si se accede a un sitio Web, se debe automáticamente bajar cualquier cantidad de cosas junto con la página HTML: ficheros GIF, código de guiones, código Java compilado, y componentes ActiveX. Algunos de ellos son benignos: los ficheros GIF no pueden hacer ningún daño, y los lenguajes de guiones están limitados en lo que pueden hacer. Java también fue diseñado para ejecutar en applets dentro de una jaula de seguridad, que previene de escribir el disco o de acceder a la memoria fuera de la jaula.

ActiveX es el extremo opuesto del espectro. Programar con ActiveX es como programar Windows -se puede hacer todo lo que se quiera. Así es que si se hace un clic en una página que baja un componente ActiveX, ese componente puede causar daño en los ficheros de su disco. Claro, los programas que se cargan en la computadoras no están restringidos a

ejecutarse dentro de un navegador Web pueden hacer lo mismo. Los virus bajados de sistemas de mensajería electrónica (Bulletin Board Systems BBSs) fueron un gran problema, pero la velocidad de la Internet amplifica esta dificultad.

La solución parece ser las “firmas digitales”, por lo cual el código es verificado para mostrar quién es el autor. Esto está basado en la idea de que los virus trabajan porque el creador puede ser anónimo, así es que si se le quita el anonimato individual se es forzado a ser responsable por sus acciones. Esto suena como un buen plan porque esto permite a los programadores ser mucho más funcional, y sospecho que esto eliminará las malas conductas. Pero si, sin embargo, un programa tiene un error destructivo no intencional este causará problemas.

La estrategia de Java es prevenir estos problemas de que ocurran, mediante la jaula de seguridad. El intérprete Java que se encuentra en su navegador Web local examina el applet en busca de instrucciones adversas cuando el applet es cargado. En particular, el applet no puede escribir ficheros en el disco o borrar ficheros (uno de los motivos principales de los virus). Los applets son generalmente considerados seguros, y dado que esto es esencial para sistemas cliente/servidor confiables, y los errores en el lenguaje Java que permiten virus son rápidamente reparados (es importante notar que el software de los navegadores actualmente implementa restricciones de seguridad, y algunos navegadores permiten seleccionar diferentes niveles de seguridad para proporcionar varios niveles de acceso a su sistema).

Se puede ser escéptico de esta restricción draconiana contra escribir ficheros en su disco local. Por ejemplo, se puede querer crear una base de datos local o guardar datos para su uso posterior fuera de línea. La visión inicial parece ser que eventualmente cualquiera puede obtener fuera de línea hacer cualquier cosa importante, pero esto, fue rápidamente impráctico (a pesar que los “dispositivos de Internet” de bajo costo algún día satisfacerán las necesidades de un segmento significante de usuarios). La solución es el “applet firmado” que utiliza encriptación de clave pública para verificar que un applet viene ciertamente de donde dice que viene. Un applet firmado puede seguir podar su disco, pero la teoría es que se puede ahora apoyarse en la responsabilidad del creador que no hace cosas viciosas. Java proporciona un marco de trabajo para firmas digitales así es que eventualmente puede permitir que un applet salga fuera de la jaula de seguridad si es necesario.

Las firmas digitales se olvidan de un tema importante, que es la velocidad con que las personas se mueven a través de la Internet. Si se baja un programa con errores y hace algo adverso. ¿Cuánto tiempo pasará para que se descubra el daño? Pueden ser días o tal vez semanas. ¿Para entonces, como se rastrearán el programa que lo hizo? ¿Y qué cosa buena hará para solucionar el daño en este punto?

## Internet versus intranet

La Web es la solución mas general para el problema de cliente/servidor, así es que tiene sentido que se pueda utilizar la misma tecnología para solucionar un grupo de problemas, en particular los problemas clásicos cliente/servidor dentro de una empresa. Con la estrategia tradicional cliente/servidor se tiene el problema de múltiples tipos de computadoras cliente, como la dificultad de la instalación de un nuevo software cliente, ambos cosas que pueden ser diestramente solucionadas con navegadores Web y programación del lado del cliente. Cuando la tecnología Web es utilizada para una red de información está restringida a una empresa en particular, esto es referido como una intranet. Las intranets proporcionan mucha mas seguridad que la Internet, dado que se puede controlar físicamente el acceso a los servidores dentro de su empresa. En términos de entrenamiento, parece ser que una vez que las personas entienden el concepto general de un navegador es mucho mas fácil para ellos tratar con las diferencias en que se ven las páginas Web y los applets, así es que la curva de aprendizaje para los nuevos sistemas parecen ser reducidas.

Los problemas de seguridad nos traen a una de las divisiones que parecen formarse automáticamente en el mundo de la programación del lado del cliente. Si su programa se esta ejecutando en la Internet, no se sabrá bajo que plataforma trabajará, y se quiere ser cuidadoso de mas y no diseminar código con errores. Se necesita algo multiplataforma y seguro, como un lenguaje de guiones o Java.

Si se esta ejecutando en una intranet, debe tener un grupo diferente de limitaciones. No es común que las máquinas sean de plataforma Intel bajo Windows. En una intranet, se es responsable por la calidad de su propio código y se pueden reparar errores cuando son descubiertos. Además, se puede tener un código legado que se estará utilizando con una estrategia cliente/servidor mas tradicional, por lo cual se deben instalar programas cliente cada vez que se realiza una actualización. El tiempo gastado en instalar actualizaciones es la razón mas convincente para migrar a los navegadores, porque las actualizaciones son invisibles y automáticas. Si se esta involucrado en algo así como una intranet, la estrategia mas sensata a tomar es el camino mas corto que permita utilizar su código base existente, en lugar de tratar de escribir todo el código nuevamente en un nuevo lenguaje.

Cuando se encara con esta desconcertante colección de soluciones para el problema de programación del lado del cliente, el mejor plan de ataque es el análisis de los costos y los beneficios. Considere las limitaciones de su problema y cual sería el camino mas corto para su solución. Dado que la programación del lado del cliente sigue siendo programación, es siempre una buena idea tomar la estrategia mas rápida para su situación en

particular. Esto es una postura agresiva para prepararse para los inevitables encuentros con los problemas del desarrollo de programas.

## Programación del lado del servidor

La totalidad del debate ha ignorado el tema de la programación del lado del servidor. ¿Que sucede cuando se hace una petición a un servidor? La mayoría del tiempo es simple “envíeme este fichero”. El navegador interpreta el fichero en una forma apropiada: como una página HTML, una imagen gráfica, un applet de Java, un programa de guiones, etc. Una petición mas complicada a un servidor generalmente involucra una petición para una transacción en una base de datos. Un escenario común involucra una petición para una búsqueda compleja en una base de datos, a la cual el servidor debe luego darle un formato un una página HTML y enviar el resultado (Claro que, si el cliente es mas inteligente mediante Java o un lenguaje de guiones, el crudo de los datos pueden ser enviados y puede dársele un formato en el cliente, que será mas rápido y tardará menos en cargar en el servidor). O se puede querer registrar el nombre de alguien en la base de datos cuando acceda a un grupo o coloque una orden, lo que involucra cambios en la base de datos. Las peticiones a la base de datos deben ser procesadas mediante algún código en el lado del servidor, que es generalmente referido como programación del lado del servidor.

Tradicionalmente, la programación del lado del servidor ha sido realizada utilizando Perl y guiones CDU, pero sistemas mas sofisticados han aparecido. Estos incluyen servidores Web basados en Java escribiendo lo que se denomina servlets. Servlets y su familia JSPs, son dos de las razones mas convincentes para las empresas que desarrollan sitios Web para moverse a Java, especialmente porque elimina los problemas de tratar con diferentes capacidades de los navegadores.

## Una arena separada: aplicaciones

Mucho del lo que se ha hablado sobre Java ha sido sobre los applets. Java es actualmente un lenguaje de programación de propósito general que puede solucionar cualquier tipo de problema -al menos en teoría. Y como se ha dicho anteriormente, puede ser la forma mas efectiva de solucionar los problemas cliente/servidor. Cuando nos movemos fuera del arena del applet (y simultáneamente liberamos las restricciones, como la en contra de la escritura del disco) se entra en un mundo de aplicaciones de propósito general que se ejecutan solas, sin el navegador, exactamente como un programa común hace. Aquí, la fortaleza de Java no solo es solo en su portabilidad, también en su facilidad para programarse. Como se verá a través de este libro, Java tiene muchas características que permiten crear

programas robustos en un período mas corto que los lenguajes de programación anteriores.

Hay que ser conciente de que esto es una ventaja a medias. Se paga por las mejoras mediante una disminución en la velocidad de ejecución (a pesar de que se está haciendo un trabajo significante en ese área -JDK 1.3, en particular, introduce lo que han llamado mejoras “hotspot” en rendimiento). Como cualquier lenguaje, Java tiene limitaciones que pueden hacerlo inapropiado para solucionar ciertos tipos de problemas de programación. Java es un lenguaje que evoluciona rápidamente, sin embargo, y cada vez que una nueva versión sale es mas y mas atractiva para solucionar un gran grupo de problemas.

## Análisis y diseño

El paradigma de la orientación a objetos es una forma nueva y diferente de pensar acerca de programar. Muchos amigos tienen problemas al comienzo para saber como enfocar un proyecto orientado a objetos. Una vez que se sabe que todo se supone es un objeto, y tanto como se aprenda a pensar mas en un estilo orientado a objetos se puede comenzar a crear “buenos” diseños que tomen ventaja de todos los beneficios que la POO puede ofrecer.

Un *método* (a menudo llamado *metodología*) es un grupo de procesos y heurística utilizados para disminuir la complejidad de un problema de programación. Muchos métodos de programación orientada a objetos han sido formulados desde el despertar de la programación orientada a objetos. Esta sección dará un sentimiento de que se estará tratando de lograr cuando utilice un método.

Especialmente en POO, la metodología es un campo de muchos experimentes, así es que es importante entender que problema el método esta tratando de resolver antes de adoptar uno. Esto es en particular verdadero con Java, en donde el lenguaje de programación pretende reducir la complejidad (comparado con C) involucrada en expresar un programa. Esto puede de echo aliviar la necesidad de metodologías mas complejas. En lugar de eso, metodologías simples pueden ser satisfactorias en Java para una clase mucho mas grande de problemas de las que se pueden manejar utilizando metodologías simples que con lenguajes procesales.

Es importante de darse cuenta que el término “metodología” es a menudo muy grande y promete mucho. Lo que sea que se haga ahora con un diseño y escritura de un programa es un método. Este puede ser un método propio, y se puede no ser conciente de hacerlo, pero es un proceso por el cual se camina mientras se crea. Si es un proceso efectivo, solo se necesitará un pequeño ajuste para trabajar en Java. Si no se esta satisfecho con su productividad y la forma en que los programas se forman, se debe

considerar adoptar un método formal, o elegir piezas de una cantidad de métodos formales.

Lo que se está haciendo a través del proceso de desarrollo, el tema más importante es este: No se pierda. Es fácil hacerlo. Mucho de los métodos de análisis y diseño pretenden solucionar la mayoría de los problemas.

Recuerde que muchos proyectos no encajan en esa categoría, así es que se puede usualmente tener éxito en el análisis y el diseño con un relativamente pequeño grupo de métodos recomendados<sup>7</sup>. Pero alguna clase de procesos, no importa que limitado se esté generalmente en su camino es una forma mucho mejor que simplemente empezar a codificar.

Es fácil también atascarse, para caer en una “parálisis de análisis”, donde se siente como que no se puede mover hacia adelante porque no se tiene asegurado cada pequeño detalle en la etapa actual. Se debe recordar, que no importa cuánto análisis haga, hay algunas cosas acerca de un sistema que no revelan solas hasta el momento del diseño, y más cosas que no se revelaran solas hasta que se encuentre codificando, o inclusive hasta que un programa este ejecutándose. Dado esto, es crucial moverse relativamente rápido a través del análisis y el diseño, implementar una prueba del sistema propuesto.

El punto vale la pena enfatizarlo. Dada la historia que se ha tenido con los lenguajes procesales, es admirable que un equipo quiera proceder cuidadosamente y entender cada detalle antes de moverse a diseñar e implementar. Ciertamente, cuando creamos una DBMS, se paga por entender las necesidades del cliente a fondo. Pero DBMS es una clase de programas, la estructura de la base de datos es el problema a ser abordado. La clase de problema de programación discutido en este capítulo es el de la variedad de “carácter general” (mi término), en donde la solución no es simplemente formar nuevamente una solución bien conocida, pero en lugar de eso involucrar una o más “factores de carácter general” -elementos por los cuales no es entendida correctamente una solución previa, y por el cual la investigación es necesaria<sup>8</sup>. El intento de analizar a fondo un problema de carácter general antes de moverse en el diseño e implementación termina en una parálisis de análisis ya que no se tiene suficiente información para solucionar este tipo de problema durante la fase de análisis. Solucionar problemas como estos requiere iteración con el ciclo completo, y esto requiere tomar un comportamiento de riesgo (lo que tiene sentido, dado que se está tratando de hacer algo nuevo y la recompensa potencial es alta). Esto

---

<sup>7</sup> Un excelente ejemplo de esto es UML Distilled 2<sup>da</sup> edición, de Martin Fowler (Addison-Wesley 2000), que reduce el a veces abrumador proceso UML a un grupo pequeño manejable.

<sup>8</sup> Mi principio para copiar y lograr estimar estos proyectos: Si hay más de un carácter general, no intente planear cuánto tiempo va a tomar o cuánto va a costar hasta que se haya creado un prototipo que trabaje. Hay aquí muchos grados de libertad.

puede parecer como que el riesgo esta sintetizado en “apresurar” una implementación preliminar, pero se puede reducir el riesgo en un proyecto de carácter general dado que se esta encontrando antes una estrategia para ver si el problema es viable. El desarrollo de productos es un manejo de riesgos.

A menudo se propone que se “construya uno para lanzarlo”. Con la POO, se puede seguir lanzando *parte* de este, pero dado que el código esta encapsulado en clases, durante la primer aprobación se produce algún diseño de clase útil y desarrolla alguna idea importante acerca del diseño que no necesita ser tirado. De esta forma, la aprobación rápida de un problema no solo produce información crítica para el siguiente análisis, diseño e implementación, también crea una base de código.

Esto dice, que si se esta viendo una metodología que contiene tremendos detalles y sugiere muchos pasos y documentos, puede ser difícil saber cuando parar. Tenga en mente que se está tratando de descubrir:

1. ¿Que son los objetos? (¿Como se divide el proyecto en sus partes componentes?)
2. ¿Que son sus interfases? (¿Que mensajes necesitará enviar a cada objeto?)

Si comienza con nada mas que los objetos y sus interfases, entonces se puede escribir un programa. Por varias razones se debe pensar en necesitar mas descripciones y documentos que estos, pero no se puede tener menos.

El proceso puede ser abordado en cinco fases, y una fase 0 que es simplemente el compromiso inicial de utilizar algún tipo de estructura.

## Fase 0: Haga un plan

Se debe primero decidir que pasos se harán para estar adentro de su proceso. Esto suena simple (de echo, *todo* esto suena simple), y a pesar de eso las personas a menudo no hacen esta decisión antes de comenzar a codificar. Si los planes son “entre y comience a codificar”, bien (A veces esto es apropiado cuando se tiene bien entendido el problema). Al menos está de acuerdo que ese es el plan.

De debe también decidir en esta fase que alguna estructura de proceso adicional es necesaria, pero no la totalidad de las cosas. Se entiende que a algunos programadores les gusta trabajar en el “modo vacaciones”, en el cual ninguna estructura es impuesta en el proceso de desarrollo de su trabajo; “Eso puede hacerse cuando este terminado”. Esto puede ser atractivo por un momento, pero se encontrará que tener algunas metas en el camino puede ayudar a enfocar y galvanizar sus esfuerzos alrededor de estas metas en lugar de atarse a una sola meta de “terminar el proyecto”. Además,

se divide el proyecto en piezas mas digeribles y hace que parezcan menos amenazantes (además las metas ofrecen mas posibilidades de celebrar).

Cuando comienzo a estudiar la estructura de una historia (así es que algún día escribiré una novela) inicialmente me resistí a la idea de la estructura, sintiendo que escribo mejor cuando simplemente dejo que fluya en la página. Pero mas tarde me he dado cuenta que cuando escribo acerca de computadoras la estructura es suficientemente clara para mi que no tengo que pensar mucho acerca de esta. Pero sigo estructurando mi trabajo, a pesar de que solo es semiconsciente en mi cabeza. Aún cuando se piense que sus planes son solo comenzar a codificar, de algún modo sigue atravesando las fases subsecuentes de hacer y contestar ciertas preguntas.

## El objetivo de la misión

Cualquier sistema que se este armando, no importa que tan complicado sea, tiene un propósito fundamental; el tema involucrado, las necesidades que satisface. Si se puede pasar por alto la interfase de usuario, los detalles específicos de hardware -o sistema-, los algoritmos de código y los problemas de eficiencia, eventualmente se encontrará la forma de comenzarlo -simple y directo. Como el tantas veces llamado *concepto elevado* de una película de Hollywood, se puede describir en una o dos sentencias. Esta descripción pura es el punto de inicio.

El gran concepto es bastante importante porque le da el tono a el proyecto; esto es el objetivo de la misión. No se está necesariamente en lo correcto la primera vez (se debe estar en una fase mas adelantada del proyecto antes de estar completamente seguro), pero se sigue tratando hasta que se esta seguro. Por ejemplo, en un sistema de control de tráfico aéreo se puede comenzar con un concepto elevado enfocado en el sistema que se esta armando; “El programa de la torre le lleva el rastro del avión”. Pero considere que sucede cuando reduce el sistema a un pequeño aeropuerto; tal vez solo hay una sola persona controlando, o ninguna. Un modelo mas útil no le interesa la solución que se esta creando tanto como la descripción del problema: “El avión, llega, descargar, servicio y carga, luego emprende el viaje”.

## Fase 1: ¿Que estamos haciendo?

En la generación previa del diseño de programa (llamada *diseño procesal*), esto es llamado “creación de *análisis de requerimientos y especificación de sistema*”. Esto, claro, donde las apuestas se pierden; nombres intimidantes de documentos que pueden convertirse en grandes proyectos en todo su derecho. Su intención es buena, sin embargo. El análisis de requerimientos dice “Haga una lista de las directivas que se utilizarán cuando el trabajo esté

terminado y el cliente esté satisfecho". La especificación del sistemas dice "He aquí una descripción de *que* hará el programa (no como) para satisfacer los requerimientos". El análisis de requerimientos es realmente un contrato entre el programador y el cliente (aún si el cliente trabaja con su empresa, o si es otro objeto o sistema). La especificación de sistema es una exploración de nivel superior en el problema y en cierto sentido un descubrimiento de que puede ser realizado y cuanto tiempo tomará. Dado que ambos requieren consenso entre personas (y dado que generalmente cambia con el tiempo), creo que es mejor mantenerlo al descubierto tanto como sea posible - idealmente, la listas y el diagrama básico- para ahorrar tiempo. Se debe tener otras restricciones que exijan que se expanda en grandes documentos, pero manteniendo el documento inicial pequeño y conciso, puede ser creado en algunas pocas sesiones de un grupo mediante tormenta de ideas con un líder que dinámicamente cree la descripción. Esto no solo implica la entrada de cada uno, también fomenta la aceptación inicial -de los que tienen el mando y el acuerdo de todos en el equipo. Tal vez mas importante, se puede iniciar un proyecto con mucho entusiasmo.

Es necesario estar enfocado en el corazón de lo que se está tratando de lograr en esta fase: determinar que supuestamente el sistema hará. La herramienta mas valiosa para esto es un grupo de lo que es llamado "casos de uso". Los casos de uso identifican las características clave en el sistema que dan a conocer algunas de las clases fundamentales que se estarán utilizando. Estos son esencialmente respuestas descriptivas a preguntas como<sup>9</sup>:

- \* "¿Quien utilizará este sistema?"
- \* "¿Que pueden hacer estos actores con este sistema?"
- \* "¿Como harán *estos* actores *eso* con este sistema?"
- \* "¿De que otra forma se puede hacer este trabajo si alguien mas lo hace, o si el mismo actor tiene un objetivo diferente?" (para revelar variaciones)
- \* "¿Que problemas se pueden suceder cuando se está haciendo eso con el sistema?" (para revelar excepciones)

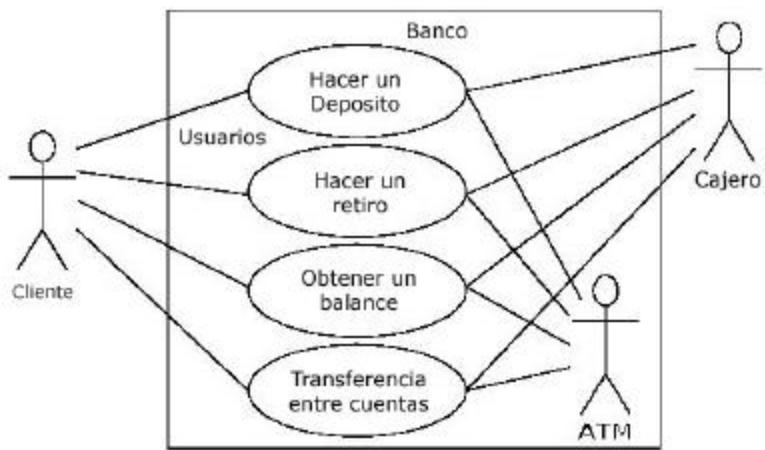
Si se está diseñando un cajero automático, por ejemplo, el caso de uso para un aspecto particular de funcionalidad del sistema es capas de describir que hará el cajero en una posible situación. Cada una de estas "situaciones" esta referida como un *escenario*, y un uso de caso puede ser considerado como una colección de escenarios. Se puede pensar en un escenario como una pregunta que comienza con: "¿Que hace el sistema si...?". Por ejemplo, "¿Que hace el cajero automático si un cliente deposita un cheque dentro de

---

<sup>9</sup> Gracias por la ayuda de James H. Jarrett.

las 24 horas, y no hay suficiente en la cuenta sin que el cheque haya sido aclarado para proporcionar el retiro deseado?”.

Los diagramas de casos de uso son intencionalmente simples para prevenir que se atasque en los detalles de la implementación del sistema prematuramente:



Cada persona que entra representa un “actor”, que es típicamente un humano u otro tipo de agente libre (Este puede ser otros sistemas de computadoras, como en el caso de “ATM”). La caja representa los límites de su sistemas. Las elipses representan los casos de uso, que son descripciones de los trabajos valiosos que pueden realizarse con el sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

No importa como esta el sistema actualmente implementado, o como se vea esta para el usuario.

Un caso de uso no necesita una horrorosa complejidad, aún si las capas bajas del sistema son complejas. Esto es solo se pretende mostrar el sistema como aparece para el usuario. Por ejemplo:



Los casos de uso producen las especificaciones determinando todas las interacciones que un usuario puede tener con el sistema. Cuando se trata de descubrir un grupo completo de casos de uso para su sistema, y una vez que se haya terminado lo que se tiene como que el corazón del sistema supuestamente tiene que hacer. Lo bueno acerca de enfocarse en los casos de uso es que ellos siempre traen lo esencial y mantienen de no quedar a la

deriva entre temas que no son críticos para obtener el trabajo terminado. Esto es, si se tiene un grupo completo de casos de uso, se puede describir su sistema y moverse en el siguiente paso. Probablemente no se obtenga todo resultado perfectamente en el primer intento, pero esto está bien. Todo quedará revelado solo en tiempo, y si se demanda una especificación perfecta en este punto se quedará atascado.

Si se queda atascado, se puede arrancar esta fase utilizando una herramienta grosera de aproximación: describiendo el sistema en algunos párrafos y luego buscar nombres y verbos. Los nombres sugieren actores, entornos de usos de caso (por ejemplo "vestíbulo"), o artefactos manipulados en los usos de caso. Los verbos pueden sugerir interacciones entre actores y casos de uso, y especifican pasos dentro el uso de caso. Se descubrirá también que los nombres y los verbos producen objetos y mensajes durante la fase de diseño (y debe notarse que los casos de usos describen interacciones entre subsistemas, así es que la técnica del "nombre y el verbo" puede ser utilizado solo como una herramienta del tipo de tormenta de ideas así es que no generan casos de uso)<sup>10</sup>.

Los límites entre un caso de uso y un actor pueden apuntar a la existencia de una interfase, pero no define una interfase de usuario. Para un proceso de definición y creación de interfaces, véase *Software for Use* por Larry Constantine y Lucy Lockwood, (Addison-Wesley Longman, 1999) o vaya a [www.ForUse.com](http://www.ForUse.com)

A pesar de que esto es magia negra, en este punto algún tipo de organización es importante. Se tiene ahora una visión general de que es lo que se está armando, así es que probablemente se sea capaz de tener alguna idea de cuánto tiempo tomará. Un montón de factores comienzan a jugar aquí. Si se estima una larga agenda entonces la empresa puede decidir no armarlo (y de esta forma utilizar sus recursos en algo más razonable -lo cual es una cosa buena). O un director puede haber decidido ya cuánto tiempo el proyecto debe tomar y tratará de influenciar la estimación. Pero es mejor tener una agenda honesta desde el comienzo y tratar con las decisiones agobiantes temprano. Ha habido un montón de intentos de comenzar con técnicas exactas para organizar una agenda (muchas de las técnicas son como las utilizadas para predecir stock de mercaderías), pero probablemente la mejor estrategia es confiar en su experiencia e intuición. Un presentimiento es probablemente correcto; se *puede* obtener algo trabajando en ese tiempo. El "doblarlo" puede convertirse en algo decente, y el 10 por ciento puede tratar con el acabado final y los detalles<sup>11</sup>. Sin embargo si quiere explicarlo, e

---

<sup>10</sup> Mas información de los casos de uso pueden ser encontrados en *Applying Use Cases* por Schneider & Winters (Addison-Wesley 1998) y *Use Case Driven Object Modeling with UML* por Rosenberg (Addison-Wesley 1999)

<sup>11</sup> Mi concepto personal de esto ha cambiado más tarde. Doblarlo y agregarle el 10 por ciento dará una estimación acertada (asumiendo que no hay muchos factores de carácter

independientemente de los lamentos y manipulaciones que suceden cuando se da a conocer semejante agenda, es justo trabajar de esa forma.

## Fase 2: Como lo haremos

En esta fase se debe comenzar con un diseño que describa como se ven que clases y como interactuarán. Una técnica excelente en determinar clases e interacciones es la tarjeta *Class-Responsibility-Collaboration*(CRC). Parte del valor de esta herramienta es que es una técnica de bajo nivel: comienza con un grupo de tarjetas de 3 x 5, y se escribe en ellas. Cada tarjeta representa una clase única, y en la tarjeta se escribirá:

1. El nombre de la clase. Es importante que este nombre capture la esencia de que hace la clase, así es que tiene sentido visto rápidamente.
2. Las “responsabilidades” de la clase: que debería hacer. Esto puede típicamente ser resumido simplemente definiendo los nombres de las funciones miembro (dado que estos nombres deben ser descriptivos en un buen diseño), pero esto no impide otras notas. Si necesita definir el proceso, vea el problema desde un punto de vista de programador perezoso: ¿Que objetos deben aparecer mágicamente para solucionar los problemas?
3. Las “colaboraciones” de la clase: ¿Que otras clases deben interactuar con esta? “Interactuar” es un término extenso intencionalmente; puede significar acumulación o simplemente que otro objeto existe que ejecutará servicios para un objeto de la clase. Colaboraciones suele considerarse el auditorio para esta clase. Por ejemplo, si se crea una clase **Petardo**. ¿Quién va a observarlo, un **Químico** o un **Espectador**? El creador va querer conocer que químicos que se utilizarán para su construcción, y luego como responderán a los colores y las formas cuando explote.

Se debe sentir como que las tarjetas deberían ser mas grandes por toda la información que se quiere meter en ella, pero son intencionalmente pequeñas, no solo para mantener sus clases pequeñas también para evitar que se entre en mucho detalle demasiado pronto. Si no se puede meter todo lo que se necesita saber acerca de la clase en la pequeña tarjeta, la clase es muy compleja (o se está siendo muy detallista, o se debe crear más de una clase). La clase ideal debe ser entendida en un solo vistazo general. La idea de las tarjetas CRC es asistir el comienzo del los primeros trazados del diseño así es que se obtiene una imagen general y se refina luego el diseño.

---

general), pero si a pesar de eso tiene que trabajar bastante diligentemente para terminar en ese tiempo. si se quiere tiempo para realmente hacerlo elegantemente y disfrutarlo en el proceso, el multiplicador correcto es mas como tres o cuatro veces, creo.

Uno de los grandes beneficios de las tarjetas CRC es en comunicación. Es mejor hacerlo en tiempo real, en un grupo, sin computadoras. Cada persona toma la responsabilidad por algunas clases (que al principio no tienen nombres u otra información). Se ejecuta una simulación para solucionar un escenario por vez, decidiendo que mensajes son enviados a los distintos objetos para satisfacer cada escenario. Cuando se comienza a mover a través de los casos de uso, se debería tener un primer corte bastante completo de su diseño.

Antes de comenzar a utilizar las tarjetas CRC, las mas exitosas experiencias de consultoría las he tenido cuando se comienza con un diseño de inicio de proyecto que involucra el estar parado de frente al equipo -que no ha armado un proyecto de POO antes- dibujando objetos en una pizarra. Hablamos de como los objetos deben comunicarse unos con otros.

Efectivamente, manejaba todas las “tarjetas CRC” en la pizarra. El equipo (que sabía lo que el proyecto supuestamente haría) verdaderamente crearon el diseño; ellos “de adueñaron” del diseño en lugar de dárselos. Todo lo que hice fue guiar el proceso haciendo las preguntas correctas, tratando las suposiciones, y tomando la información que me entregaba el equipo para modificar las conjeturas. Lo verdaderamente hermoso del proceso fue que el equipo aprendía como hacer un diseño orientado a objetos no repasando ejemplos abstractos, pero trabajando en un diseño que era mas interesante para ellos en el momento: ellos.

Una vez que se ha comenzado con un grupo de tarjetas CRC, de debe querer crear una descripción mas formal de su diseño utilizando UML<sup>12</sup>. No se necesita utilizar UML, pero puede ser útil, especialmente si se quiere colocar en un diagrama en la pared para que todos piensen, lo cual es una buena idea. Una alternativa de UML es una descripción textual de los objetos y sus interfaces, o , dependiendo de su lenguaje de programación, el código mismo<sup>13</sup>.

UML también proporciona una notación mediante diagramas adicional para describir el modelo dinámico de sus sistema. Esto es útil en situaciones en las cuales las transiciones de estado de un sistema o subsistema son suficientemente predominantes que necesitan sus propios diagramas (como en un sistema de control). Se puede también necesitar realizar una descripción de las estructuras de datos, para sistemas o subsistemas en los cuales los datos son un factor dominantes (como una base de datos).

Como se sabrá, se ha terminado con la fase 2 cuando se ha descrito todas los objetos y sus interfaces. Bueno, la mayoría de ellos -hay usualmente un pequeño grupo de ellos que se deslizan a través de grietas y no se hacen

---

<sup>12</sup> Para personas que recién comienzan, recomiendo el anteriormente mencionado *UML Distilled* 2<sup>da</sup> edición.

<sup>13</sup> Python ([www.Python.org](http://www.Python.org)) es a menudo utilizado como “pseudocódigo ejecutable”

conocer hasta la fase 3. Pero esto esta bien. Todo lo que respecta de esto es que eventualmente se ha descubierto todos los objetos. Es lindo descubrirlos temprano en el proceso, pero la POO proporciona una estructura suficiente así es que no es tan malo que se descubran objetos mas tarde. De hecho, el diseño de un objeto tienden a suceder en cinco etapas, a través del proceso de desarrollo de un programa.

## Cinco etapas para el diseño de un objeto

La vida de un objeto durante el diseño no esta limitada a el tiempo en que se esta escribiendo el programa. En lugar de eso, el diseño de un objeto aparece a través de una secuencia de etapas. Es muy útil tener esta perspectiva porque ya no se espera que las cosas salgan a la perfección en el momento; en lugar de eso, se da cuenta que el entendimiento de lo que un objeto hace y como se debe ver a través del tiempo. Esta vista también se aplica al diseño de varios tipos de programas; el patrón para un tipo de programa en particular emerge a través de luchar una y otra vez con esos problemas (esto se pone en crónicas en el libro *Thinking in Patterns with Java* que se puede bajar en [www.BruceEckel.com](http://www.BruceEckel.com)). Los objetos, también, tienen sus patrones que emergen a través del entendimiento, utilización y reutilización.

**1. Descubrimiento del objeto.** Esta etapa se sucede durante el análisis inicial de un programa. Los objetos pueden ser descubiertos buscando factores externos y límites, duplicación de elementos en el sistema y las mas pequeñas unidades conceptuales. Algunos objetos son obvios si ya se tiene un grupo de librerías de clases. El populacho entre clases sugieren que las clases base y la herencia pueden aparecer al instante, o mas tarde en el proceso.

**2. Armado de objetos.** Casi en el momento que se este creando un objeto se descubrirán las necesidades de nuevos miembros que no aparecen durante el descubrimiento. Las necesidades internas del objeto pueden requerir otras clases para soportarlo.

**3. Construcción del sistema.** Una ves mas, muchos de los requerimientos de un objeto pueden aparecer en esta etapa tardía. Como se ha aprendido, los objetos evolucionan. La necesidad de comunicación y interconexión con otros objetos en el sistema pueden cambiar las necesidades de sus clases o requerir nuevas clases. Por ejemplo, se puede descubrir las necesidades para facilitar o ayudar alguna clase, como una lista enlazada, que contiene poca o no información del estado y simplemente ayuda a funciones de otras clases.

**4. Extensión del sistema.** A medida que se agreguen nuevas características a un sistema de puede descubrir que su diseño previo no soporta extensión fácil del sistema. Con esta nueva información, se pude reestructurar partes del sistema, posiblemente agregando nuevas clases o nuevas jerarquías.

**5. Reutilización de objetos.** Esta es la verdadera presión sobre la clase. Si alguno trata de reutilizar esta en una situación totalmente nueva, probablemente ellos descubran algún defecto. Así como se cambia una clase para adaptarla a mas nuevos programas, los principios generales de la clase se hacen mas claros, hasta que se tenga un tipo verdaderamente reutilizable. Sin embargo, no espere que muchos objetos de un sistema de sistema sean reutilizables -es perfectamente aceptable para el grueso de sus objetos que sean específicos del sistema. Los tipos reutilizables tienden a ser menos comunes, y ellos deben solucionar problemas mas generales para poder ser reutilizables.

## Líneas guía para el desarrollo de objetos

Estas etapas sugieren algunas guías cuando se esté pensando en desarrollar clases:

1. Se debe dejar que un problema específico genere una clase, entonces deje que la clase crezca y madure durante la solución de otros problemas.
2. Se debe recordar que, al descubrir las clases que se necesitan (y sus interfaces) ya se ha hecho la mayor parte del diseño del sistema. Si ya ha encontrado esas clases, esto puede ser un proyecto fácil.
3. No se debe forzar a uno mismo a conocer todo en el comienzo; aprenda mientras marcha. Esto sucederá de todas formas.
4. Se debe comience a programar: Se puede obtener algo del trabajo y así probar o desaprobar el diseño. No tenga miedo de terminar escribiendo código del estilo procesal -las clases dividen el problema y ayudan a controlar la anarquía y la entropía. Las malas clases no echan a perder las buenas clases.
5. Siempre se debe mantener la simpleza. Los objetos transparentes con utilidades obvias son mejores que las interfaces complicadas. Cuando los puntos de decisión llegan, se debe utilizar la estrategia Occam's Razor: Se debe considerar las opciones y seleccionar una que sea la mas simple, porque las clases simples son siempre las mejores. Se debe comenzar pequeño y simple, y se puede extender la interfase de la clase cuando se entienda que es mejor. En el momento de seguir, es difícil remover elementos de una clase.

## Fase 3: Arme lo principal

Esta es la transformación inicial de el grueso diseño en un cuerpo de código que se puede compilar y ejecutar para ser probado, y en especial que al que se le puede probar o desaprobar la arquitectura. Esto no es un proceso de un

solo paso, pero en lugar de eso se comienza con una serie de pasos que interactivamente arman el sistemas, como se verá en la fase 4.

La meta es encontrar el corazón de la arquitectura del sistema que necesita ser implementada para obtener un sistema ejecutable, no importa que tan incompleto está ese sistema en el paso inicial. Estamos creando un esqueleto que podrá montar con iteraciones mas adelante.

También estamos armando la primera de muchas integraciones del sistema y la primeras pruebas y dando información acerca de cómo el sistema se verá y como va progresando.

Idealmente, estamos expuesto a algunos riesgos de criticas. Probablemente también descubramos cambios y mejoras que puede hacerse a la arquitectura original-cosas que no hubiera aprendido si no se implementaba el sistema.

Parte del armado del sistema son las pruebas reales que obtendremos de los ensayos contra su análisis de requerimientos y la especificación (de la forma en que existan). Debemos asegurarnos de que las pruebas verifican los requerimientos y los usos de caso. Cuando el corazón del sistema es estable, estamos listos para movernos y agregar mas funcionalidad.

## Fase 4: Iterando los casos de uso.

Una vez que el esqueleto del corazón esta corriendo, cada grupo de características que agregue es un pequeño proyecto por si mismo. Usted agrega un grupo de característica durante una iteración, un período de desarrollo razonablemente pequeño período de desarrollo.

¿Cuan grande es una iteración? Idealmente, cada iteración dura de una a tres semanas (esto puede variar basado en el lenguaje de implementación). Al final de cada período, usted tiene un sistema integrado y verificado con mas funcionalidad de la que tenía antes. Pero lo que es particularmente interesante es lo básico para la iteración: un solo uso de caso. Cada uso de caso es un paquete de funcionalidades relacionadas que son armadas durante una iteración. No solo nos esta dando una mejor idea de cual es el alcance que un uso de caso puede tener, también nos esta dando mas validez a la idea de un uso de caso, dado que el concepto no es descartado luego del análisis y del diseño, además mas bien es una unidad fundamental de desarrollo a lo largo del proceso de armado del software.

Nos detenemos cuando alcanzamos la funcionalidad esperada o llega una fecha límite y el cliente puede ser satisfecho con la versión actual.  
(Recordemos que es software es un negocio de suscripciones.) Porque el proceso es interactivo, tenemos muchas oportunidades de enviar un producto antes de que se termine; los proyectos de código de dominio

público trabajan exclusivamente en un ambiente iterativo con retro alimentación, que es precisamente lo que lo hace exitoso.

Un proceso de desarrollo iterativo se valora mas por muchas razones. Usted puede dar a conocer y resolver riesgos críticos de forma temprana, los clientes tienen amplias oportunidades para cambiar de idea, la satisfacción del programador es alta, y el proyecto puede ser dirigida con mas precisión. Pero un beneficio mas importante es la comunicación con el stakeholders, que puede ver el estado actual exacto del producto donde todos mienten. Esto puede reducir o eliminar la necesidad de reuniones que entumecen la mente para presentar estados e incrementar la confidencialidad y soporte de los stakeholders.

## Fase 5: Evolución.

Este es el punto del ciclo de desarrollo que tradicionalmente es llamado “mantenimiento”, lograr capturar todos los términos que puedan significar todo de “lograr que trabaje de la forma que fue realmente supuesta en el primer lugar” para “agregar aspectos que el cliente olvidó mencionar” para la mas tradicional “reparación de errores que se descubren” y “agregar nuevos aspectos que surgen de las necesidades.” Así muchas pequeñas interpretaciones equivocadas son aplicadas a el término “mantenimiento” que son tomadas como una calidad un poco engañosa, en parte porque sugiere que estamos actualmente armando un programa prístino y todo lo que necesita hacer es cambiar partes, aceitarlas y evitar que se oxiden. A lo mejor hay un mejor término para describir que esta sucediendo.

Usaremos el término *evolución*<sup>14</sup>. Esto es, “No se logrará de primera, así que hay que permitirse la amplitud de aprender, regresar y hacer cambios.” Se puede necesitar hacer un montón de cambios dado que se ha aprendido y entendido el problema mas profundamente. La elegancia que producida si se evoluciona hasta obtener lo correcto se recompensará, a corto y a largo plazo. Evolución es donde un programa se dirige desde lo bueno a lo mejor, y donde aquellos temas que no realmente se entendieron en los primeros pasos se aclaran. Es también donde las clases pueden evolucionar desde un proyecto con un único uso a un recurso reutilizable.

“Obtener lo correcto” no es solo que el programa trabaje de acuerdo con los requerimientos y los casos de uso. Esto también significa que la estructura interna del código tenga sentido para usted, y sienta que todo encaja bien, sin sintaxis complicada, objetos muy grandes, o pequeñas partes de código desgarbados. Además debemos tener un poco sentido en la estructura del

---

<sup>14</sup> Al menos un aspecto de la evolución es cubierto por Martín Fowler's en el libro Refactoring: improving the design of existing code (Addison-Wesley 1999), which uses Java examples exclusively.

programa que sobrevivirá a los cambios que son inevitables durante su tiempo de vida y aquellos cambios que pueden ser fáciles y limpios. Esto no es un echo menor. No solo debe únicamente entender que esta creando, también como el programa evolucionará (lo que yo llamo el *vector de cambio*). Por fortuna, la programación orientada a objetos son particularmente adepto a soportar esta continua modificación-los límites creados por los objetos son los que tienden a mantener la estructura de quiebres. Esto también permite hacer cambios-algunos que puedan ser drásticos en un programa procesal-sin causar terremotos en su código. De hecho, el soporte para la evolución puede ser el beneficio mas importante de la POO.

Con la evolución, crearemos algo que al menos se aproxima a lo que pensamos que estamos creando, y entonces lo examinaremos, lo comparamos con los requerimientos, y vemos donde esta débil. Entonces podemos regresar y corregirlo rediseñándolo y implementando nuevamente las porciones del programa que no trabajan correctamente. Podemos necesitar solucionar un problema o un aspecto de un problema, muchas veces antes de obtener la solución correcta<sup>15</sup>. (Un estudio de diseño de patrones es útil aquí (Podemos encontrar información en *Thinking in Patterns with Java*, el cual podemos bajar en [www.BruceEckel.com](http://www.BruceEckel.com))).

La evolución también sucede cuando creamos un sistema, vemos que se corresponde con los requerimientos, y entonces descubrimos que no es lo que necesitamos actualmente. Cuando veamos un sistema en operación, encontraremos que realmente queremos solucionar un problema diferente. Si pensamos este tipo de evolución va a suceder, entonces debemos armar su primer versión lo mas rápido posible así podemos encontrar si es ciertamente lo que queremos.

Tal vez lo mas importante para recordar es que por defecto-por definición en realidad-si modificamos una clase, su padre y sus hijos deben seguir funcionando. No debemos tener miedo a la modificación (especialmente si tenemos creadas un grupo de unidad de verificación para saber si son correctas sus modificaciones).

Las modificaciones no deben necesariamente romper el programa, y cualquier cambio en los resultados deben ser limitados a una subclase y o a colaboradores específicos de la clase que cambiemos.

---

<sup>15</sup> Esto es algo como “rápidamente hacer un prototipo”, donde se supone que crearemos una rápida y sucia versión con la que podemos aprender acerca del sistema, y luego tirar nuestro prototipo y crear el correcto. Los problemas con hacer un prototipo rápido es que las personas no tiran el prototipo, en lugar de eso arman sobre el. Combinado el agujero en la estructura en la programación de los procedimientos, esto a menudo termina en sistemas desordenados que son caros de mantener.

## El planeamiento paga.

Por su puesto no construiremos una casa sin un montón de cuidadosos planos. Si construimos una cubierta o una casa para el perro, los planos no son tan elaborados, pero probablemente comenzemos con algún tipo de esquema para guiarlos. El desarrollo de software se fue a los extremos. Por un largo tiempo, las personas no tenían mucha estructura en sus desarrollos, pero los proyectos grandes comenzaban a fracasar.

Reaccionando a esto, terminamos con metodologías que tenían una cantidad intimidatoria de estructura y detalle, en un principio para aquellos grandes proyectos. Estas metodologías que asustaban mucho de usar-se veían como gastando todo el tiempo escribiendo documentos y nunca programando (A menudo este era lo que sucedía). Espero que lo que mostramos aquí sugiera un camino intermedio-una escala variable.

Utilicemos una aproximación que se adapte a las necesidades (y a nuestra personalidad). No importa que tan pequeña elija hacerlas, algún tipo de plan hará una mejora grande en el proyecto, lo opuesto a lo que sucederá si no tiene ningún plan. Recordemos que, estimando, un 60 por ciento de los proyectos fracasan (algunos estiman un 70 por ciento!).

Siguiendo un plan-preferentemente uno corto y simple-y, comenzando con el diseño de la estructura antes de codificar, descubriremos que las cosas se dan juntas mucho mas fácilmente que si nos agarramos y comenzamos cabalgar. También nos sentiremos mucho mas satisfechos. Por experiencia propia el echo de comenzar con una elegante solución es profundamente satisfactoria a un nivel totalmente diferente; se siente mas cerca del arte que de la tecnología. Y la elegancia siempre tiene recompensa; No es una búsqueda de frivolidad. No solo le daremos a el programa la facilidad al construir y depurar, será mas fácil de entender y mantener, y esto es con lo que se maneja el valor financiero.

## Programación extrema

He estudiado técnicas de análisis y diseño, de vez en cuando, desde que me gradué en la escuela. El concepto de programación extrema (XP) es el mas radical, y encantador que he visto. Se puede encontrar su historia en *Extreme Programming Explained by Kent Beck* (Addison-Wesley, 2000) y en la Web en [www.xprogramming.com](http://www.xprogramming.com).

XP es una filosofía acerca del trabajo de programación y un grupo de instrucciones para hacerlo. Alguna de estas instrucciones son reflejadas en otras metodologías recientes, pero las dos mas importantes y distintivas contribuciones, en mi opinión, son “escribe las pruebas primero” y “programación en pareja”. A pesar que discute fuertemente de todo el

proceso, Beck apunta a que si adoptamos solo estas dos prácticas aumentaremos enormemente nuestra productividad y confianza.

## Escribir primero las pruebas

Las pruebas, tradicionalmente son relegadas a la última parte del proyecto, después de que “tenemos todo el proyecto, solo para estar seguro”. Tiene implícitamente una prioridad baja, y las personas que se especializa en eso no nos esta dando un montón de estados y están a menudo acordonados en un sótano, lejos de los “programadores reales”. Los equipos de prueba tienen que responden amablemente, yendo mas lejos que sus ropas negras y ríen con alegría cuando quiebran algo (para ser honesto, he tenido ese sentimiento cuando quiebro compiladores).

XP completamente revoluciona el concepto de prueba dando igual (o a veces mayor) prioridad que al código. De hecho, se escriben las pruebas *antes* que el código que será probado, y las pruebas se quedan con el código para siempre. Las pruebas debes ser ejecutadas con éxito cada vez que realiza una integración del proyecto (que a menudo, es mas de una vez al día).

Escribir las pruebas primero tiene importantes efectos.

Primero, fuerza a una clara definición de la interfase de una clase. Esto a menudo sugiere que las personas “imaginan la clase perfecta para solucionar un problema particular” como una herramienta con la que trataremos de diseñar el sistema. La estrategia de prueba de XP va mas lejos que eso-especifica exactamente como se debe ver una clase, a el consumidor de esa clase, y exactamente como la clase debe comportarse. No en términos inciertos. Podemos escribir todos los versos, o crear todos los diagramas que queramos, describiendo como una clase debe comportarse y como se vera, pero nada es mas real que un conjunto de pruebas. El molde es una lista de deseos, pero las pruebas son un contrato que es impuesto por el compilador y el programa que se ejecuta. Es difícil imaginar una descripción mas concreta de una clase que las pruebas.

Cuando creamos las pruebas, estamos forzados a pensar en la totalidad de la clase y a menudo descubriremos funcionalidades que necesitamos que puede que nos hayamos olvidado mientras experimentábamos con diagramas UML, tarjetas CRC, casos de usos, etc.

El segundo mas importante efecto de escribir las pruebas primero llega al correr las pruebas cada vez que armamos el software. Esta actividad nos otorga la otra mitad de las pruebas que ejecuta el compilador. Si miramos la evolución de los lenguajes de programación en esta perspectiva, veremos un avance en las tecnologías que actualmente evolucionan alrededor de las pruebas. El lenguaje ensamblador prueba solo la sintaxis, pero C impuso algunas restricciones semánticas y estas previenen de cometer cierto tipo de

errores. Los lenguajes de POO imponen aun mas restricciones semánticas, que si pensamos en ellas son actualmente formas de probar. “¿Es este tipo de dato utilizado correctamente?” o “¿Es esta función llamada correctamente?” son los tipos de pruebas ejecutadas por el compilador o en tiempo de ejecución del sistema. Vemos los resultados de tener estos tipos de pruebas armados en el lenguaje: las personas son capaces de escribir sistemas mas complejos, y hacerlos trabajar, con mucho menos tiempo y esfuerzo. He quedado perplejo acerca de por que es esto, pero ahora me he dado cuenta que esta es la prueba: hacemos algo mal, y la segura red de pruebas incluidas nos indica que tenemos un problema e indica donde es.

Pero el sistema de pruebas incluido ofrecido por el diseño del lenguaje puede solo ir hasta ahí. En algún momento, *nosotros* debemos dar un paso adelante y agregar el resto de las pruebas que producen un grupo completo (en cooperación con el compilador y el sistema en tiempo de ejecución) que verifique todo nuestro programa. ¿Y simplemente teniendo al compilador mirando sobre nuestro hombro, no queremos que estas pruebas nos ayuden desde el comienzo? Es por esto que escribimos las pruebas al comienzo, y las ejecutamos automáticamente cada vez que armamos el sistema. Las pruebas extienden la red de seguridad que proporciona el lenguaje.

Una de las cosas que he descubierto acerca del uso de lenguajes de programación mas y mas poderosos es que me animo a intentar experimentos mas descarados, porque se que el lenguaje me evitará que gaste mi tiempo en perseguir errores. El esquema de pruebas de XP hace lo mismo con su proyecto entero. Porque sabemos que las pruebas encontraran cualquier problema que introduzca ( y regularmente agrega nuevas pruebas a medida que pensemos en ellas), podemos hacer grandes cambios cuando necesitemos sin preocuparnos en llevar el proyecto entero a un completo desastre. Esto es increíblemente poderoso.

## Programación en pareja

La programación en pareja va en contra del fuerte individualismo al que hemos sido adoctrinados desde el comienzo, a través de la escuela (donde acertamos o fallamos nosotros mismos, y el trabajar con nuestros vecinos es considerado “hacer trampa”), y los medios, especialmente las películas de Hollywood en el que el héroe usualmente pelea contra el conformismo de los cortos de mente<sup>16</sup>. Los programadores, también son considerados parangones del individualismo- “codificadores vaqueros” como a Larry Constantine le gustaba decir. Y a pesar de eso XP, que por si sola batalla contra el pensamiento convencional, dice que el código debe ser escrito por

---

<sup>16</sup> A pesar de que esto puede ser una perspectiva mas americana, las historias de Hollywood alcanzan a todos.

dos personas por estación de trabajo. Y que esto debe ser hecho en un área con un grupo de estaciones de trabajo, sin las barreras a las cuales personas del plantel de diseño esta tan encariñada. De echo, Beck dice que la primera tarea para convertir en XP es llegar con un destornillador, una llave Allen y quitar todo lo que este en el camino<sup>17</sup>.

El valor de la programación en pareja es que en el momento en que una persona esta codificando la otra persona esta pensando acerca de esto. La persona que piensa mantiene una imagen general en mente-no solo la imagen del problema a mano, también las instrucciones de XP. Si dos personas están trabajando, es menos probable que uno de ellos se vaya diciendo, “No quiero escribir las pruebas primero”, por ejemplo. Y si el codificador se queda parado, pueden intercambiar lugares. Si los dos se quedan parados, sus meditaciones pueden ser alcanzadas por alguien mas en el área de trabajo que pueda contribuir. Trabajar en parejas mantiene las cosas fluyendo y en camino. Probablemente mas importante, hace la programación mucho mas social y divertida.

He comenzado a utilizar programación en pareja durante los períodos de ejercicios en alguno de mis seminarios y parece que mejora significativamente la experiencia de cada uno.

## Por que se sucede Java

La razón por la que Java ha sido tan exitoso es que la meta es solucionar muchos de los problemas a los que se enfrentan los desarrolladores hoy. La meta de java es mejorar la productividad. Esta productividad llega de varias formas, y el lenguaje es diseñado para ayudarnos lo mas que se pueda, ayudándonos un poco con lo posible mediante reglas arbitrarias o algún requerimiento que usemos en un grupo particular de prestación. Java es diseñado para ser práctico; las decisiones de diseño son basadas en proporcionarle los máximos beneficios a el programador.

---

<sup>17</sup> Incluido (especialmente) el sistema PA. Una vez trabajé en una compañía que insistía en transmitir cada llamada que llegaba a cada ejecutivo, esto interrumpía constantemente nuestra productividad (pero los directores no comenzaban a entender lo vergonzoso que es un servicio tan honorable como el PA). Finalmente, cuando nadie estaba mirando comenzaba a atacar los cables de los parlantes.

## Los sistemas son fáciles de expresar y de entender

Las clases están diseñadas para introducir el problema tendiendo a expresarlo de mejor manera. Esto significa que cuando escribimos el código, estamos describiendo nuestra solución utilizando la terminología del espacio del problema (“Coloca el clip en la papelera”) en lugar de utilizar los términos de la computadora, cual es la solución en el espacio (“Coloque el bit en el chip que significa que el relee cerrará”). Usted se maneja con conceptos de alto nivel y puede hacer mucho mas con una simple línea de código.

Los otros beneficios de esta facilidad de expresión es el mantenimiento, que (si se puede creer en los reportes) tomo una gran cantidad del costo del tiempo de desarrollo. Si un programa es fácil de entender, entonces es fácil de mantener.

Esto también reduce el costo de crear y mantener la documentación.

## Máxima ventaja con librerías

La manera mas rápida de crear un programa es utilizar código que ya este escrito: una librería. El logro mas grande en Java es hacer las librerías fáciles de utilizar. Esto es llevado a cabo fundiendo librerías dentro de nuevos tipos de datos (clases), así es que introduciendo una librería significa agregar nuevos tipos al lenguaje. Dado que el compilador tiene cuidado de cómo la librería es utilizada-garantizando la correcta inicialización y limpieza, y asegurándose que las funciones son llamadas de forma adecuada-podemos focalizar nuestra atención en lo que queremos que la librería haga, no como debemos hacerlo.

## Manejo de error

El manejo de error en C es un problema notorio, y uno que a menudo es ignorado-cruzar los dedos es usualmente involucrado. Si usted esta desarrollando un largo y complejo programa, no hay nada peor que tener un error enterrado en algún lugar sin indicio de donde viene. El *manejo de excepciones* de Java es una forma de garantizar que un error es indicado, y que algo sucede como resultado.

## Programando a lo grande

Muchos lenguajes tradicionales tienen limitaciones de armado para programas grandes y complejos. Por ejemplo BASIC, puede ser genial para obtener rápidas soluciones en grupo para cierta clase de problemas, pero si el programa tiene mas de algunas pocas páginas de largo, o nos aventuramos fuera de los dominios de problemas normales del lenguaje, es como nadar por un fluido cada vez mas viscoso. No hay líneas claras que nos digan cuando el lenguaje esta fallando o siquiera donde, lo ignoraremos. No diremos, "Mi programa BASIC solo creció demasiado; tendré que rescribirlo en C!" En lugar de eso, se tratará de calzar unas líneas mas para agregar una nueva característica. Así los costos extras silenciosamente crecen.

Java está diseñado para ayudar a *programar a lo largo*; esto es, borrar el límite de crecimiento de complejidad entre un programa pequeño y uno largo. Ciertamente no se necesita el uso de POO cuando estamos escribiendo una utilidad del estilo de "Hola mundo", pero las características están cuando las necesitamos. Y el compilador es agresivo a la hora de encontrar la causa de errores en pequeños y grandes programas de la misma forma.

## Estrategias para el cambio

Si se compran acciones de POO, la siguiente pregunta es que es probable es. "Como puedo lograr que mis representantes, colegas, departamento o iguales comiencen a utilizar objetos?" Piense como una persona-un programador independiente- puede aprender el uso de un nuevo lenguaje y de un nuevo paradigma de programación. Ya lo ha hecho primero. Se comienza con educación y ejemplos; luego se inicia un juicio de un proyecto para obtener las básicas sin hacer algo muy confuso. Luego viene el proyecto en el "mundo real" que actualmente hace algo útil.

Desde el comienzo del primer proyecto se continúa la educación leyendo, preguntando a expertos e intercambiando consejos con sus amigos. Esta es la aproximación que muchos programadores experimentados sugieren para migrar a Java. Migrar una compañía entera introducirá por supuesto cierta dinámica de grupo, y se ayudará en cada paso a recordar como una persona pudo hacerlo.

## Instrucciones

He aquí algunas instrucciones a considerar cuando hacemos una transición a POO o Java:

## 1. Entrenamiento

El primer paso es algo de educación. Recordemos que la compañía invierte en código, y trata de no echar todo a perder por seis o nueve meses donde cada uno se esfuerza para saber como funciona la interfase. Tomemos un pequeño grupo para adoctrinar, preferentemente compuesto por personas que son curiosas, que trabaje bien juntas y que puedan funcionar como su propia red de soporte cuando estén aprendiendo Java.

Una alternativa próxima que a veces es sugerida es la educación de toda la empresa por niveles, incluyendo un vistazo inicial de cursos de estrategia de dirección al igual que de diseño y programación para las personas que arman el proyecto. Esto es especialmente bueno para las pequeñas compañías que hacen movimientos fundamentales en la forma que hacen las cosas, o a nivel de división de grandes compañías. Sin embargo, dado que el costo es alto, algunos pueden elegir comenzar con proyectos a nivel de entrenamiento, un proyecto piloto (posiblemente un mentor externo) y dejar que el equipo de proyecto comience a educar al resto de la compañía.

## 2. Proyectos de bajo riesgo

Se puede intentar primero con un proyecto de bajo riesgo y permitirnos errores. Una vez que se ha ganado alguna experiencia, se puede iniciar otros proyectos con miembros de este primer equipo o utilizar miembros del equipo como soporte técnico de POO. El primer proyecto puede no trabajar correctamente al principio, por lo que no debe ser una misión crítica para la compañía. Debe ser simple, auto contenida, e instructiva; esto significa que debe incluir la creación de clases que serán significativas para otros programadores en la empresa cuando les toque el turno de aprender Java.

## 3. Modelo para el éxito

Se buscarán ejemplos de buenos diseños orientados a objetos antes de comenzar. Hay una buena probabilidad de que alguien ya haya solucionado el problema antes y aunque no se la solución exacta a su problema podremos aplicar lo que hemos aprendido sobre abstracción para modificar un diseño para que cumpla sus necesidades. Este es el concepto general del *diseño de patrones* cubierto en *Thinking in Patterns with Java* el que se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com).

## 4. Use librerías de clases existente

La principal motivación económica para migrar a POO es la facilidad de uso del código existente en la forma de librerías de clases (en particular, la Librería Estándar de Java, el que es cubierto a lo largo de este libro). Se obtendrá el ciclo de desarrollo mas corto cuando se pueda crear y utilizar

objetos de una librería propia. Sin embargo, algunos nuevos programadores no entienden esto, ignoran la existencia de clases que pueden existir o fascinados con el lenguaje, decidir escribir clases que ya existan. El éxito con la POO y Java puede ser optimizado si se hace un esfuerzo de búsqueda y reutilizar código de otras personas en el proceso de transición.

## 5. No rescribir código existente en Java

Usualmente el mejor uso del tiempo tomar código existente y funcional y rescribirlo en Java (Si debe insertarlo en objetos, puede crear una interfase con código C o C++ usando la Interfase Nativa de Java, descrita en el apéndice B.). Los beneficios se incrementan, especialmente si el código es escrito para reutilizarse. Pero las posibilidades son que no se vea el dramático incremento en la productividad que se espera en los primeros proyectos a no ser que el proyecto sea nuevo. Java y la POO se lucen mejor cuando se toma un proyecto de un concepto a la realidad.

## Manejo de obstáculos

Si usted es un director, su trabajo es proporcionar recursos para su equipo, superar las barreras para su éxito, y en general para proporcionar un ambiente productivo y agradable, así su equipo es mas prometedor de lograr esos milagros que siempre son requeridos.

Migrar a Java corresponde con las tres categorías, y sería maravilloso si tampoco no costara nada. Aunque migrar a Java puede ser barato-dependiendo de sus restricciones económicas-que las alternativas para un equipo de programadores C (y probablemente para programadores de otros lenguajes procesales), no es gratis y hay obstáculos de los cuales deberemos ser conscientes antes de tratar de cautivarse con una migración a Java sin su compañía y embarcarse en esta empresa solo.

## Costos iniciales

El costo de migrar a Java es mas que solo la adquisición de los compiladores de Java (El compilador de Java de Sun es gratis, así que difícilmente es un obstáculo). Sus costos a mediano y largo plazo serán minimizados si invierte en entrenamiento (y posiblemente tutelando su primer proyecto) y también si identifica y compra librerías de clases que solucionen su problema en lugar de intentar crear sus propias librerías. Estos son costos altos que deben ser divididos en factores en un propósito relista. Además, hay costos ocultos de pérdida de productividad del aprendizaje y un nuevo ambiente de trabajo. El entrenamiento y la presencia de un tutor pueden ciertamente minimizar estos, pero los miembros del equipo deben superar sus propios problemas para entender la nueva tecnología. Durante este proceso

cometerán muchos errores (esto es una herramienta, el conocimiento de los errores es el camino mas rápido de aprendizaje) y serán menos productivos. Aun así, con algunos tipos de problemas de programación, las clases correctas y el ambiente de desarrollo adecuado, es posible ser mas productivo aprendiendo Java (aun considerando que se están cometiendo muchas equivocaciones y escribiendo pocas líneas de código por día) que lo que sería quedándose con C.

## Temas de rendimiento

Una pregunta usual es, “La POO no hace automáticamente mis programas mas grandes y pesados?” La respuesta es, “Eso depende”. Las características de seguridad extras en Java son tradicionalmente penalizadas como en lenguajes como C++. Tecnologías como “hotspot” y las tecnologías de compilación han mejorado significativamente la velocidad en muchos casos y los esfuerzos continúan en camino a mejorar el rendimiento.

Cuando el enfoque es un rápido prototipo, se colocan componentes lo más rápido posible ignorando los temas de rendimiento. Si usted esta utilizando cualquiera de las librerías de terceros, estas usualmente optimizadas por sus vendedores; de cualquier forma no es un tema que se tenga cuando se desarrolla rápidamente. Cuando tenemos un sistema que queremos, si es lo suficientemente pequeño y rápido, entonces se habrá terminado. Si no, se deberá ajustar con una herramienta de prueba, mirando primero acelerando lo que se pueda hacer escribiendo nuevamente pequeñas porciones de código. Si esto no ayuda, buscaremos modificaciones que puedan ser hechas en la implementación subyacente así es que ningún código que use una clase particular necesita ser cambiada. Solo sin nada mas soluciona el problema se cambiará el diseño. El echo de que el rendimiento es tan crítico en esa porción del diseño es un indicador que puede ser parte del criterio primario de diseño. Se tiene el beneficio de encontrar el problema temprano utilizando un rápido desarrollo.

Si se encuentra una función que es particularmente un cuello de botella, se puede rescribir esta en C o C++ usando métodos nativos de Java, es tema del apéndice B.

## Errores comunes de diseño

Cuando un equipo comienza con POO y Java, los programadores típicamente cometen una serie de errores de diseño. Esto a menudo sucede a causa de una falta de respuesta de los expertos durante el diseño y la implementación de proyectos tempranos, porque no expertos han desarrollado sin la compañía y porque hay mucha resistencia para retener asesores. Es fácil sentir que entendemos rápidamente la POO en el ciclo y nos vamos por un camino equivocado. A veces lo que es obvio para alguien

experimentando con el lenguaje puede resultar un gran debate para un principiante. Mucho de este trauma puede ser evitado utilizando un experto con experiencia de afuera que sirva de tutor nos entrene.

## Java versus C++?

Java se ve mucho como C++, y es natural que veamos que C++ puede ser remplazado con Java. Comencemos a cuestionarnos esta lógica. Por una cosa, C++ aun así tiene algunas características que Java no, y a pesar de que ha habido un montón de promesas acerca de que Java será algún día igual de rápido o mas rápido que C++, vimos mejoras en la estabilidad pero no logros dramáticos. También, vemos parece estar en un continuo interés en C++, así es que no piense que ese lenguaje se desaparecerá pronto (Los lenguajes parecen merodear. Hablando de uno de mis "Seminarios de Java Intermedio-avanzado", Allen Holub defendió que el lenguaje mas comúnmente utilizado era Rexx y COBOL, en ese orden).

Comienzo a creer que la fuerza de Java pelea en una arena algo diferente que C++. C++ es un lenguaje que no intenta adecuarse a un molde. Ciertamente ha sido adaptado en un varias formas para solucionar problemas particulares. Algunas herramientas de C++ combinan librerías, modelos de componentes, y herramientas de generación de código para solucionar el problema de desarrollo de aplicaciones finales para el usuario con ventanas (para Microsoft Windows). Y a pesar de eso, que usan la mayoría de los desarrolladores Windows? Microsoft's Visual Basic (VB). Esto a pesar del echo que VB produce un tipo de código que comienza a ser inmanejable cuando el programa tiene algunas páginas de largo (y la sintaxis puede ser absolutamente desconcertante). Casi tan exitoso y popular es VB, como un no muy buen ejemplo de lenguaje de diseño. Sería bueno tener la simplicidad y poder de VB sin el código inmanejable resultante. Y esto es lo que pienso que Java se lucirá: como el "siguiente VB". Se pueden o no se pueden estremecer por esto, pero si pensamos acerca de esto: a tal grado Java intenta hacer fácil al programador solucionar problemas a nivel de aplicación como redes de trabajo y interfaces de usuarios en plataformas cruzadas, y aun así con un diseño de lenguaje que permite la creación de código largo y flexible. Agreguemos a esto el echo de que Java tiene el sistema de verificación de tipos y manejo de errores mas robusto que he visto en un lenguaje y podemos dar un significante salto para adelante en la productividad al programar.

¿Se debe utilizar Java en lugar de C++ para un proyecto? Aparte de los applets, hay dos temas a considerar. Primero, si se quiere utilizar una gran cantidad de librerías de C++ existentes (y ciertamente se tiene un montón de productividad ganada aquí), o si se tiene código base C o C++, Java tal vez reduzca la velocidad de caída del desarrollo en lugar de acelerarla.

Si se está desarrollando todo el código primariamente de uno poco prolífico, entonces la simplicidad de Java sobre C++ acortará significativamente el tiempo de desarrollo—la evidencia anecdótica (historias de equipos C++ que hablan de cómo cambiaron a Java) dicen que doblaron la velocidad de desarrollo. Si el rendimiento de Java no importa o puede compensarlo de alguna manera, tiempo total de comercialización hace difícil elegir C++ sobre Java.

El gran tema es el rendimiento. El intérprete de Java es lento, desde 20 a 50 veces más lento que C en intérpretes originales. Esto ha mejorado mucho, pero sigue siendo un valor muy alto. Lo que las computadoras son acerca de la velocidad; si no son significativamente más rápidas para hacer algo entonces se hace a mano (He oído esto sugiriendo que se comienza con Java para ganar un tiempo de desarrollo, entonces si necesita una mayor velocidad de ejecución se utiliza una herramienta y bibliotecas de soporte para convertir el código a C++).

La clave para hacer Java factible para la mayoría de los proyectos de desarrollo es la apariencia de mejoras en la velocidad llamadas a veces compiladores “just-in time” (JIT), la tecnología de propiedad de Sun’s llamada “hotspot”, y los compiladores de código nativo eliminan la tan pregonada ejecución de programas en plataforma cruzada pero pueden llevar más cerca la velocidad del ejecutable más cerca de C y C++. Y compilar de forma cruzada un programa en Java puede ser mucho más fácil que hacerlo en C o en C++ (en teoría, solo se compila nuevamente, pero la promesa fue hecha antes para otros lenguajes).

Se pueden encontrar comparaciones de Java con C++ y observaciones acerca de Java en los apéndices de la primera edición de este libro (Disponible en este CDROM, de misma forma que en [www.BruceEckel.com](http://www.BruceEckel.com)).

## Resumen

Este capítulo intenta dar un amplio sentido de los temas de la programación orientada a objetos y Java, incluyendo por qué la POO es diferente, y por qué Java en particular es diferente, conceptos de metodologías de la POO y finalmente los tipos de temas que se encuentran cuando se migra una compañía a la POO y Java.

POO y Java no es para cualquiera. Es importante evaluar sus necesidades y decidir cuándo Java va a satisfacer óptimamente esas necesidades, o si es tal vez es mejor desenchufarse con otro sistema de programación (incluyendo el actualmente utilizado). Se sabe que las necesidades serán muy especializadas para el futuro predecible y si las restricciones específicas no pueden ser satisfechas por Java, entonces se deberá investigar las

alternativas<sup>18</sup>. Aun si se elige eventualmente Java como el lenguaje, al menos se entenderán que opciones hay y tener una clara visión de por qué se tomó esa dirección.

Se sabe que un programa procesal se ve como: definiciones de datos y llamadas a funciones. Para encontrar el significado de cómo un programa en el que hay que trabajar un poco, mirar a través de las llamadas a funciones y los conceptos de bajo nivel para crear un modelo en la mente. Esta es la razón por la que se necesitan representaciones intermedias cuando diseñamos programas procesales-por si solos, estos programas tienden a ser confusos porque los términos de la expresión son orientados mas cerca de la computadora que del problema que se está intentando solucionar.

Dado que Java agrega muchos nuevos altos conceptos de los que encontrará en un lenguaje procesal, se asumirá naturalmente que el main() en un programa Java será mucho mas complicado que su equivalente en C. Aquí se sorprenderá placenteramente: Un programa Java bien escrito es generalmente mas simple y mas fácil de entender que su programa equivalente en C. Lo que se verá son definiciones de objetos que representen conceptos en el espacio de su problema (en lugar de los temas de la representación en computadoras) y mensajes enviados a esos objetos para representar las actividades en ese espacio. Uno de los deleites de la programación orientada a objetos es esa, con un programa bien diseñado, es fácil entender el código para leerlo. Usualmente hay una gran cantidad de código también, porque muchos de los problemas serán resueltos utilizando nuevamente código de librerías existentes.

---

<sup>18</sup> En particular, recomiendo mirar Pitón (<http://www.Python.org>).

# 2: Todo es un objeto

A pesar de que es basado en C++, Java es mas que un lenguaje orientado a objetos “puro”.

C++ y Java son lenguajes híbridos, pero en Java los diseñadores sintieron que la hibridación no era tan importante como lo fue en C++. Un lenguaje híbrido permite muchos estilos de programa; la razón por la que C++ es un híbrido es para soportar la compatibilidad hacia atrás con el lenguaje C. Dado que C++ es un conjunto superior de C, esto incluye muchas de las características indeseables que pueden hacer algunos aspectos de C++ demasiado complicados.

El lenguaje Java asume que no queremos solo programación orientada a objetos. Esto significa que antes que comencemos deberemos mover nuestra mente a un mundo orientado a objetos (a no ser que ya la tengamos ahí). El beneficio de este esfuerzo inicial es la habilidad de programar en un lenguaje que es muy simple de aprender y de utilizar que otros muchos lenguajes OO. En este capítulo veremos los componentes básicos de un programa Java y aprenderemos que en Java, todo es un objeto, incluso el programa.

## Manipulamos objetos con referencias

Cada lenguaje de programación tiene su propios métodos para manipular datos. A veces los programadores deben ser constantemente conscientes de que tipo de manipulación está haciendo. ¿Se esta manipulando el objeto directamente, o esta lidiando con algún tipo de representación indirecta (un puntero en C o C++) que debe ser tratado con una sintaxis especial?

Todo esto es simplificado en Java. Tratamos todo como un objeto, así que hay una sola sintaxis consistente que utilizaremos en todos lados. De la misma forma que *trataremos* todo como un objeto, el identificador que manipularemos es actualmente una “referencia” a un objeto<sup>1</sup>. Se puede

---

<sup>1</sup> Esto puede ser un punto de explosión. Existen aquello que dicen “claramente, es un puntero”, pero esto presume de una implementación de capas inferiores. Además, las referencias en Java son mucho mas semejantes a las de C++ que a punteros es sus sintaxis. En la primera edición de este libro, elegí inventar un nuevo término, “manejar”, porque las

imaginar esta escena como un televisor (el objeto) con un control remoto (la referencia). El tiempo que se tenga esta referencia, se tendrá conexión con el televisor, pero cuando se le indica “cambie el canal” o “baje el volumen” lo que se está manipulando es la referencia, que es la que modifica el objeto. Si se queremos movernos por el cuarto y seguir teniendo control sobre el televisor tomaremos la referencia-remoto y la llevaremos con nosotros, no el televisor.

Además, el control remoto puede quedarse en nuestra propiedad, sin televisor. Esto es, solo porque tener una referencia no significa que tengamos que estar conectados a ella. Así es que si queremos mantener una palabra o sentencia, crearemos una referencia **String**:

```
| String s;
```

Pero aquí hemos creado solo la referencia, no el objeto. Si decidimos enviar un mensaje a `s` en este momento, obtendremos un error (en tiempo de ejecución) porque `s` no está actualmente asociado con nada (no hay televisor). Una práctica segura es siempre inicializar una referencia cuando se es creada:

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

Sin embargo, esto utiliza una característica especial de Java: las cadenas pueden ser inicializadas con texto entre comillas. Normalmente, se deberá utilizar un tipo mas general de inicialización para los objetos.

## Debemos crear todos los objetos

Cuando se crea una referencia, se conecta con un nuevo objeto. Generalmente se hace con la palabra clave **new**. Esta palabra indica algo así

---

referencias de C++ y las de Java tienen importantes diferencias. Me retiré de lo que es C++ y no confundí a los programadores de C++ a quienes asumí como la mayor audiencia para Java. En la 2da edición, decidí que “referencia” es un término mas comúnmente utilizado, y que todos cambian de C++ que tienen que hacer mucho mas frente a esa terminología de referencias, así es que pueden saltar en dos pies igualmente. Sin embargo, hay personas que les desagrada también el término “referencia”. Leí en un libro que es “completamente equivocado decir que Java soporta paso por “referencia” porque los identificadores de objetos en Java (de acuerdo con ese autor) son *actualmente* “referencias a objetos”. Y (ahí va nuevamente) todo es *actualmente* pasado por valor. Así es que no estamos pasando por referencia, estamos “pasando una referencia a un objeto por valor”. Se puede discutir de la precisión de tan complicadas explicaciones, pero pienso que mi aproximación simplifica entender el concepto con el que dejamos mal parado todo (bueno, los abogados del lenguaje pueden reclamar que estoy mintiendo, pero diré que estoy proporcionando una abstracción apropiada).

como “Hágame como uno nuevo de esos objetos”. Así que en el siguiente ejemplo podemos decir:

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

Esto no solo significa “Hágame como un nuevo **String**”, también indica como crear ese **String** dando una cadena de caracteres como información inicial.

Claro que, **String** no es el único tipo que existe. Java ya viene con una abundancia de tipos ya hechos. Lo que es mas importante que se puede hacer es crear sus propios tipos. De hecho, esta es la actividad fundamental en la programación Java, y esto es lo que aprenderá en el resto de este libro.

## Donde se almacena

Es útil visualizar algunos aspectos de cómo las cosas son colocadas cuando el programa esta corriendo, en particular como la memoria es arreglada. Hay seis diferentes lugares donde se almacenan los datos.

1. **Registros.** Esta es la forma mas rápida de almacenar los datos porque se encuentra en un lugar muy diferente a cualquier otro dato almacenado: adentro del procesador. Sin embargo el numero de registros es muy limitado, así es que los registros son reubicados por el compilador de acuerdo con sus necesidades. No se tiene un control directo y no se ve ninguna evidencia en sus programas de que los registros verdaderamente existan.
2. **La pila.** Se encuentra en el área de la memoria RAM (Memoria de acceso aleatorio), pero tiene apoyo del procesador mediante el *puntero de pila*. El puntero de pila es movido hacia abajo para crear un nuevo espacio en memoria y hacia arriba para liberar esa memoria. Esto es una forma muy rápida y eficiente para ubicar espacio de almacenamiento, seguido solo por los registros. El compilador Java debe saber, cuando es creado el programa el tamaño exacto del tiempo de vida de todos los datos que son almacenados en la pila, porque debe generar el código para mover el puntero de pila arriba y abajo. Este restriccción de lugar limita la flexibilidad de sus programas, así es que aunque algún tipo de almacenamiento existe en la pila-en particular, referencias a objetos-los objetos por si solos no son colocados en la pila.
3. **El heap.** Esta es una especie de piscina de memoria de propósito general (también en la memoria RAM) donde viven todos los objetos. Lo bueno del heap es que, a diferencia de la pila, el compilador no necesita saber cuanto espacio necesita en el heap para algo o cuanto tiempo va a permanecer ahí. Donde sea que necesitemos crear un objeto, simplemente se escribirá el código utilizando **new** y el espacio

es asignado en el heap cuando el código es ejecutado. Claro que hay un precio que pagar por esta flexibilidad: toma mas tiempo asignar memoria en el heap que hacerlo en la pila (esto si se *pudiera* crear objetos en la pila en Java, como se puede hacer en C++).

4. **Almacenamiento estático.** “Estático” es utilizado en el sentido de “en una posición fija” (a pesar de que es también en la RAM). El almacenamiento contiene datos que están disponibles en cualquier momento de la ejecución de un programa. Se puede utilizar la palabra clave **static** para especificar un elemento particular que es estático, pero los objetos en Java nunca son colocados en esta zona de almacenamiento.
5. **Almacenamiento constante.** Los valores constantes son colocados directamente en el código del programa donde están a salvo de que alguien los cambie. A veces las constantes son acordonadas por un anillo de seguridad por si mismos así que opcionalmente pueden ser colocados en la memoria de solo lectura (ROM).
6. **Almacenamiento no-RAM.** Si los datos se encuentran completamente fuera del programa, pueden existir cuando el programa no esta ejecutándose. Los dos ejemplos primarios de estos son los *objetos en flujo* (*streamed objects*), los cuales son transformados en flujos de bytes, generalmente para ser enviados a otra máquina, y los objetos persistentes que son colocados en disco y no se borran cuando el programa es terminado. El truco con estos tipos de almacenamiento es convertir los objetos en algo que pueda existir en otro medio, y a pesar de eso puedan ser resucitados en un objeto basado en RAM regular cuando sea necesario. Java proporciona soporte para *persistencia liviana* (*lightweight persistence*) y futuras versiones pueden proporcionar soluciones completas de persistencia.

## Casos especiales: tipos primitivos

Este es el grupo de tipos que obtienen un tratamiento especial; podemos pensar en esto como tipos “primitivos” que se utilizarán a menudo en los programas. La razón para este tratamiento especial es que crear un objeto con **new**-especialmente si es pequeño, o una simple variable-no es muy eficiente dado que **new** coloca los objetos en el heap. Para esos tipos Java retrocede a una estrategia de C y C++. Esto es, en lugar de crear la variable utilizando **new**, una variable “automática” es creada que *no es* una

*referencia*. Esta variable almacena un valor, y es colocada en la pila que es mucho mas eficiente.

Java determina el tamaño de cada tipo primitivo. Estos tamaños no cambian de una a otra arquitectura como la mayoría de los lenguajes. Estos tamaños invariables es una razón para que los programas Java sean portátiles.

Tipo primitivo	Tamaño	Mínimo	Máximo	Tipo de envoltura
<b>boolean</b>	-	-	-	<b>Boolean</b>
<b>char</b>	16 bits	Unicode o	Unicode $2^{16-1}$	<b>Carácter</b>
<b>byte</b>	8 bits	-128	+127	<b>Byte</b>
<b>short</b>	16 bits	$-2^{15}$	$+2^{15-1}$	<b>Short</b>
<b>int</b>	32 bits	$-2^{31}$	$+2^{31-1}$	<b>Integer</b>
<b>long</b>	64 bits	$-2^{63}$	$+2^{63-1}$	<b>Long</b>
<b>float</b>	32 bits	IEEE <sub>754</sub>	IEEE <sub>754</sub>	<b>Float</b>
<b>double</b>	64 bits	IEEE <sub>754</sub>	IEEE <sub>754</sub>	<b>Double</b>
<b>void</b>	-	-	-	<b>Void</b>

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

El tamaño del tipo **boolean** no es esta explícitamente definido; solo esta capacitado para tomar valores literales **true** o **false**.

Los tipos de datos primitivos también tienen clases “envolturas” (wrapper). Que significa que si quiere crear un tipo no primitivo en el heap para representar un tipo primitivo, puede utilizar la envoltura asociada. Por ejemplo:

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

O también podemos utilizar:

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

Las razones para hacer esto serán mostradas en un capítulo mas adelante.

## Números de precisión alta

Java incluye dos clases para representar aritmética de precisión alta:

**BigInteger** y **BigDecimal**. A pesar de que se aproximan a ajustar con alguna de las categorías de clase de “envoltura”, ninguna tiene una primitiva análoga.

Ambas clases tienen métodos que proporcionan analogías para las operaciones que se ejecutan con los tipos primitivos. Esto es, si se quiere hacer algo con un **BigInteger** o un **BigDecimal** se puede hacer también con **int** o **float**, simplemente se utilizan llamadas a métodos en lugar de

operaciones. También, dado que son mas evolucionadas, las operaciones serán mas lentas. Se esta cambiando velocidad por exactitud.

**BigInteger** soporta enteros de precisión arbitraria. Esto significa que se puede representar valores enteros de cualquier tamaño sin perder información durante las operaciones.

**BigDecimal** se utiliza para números de punto fijo de precisión arbitraria; se pueden utilizar estos para cálculos monetarios exactos por ejemplo.

Consulte la documentación en línea por detalles acerca de los constructores y los métodos que se pueden llamar para estas dos clases.

## Arreglos en Java

Virtuamente todos los lenguajes de programación soportan arreglos. Usar arreglos en C o en C++ es peligroso porque aquellos esos arreglos son simplemente bloques de memoria. Si un programa accede a un arreglo fuera de este bloque o utiliza la memoria antes de la inicialización (los errores mas comunes de programación) se obtienen resultados impredecibles.

Una de las principales metas de java es la seguridad, así es que muchos de los problemas que son flagelos de los programadores de C y C++ no se repiten en Java. Un arreglo en Java esta garantizado a ser iniciado y no pueden ser accedidos fuera de ese rango. El chequeo de rango incluye un pequeño precio a pagar de memoria en cada arreglo así como la verificación del índice en tiempo de ejecución, pero asumiendo que es seguro e incrementa la productividad es un precio justo.

Cuando crea un arreglo de objetos, esta creando en realidad un arreglo de referencias, y cada una de estas referencias es automáticamente inicializada a un valor especial que esta indicado con la palabra clave **null**. cuando Java ve **null** reconoce que la referencia en cuestión no esta apuntando a un objeto. Se debe asignar un objeto a cada referencia antes de utilizarlo, y si intentamos utilizar una referencia **null**, el problema será reportado en tiempo de ejecución. Estos, errores típicos de arreglos son prevenidos en Java.

También se pueden crear arreglos de primitivas. En este caso, también el compilador garantiza la inicialización agregando ceros en la memoria de ese arreglo. Todo será explicado en detalle en capítulos siguientes.

# No se necesita nunca destruir un objeto

En muchos lenguajes de programación, el concepto de tiempo de vida de una variable ocupa una significativa parte del esfuerzo de programación. ¿Cuanto tiempo estuvo la última variable? ¿Si se supone que hay que destruirla, cuando deberías hacerlo? Las confusiones por tiempos de vida de variable acumulan una gran cantidad de errores, y esta sección muestra como Java simplifica el tema del trabajo de limpieza.

## Alcance

Muchos lenguajes procesales tienen el concepto de *alcance (scope)*. Este determina la visibilidad y el tiempo de vida de los nombres definidos en determinado alcance. En C, C++ y Java, el alcance está determinado por las llaves { y }. Así es que dado un ejemplo:

```
{  
    int x = 12;  
    /* solo esta disponible x */  
    {  
        int q = 96;  
        /* x y q están disponibles */  
    }  
    /* solo x esta disponible */  
    /* q esta "fuera de alcance" */  
}
```

Una variable definida sin alcance esta disponible solo en el final de ese alcance.

La sangría hace el código mas fácil de leer. Dado que Java es un lenguaje de forma libre, los espacios extra, tabulaciones y retornos de carro no afectan el problema resultante.

Se ve que *no es posible* hacer lo siguiente, aunque sea legal en C y en C++:

```
{  
    int x = 12;  
    {  
        int x = 96; /* ilegal */  
    }  
}
```

El compilador indica que la variable x ya esta definida. Esta habilidad que tienen C y C++ de “esconder” una variable en un largo alcance no esta permitida porque los diseñadores de Java piensan que da lugar a programas confusos.

## Alcance de los objetos

Los objetos de java no tienen el mismo tiempo de vida que las primitivas. Cuando se crea un objeto utilizando **new**, este queda colgado pasado el final del alcance. De esta manera si se usa:

```
{  
    String s = new String("una cadena");  
} /* final del alcance */
```

La referencia a **s** deja de existir en el final del alcance. Sin embargo, el objeto **String** al que **s** fue apuntado sigue ocupando memoria. En este pequeño código, no hay forma de acceder al objeto porque la única referencia al objeto está fuera del alcance. En capítulos siguientes se verá como la referencia a el objeto puede ser pasada y duplicada en el transcurso del programa.

Nos olvidamos de eso porque los objetos creados con **new** siguen presentes tanto como se quiera, la totalidad de los problemas escabrosos de programar C++ simplemente se desvanecen en Java. Los problemas mas grandes parecen ocurrir en C++ porque no se tiene ayuda del lenguaje acerca de si los objetos están disponibles cuando se necesitan. Y mas importante, en C++ se debe asegurar que destruye los objetos cuando se ha terminado con ellos.

Esto plantea una pregunta interesante. ¿Si Java deja los objetos dando vueltas por ahí, que evita que llenen la memoria y detengan el programa? Este es exactamente el tipo de problema que sucede en C++. Aquí es donde un poco de magia se sucede. Java tiene un *recolector de basura(garbage collector)*, que observa todos los objetos cuando son creados con **new** y resuelve cuales no son referenciados mas. entonces libera la memoria de esos objetos, para que pueda ser utilizada por nuevos objetos. Esto significa que no es necesario preocuparse por reclamar memoria. simplemente se crean objetos, y cuando no se utilizan mas se van a ir solos. Esto elimina cierto tipo de problema: el llamado “filtración de memoria” (*memory leak*), en donde el programador olvida liberar memoria.

## Creando nuevos tipos de datos: clases

¿Si todo es un objeto, que determina como un objeto se verá y comportará? Dicho de otra forma; ¿Que establece el *tipo* de un objeto? Siempre se espera una palabra clave llamada “tipo”, y ciertamente eso tiene sentido. Históricamente sin embargo, muchos lenguajes orientados a objetos han utilizado la palabra clave **class** que significa “voy a indicarte como se verá un

nuevo tipo de objeto". La palabra clave **class** (que es tan común que no le voy a colocar mas negrita en este libro) esta seguida por el nombre del nuevo tipo. Por ejemplo:

```
| class NombreDelTipoA { /* el cuerpo de la clase va aquí */ }
```

Esto introduce un nuevo tipo, así es que ahora se puede crear un objeto de este tipo utilizando **new**:

```
| NombreDelTipoA a = new NombreDelTipoA ();
```

En **AtypeName**, el cuerpo de la clase consiste solo en un comentario (el comienzo, las llaves y lo que hay adentro, que será discutido mas adelante en este capítulo), así que no es mucho lo que se puede hacer con esto. De hecho, no se le puede decir que haga mucho de nada (esto es, no se le puede enviar ningún mensaje interesante) hasta que no se definan algunos métodos para esto.

## Campos y métodos

Cuando se define una clase (y todo lo que se hace en Java es definir clases, crear objetos con estas clases, y enviar mensajes a aquellos objetos), se pueden colocar dos tipos de elementos en las clases: datos miembro (a veces llamados *campos*), y funciones miembro (llamadas típicamente *métodos*). Un dato miembro es un objeto de cualquier tipo con el que nos comunicamos por medio de su referencia. Pueden ser uno de los tipos primitivos (que no son referencias). Si es una referencia a un objeto, debe inicializarse la referencia para poder conectarlo a un objeto actual (utilizando **new**, como se ha visto ya) en una función especial llamada *constructor* (descrito totalmente en el Capítulo 4). Si es un tipo primitivo podemos iniciararlo directamente en el punto de la definición de la clase (como se ha visto anteriormente, las referencias pueden también ser inicializadas en el momento de la definición).

Cada objeto tiene su propio espacio para sus datos miembro; los datos miembros no son compartidos entre objetos. Aquí hay un ejemplo de una clase con algunos datos miembros.

```
| class DataOnly {
|     int i;
|     float f;
|     boolean b;
| }
```

Esta clase no hace nada, pero podemos crear un objeto:

```
| DataOnly d = new DataOnly();
```

Se pueden asignar valores a los miembros datos, pero primero debe conocer como se hace referencia a un miembro de un objeto. Este se logra comenzando con el nombre del objeto referenciado, seguido por un periodo (punto), seguido por el nombre del miembro adentro del objeto:

```
| referenciaAOBJETO.miembro  
Por ejemplo:
```

```
| d.i = 47;  
| d.f = 1.1f;  
| d.b = false;
```

Es también posible que un objeto pueda tener adentro otros objetos que contienen datos que se quieran modificar. Para esto, solo mantenga “conectado los puntos”. Por ejemplo:

```
| miAvion.tanqueIzquierdo.capacidad = 100;
```

La clase **DataOnly** no puede hacer mucho, solo guardar datos, porque no tiene funciones miembros (métodos). Para entender como trabajan estos, se deberá primero entender los *argumentos* y *valores de retorno* que serán explicados mas adelante.

## Valores por defecto de miembros primitivos

Cuando un tipo de datos es miembro de una clase, es garantido obtener un valor por defecto si no se ha inicializado:

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 por defecto garantizados por Java cuando una variable es utilizada como *miembro de una clase*. Esto asegura que las variables miembro de tipos primitivos serán siempre inicializadas (algo que C++ no hace), reduciendo una fuente de errores. Sin embargo, estos valores iniciales pueden no ser los correctos o legales para el programa que se está escribiendo. Es mejor explícitamente inicializar las variables.

Esto garantiza no aplicar a variables “locales”-aquellas que no son campos de una clase. De esta manera, si dentro de una definición de función tenemos:

```
| int x;
```

Entonces, x será algún valor arbitrario (como en C y C++); no será automáticamente inicializada a cero. Es nuestra responsabilidad asignar un valor apropiado antes de utilizar la variable x. Si se olvida, Java definitivamente mejora a C++; se tendrá un error de compilación indicando que la variable puede no haber sido iniciada (Muchos compiladores C++ advertirán la presencia de variables sin iniciar, pero en Java estos son errores).

## Métodos, argumentos, y valores de retorno

Hasta ahora el término *función* ha sido utilizado para describir una subrutina nombrada. El término que es mas comúnmente utilizado en Java es *método*, como “una forma de hacer algo”. Si deseamos, podemos continuar pensando en términos de funciones. Es realmente una diferencia sintáctica, pero desde ahora en adelante “método” será utilizado en este libro preferiblemente antes que “función”.

Los métodos en java determinan los mensajes que un objeto puede recibir. En esta sección aprenderemos que tan simple es definir un método.

Las partes fundamentales de un método son, el nombre, los argumentos, el tipo retornado y el cuerpo. He aquí la forma básica:

```
tipoRetornado nombreDelMetodo ( /* lista de argumentos */ ) {  
    /* Cuerpo del método */  
}
```

El tipo retornado es el tipo del valor que el método retorna luego de llamarlo. La lista de argumentos entrega a el método los tipos y nombres de la información que se pasa al método. El nombre del método y la lista de argumentos juntos excepcionalmente identifican el método.

Los métodos en Java pueden ser creados solo como parte de una clase. Un método puede ser llamado solo por un objeto<sup>2</sup>, y ese objeto puede ser capaz de ejecutar los métodos llamados. Si se intenta llamar el método equivocado para un objeto, se obtendrá un mensaje de error en tiempo de compilación. Llamamos un método de un objeto nombrando el objeto seguido de un período (punto), seguido por el nombre del método y su lista de argumentos, de la siguiente forma:

**nombreDeObjeto.nombreDeMetodo(arg1, arg2, arg3)**. Por ejemplo, supongamos que tenemos un método **f()** que no tiene argumentos y retorna

---

<sup>2</sup> Los métodos estáticos (**static**), que aprenderemos mas adelante, pueden ser llamados por la clase, sin un objeto.

un valor del tipo **int**. Entonces, si tenemos un objeto llamado **a** para cada **f()** que pueda ser llamado, colocaremos esto:

```
| int x = a.f();  
| El tipo retornado debe ser compatible con el tipo de x.
```

Este acto de llamar un método es comúnmente llamado *enviar un mensaje a un objeto*. En el siguiente, el mensaje es **f()** y el objeto es **a**. La POO es también resumida como simplemente “enviar mensajes a objetos”.

## La lista de argumentos

La lista de argumentos del método especifica que información pasaremos a el método. Como podemos adivinar, esta información-al igual que cualquiera en Java-toma la forma de objetos. Así es que, lo que especifiquemos en la lista de argumentos son los tipos de los objetos que se pasarán y el nombre que utilizará cada uno. Como en cualquier situación en Java donde parece ser que se manejan objetos por todos lados, estamos pasando actualmente referencias<sup>3</sup>. El tipo de la referencia debe ser el correcto, sin embargo. Si el argumento supuestamente es un **String**, lo que debemos pasar debe ser una cadena.

Considerando un método que toma un **String** como argumento. He aquí la definición, que debe ser colocada con una definición de clase para ser compilada:

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

Este método nos muestra cuantos bytes son requeridos para almacenar la información en un **String** particular. (Cada **char** en un **String** es de 16 bits, o dos bytes de largo para soportar caracteres Unicode). El argumento de este tipo es **String** y es llamado **s**. Una vez que **s** es pasado al método, se puede tratar simplemente como otro objeto (Podemos enviarle mensajes). Aquí, el método **length()** es llamado, que es uno de los métodos de la clase **Strings**; este retorna el numero de caracteres en una cadena.

Como podemos ver el uso de la palabra clave **return**, hace dos cosas. Primero, significa “dejando el método, he terminado”. Segundo, si el método produce un valor, el valor es colocado inmediatamente después de la instrucción **return**. En este caso, el valor returned es producido por la evaluación de la expresión **s.length() \* 2**.

---

<sup>3</sup> Con la usual excepción de los anteriormente “especiales” tipos de datos **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**. En general, si bien pasamos objetos, lo que realmente tratamos de decir es que pasamos referencias a objetos.

Podemos retornar cualquier tipo que deseemos, pero si no se debe retornar nada, se debe hacer indicando que el método retorna **void**. He aquí algunos ejemplos:

```
boolean bandera() { return true; }
float baseLogaritmicaNatural() { return 2.718f; }
void nada() { return; }
void nada2() {}
```

Cuando el tipo de retorno es **void**, la palabra clave **return** solo es utilizada para salir del método, y es por consiguiente innecesaria cuando se alcanza el final del método. Se puede retornar de un método en cualquier punto, pero si el tipo de retorno no es **void** el compilador forzará (con mensajes de error) a retornar el tipo apropiado independientemente de donde retorne.

En este punto, se puede ver como un programa simplemente es una acumulación de objetos con métodos que toman otros objetos como argumento y se envían mensajes a esos otros objetos. Esto es efectivamente mucho de lo que lo que sigue, pero en el siguiente capítulo veremos como hacer el trabajo de bajo nivel tomando decisiones con un método. Para este capítulo, enviar mensajes es suficiente.

## Armando un programa Java

Hay otros muchos temas que debemos entender antes de entender el primer programa Java.

### Visibilidad de nombres

Un problema en cualquier lenguaje de programación, es el control de nombres. Si usamos un nombre en un módulo del programa, y otro programa usa el mismo nombre en otro módulo, como distinguimos un nombre de otro y prevenir que los nombres entren en “conflicto”? En C esto es un problema porque un programa es un mar inmanejable de nombres. Las clases en C++ (de donde están basadas las clases de Java) anidan funciones con clases así es que no pueden entrar en conflicto con nombres anidados en otras clases. Sin embargo, C++ sigue permitiendo datos y funciones globales, así es que los conflictos siguen siendo posibles. Para solucionar este problema, C++ introduce los *espacios de nombres* utilizando palabras clave.

Java es capaz de evitar todo esto con una estrategia nueva. Para producir nombres sin ambigüedad de una librería, el que fija las condiciones no es diferente a un nombre de dominio de la Internet. De hecho, los creadores de Java quieren que se utilice los nombres de dominios de la Internet en reversa dado que es garantido que sean únicos. Dado que mi dominio es

**BruceEckel.com**, mi librería de utilidades de vulnerabilidades puede ser llamada **com.bruceeckel.utilidades.vulnerabilidades**. Luego de que se invierte el nombre de dominio, los puntos representan subdirectorios.

En Java 1.0 y Java 1.1 las extensiones de dominio **com, edu, org, net**, etc., estaban capitalizados por convención, así que la librería aparecería:

**COM.bruceeckel.utilidades.vulnerabilidades**. Parte del camino a través del desarrollo de Java 2, sin embargo, fue descubrir que esto causaba problemas, y la totalidad del nombre del paquete es en minúsculas.

Este mecanismo, da a entender que todos sus ficheros automáticamente están en sus propios espacios de nombres, y cada clase con un fichero debe ser un único identificador. Así no necesitamos aprender herramientas especiales de lenguaje para solucionar este problema-el lenguaje se encarga de esto por nosotros.

## Utilizando otros componentes

En el momento que se quiera usar una clase predefinida en un programa, el compilador debe saber donde encontrarla. Por su puesto, la clase debe existir en el mismo fichero de código fuente de donde es llamado. En ese caso, simplemente utilizamos la clase-aun si la clase no esta definido hasta tarde en el fichero. Java elimina el problema de la “referencia anticipada” así es que no se necesita pensar en ello.

¿Que sucede con las clases que están en otros ficheros? Debemos suponer que el compilador debería ser lo suficientemente inteligente para simplemente ir y encontrarla, pero eso es un problema. Imaginemos que se quiere utilizar una clase de un nombre particular, pero mas de una definición de esa clase existe (presumiblemente son definiciones diferentes). O peor, imaginemos que se esta escribiendo un programa y cuando se esta armando agregamos una nueva clase en la librería que entra en conflicto con el nombre de una clase ya existente.

Para solucionar este problema, debe eliminar todas las ambigüedades potenciales. Esto es logrado indicándole a el compilador de Java exactamente cuales clases queremos usar mediante la palabra clave **import**. **import** le indica al compilador que traiga de un *paquete*, cual es la librería de clases (en otros lenguajes, una librería puede consistir en funciones y datos de la misma forma que clases, pero recuerde que todo el código en Java debe ser escrito adentro de una clase).

La mayoría del tiempo se estará utilizando componentes de la librería estándar de Java que vienen con su compilador. Con ellas, no hay necesidad de preocuparnos acerca del largo, nombres de dominios inversos; simplemente se indica, por ejemplo:

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

para decirle al compilador que se quiere utilizar la clase de Java **ArrayList**. Sin embargo, **util** contiene muchas clases y deberíamos utilizar muchas de ellas sin declarar explícitamente todas ellas. Esto es fácil de lograr usando '\*' para indicar un comodín:

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

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

## La palabra clave **static**

Por lo general, cuando creamos una clase estamos describiendo que objetos de esa se ven y como se comportarán. No tenemos nada hasta que se crea un objeto de esa clase con **new**, y en ese punto el espacio de almacenamiento de los datos es creado y los métodos se hacen disponibles.

Pero hay dos situaciones en la que ese acercamiento no es suficiente. Una es si se quiere tener un solo lugar para almacenar cierto tipo de dato, sin importar cuantos objetos son creados, o tal vez aun si el objeto no ha sido creado. La otra es si se necesita un método que no esta asociado con un objeto de la clase. Esto es, se necesita un método que se pueda llamar aun si no se han creado objetos. Podemos lograr estas dos cosas con la palabra clave **static**. Cuando se dice que algo es estático, significa que el dato o método no está atado a una instancia particular de un objeto de una clase. Así que si nunca se creo un objeto de esa clase podemos llamar un método estático o acceder a una parte de un dato estático. Con datos y métodos no estáticos se debe crear un objeto y utilizarlo para acceder a los datos o métodos, dado que con los métodos y datos no estáticos debemos conocer el objeto en particular con el que se va a trabajar. Claro, que dado que los métodos estáticos no necesitan objetos para ser creados antes de utilizarlos, no pueden *directamente* acceder a miembros o métodos no estáticos simplemente llamando a otros miembros sin referenciar a un nombre de objeto (dado que los miembros no estáticos y métodos deben ser ligados a un objeto particular).

Algunos lenguajes orientados a objetos utilizan los términos *datos de la clase* y *métodos de la clase*, dando a entender que los datos y métodos existen para la clase entera como un todo, y no para objetos particulares de la clase. A veces en la literatura de Java se utiliza esos términos también.

Para hacer un dato miembro o método estático, debemos simplemente colocar la palabra clave **static** antes de la definición. Por ejemplo, lo siguiente produce un miembro de dato estático y lo inicializa:

```
class pruebaEstatica {
```

```
    static int i = 47;  
}
```

Ahora, si creamos dos objetos **pruebaEstatica**, solo tendrán un lugar donde almacenan **PruebaEstatica.i**. Ambos objetos compartirán el mismo **i**. Consideremos:

```
| PruebaEstatica st1 = new PruebaEstatica();  
| PruebaEstatica st2 = new PruebaEstatica();
```

En este punto, **st1.i** y **st2.i** tienen el mismo valor de 47 dado que se refiere a la misma dirección de memoria.

Hay dos formas de referirse a una variable estática. Como se indica mas arriba, se puede nombrar mediante el objeto, indicando por ejemplo, **st2.i**. También podemos referirnos a ellos con el nombre de la clase, algo que no se puede hacer con un miembro no estático (Esta es la forma preferible de referirse a una variable estática dado que enfatiza su naturaleza estática).

```
| PruebaEstatica.i++;
```

El operador **++** incrementa la variable. En este momento, ambos, **st1.i** y **st2.i** tienen el valor **48**.

Lógica similar se aplica a métodos estáticos. Puede referirse a un método estático a través de un objeto como con cualquier método, o con la sintaxis especial adicional **nombreDeClase.metodo()**. Si se define un método estático de una forma similar:

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

Se puede ver que el método de **DiversionEstatica** llamado **incr()** incrementa el dato estático **i**. Se puede llamar a **incr()** de la forma típica, a través de un objeto.

```
| DiversionEstatica sf = new DiversionEstatica();  
| sf.incr();
```

O, como **incr()** es un método estático, podemos llamarlo directamente a través de su clase:

```
DiversionEstatica.incr();
```

Dado que **static**, fue aplicado a un dato miembro, definitivamente cambia la forma en que el dato es creado (uno para cada clase contra el no estático uno para cada objeto), cuando es aplicado a un método no es tan dramático. Un uso importante de **static** para métodos es permitir que se llame a un método sin crear un objeto. Esto es esencial, como veremos, en la definición del método **main()** que es el punto de entrada para ejecutar una aplicación.

Como cualquier método, un método estático puede crear o utilizar nombres de objetos de su tipo, así es que muchas veces utilizado como “pastor” para bloquear instancias de su propio tipo.

# Su primer programa Java

Finalmente, he aquí el programa<sup>4</sup>. Este comienza desplegando una cadena, y luego, la fecha, utilizando la clase **Date** de la librería estándar de Java. Se puede ver que un estilo de comentario adicional es introducido aquí: el ‘//’, que es un comentario hasta el final de la línea:

```
// HolaFecha.java
import java.util.*;
public class HolaFecha {
    public static void main(String[] args) {
        System.out.println("Hola, hoy es: ");
        System.out.println(new Date());
    }
}
```

Al comienzo de cada fichero de programa se deberá colocar el comando **import** para traer cualquier clase extra que se necesite para el código en ese fichero.

Note que dice “extra”; esto es porque ciertas librerías de clases son automáticamente traídas en cada fichero Java: **java.lang**. En un navegador se puede ver la documentación de Sun (si se han bajado los programas de [java.sun.com](http://java.sun.com), de otra forma deberá instalarse la documentación). Si se mira la lista de paquetes, se verá todas las diferentes librerías de clases que vienen con Java. Si se selecciona **java.lang**, traerá una lista de clases que son parte de la librería. Dado que **java.lang** está implícitamente incluida en cada fichero de código, estas clases están disponibles automáticamente. No hay una clase **Date** listada en **java.lang**, lo que significa que debemos importar otra librería para usarla. Si se conoce la librería en donde se encuentra una clase particular, se deberá ver todas las clases. Se puede seleccionar “Tree” en la documentación de Java. Ahora se pueden encontrar cada clase por separado de las que vienen con Java. Entonces se puede buscar **Date** con la función “Find”. Se verá que está listada como **java.util.Date**, lo que indica que se encuentra en la librería util y se debe colocar **import java.util.\*** para poder utilizar **Date**.

---

<sup>4</sup> Algunos ambientes de programación destellarán el programa en la pantalla y lo cerrarán sin que exista la posibilidad de ver los resultados. Podemos colocar este pequeño código en el final del **main()** para introducir una pausa en la salida.

```
try {
    System.in.read();
} catch (Exception e) {}
```

Esto detendrá la salida hasta que se presione “Entrada” (o algunas otras teclas). Este código involucra conceptos que no serán introducidos hasta mucho mas adelante en este libro, así es que no intente entenderlo ahora, pero hará la trampa.

Si se regresa al principio, y se selecciona **java.lang** y luego **System**, se pobra ver que la clase **System** tiene varios campos, y si se selecciona **out** se descubrirá que hay un objeto estático **PrintStream**. Dado que es estático no se necesita crear nada. El objeto **out** esta siempre y simplemente se puede utilizar. Lo que se puede hacer con este objeto out esta determinado por su tipo: un **PrintStream**. Convenientemente, **PrintStream** es mostrado en la descripción como un vínculo, de esta forma se puede ver la lista de todos los métodos que se pueden llamar para **PrintStream**. Hay muchos de ellos que son cubiertos mas adelante en este libro. Por ahora estamos interesados en **println()**, que en efecto significa “imprima lo que le estoy dando en la consola y termine con un final de línea”. Esto, en cualquier programa que se escriba podemos hacer **System.out.println("cosas")** en cualquier lugar para imprimir algo en la consola.

El nombre de la clase es el mismo que el del fichero. Cuando se crea un programa independiente como este, una de las clases en el fichero debe tener el mismo nombre que el fichero (El compilador se queja si no lo hacemos de esta forma). esta clase debe contener un método llamado **main()** con la siguiente firma:

```
| public static void main(String[] args) {  
|     La palabra clave public significa que el método esta disponible para el  
|     mundo exterior (descrito en detalle en el capítulo 5). El argumento para  
|     main() es un arreglo de objetos String. Los args no serán utilizados en este  
|     programa, pero el compilador insiste en que ellos estén allí porque sostienen  
|     los argumentos invocados en la línea de comandos.
```

La línea que imprime la fecha es muy interesante:

```
| System.out.println(new Date());  
| Considerando el argumento: un objeto Date es creado justo para enviar el  
|     valor a println(). Tan pronto como este comando es terminado, el objeto  
|     Date es innecesario, y el recolector de basura puede venir y llevárselo en  
|     cualquier momento. No hay necesidad de preocuparse por ello.
```

## Compilando y ejecutando

Para compilar y ejecutar este programa, y todos los demás programas en este libro, se debe tener un ambiente de programación Java. Hay algunos ambientes desarrollados por terceros, pero este libro asumirá que estamos utilizando el JDK de Sun, que es gratis. Si se esta utilizando otro sistema de desarrollo, se necesitará ver la documentación para el sistema para determinar como compilar y ejecutar programas.

En la Internet, en [java.sun.com](http://java.sun.com) se puede encontrar información y enlaces que puede dirigirnos a través del proceso de bajar e instalar JDK para una plataforma en particular.

Una vez que JDK esta instalado y configurada la ruta podremos encontrar **javac** y **java**, bajemos y desempaquetemos el código fuente de este libro (se puede encontrar en el CD ROM que viene con este libro, o en [www.BruceEckel.com](http://www.BruceEckel.com)). Esto creará un subdirectorio para cada capítulo de este libro. Podemos movernos a el subdirectorio co2 y escribir:

```
| javac HelloDate.java
```

Este comando puede no producir respuesta. Si se tiene algún tipo de mensaje de error quiere decir que no ha instalado JDK correctamente y se necesita investigar ese problema.

Por otro lado, si solo se obtiene el indicador de línea de comando nuevamente, se podrá escribir:

```
| java HelloDate
```

y tendrá el mensaje y la fecha como salida.

Este es el proceso con el que se compila y ejecuta cada uno de los programas de este libro. Sin embargo, podrá verse que el código fuente de este libro también tiene un fichero llamado **makefile** en cada capítulo, y este contiene comandos “make” para armar automáticamente los ficheros para ese capítulo. En la página del libreto en [www.BruceEckel.com](http://www.BruceEckel.com) hay detalles de como utilizar los makefiles.

## Comentarios y documentación incluida

Hay dos tipos de comentarios en Java. El primero es el tradicional comentario estilo C que fue heredado por C++. Estos comentarios comienzan con un /\* y continúan, posiblemente a través de muchas líneas, hasta un \*/. Nótese que muchos programadores comienzan cada línea de un comentario continuo con un \*, así que se podrá ver:

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

Recordemos, sin embargo que todo lo que está entre el /\* y el \*/ es ignorado, así es que no hay diferencia en decir:

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

La segunda forma de comentar llega con C++. Estos son los comentarios de una sola línea, estos comienzan con un // y continúan hasta el final de la línea. Este tipo de comentario es conveniente y comúnmente utilizado porque es fácil. No se necesita cazar a través del teclado en búsqueda de un /

y luego un \* (en lugar de eso, se presiona la misma tecla dos veces), y no se necesita cerrar el comentario. Así es que también se podrá ver:

```
| // Este es un comentario de una sola linea.
```

## Comentarios de documentación

Una de las partes de Java mas consideradas, es que los diseñadores no consideran que escribir código sea la única parte importante de la actividad—piensan también en documentarlo. Posiblemente el problema mas grande con documentar código es mantenerlo documentado. Si la documentación y el código están separados, comienza a molestar cambiar la documentación cada vez que cambiamos el código. La solución es simple: unir el código con la documentación. La manera mas fácil de hacer esto es colocar todo en el mismo fichero. Para completar esta imagen, sin embargo, se necesita una sintaxis especial para indicar documentación especial y herramientas para extraer esos comentarios y colocarlos en una forma útil. Esto es lo que Java ha hecho.

La herramienta para extraer los comentarios es llamada *javadoc*. Utiliza tecnología del compilador de Java para buscar etiquetas de comentarios especiales que se colocan en los programas. No solo extrae la información indicada por esas etiquetas, también extrae los nombres de las clases, los métodos que contienen los comentarios. De esta forma se puede olvidar del poco trabajo para generar una documentación de programa decente.

La salida de javadoc es un fichero HTML que se puede ver con un navegador Web. Esta herramienta permite crear y mantener un único fichero y automáticamente generar documentación útil. Dado que javadoc tiene una norma para crear documentación, y es tan fácil como cualquiera pueda esperar o incluso exigir de una documentación con todas las librerías Java.

## Sintaxis

Todos los comandos se suceden con comentarios `/**`. Los comentarios terminan con `*/` como es usual. Hay dos formas principales de utilizar javadoc: insertando HTML o utilizando “etiquetas de documentación” (Doc tags). Las etiquetas de documentación son comandos que comienzan con un ‘@’ y son colocadas al comienzo de una línea de comentario (Un ‘\*’ inicial, sin embargo, es ignorado).

Hay tres “tipos” de comentarios de documentación, que corresponden con el elementos del comentario que preceden: clase, variable o método. Esto es, un comentario de una clase aparece a la derecha luego de una definición de clase; un comentario de variable aparece inmediatamente antes que la

definición de la variable, y un comentario de un método aparece inmediatamente antes de la definición del método. Como el ejemplo simple:

```
/** Comentario de clase */
public class docTest {
    /** Comentario de variable */
    public int i;
    /** Comentario de método */
    public void f() {}
}
```

Note que javadoc procesará los comentarios de documentación solo para los miembros **public** y **protected**. Los comentarios para miembros **private** y “frendly” (vea el capítulo 5) son ignorados y no se obtendrá ninguna salida (sin embargo, se puede utilizar la bandera -private para incluir miembros private igual). Esto tiene sentido, dado que solo los miembros públicos y protegidos están disponibles fuera del fichero, que es la perspectiva del cliente programados. Sin embargo, todos los comentarios de las clases son incluidos en la salida.

La salida para el código siguiente es un fichero HTML que tiene el mismo formato que el resto de la documentación de java, así es que los usuarios estarán cómodos con el formato y fácilmente pueden navegar en sus clases. Sería muy bueno entrar el código anterior, enviarlo a través de javadoc y ver el fichero HTML para ver los resultados.

## HTML incrustado

Javadoc pasa comandos HTML hacia el documento HTML generado. Esto permite un uso completo de HTML; sin embargo el principal motivo es permitir formatear el código.

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

Podemos también usar HTML justo como se desearía en cualquier otro
documento para dar formato a texto regular en sus descripciones.
/**
 * Posemos <em>tambien</em> insertar una lista:
 * <ol>
 * <li> Primer item
 * <li> Segundo item
 * <li> Tercer item
 * </ol>
 */
```

Debe notarse que con los comentarios de documentación, los asteriscos en el comienzo de una línea son eliminados por javadoc, junto con los espacios iniciales. Javadoc le da un formato a todo lo este de acuerdo con la apariencia de la documentación estándar. No se pueden utilizar

encabezamientos como `<h1>` o `<hr>` en el código HTML insertado porque javadoc inserta sus propios encabezamientos y los demás interfieren con ellos.

Todos los tipos de comentarios de documentación-clases, variables y métodos-pueden soportar HTML incrustado.

## @see: refiriéndose a otras clases

Los tres tipos de comentarios de documentación (clases, variables y métodos) pueden contener etiquetas `@see`, que permite referirse a la documentación en otras clases. Javadoc generará vínculos HTML con las etiquetas `@see` hacia otra documentación. Las formas son:

```
| @see nombre-de-clase
| @see nombre-de-clase-totalmemente-calificado
| @see nombre-de-clase-totalmemente-calificado#nombre-de-metodo
Cada uno agrega un vínculo “See Also” en la documentación generada.
Javadoc no verificará los vínculos así que se deberá estar seguro de que son
válidos.
```

## Etiquetas de documentación de clase

Junto con el código HTML insertado y las referencias `@see`, la documentación de clase puede incluir etiquetas para información de la versión y el nombre del autor. La documentación de clase puede ser utilizada por *interfaces* (vea el capítulo 8).

### @version

Esta es la forma:

```
| @version información-de-versión
en donde la información-de-versión es cualquier información significativa
que se vea que se debe incluir. Cuando la bandera -version es colocada
junto con javadoc en la línea de comando, la información de la versión será
colocada especialmente en la documentación generada en HTML.
```

### @autor

Esta es la forma:

```
| @autor información-del-autor
```

en donde la **información-del-autor** es, presumiblemente su nombre, pero que también puede incluir su dirección de correo y otra información apropiada.

Cuando la bandera **-autor** es colocada en la línea de comandos al invocar javadoc la información del autor será colocada especialmente en la documentación generada.

Se pueden tener varias etiquetas de autor para una lista de autores, pero deben ser colocadas de forma consecutiva. Toda la información de los autores será puesta junta en un solo párrafo en el código HTML generado.

### @since

Esta etiqueta indica la versión de este código que da inicio a un aspecto particular. Se verá aparecer en la documentación HTML de Java para indicar que versión de JDK esta utilizando.

## Etiquetas de documentación de variables

La documentación variable puede incluir solo HTML insertado y referencias **@see**.

## Etiquetas de documentación de métodos

De la misma forma que la documentación insertada y las referencias **@see**, los métodos permiten etiquetas de documentación para los parámetros, valores de retorno y excepciones.

### @param

Esta es la forma:

```
| @param nombre-de-parámetro descripción  
| en donde el nombre-de-parámetro es el identificador en la lista de  
parámetros, y la descripción es el texto que puede continuar en las  
subsecuentes líneas. La descripción es considerada terminada cuando una  
nueva etiqueta de documentación es encontrada. Se pueden tener cualquier  
número de ellas, presumiblemente una por cada parámetro.
```

## **@return**

Esto es de la siguiente forma:

```
@return descripción
```

donde **descripción** le da un significado al valor retornado. Puede continuar el las líneas subsiguientes.

## **@throws**

Las excepciones son mostradas en el capítulo 10, pero brevemente son objetos que pueden ser “lanzados” afuera del método si ese método falla. A pesar de que solo una excepción puede emerger cuando se llama a un método, un método particular puede producir cualquier número de tipos diferentes de excepciones, las cuales, todas necesitan descripciones. Así es que la forma para la etiqueta de excepciones es:

```
| @throws nombre-de-clase-totamente-calificado descripción  
en donde nombre-de-clase-totamente-calificado da un nombre no ambiguo  
de una clase de excepción que está definida en alguna parte, y descripción  
(que puede continuarse en las subsiguientes líneas) indican el tipo  
particular de excepción que puede emerger de una llamada a un método.
```

## **@deprecated**

Esto es utilizado para marcar características que se encuentran reemplazadas por una característica mejorada. La etiqueta **deprecated** sugiere que no se use mas esa característica particular, dado que en un futuro será eliminada. Un método que es marcado como **@deprecated** causa que el compilador emita una alerta si es utilizada.

# Ejemplo de documentación

He aquí el primer programa Java nuevamente, esta vez con agregados de comentarios de documentación.

```
//: c02>HelloDate.java  
import java.util.*;  
/** El primer ejemplo de programa de Thinking in Java.  
 * Despliega una cadena y la fecha de hoy date.  
 * @author Bruce Eckel  
 * @author www.BruceEckel.com  
 * @version 2.0  
 */  
public class HelloDate {  
    /** Único punto de entrada a la clase y a la aplicación  
     * @param args arreglo de argumentos Strings  
     * @return No hay valor de retorno
```

```

 * @exception exceptions No se lanzan excepciones
 */
public static void main(String[] args) {
    System.out.println("Hola, hoy es: ");
    System.out.println(new Date());
}
} //:~

```

La primer línea del fichero usa mi propia técnica de poner un ‘:’ como marca especial para la línea de comentario que contiene el nombre de fichero fuente. Esa línea tiene la información de ruta a el fichero (en este caso, **co2** indica capítulo 2) seguido del nombre del fichero<sup>5</sup>. La última línea también termina con un comentario, y este indica el final del listado del código fuente, que permite automáticamente extraer del texto de este libro y verificarlo con un compilador.

## Estilo de código

Es estándar no oficial de Java es llevar a mayúscula la primer letra de un nombre de clase. Si el nombre de clase consiste en varia palabras, irán todas juntas (esto es, sin utilizar caracteres de subrayado para separar los nombres), y la primer letra de cada palabra es llevada a mayúscula, como aquí:

```

class TodosLosColoresDelArcoiris { // ...
Para al menos todo lo demás: métodos, campos (variables miembro), y
referencias a nombres de objetos, el estilo aceptado es el mismo que para las
clases excepto que la primer letra del identificador es minúscula. Por
ejemplo:

```

```

class TodosLosColoresDelArcoiris {
    int unEnteroRepresentandoColores;
    void cambieElTonoDeLosColores (int nuevoTono) {
        // ...
    }
    // ...
}

```

Por supuesto debemos recordar que el usuario debe también escribir todos esos nombres largos, y ser compasivo.

El código Java que se verá en las librerías Sun también siguen la ubicación de las llaves de apertura y cierre que se verá en este libro.

---

<sup>5</sup> Una herramienta que he creado usando Python ([www.Python.org](http://www.Python.org)) usa esta información para extraer los archivos de código, colocarlos en los subdirectorios apropiados y crear makefiles.

# Resumen

En este capítulo se vio suficiente de la programación Java para entender como escribir un simple programa, y se tiene una visión general del lenguaje y de algunas de sus ideas básicas. sin embargo, los ejemplos hasta ahora son todos del tipo “se hace esto, luego aquello, luego se hará algo mas”. Que si se quiere que el programa haga elecciones, como “si el resultado entregado es rojo, hago eso; si no, entonces hago otra cosa”? El soporte en Java para esta actividad fundamental de la programación será cubierta en el siguiente capítulo.

# Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Siguiendo el ejemplo **HelloDate.java** en este capítulo, cree un programa “Hola, mundo” que simplemente imprima esa frase. Necesitará solamente un método en esa clase (el “main” es el que se ejecuta cuando el programa se inicia). Recuerde que es estático y debe incluir la lista de argumentos aunque no la utilice. Compile el programa con **javac** y ejecútelo utilizando **java**. Si está utilizando un ambiente de desarrollo diferente que el JDK, aprenda como compilar y ejecutar programas en ese ambiente.
2. Encuentre los fragmentos de código que involucran **ATypeName** y colóquelos dentro de un programa que pueda compilar y ejecutar.
3. Coloque los fragmentos de código de **Data Only** dentro de un programa que pueda compilar y ejecutar.
4. Modifique el ejercicio 3 a fin de que los valores de los datos en **DataOnly** sean asignados e impresos en **main()**.
5. Escriba un programa que incluya y llame el método **storage()** definido como un fragmento de código en este capítulo.
6. Convierta los fragmentos de código de **StaticFun** en un programa que se pueda ejecutar.
7. Escriba un programa que imprima tres argumentos tomados de la línea de comando. Para hacer esto, necesitará indexar dentro del arreglo de **Strings** de la línea de comandos.
8. Convierta el ejemplo **AllTheColorOfTheRainbow** en un programa que compile y corra.

9. Encuentre el código de la segunda versión de **HelloDate.java**, que simplemente es un ejemplo de comentarios de documentación. Ejecute **javadoc** para el fichero y vea los resultados con su navegador.
10. Convierta docTest en un fichero que compile y corra, luego ejecute **javadoc** con él. Verifique la documentación resultante con su navegador.
11. Agregue una lista de ítem HTML a la documentación del ejercicio 10.
12. Tome el programa del ejercicio 1 y agréguele comentarios de documentación. Extracte estos comentarios de documentación en un fichero HTML utilizando **javadoc** y véalos con su navegador.

# 3: Controlando el flujo del programa

Como una criatura sensible, un programa debe manipular su mundo y hacer elecciones durante su ejecución.

En Java se manipulan objetos y datos utilizando operadores, y se realizan elecciones con instrucciones de control. Java fue heredado de C++, así es que muchos de estas instrucciones y operadores serán ser familiares a programadores C y C++. Java también ha agregado algunas mejoras y simplificaciones.

Si nos encontramos moviéndonos torpemente un poco en este capítulo, asegurémonos mediante el CD ROM multimedia dentro de este libro: *Thinking in C: Fundations for Java y C++* Contiene lecturas de audio, diapositivas, ejercicios y soluciones específicamente diseñados para darnos la velocidad con la sintaxis de C necesaria para aprender Java.

## Utilizando operadores Java

Un operador toma uno o mas argumentos para producir un nuevo valor. Los argumentos están en una forma diferente que las llamadas a métodos comunes, pero el efecto es el mismo. Nos deberemos sentir razonablemente confortable con los conceptos generales de operadores de nuestra experiencia previa programando. Suma (+), resta y menos unitario (-), multiplicación (\*), división (/) y asignación (=) igual que en cualquier lenguaje.

Todos los operadores producen un valor de sus operandos. Además, un operador puede cambiar el valor de un operando. Esto es llamado un *efecto secundaria*. El uso mas común de los operandos que modifican operandos es generar el efecto secundario para simplemente utilizarlas en operaciones sin efectos secundarios.

Casi todos los operadores trabajan solo con primitivas. La excepción es '=', '==' y '!=', que trabajan con todos los objetos (y son un punto de confusión para los objetos). además, la clase **String** puede utilizar '+' y '+='.

## Precedencia

La precedencia de los operadores define como una expresión evalúa cuando varios operadores están presentes. Java tiene reglas específicas que determinan el orden de evaluación. La forma más fácil de recordar esto es que la multiplicación y la división sucede antes que la suma y la resta. Los programadores a menudo olvidan las reglas de precedencia, así es que se suele utilizar paréntesis para crear un orden explícito de evaluación. Por ejemplo:

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

tiene diferente significado que la misma instrucción con un grupo particular de paréntesis:

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

## Asignación

La asignación se realiza con el operador `=`. Esto significa “tome el valor del lado derecho (a veces llamado el *rvalue*) y cópielo en el lado izquierdo (a veces llamado *lvalue*). Un rvalue es cualquier constante, variable o expresión que pueda producir un valor, y un lvalue que debe ser una variable distinta (Esto es, un valor físico para almacenar un valor). Por ejemplo, podemos asignar un valor constante a una variable (`A = 4;`), pero no podemos asignar nada a un valor constante-no puede ser un lvalue (no podemos decir `4 = A;`).

La asignación de primitivas es mas directo. Dado que la primitiva almacena el valor actual y no la referencia a un objeto, cuando asignamos primitivas copiamos el contenido de un lugar a otro. Por ejemplo, si decimos `A = B` para primitivas, el contenido de `B` será copiado en `A`. Si modificamos `A, B` naturalmente no es afectado por esta modificación. Como programador esta es la forma en la que se esperará en la mayoría de las situaciones.

Cuando asignamos objetos, sin embargo, las cosas cambian. En el momento en que manipulamos un objeto, lo que se está manipulando es la referencia, así es que cuando asignamos “de un objeto a otro” estamos actualmente copiando una referencia de un lugar a otro. Esto significa que si decimos `C = D` para objetos, ambos terminan apuntando a el mismo objeto que apuntaba `D`. El siguiente ejemplo demostrará esto:

He aquí el ejemplo:

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.
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 simple y dos instancias de ella (**n1** y **n2**) son creadas con un **main()**. El valor **i** en cada clase **Number** se le asigna un valor diferente, y **n2** es asignado a **n1**, y **n1** es cambiado. En muchos lenguajes de programación se esperaría que **n1** y **n2** sean independientes en todo momento, pero a causa de que se asigna una referencia en la salida se vera:

```

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

```

Si se cambia el objeto **n1** se puede ver que se cambia el objeto **n2** de la misma forma! Esto es porque ambos, **n1** y **n2** contienen la misma referencia, que es puntero a el mismo objeto (La referencia original que estaba en **n1** y apuntaba a el objeto que guardaba fue sobreescrita durante el asignación y efectivamente perdido; el objeto será limpiado por el recolector de basura).

Este fenómeno es también llamado *aliasing* es la forma fundamental con la que Java trabaja con objetos. Pero que si no se quiere que ocurra esto en este caso? Se puede privar de asignar la referencia y decir:

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

Esto mantiene dos objetos separados en lugar de eliminar uno y tratar a **n1** y **n2** como el mismo objeto, pero como verá mas adelante realizar la manipulación de los campos de un objeto es desordenado y va contra los principios de una buen diseño de programación orientada a objetos. Esto no es un tema trivial, así es que se debería leer el apéndice A, que es devoto del aliasing. Por lo pronto, deberemos tener en mente que la asignación de objetos puede tener sorpresas.

## Aliasing en la llamada a métodos

El aliasing puede ocurrir cuando pasamos un objeto a un método:

```

//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.
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, el método **f()** puede parecer que hace una copia de su argumento **Letter** y en el alcance de un método. Pero una vez mas una referencia es pasada de esta forma en la línea:

```

| y.c = 'z';
actualmente es cambiado el objeto fuera de f(). La salida muestra esto:

```

```

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

```

Aliasing y esta solución es un tema complejo, y a pesar que se debe esperar hasta el apéndice A para todas las respuestas, se deberá estar alerta de esto en este punto así es que vamos a poder tener algunas dificultades.

## Operadores matemáticos

Los operadores matemáticos básicos son los mismos que los disponibles en muchos lenguajes de programación; suma (+), resta (-), división (/), multiplicación (\*) y módulo (%), que produce el resto de una división entera). La división entera trunca, en lugar de redondear, el resultado.

Java también utiliza una notación taquigráfica para ejecutar una operación y una asignación en el mismo momento. Esto se logra con un operador seguido de un signo de igual, y es consistente con todos los operadores en el lenguaje (donde quiera que tenga sentido). Por ejemplo, para agregar 4 a la variable **x** y asignando el resultado **x**, utilice: **x += 4**.

Este ejemplo muestra el uso de los operadores matemáticos:

```

//: c03/MathOps.java
// Demonstrates the mathematical operators.
import java.util.*;
public class MathOps {
    // Create a shorthand to save typing:
    static void prt(String s) {
        System.out.println(s);
    }
    // shorthand to print a string and an int:
    static void pInt(String s, int i) {

```

```

        prt(s + " = " + i);
    }
    // shorthand to print a string and a float:
    static void pFlt(String s, float f) {
        prt(s + " = " + f);
    }
    public static void main(String[] args) {
        // Create a random number generator,
        // seeds with current time by default:
        Random rand = new Random();
        int i, j, k;
        // '%' limits maximum value to 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);
        // Floating-point number tests:
        float u,v,w; // applies to doubles, too
        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);
        // the following also works for
        // char, byte, short, int, long,
        // and 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);
    }
} //:~

```

La primera cosa que se verá son algunos métodos taquigráficos para imprimir: el método **prt()** imprime un **String**, el **pInt()** imprime un **String** seguido por un **int** y el **pFlt()** imprime un **String** seguido por un **float**. Por supuesto, se termina siempre utilizando **System.out.println()**.

Para generar números, el programa primero crea un objeto **Random**. Como ningún argumento es pasado durante la creación, Java utiliza la hora actual como semilla para el generador de números aleatorios. El programa genera una determinada cantidad de diferentes tipos de números aleatorios con el objeto **Random** simplemente llamando a diferentes métodos: **nextInt()**, **nextLong()**, **nextFloat()** o **nextDouble()**.

El operado módulo, que utilizamos con el resultado del generador de números aleatorios, limita el resultado a una cota superior del operador menos uno (99 en este caso).

## Operadores unitarios menos y mas

Los operadores unitario menos (-) y mas (+) son los mismos operadores menos y mas binarios. El compilador se imagina el uso que se pretende dar por la forma de escribir la expresión. Por ejemplo, la instrucción:

```
| x = -a;  
tiene un significado obvio. El compilador es capaz de figurárselo:  
| x = a * -b;  
pero el lector puede confundirse, así que es mas claro decir:  
| x = a * (-b);  
El menos unitario produce el negativo del valor. El unitario mas proporciona una simetría con el menos unitario, así es que no tiene efecto.
```

## Incremento y decremento propio

Java, como C esta lleno de atajos. Los atajos pueden hacer el código mas fácil de escribir, y a también fácil o difícil de leer.

Dos de los mas lindos atajos son los operadores de incremento y decremento (muchas veces refiriéndose como los incremento y decrementos propios). El operador de decremento es -- y significa “decremente en una unidad”. El operador de incremento es ++ y significa “incremente en una unidad”. Si a es un **int**, por ejemplo, la expresión **++a** es equivalente a (**a = a + 1**). Los operadores de incremento y decremento producen el valor de la variable como resultado.

Hay dos versiones para cada tipo de operador, a menudo llamadas las versiones prefijo y postfijo. Incremento previo significa que el operador ++ aparece antes que la variable o expresión, y incremento subsiguiente significa que el operado ++ aparece luego de la variable o expresión. Para los incrementos y decrementos previos (i.e., **++a** o **--a**), la operación es realizada y el valor es producido. Para los incrementos y decrementos subsiguientes (i.e. **a++** y **a--**), el valor es producido, luego la operación se realiza. Como ejemplo:

```
//: c03:AutoInc.java  
// Muestra los operadores ++ y --.  
public class AutoInc {  
    public static void main(String[] args) {  
        int i = 1;  
        prt("i : " + i);  
        prt("++i : " + ++i); // incremento previo  
        prt("i++ : " + i++); // incremento subsiguiente
```

```

        prt("i : " + i);
        prt("--i : " + --i); // decremento previo
        prt("i-- : " + i--); // decremento subsiguiente
        prt("i : " + i);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~

```

La salida para el programa es:

```

i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1

```

Se puede ver que para la forma prefija se obtiene el valor luego que la operación ha sido ejecutada, pero en la forma postfija se obtiene el valor antes que la operación es realizada. Estos son los únicos operadores (otros involucran asignaciones) que tiene efectos distintos de acuerdo al lado donde se ponga (esto es, cambian el operador antes que se use su valor.).

El operador de incremento es una explicación para el nombre C++, implica “un paso adelante de C”. En un discurso temprano acerca de Jaca, Bill Jow (uno de sus creadores), dijo que “Java=C++-” (C mas mas menos menos), sugiriendo que Java es C++ con las partes innecesariamente duras removidas y sin embargo un lenguaje mucho mas simple, y aun java no es mucho mas simple que C++.

## Operadores relacionales

Los operadores relacionales generan un resultado del tipo **boolean**. Ellos evalúan la relación entre los valores de los operadores. Una expresión relacional produce **true** si la relación es verdadera y **false** si la relación es falsa. Los operadores de relación son menor que (<), mayor que (>), menor igual que (<=), mayor igual que (>=), equivalente (==), y no equivalente (!=). Equivalencia y no equivalencia trabajan con todos los tipos de datos , pero las otras comparaciones no funcionan con el tipo **boolean**.

### Examinando la equivalencia entre objetos

Los operadores relacionales == y != también trabajan con todos los objetos, pero su significado a veces es confuso para el programados que recién comienza con Java. He aquí un ejemplo:

```

//: c03: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)** imprimirá el resultado de una comparación **boolean** sin ella. Seguramente la salida tendría que ser **true** y luego **false**, dado que ambos objetos Integer son los mismos. Pero a pesar de que el *contenido* de los objetos son el mismo, las referencias no son las mismas y los operadores `==` y `!=` comparan referencias a objetos. Así es que la salida es **false** y luego **true**. Naturalmente, esto sorprende a las personas al comienzo.

Que si se desea saber si el contenido actual de dos objetos son iguales? Para esto se utiliza el método especial **equals()** que existe para todos los objetos (no primitivos, que trabajen bien con `==` y `!=`). Se utiliza de la siguiente forma:

```

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 **true**, como se esperaba. Ha, pero, no es tan simple como esto. Si se crean clases completamente nuevas, como esta:

```

//: c03: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 lo convencional: el resultado es **false**. Esto es porque el comportamiento por defecto de **equals()** es comparar referencias. Así es que si no se *sustituye* **equals()** la clase no tendrá el comportamiento deseado. Desafortunadamente, no aprenderemos a sustituir hasta el capítulo 7, pero hay que tener cuidado con la forma que **equals()** se comporta puede ayudar a evitar alguna pena mientras tanto.

La mayoría de las librerías de clase de Java implementan **equals()** así es que se comparan el contenido de los objetos en lugar de sus referencias.

## Operadores lógicos

Los operadores lógicos AND (`&&`), OR (`||`) y NOT (`!`) producen un valor **boolean true** o **false** basado en la relación lógica de sus argumentos. Este ejemplo utiliza operadores relacionales y lógicos.

```
//: c03:Bool.java
// Relational and logical operators.
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 int como boolean no
        // es legal 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);
    }
} //:~
```

Se puede aplicar AND, OR o NOT únicamente a valores **boolean**. No se puede utilizar un valor que no es **boolean** como si lo fuera en una expresión lógica de la misma forma que se hace en C y C++. Se puede ver que falla cuando se intenta hacer. Las expresiones que siguen, sin embargo producen valores **boolean** utilizando comparaciones relacionales, luego utilizan operadores lógicos en los resultados.

La salida listado se vería como esto:

```
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
```

Se puede notar que un valor **boolean** es automáticamente convertido a una forma de texto apropiada si es utilizado donde se espera un **String**.

Podemos remplazar la definición de int en el programa antes citado con otro tipo de datos primitivo que no sea **boolean**. Hay que ser cuidadoso, sin embargo, que la comparación de punto flotante es muy estricta. Un número que es una diminuta fracción diferente de otro número será “no igual”. Un numero que es un poquito mas grande que cero no será cero.

## Cortocircuitado

Cuando se trata con operadores lógicos nos topamos con un fenómeno llamado “cortocircuitado”. Esto significa que la expresión será evaluada solo *hasta* la verdad o falsedad de la expresión entera pueda ser claramente determinada. Como resultado, todas las partes de la expresión pueden no ser evaluadas. Aquí hay un ejemplo que demuestra cortocircuitado:

```
//: c03:ShortCircuit.java  
// Demonstrates short-circuiting behavior.  
// with logical operators.  
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 prueba realiza una comparación del argumento y retorna verdadero o falso. También imprime información para indicar que ha sido llamada. Las pruebas son llamadas utilizando la expresión:

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

Naturalmente, podemos pensar que estas tres funciones son ejecutadas, pero la salida muestra otra cosa:

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

La primer prueba produce un resultado **true**, así es que la evaluación de la expresión continúa. Sin embargo, la segunda prueba produce un resultado **false**. Dado que esto significa que la totalidad de la expresión será **false**, porque continuar el resto de la expresión? Esto puede ser caro. La razón para cortocircuitar, de echo es precisamente esa; podemos obtener un incremento potencial de rendimiento si todas las partes de una expresión no necesitan ser evaluadas.

## Operadores a nivel de bits.

Los operadores a nivel de bits permiten manipular bits de forma individual en un tipo de dato primitivo. Los operadores a nivel de bits realizan álgebra de Bool con los bits correspondientes de los argumentos para producir un resultado.

Los operadores a nivel de bits son los orientados a bajo nivel de C; podemos a menudo manipular hardware directamente y tener que determinar bits en registros de hardware. Java originalmente fue diseñado para incrustar en televisores, así es que la orientación de bajo nivel tiene sentido. Sin embargo, probablemente no se utilice mucho operadores a nivel de bits.

El operador a nivel de bits AND (**&**) produce un bit uno en la salida si ambas entradas son uno; de otra forma produce un cero. El operador OR (**|**) produce un uno si alguna de las entradas son uno y cero si ambas entradas son cero. El operador NOT (**~**, también llamado operador *complemento*) es un operador unitario; toma solo un argumento (Todos los demás operadores son binarios). El operador NOT produce el opuesto de la entrada a nivel de bits-uno si el bit de entrada es cero, un cero si es uno.

Los operadores a nivel de bits y los operadores lógicos utilizan los mismos caracteres, así que es muy útil tener dispositivos nemotécnicos para ayudarnos a recordar los significados: dado que los bits son “pequeños”, hay solo un carácter en las operaciones a nivel de bits. Los operadores pueden ser combinados con el signo **=** para combinar la operación y la asignación: **&=**, **|=** y **^=** son todas legítimas (Dado que **~** es un operador unitario no puede ser combinado con el signo de **=**).

El tipo **boolean** es tratado como un valor de un solo bit así que es un algo diferente. Se puede ejecutar un operación a nivel de bits AND, OR y XOR, pero no podemos ejecutar una NOT a nivel de bits (presumiblemente para prevenir confusión con el NOT lógico). Para valores **boolean** se incluye un operador XOR lógico que no está incluido en la lista de operadores “lógicos”.

Se esta impeditido de utilizar valores **boolean** en expresiones de desplazamiento, como será descrito mas adelante.

## Operadores de desplazamiento

Los operadores de desplazamiento pueden también manipular bits. Estos pueden ser utilizados exclusivamente con tipos primitivos enteros. El operador de desplazamiento a la izquierda (`<<`) produce que el operando se desplace a la derecha el numero de bits especificados luego del operador (insertando ceros en los bits de menor orden). El operador de desplazamiento a la derecha con signo (`>>`) produce que el operando a la izquierda se desplace a la derecha el numero de bits especificado luego del operador. El operador `>>` utiliza *extensión de signo*: si el valor es positivo, se insertan ceros en los bits de mayor orden; si el valor es negativo, son insertados unos en los bits de mayor orden. Java también agrega el desplazamiento a la derecha sin signo `>>>`, que usa *extensión cero*: independientemente del signo, ceros son insertados en los bits de mayor orden. Este operador no existe en C o C++.

Si se desplaza un **char**, **byte** o **short**, será convertido a **int** antes de que el desplazamiento tenga lugar, y el resultado será un **int**. Solo los cinco bits de orden mas bajo del lado derecho serán utilizados. Esto previene que se desplacen mas que el número de bits en un entero. Si se esta operando con un **long**, tendremos un resultado long. Solo los seis bits de menor orden del lado derecho serán utilizados así es que no se puede desplazar mas que el número de bits en un **long**.

Los desplazamientos pueden ser combinados con el signo de igual (`<<=` o `>>=` o `>>>=`). El lvalue es remplazado por el lvalue desplazado con el rvalue. Aquí hay un problema, sin embargo, con el desplazamiento sin signo combinado con la asignación. Si se usa con **byte** o **short** no se obtendrán los resultados correctos. En lugar de eso, estos son convertidos a **int** y desplazados a la derecha, y serán truncados cuando se asignen nuevamente a sus variables, así se obtendrá un -1 en esos casos. El ejemplo que sigue demuestra esto:

```
//: c03:URShift.java
// Prueba de utilización de desplazamiento sin signo a la derecha.
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 valor resultante no es asignado de vuelta a **b**, pero es impreso directamente y es comportamiento esperado es correcto.

Aquí hay un ejemplo que demuestra el uso de todos los operadores que involucran bits:

```

//: c03:BitManipulation.java
// Usando operadores a nivel 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("1 << 5", 1 << 5);
pBinLong("1 >> 5", 1 >> 5);
pBinLong("(~1) >> 5", (~1) >> 5);
pBinLong("1 >>> 5", 1 >>> 5);
pBinLong("(~1) >>> 5", (~1) >>> 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 al final, **pBinInt()** y **pBinLong()** toman un **int** o un **long** respectivamente, y los imprimen en formato binario con una descripción. Se puede ignorar esta implementación por ahora.

Se puede notar el uso de **System.out.print()** en lugar de **System.out.println()**. El método **print()** no produce un salto de línea, lo que permite desplegar una sola línea de una sola vez.

Igualmente como se demuestra el efecto de todos los operadores a nivel de bit para **int** y **long**, este ejemplo muestra el los valores mínimo, máximo, +1 y -1 para **int** y **long** así que se puede observar como se ven. Puede notarse que el bit mas alto representa el signo: 0 significa positivo y 1 negativo. La salida para la parte del **int** se vería como esto:

```
-1, int: -1, binary:  
111111111111111111111111111111111111111111111  
+1, int: 1, binary:  
000000000000000000000000000000000000000000000000001  
maxpos, int: 2147483647, binary:  
0111111111111111111111111111111111111111111111111  
maxneg, int: -2147483648, binary:  
1000000000000000000000000000000000000000000000000000  
i, int: 59081716, binary:  
00000011100001011000001111110100  
~i, int: -59081717, binary:
```

```

1111110001111010011110000001011
-i, int: -59081716, binary:
1111110001111010011110000001100
j, int: 198850956, binary:
00001011110110100011100110001100
i & j, int: 58720644, binary:
00000011100000000000000110000100
i | j, int: 199212028, binary:
000010111101111101110111111100
i ^ j, int: 140491384, binary:
0000100001011111011101001111000
i << 5, int: 1890614912, binary:
0111000010110000011111010000000
i >> 5, int: 1846303, binary:
00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
1111111111000111101001111100000
i >>> 5, int: 1846303, binary:
00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
0000111110001111010011111000000

```

La representación binaria de los números es referida como *complemento a dos con signo*.

## Operador ternario if-else

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

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

Si *boolean-exp* se evalúa como **true**, *value0* es evaluado y su resultado es el producido por el operador. Si *boolean-exp* es **false**, *value1* es evaluado y su resultado es el producido por el operador.

Claro que, se puede utilizar un comando **if-else** (descrito mas adelante), pero el operador ternario es muy conciso. De la misma forma que C (de donde este operador es originario) se enorgullece de ser un lenguaje conciso, y el operador ternario puede ser introducido en parte por eficiencia, se debe ser un poco cuidadoso al utilizarse en las básicas de todos los días-es fácil producir código ilegible.

El operador condicional puede ser utilizado por sus efectos de lado o por el valor que produce, pero en general el valor que genera el operador lo distingue del **if-else**. He aquí un ejemplo:

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

Como se puede ver, este código es mas compacto que el que se necesita para escribir sin el operador ternario:

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

La segunda forma es mas fácil de entender, y no necesita mucho mas escritura. Así es que debemos asegurarnos de deliberar las razones cuando elijamos el operador ternario.

## El operador coma

La coma es utilizado en C y C++ no solo como separador en listas de argumentos de funciones, también como operador para evaluaciones secuenciales. El único lugar donde el *operador coma* es utilizado en Java es en los bucles **for**, que serán descritos mas adelante en este capítulo.

## Operador de **String** +

Aquí tenemos un uso especial de un operador en Java: el operador + puede ser utilizado para concatenar cadenas, como ya hemos visto. Parece un uso natural del + aun si no corresponde con la forma tradicional que el signo es utilizado. Esta habilidad que parecía una buena idea en C++, de tal manera que la *sobrecarga de operadores* fue agregada a C++ para permitir al programador agregar significado a casi cualquier operador.

Desafortunadamente, la sobrecarga de operadores combinada con algunas otras restricciones en C++ se convirtió en un rasgo bastante complicado para los programadores para diseñar dentro de sus clases. A pesar que la sobrecarga de operadores sería mucho mas fácil de implementar en Java que en C++, este rasgo sigue considerándose muy compleja, así es que los programadores Java no pueden implementar sus propios operadores sobrecargados como los programadores C++.

El uso del + en **String** tiene algunos comportamientos interesantes. Si una expresión comienza con un tipo **String**, todos los operandos que siguen deben ser del tipo **String** (recordemos que el compilador convertirá secuencias de caracteres entre comillas a tipo **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 del tipo **String** en lugar de sumarlos entre ellos primero. Ahora si colocamos:

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

Java convertirá `x + y + z` en un String y luego lo concatenará a `sString`.

## Dificultades comunes utilizando operadores

Una de las dificultades cuando utilizamos operadores es tratar de quitar los paréntesis incluso cuando se tiene la mas pequeña incertidumbre acerca de como una expresión se evaluará. Esto sigue siendo verdad en Java.

Un error extremadamente común en C y C++ se vería como esto:

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

El programador trató de probar la equivalencia (`==`) en lugar de realizar una asignación. En C y C++ el resultado de esta asignación siempre será **true** si `y` no es cero, y probablemente se tendrá un bucle infinito. En Java, el resultado de esta expresión no es un tipo **boolean**, y el compilador espera un **boolean** y no lo convertirá a **int**, así es que convenientemente entregara un error en tiempo de compilación y se encontrará el problema antes que se intente correr el programa. Así es que es escollo jamás sucederá en Java (El único momento en que no recibirá un error en tiempo de compilación es cuando `x` e `y` son del tipo **boolean**, caso en el cual `x = y` es una expresión legal, y en el caso anterior, probablemente un error).

Un problema similar en C y C++ es utilizando los AND y OR a nivel de bits en lugar de las versiones lógicas. AND y OR a nivel de bits uso un solo carácter (`&` o `|`) y AND y OR lógicos usan dos (`&&` y `||`). Igual que con `=` y `==`, es fácil escribir un solo carácter en lugar de dos. En Java, el compilador previene contra esto porque no deja utilizar caballerescamente un tipo donde no pertenece.

## Operadores de conversión

Java puede cambiar automáticamente un tipo de datos en otro cuando sea apropiado. Para algunas ocasiones, si se asigna un valor entero a una variable de punto flotante, el compilador automáticamente convertirá en **int** en **float**. La conversión permite hacer este tipo de cosas de forma explícita, o forzarla cuando no sucedería normalmente.

Para realizar una conversión, se coloca el tipo de dato deseado (incluidos todos los modificadores) adentro de paréntesis a la izquierda del un valor. He aquí un ejemplo:

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

```
|     long l2 = (long)200;  
| }
```

Como se puede ver, es posible realizar una conversión de un valor numérico igual que para una variable. En ambos casos mostrados aquí, sin embargo, la conversión es superflua, dado que el compilador automáticamente promover un valor **int** a un valor del tipo **long** cuando sea necesario. Sin embargo, está permitido utilizar conversiones superfluas para marcar un punto o hacer el código más claro. En otras situaciones, una conversión puede ser esencial sencillamente para hacer que el código compile.

En C y C++, la conversión de tipo puede causar algunos dolores de cabeza. En Java, la conversión es segura, con la excepción que cuando se ejecuta una conversión muchas veces llamada estrecha (*narrowing*) esto es, cuando partimos de un tipo que puede almacenar más información hacia un tipo que no almacena la totalidad) se corre el riesgo de perder información. Aquí el compilador fuerza a hacer una conversión, en efecto diciendo “esto puede ser una cosa peligrosa para hacer si quiere que lo haga igual debe hacer la conversión explícitamente”. Con una *conversión que se ensancha* (*widening*) no es necesario porque el nuevo tipo puede almacenar más que el tipo viejo así es que no se pierde información.

Java permite convertir cualquier tipo a cualquier otro tipo primitivo, excepto para **boolean**, no permite ninguna conversión. Los tipos de clases no permiten conversiones. Para convertir uno en otro debe haber un método especial (**String** es un caso especial, y se verá más adelante en este libro que objetos pueden convertirse con una familia de tipos, un **Oak** puede ser convertido en un **Tree** y a la inversa, pero no a un tipo extraño como **Rock**).

## Literales

Normalmente cuando insertamos un valor literal en un programa el compilador sabe exactamente qué tipo utilizar con él. A veces, sin embargo, el tipo es ambiguo. Cuando esto sucede debemos guiar el compilador agregando información extra en la forma con que los caracteres asociados con el valor literal. El siguiente código muestra esos caracteres:

```
//: c03:Literals.java  
class Literals {  
    char c = 0xffff; // max char hex value  
    byte b = 0x7f; // max byte hex value  
    short s = 0x7fff; // max short hex value  
    int i1 = 0x2f; // Hexadecimal (lowercase)  
    int i2 = 0X2F; // Hexadecimal (uppercase)  
    int i3 = 0177; // Octal (leading zero)  
    // Hex and Oct also work with long.  
    long n1 = 200L; // long suffix  
    long n2 = 200l; // long suffix  
    long n3 = 200;  
    //! long 16(200); // not allowed
```

```

float f1 = 1;
float f2 = 1F; // float suffix
float f3 = 1f; // float suffix
float f4 = 1e-45f; // 10 to the power
float f5 = 1e+9f; // float suffix
double d1 = 1d; // double suffix
double d2 = 1D; // double suffix
double d3 = 47e47d; // 10 to the power
} //:~

```

Los hexa decimales (base 16), que funcionan con todos los tipos de datos, se indican con un **0x** o **0X** adelante seguidos o por un dígito (0-9) o por una letra (a-f) en mayúscula o minúscula. Si intentamos inicializar una variable con un valor mayor que el que se puede almacenar (independiente de la forma del valor numérico), el compilador dará un mensaje de error. Se puede notar en al código anterior los valores máximos para **char**, **byte** y **short**. Si se exceden estos valores, el compilador automáticamente creará el valor entero e indicará que se necesita convertir el tipo. Se sabrá que esta superando la línea.

Los octales (base 8) de indican con un cero al comienzo del numero y se utilizan dígitos de 0 a 7. No hay representación literal para números binarios en C, C++ o Java.

Un carácter que se arrastra luego de el valor literal establece su tipo. En mayúscula o en minúscula la **L** significa **long**, en mayúscula o en minúsculas **F** significa **float** y la **D** significa **double**.

Los exponentes utilizan una notación que se siempre se encuentran antes del desmayo: **1.39 e-47f**. En ciencia e ingeniería, ‘e’ se refiere a la base natural logarítmica, aproximadamente 2.718. (Un valor **double** mas preciso está disponible en Java como **Math.E**). Este es utilizado en expresiones exponenciales como  $1.39 \times 10^{-47}$ , que significa  $1.39 \times 2.718^{-47}$ . Sin embargo, cuando FORTRAN fue inventado se decidió que **e** naturalmente significa “diez a la potencia”, que es una decisión curiosa porque FORTRAN fue diseñado para la ciencia e ingeniería y uno puede pensar que sus diseñadores podrían ser sensibles acerca de introducir tal como una ambigüedad<sup>1</sup>. En cualquier evaluación, este hábito fue seguido en C, C++ y

---

<sup>1</sup> John Kirkham escribió, “He estado computando en 1962 utilizando FORTRAN II en una IBM 1620. En este momento, y durante todos los años 60 y en los 70, FORTRAN fue un lenguaje en mayúsculas. Esto probablemente comenzó porque muchos de los dispositivos al inicio fueron viejas unidades de teletipo que utilizaban código Baudot de 5 bit, y no tenían capacidad de representar minúsculas. La letra ‘E’ en la notación exponencial fue también siempre mayúscula y nunca se confundió con el logaritmo en base natural ‘e’, que siempre fue minúscula. La ‘E’ simplemente se mantuvo de pie para exponencial, que fue por la base del sistema de numeración utilizado-usualmente 10. En el momento que octal fue también ampliamente utilizado por los programadores. A pesar que yo nunca distinguí cual usar, si tengo que ver un numero octal en notación exponencial nunca hubiera considerado que fuera de base 8. El primer momento que recuerdo ver un uso exponencial de la letra

ahora en Java. Así es que si se está acostumbrado a pensar en términos de **e** como la base de los logaritmos naturales, se deberá realizar una traslación mental cuando se use una expresión como **1.32e-47f** en Java; esto significa  $1.39 \times 10^{-47}$ .

Se debe notar que no se necesita utilizar el último carácter cuando el compilador puede figurarse el tipo apropiado. Con

```
| long n3 = 200;  
no hay ambigüedad, así es que una L luego del 200 puede ser superfluo. Sin  
embargo con  
| float f4 = 1e-47f; // 10 a la potencia  
el compilador normalmente toma los números exponentiales como dobles,  
así es que sin el último carácter f dará un error indicando que se debe  
utilizar una conversión de tipo explícitamente double a float.
```

## Promoción

Se descubrirá que si se realiza una operación matemática o a nivel de bits en tipos de datos primitivos que son menores que un **int** (esto es, **char**, **byte**, o **short**), estos valores serán ascendidos a **int** antes de realizar las operaciones, y el valor resultante será del tipo **int**. Así es que si se desea asignar nuevamente a un tipo más pequeño, se deberá utilizar conversiones de tipo (Y, dado que se está asignando a un tipo más pequeño, se podrá perder información). En general, los tipos más grandes es el que determina el tamaño del resultado de una expresión; si multiplicamos un **float** y un **double**, el resultado deberá ser **double**; si sumamos un **int** y un **long**, el resultado será **long**.

## Java no tiene “sizeof”

En C y C++, el operador **sizeof()** satisface una necesidad específica: indica el número de bytes asignados para ítem de datos. La necesidad más urgente para **sizeof()** en C y C++ es la portabilidad. Los diferentes tipos de datos pueden ser de diferentes tamaños en diferentes máquinas, así es que el programador debe averiguar cuan grandes son los tipos que hay cuando se realizan operaciones sensibles a el tamaño. Por ejemplo, una computadora puede almacenar enteros en 32 bits, mientras que otra puede almacenar enteros de 16 bits. Los programas pueden almacenar valores grandes de enteros en la primer máquina. Como se debe imaginar, la portabilidad es un enorme dolor de cabeza para los programadores C y C++.

---

minúscula ‘e’ fue a finales de los 70 y también me sentí confundido. El problema se origina lentamente en FORTRAN, no es sus comienzos. Tenemos actualmente funciones para usar que realmente queremos utilizar la base de logaritmos naturales, y existen todas en mayúsculas”.

Java no necesita un operador `sizeof()` para este propósito porque todos los tipos de datos son del mismo tamaño en todas las máquinas. No se necesita pensar acerca de la portabilidad en este nivel—esta diseñado dentro del lenguaje.

## Revisión de precedencia

Enzima de escucharme quejar sobre la complejidad de recordar la precedencia de operadores durante uno de mis seminarios, un estudiante sugirió un nemotécnico que es simultáneamente una observación: “Ulcer Addicts Really Like C A lot.” (Los adictos a las úlceras realmente les gusta C un montón).

Nemotécnico	Tipo de operador	Operadores
Ulcer	Unitario	+ - + + --
Addicts	Aritmético (y desplazamiento)	* / % + - << >>
Really	Relacional	> < >= == !=
Like	Lógico (y a nivel de bits)	&&    &   ^
C	Condicional (ternarios)	A > B ? X : Y
A lot	Asignación	= (y asignación compuesta como *=)

Claro, que con el desplazamiento y los operadores a nivel de bits distribuidos por la tabla no es un nemotécnico perfecto, pero para operaciones que no son a nivel de bit trabaja.

## Un compendio de operadores

El siguiente ejemplo muestra que tipos de datos primitivos puede ser utilizado con un operador particular. Básicamente, es el mismo ejemplo repetido una y otra vez, pero utilizando diferentes tipos de datos. El fichero compilara sin error dado que las líneas que pueden causar error están comentados con un `//!`.

```
//: c03:AllOps.java
// Prueba todos los operadores para todos los
// datos primitivos para mostrar con los
// aceptados por el compilador Java.
class AllOps {
    // Para aceptar los resultados de una prueba booleana:
    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 a nivel de bits:
//! x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
//! x = x << 1;
//! x = x >> 1;
//! x = x >>> 1;
// Asignación compuesto:
//! x += y;
//! x -= y;
//! x *= y;
//! x /= y;
//! x %= y;
//! x <= 1;
//! x >= 1;
//! x >>= 1;
//! x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! 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 a nivel 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);
// Asignamiento compuesto:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
///! 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);
// Operadores a nivel de bits:
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);
// Asignamientos compuestos:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Conversiones:
//! 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 a nivel 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);
// Asignaciones compuestas:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Conversiones:
///! 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 a nivel de bits:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
}

```

```

x = x << 1;
x = x >> 1;
x = x >>> 1;
// Asignamientos compuestos:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Conversiones:
//! 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 a nivel de bits:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Asignamientos compuestos:
    x += y;
}

```

```

x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Conversiones:
//! 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 a nivel de bits:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignaciones compuestas:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <= 1;
}

```

```

    //! x >= 1;
    //! x >>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! 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 a nivel de bits:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Asignamiento compuesto:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
}

```

```

    // Casting:
    //! 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;
}
} //:~

```

Se puede notar que **boolean** es un poco limitado. Se puede asignar los valores **true** y **false**, y se puede verificar por verdad o falsedad, pero no puede sumar tipos boolean o realizar otro tipo de operaciones con ellos.

En **char**, **byte** y **short** se puede ver el efecto de promoción con los operadores aritméticos. Cada operador aritmético de cualquiera de estos tipos tiene un resultado del tipo **int**, que debe ser explícitamente convertido a el tipo original (una conversión a un tipo inferior puede hacer perder información) para asignar nuevamente al tipo anterior. Con los valores del tipo **int**, sin embargo, no se necesita convertir el tipo, porque todo ya se encuentra con el tipo **int**. No se puede ser calmado al pensar que todo es seguro, sin embargo. Si se multiplican dos valores del tipo **int** que son suficientemente grandes, se puede obtener un desbordar el resultado. El siguiente ejemplo demuestra esto:

```

//: c03:Overflow.java
// Sorpresa! Java indica desbordamiento.
public class Overflow {
    public static void main(String[] args) {
        int big = 0x7fffffff; // valor maximo para int
        prt("big = " + big);
        int bigger = big * 4;
        prt("mayor = " + bigger);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} //:~

```

La salida de esto es:

```

big = 2147483647
bigger = -4

```

y no se tienen errores o advertencias del compilador, y no hay excepciones en tiempo de ejecución. Java es bueno, pero no *tan* bueno.

Las asignaciones compuestas *no* necesitan conversiones de tipo para **char**, **byte** o **short**, aún si realizan promociones que tienen los mismos resultados que las operaciones aritméticas directas. Por otro lado, la ausencia de la conversión de tipos naturalmente simplifica el código.

Se puede ver que, con excepción del tipo **boolean**, cualquier tipo primitivo puede ser convertido a cualquier otro tipo primitivo. Nuevamente, debe

tener cuidado de los efectos de hacer mas pequeño un tipo cuando se convierte a un tipo inferior, de otra forma puede perder sin saberlo información al convertir el tipo.

# Control de ejecución

Java utiliza todas las instrucciones de control de ejecución de C, así es que se verá familiar si se ha programado con C o C++. Muchos lenguajes de programación procesales tienen algún tipo de instrucciones de control, y a menudo de pasan entre lenguajes. En Java, las palabras clave incluyen **if-else**, **while**, **do-while**, **for**, y una instrucción de control llamada **switch**. Java no soporta, sin embargo, el muy maldecido **goto** (que todavía puede ser la forma mas conveniente forma de solucionar cierto tipo de problemas). Aún se puede realizar un salto del tipo de **goto**, pero es mucho mas limitado que el típico.

## true y false

todas las instrucciones condicionales usan la verdad o falsedad de una expresión condicional para determinar el camino de ejecución. Un ejemplo de expresión condicional es **A == B**. Estas utilizan el operador condicional **==** para ver si el valor de **A** es equivalente a el valor de **B**. La expresión retorna **true** o **false**, Cualquiera de los operadores relacionales que se han visto antes en este capítulo pueden ser utilizados para producir expresiones condicionales. Debe notarse que Java no permite el uso de un número como tipo **boolean**, aun cuando esto esta permitido en C y en C++ (donde verdadero en un valor distinto de cero y falso es cero). Si se quiere utilizar un tipo que no sea **boolean** en una prueba del tipo **boolean**, como **if(a)**, debemos convertirlo a el valor **boolean** utilizando una expresión condicional, como es **if(a != 0)**.

## if-else

La instrucción **if-else** es probablemente la forma mas básica de controlar el flujo de un programa. El **else** es opcional, así es que se puede utilizar **if** de dos formas:

```
| if(expresión booleana)
|   instrucciones
|
| o
|
| if(expresión booleana)
|   instrucciones
| else
|   instrucciones
```

La condición debe producir un resultado del tipo **boolean**. Las *instrucciones* es una instrucción terminada con un punto y coma o una instrucción compuesta, que es un grupo de instrucciones encerrado entre llaves. En cualquier momento que la palabra “*instrucción*” sea utilizada, siempre implica que la instrucción pueda ser simple o compuesta.

A forma de ejemplo de **if-else**, he aquí un método **test()** que nos indicará si un número para adivinar es superior, inferior o igual a un número secreto:

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

Estos son cosas normales que se le piden a el cuerpo de una instrucción de control de flujo así que se puede determinar fácilmente donde comienza y donde termina.

## return

La palabra clave **return** tiene dos propósitos: especifica que valor de retorno tendrá un método (si no tiene un tipo **void** como retorno) y produce un retorno inmediato de ese valor. El método **test()** anterior puede ser reescrito para tomar ventaja de esto.

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

Aquí no hay necesidad de la palabra clave **else** ya que el método no continua luego de ejecutar un **return**.

## Iteración

Los bucles de control **while**, **do-while** y **for** son clasificados como *instrucciones de interacción*. Una *instrucción* se repite hasta que la expresión de control booleana evaluada es falsa. La forma del bucle **while** es

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

La *expresión booleana* es evaluada una vez al inicio del bucle y nuevamente cada nueva iteración de la *instrucción*.

He aquí un simple programa ejemplo que genera números al azar hasta que una condición particular es encontrada:

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

Utiliza un método estático llamado **random()** en la librería **Math**, que genera un valor **double** entre 0 y 1 (Esto incluye 0 pero no 1). La expresión condicional para el **while** dice “manténgase haciendo este bucle hasta que el numero es 0.99 o mayor”. Cada vez que se ejecute este programa obtendremos una lista de números de diferente tamaño.

## do-while

la forma del **do-while** es

```
do
    instrucción
while(expresión booleana)
```

la única diferencia entre **while** y **do-while** es que la instrucción en **do-while** se ejecuta siempre por lo menos una vez, aún si la expresión se evalúa como falsa la primera vez. En un **while**, si la condición es falsa la primera vez la instrucción nunca se ejecuta. En la práctica, **do-while** es menos común que **while**.

## for

Un bucle **for** realiza un inicialización antes de la primer iteración. Luego realiza pruebas condicionales y, al final de cada iteración, alguna forma de “adelantamiento de a pasos”. La forma del bucle **for** es:

```
for (inicialización; expresión booleana; paso)
    instrucción
```

Cualquiera de las expresiones *inicialización*, *expresión booleana* o *paso* no pueden ser vacías. La expresión es probada luego de cada iteración, y tan pronto como se evalúa en **false** la ejecución continúa en la línea seguida por la instrucción **for**. En el final de cada bucle, paso se ejecuta.

Los bucles **for** usualmente son utilizados para tareas de “cuenta”

```
//: c03>ListCharacters.java
// Demuestra los bucles "for" listando
// todos los caracteres ASCII.
public class ListCharacters {
    public static void main(String[] args) {
        for( char c = 0; c < 128; c++)
            if (c != 26 ) // Borrado de pantalla ANSI
                System.out.println(
                    "value: " + (int)c +
                    " character: " + c);
    }
} //:~
```

Se puede ver que la variable **c** esta definida en el punto donde va a ser utilizada, dentro de la expresión de control del bucle **for**, antes que al comienzo de el bloque indicado por la llave de apertura. El alcance de **c** es la expresión controlada por el **for**.

Los lenguajes tradicionales como C requiere que todas las variables sean definidas en el comienzo del bloque así es que cuando el compilador crea un bloque puede asignar el espacio para esas variables. En Java y en C++ se puede desparramar las declaraciones de variables por todo el bloque, definiéndolas en el punto en que se necesiten. Esto permite un estilo de código mas natural y hace el código mas fácil de entender.

Se pueden definir múltiples variables con la instrucción **for**, pero estas deben ser del mismo tipo:

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

la definición del tipo **int** en la instrucción **for** cubre **i** y **j**. La habilidad para definir variables en la expresión de control está limitada por el bucle **for**. No se puede utilizar este método con ninguna otra instrucción de selección o iteración.

## El operador coma

En los comienzos de este capítulo se dijo que el *operadorcoma* (no el *separadorcoma*, que es utilizado para separar definiciones y argumentos de funciones) tiene solo un uso en Java: en la expresión de control de un bucle **for**. En la inicialización y en la parte de pasos de la expresión de control se pueden tener varias instrucciones separadas por comas, y aquellas instrucciones serán evaluadas de forma secuencial. El trozo de código anterior utiliza esa habilidad. He aquí 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);
        }
    }
} //:~
```

Y su salida:

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

Como se puede ver en ambos en la inicialización y en las parte de pasos las instrucciones son evaluadas en orden secuencial. Además, la parte de la inicialización puede tener cualquier número de definiciones *de un tipo*.

## break y continue

Adentro del cuerpo de cualquiera de las instrucciones de la iteración se puede tener control del flujo del bucle utilizando **break** y **continue**. **break** se sale del bucle sin ejecutar el resto de las instrucciones en el bucle. **continue** detiene la ejecución de la actual iteración y regresa al comienzo del bucle para comenzar en la siguiente iteración.

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

```
//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.
public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Fuera del bucle
            if(i % 9 != 0) continue; // Siguiente iteración
            System.out.println(i);
        }
        int i = 0;
        // Un "bucle infinito":
        while(true) {
```

```

        i++;
        int j = i * 27;
        if(j == 1269) break; // fuera del bucle
        if(i % 10 != 0) continue; // Arriba en el bucle
        System.out.println(i);
    }
}
} //:~

```

En el bucle **for** el valor de **i** nunca llega a 100 porque la instrucción **break** sale del bucle cuando **i** es **74**. Normalmente, se usará una línea **break** como esta solo si no se conoce cuándo va a suceder la condición de terminación. La instrucción **continue** produce que la ejecución regrese a el comienzo de la iteración del bucle (de esta manera se incrementa **i**) cuando quiera que **i** no sea divisible entre 9. Cuando esto se sucede el valor es impreso.

La segunda parte muestra un "bucle infinito" que puede, en teoría, continuar para siempre. Sin embargo, dentro del bucle hay una instrucción **break** que saldrá del bucle. Además, se puede ver que **continue** regresa a el inicio del bucle sin completar el resto (Estas impresión se sucede en el segundo bucle solo 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

```

El valor de 0 es impreso porque  $0 \% 9$  produce 0.

Una segunda forma de bucle infinito es **for(;;)**. El compilador trata **while(true)** y **for(;;)** de la misma forma así es que cualquiera que se use es por preferencias de programación.

## El infame "goto"

La palabra clave **goto** ha estado presente en los lenguajes de programación desde el comienzo. Ciertamente, **goto** fue el génesis del control de programa en lenguaje ensamblador: "Si se da la condición A, entonces salte aquí, de otra forma salte aquí". Si se lee el código ensamblador que se genera al final de virtualmente cualquier compilador, se puede ver que el control del programa tiene muchos saltos. Sin embargo, **goto** es un salto a nivel de código fuente, y esto es lo que ha traído descrédito. ¿Si un programa siempre salta de un punto a otro, no hay una forma de reorganizar el código

para que el control de flujo no salte? **goto** cae en una verdadera desaprobación luego de la publicación de el famoso trabajo “goto considerado dañino” por Edsger Dijkstra, y desde que maltratar a **goto** ha sido un deporte popular, gracias a los abogados de la expulsión de palabras clave se ha echado a correr.

Como es típico en situaciones como esa, la tierra media es lo mas fructífero. El problema no es el uso de **goto**, sino la sobre utilización de **goto**-en situaciones poco comunes **goto** es actualmente la mejor manera de estructurar el control de flujo.

A pesar que **goto** es una palabra reservada en Java, no es utilizada en el lenguaje; Java no tiene **goto**. Sin embargo, hace algo que se ve un poco como salto atado con las palabras clave **break** y **continue**. No es un salto porque mas bien es una forma de quebrar una instrucción de iteración. La razón es a menudo olvidar las discusiones de **goto** porque utiliza el mismo mecanismo: una etiqueta.

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

```
| etiqueta1:
```

Al único lugar donde es útil en Java es justo antes de una instrucción de iteración. Y esto significa *justo antes*-no es bueno colocar otra instrucción entre la etiqueta y la iteración. Y la única razón para colocar una etiqueta después de una iteración es si esta anidada a otra iteración o un switch dentro de ella. Esto es porque las instrucciones **break** y **continue** serán normalmente interrumpidas solo en el bucle actual, pero cuando son utilizadas con una etiqueta serán interrumpidos encima de donde la etiqueta existe:

```
etiqueta 1:  
iteración exterior {  
    iteración interior {  
        //...  
        break; // 1  
        //...  
        continue; // 2  
        //...  
        continue etiqueta1; // 3  
        //...  
        break etiqueta 1; // 4  
    }  
}
```

En el caso uno, **break** quiebra la iteración interior y se termina en la iteración exterior. En el caso 2, **continue** regresa al inicio la iteración interior. Pero en el caso 3, la **etiqueta1** del **continue** quiebra la iteración interior y la iteración exterior, todo allá en la **etiqueta1**. Entonces de echo continúa la iteración, pero comenzando con la iteración externa. En el caso 4, la **break etiqueta1** también quiebra mas allá de la **etiqueta1**, pero no ingresa nuevamente la iteración. Actualmente no sale de ambas iteraciones.

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

```
public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer: // No puede haber instrucciones aquí
        for(; true ;) { // bucle infinito
            inner: // No puede haber instrucciones aquí
            for(; i < 10; i++) {
                prt("i = " + i);
                if(i == 2) {
                    prt("continua");
                    continue;
                }
                if(i == 3) {
                    prt("break");
                    i++; // De otra forma nunca
                    // seré incrementado.
                    break;
                }
                if(i == 7) {
                    prt("continua por afuera");
                    i++; // De otra forma nunca seré
                    // incrementado.
                    continue outer;
                }
                if(i == 8) {
                    prt("quiebre para afuera");
                    break outer;
                }
                for(int k = 0; k < 5; k++) {
                    if(k == 3) {
                        prt("continua en el interior");
                        continue inner;
                    }
                }
            }
        }
        // No se puede quebrar o continuar
        // con etiquetas aquí
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~
```

Esto utiliza el método **prt()** que ha sido definido en otros ejemplos.

Debe notarse que **break** sale del bucle **for**, y la expresión de incremento no ocurre hasta que se termine el bucle **for**. Dado que **break** evita la expresión de incremento, el incremento es realizado directamente en el caso de **i == 3**. La instrucción **continue outer** en el caso de **i == 7** también salta al inicio del bucle y evita el incremento, así es que se incrementa directamente también.

He aquí la salida:

```

i = 0
continua en el interior
i = 1
continua en el interior
i = 2
continua
i = 3
break
i = 4
continua en el interior
i = 5
continua en el interior
i = 6
continua en el interior
i = 7
continua por afuera
i = 8
quiebre para afuera

```

Si no fuera por la instrucción **break outer**, podría no ser la forma de salir del bucle exterior desde adentro del bucle interior, dado que **break** por si solo puede salir del bucle mas interno (Lo mismo es cierto para **continue**).

Claro que en los casos donde se sale de un bucle y también salir del método se puede utilizar simplemente **return**.

Aquí podemos ver una demostración de las instrucciones **break** etiquetado y **continue** para bucles **while**:

```

//: c03:LabeledWhile.java
// Java's "labeled while" loop.
public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            prt("salida del bucle while");
            while(true) {
                i++;
                prt("i = " + i);
                if(i == 1) {
                    prt("continuar");
                    continue;
                }
                if(i == 3) {
                    prt("continuar afuera");
                    continue outer;
                }
                if(i == 5) {
                    prt("quebrar");
                    break;
                }
                if(i == 7) {
                    prt("quebrar para afuera");
                    break outer;
                }
            }
        }
    }
}

```

```

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

```

Las mismas reglas se cumplen para **while**:

1. Una instrucción **continue** simple salta a el comienzo del bucle mas interno y continúa.
2. Una instrucción **continue** etiquetada salta a la etiqueta y entra en el bucle inmediatamente después de la etiqueta.
3. Un **break** “desciende inmediatamente al final” del bucle.
4. Un **break** etiquetado desciende al final del bucle indicado por la etiqueta.

La salida de este método lo aclara:

```

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 utilizar etiquetas en Java es cuando se tiene bucles anidados y se quiera realizar un break o un continue a través de uno o mas niveles anidados.

En el trabajo de Dijkstra's “goto considerado dañino”, lo que el específicamente objetaba eran las etiquetas, no el goto. El observaba que el número de errores parecían incrementarse con el número de etiquetas en un programa. Las etiquetas y las instrucciones goto hacen que sea difícil de analizar la sintaxis de los programas, dado que introduce ciclos en la gráfica de ejecución. Se puede ver que las etiquetas en Java no padecen de este problema, ya que son limitadas a un sitio y no pueden ser utilizadas para transferir el control en una manera ad hoc. Es también interesante de destacar que este es un caso en donde dadas las características del lenguaje es mas útil restringir el poder de la instrucción.

## switch

**switch** es también clasificada como una *instrucción de selección*. Esta selecciona de un conjunto de segmentos de código basadas en el valor de una expresión integral. Su forma es:

```
switch(selector-integral) {  
    case valor-integral1 : instrucción; break;  
    case valor-integral2 : instrucción; break;  
    case valor-integral3 : instrucción; break;  
    case valor-integral4 : instrucción; break;  
    case valor-integral5 : instrucción; break;  
    // ...  
    default: instrucción;  
}
```

*selector-integrales* una expresión que produce un valor integral. La instrucción **switch** compara el resultado de *selector-integral* con cada *valor integral*. Si se encuentra una coincidencia, la *instrucción* (simple o compuesta) correspondiente se ejecuta. Si no se encuentra ninguna coincidencia, se ejecuta la instrucción en **default**.

Se verá en la definición anterior que cada **case** termina con un **break**, que hace que la ejecución salte al final del cuerpo del **switch**. Esta es la forma convencional de armar una instrucción **switch**, pero la instrucción **break** es opcional. Si no se encuentra, el código de la siguiente instrucción **case** se ejecuta hasta que un **break** es encontrado. A pesar de que no se quiera este tipo de comportamiento, puede ser útil para un programador experimentado. La última instrucción, que continúa el **default**, no tiene un **break** porque la ejecución termina donde el **break** debería salir. Se puede colocar un **break** al final de la instrucción **default** sin daño si considera importante por un tema de estilo.

La instrucción **switch** es una forma limpia de implementar una selección con varios caminos (i.e., seleccionando de una determinada cantidad de diferentes caminos de ejecución), pero esto requiere un selector que evalúe un valor integral como lo es **int** o **char**. Si se desea utilizar, por ejemplo, una cadena o un valor de punto flotante como selector, no funcionará con la instrucción **switch**. Para tipos que no sean integrales, deberá utilizar una serie de instrucciones **if**.

He aquí un ejemplo que genera letras al azar y determina cual de ellas son vocales y cual de ellas son consonantes:

```
//: c03:VowelsAndConsonants.java  
// Demonstrates the switch statement.  
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 + ": ");  
            if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')  
                System.out.println("vowel");  
            else  
                System.out.println("consonant");  
        }  
    }  
}
```

```

        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");
        }
    }
}
} //:~

```

Dado que **Math.random()** genera un valor entre 0 y 1, necesita solo multiplicar por el límite superior del rango de números que desea producir (26 por las letras del alfabeto) y agregar un desplazamiento para establecer el límite inferior.

Aunque a pesar de que parece estar comutando caracteres, la instrucción **switch** está utilizando el valor integral del carácter. El carácter entre comillas simples en la instrucción **case** también produce valores integrales que pueden utilizarse para la comparación.

Debe notarse que las instrucciones **case** pueden ser “apiladas” arriba de cada una de las otras para proporcionar múltiples coincidencias para una segmento de código particular. Se debe ser cuidadoso con que es esencial colocar una instrucción **break** al final de cada **case**, de otra forma el control continuará simplemente ejecutando el siguiente **case**.

## Detalles de cálculo

La instrucción:

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

merece una mirada detallada. **Math.random()** produce un valor **double**, así es que el valor 26 es convertido a un **double** para ejecutar la multiplicación, lo que también produce un valor del tipo **double**. Esto significa que ‘a’ debe ser convertido a **double** para realizar la suma. El resultado **double** es regresado a el tipo **char** con una conversión.

¿Que es lo que el convertir a el tipo **char** hace? ¿Esto es, si se tiene el valor 29.7 y convertimos el tipo a **char**, el valor resultante es 30 o 29? La respuesta a esto puede verse en el ejemplo:

```
| //: c03:CastingNumbers.java
```

```

// Que sucede cuando convertimos un tipo float
// o double en un valor entero?
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 es que la respuesta de convertir el tipo **float** o **double** a un valor entero siempre trunca.

La segunda pregunta atañe a **Math.random()**. ¿Produce un valor de cero a uno, inclusive o el valor 1 no está incluido? ¿En lengua matemática, esto es  $(0,1)$ , o  $[0,1]$ , o  $(0,1]$  o  $[0,1)$ ? (Los paréntesis rectos significan “incluye” y los paréntesis significan “no incluye”) Nuevamente, un programa probara la respuesta:

```

//: c03:RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?
public class RandomBounds {
    static void usage() {
        System.out.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, escribiremos una de las dos siguientes líneas de comandos:

```
| java RandomBounds lower
  0
```

```
| java RandomBounds upper
```

En ambos casos estamos forzados a terminar al programa manualmente, así es que *aparecerá* que **Math.random()** nunca produce 0.0 o 1.0. Pero esto es un experimento que puede ser engañoso. Si consideramos<sup>27</sup> que hay alrededor de  $2^{62}$  diferentes fracciones del tipo double entre 0 y 1, la probabilidad de alcanzar cada uno de los valores experimentalmente excede el tiempo de vida de una computadora, o inclusive la de un experimentador. Entrega que 0.0 *esta* incluido en la salida de **Math.random()**. O lo que, en lenguaje matemático, es [0,1).

## Resumen

Con este capítulo concluye el estudio de los aspectos que aparecen en la mayoría de los lenguajes de programación: cálculos, precedencia, tipos de conversiones y selección e iteración. Ahora estamos listos para comenzar a dar pasos que nos acerquen al mundo de la programación orientada a objetos. El siguiente capítulo cubre importantes temas acerca de inicialización y limpieza de objetos, seguido del capítulo con el esencial concepto de implementación oculta.

<sup>27</sup> Chuck Allison escribió: El número total de números en un sistema de punto flotante es  $2(M-m+1)b^{(p-1)+1}$  donde **b** es la base (usualmente 2), **p** es la precisión (dígitos en la mantisa), **M** es el largo del exponente, y **m** es el exponente más pequeño. IEEE<sub>754</sub> utiliza: **M = 1023, m = -1022, p = 53, b = 2**

así es que el número total de números es

$$\begin{aligned}
 & 2(1023+1022+1)2^{52} \\
 & = 2((2^{10}-1)+(2^{10}-1))2^{52} \\
 & = (2^{10}-1)2^{54} \\
 & = 2^{64}-2^{54}
 \end{aligned}$$

La mitad de estos números (corresponden a exponentes en el rango [-1022,0]) que son menos que 1 en magnitud (ambos positivos y negativos), así es que 1/4 de esa expresión, o  $2^{62} - 2^{52} + 1$  (aproximadamente  $2^{62}$ ) está en el rango de [0,1). Vea mi trabajo en <http://www.freshsources.com/1995006a.htm> (al final del texto).

# Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Hay dos expresiones en la sección etiquetada “precedencia” al inicio de este capítulo. Coloque estas expresiones en un programa y demuestre que producen diferentes resultados.
2. Coloque los métodos `ternary()` y `alternative()` en un programa que trabaje.
3. De las secciones etiquetadas “if-else” y “return”, coloque los métodos `test()` y `test2()` en un programa que trabaje.
4. Escriba un programa que imprima valores desde uno a 100.
5. Modifique el ejercicio 4 para que el programa salga utilizando la palabra clave `break` con el valor 47. Trate de utilizar `return` en su lugar.
6. Escriba una función que tome dos argumentos `String`, y utilice todas las comparaciones `boolean` para comparar los dos `Strings` e imprima el resultado. Para el `==` y el `!=`, también realice la prueba con `equals()`. En el `main()`, llame su función con diferentes objetos `String`.
7. Escriba un programa que genere 25 números al azar enteros. Para cada valor, utilice una instrucción `if-then-else` para clasificar si es mayor, menor o igual a un segundo valor generado al azar.
8. Modifique el ejercicio 7 para que el código circule con un bucle `while` “infinito”. Debe ejecutarse hasta que se interrumpa con el teclado (típicamente presionando Control-C).
9. Escriba un programa que utilice dos bucles `for` anidados y el operador módulo (%) para detectar e imprimir números primos (números enteros que solo sean divisibles por si mismos y por la unidad).
10. Cree una instrucción `switch` que imprima un mensajes por cada `case`, y coloque el `switch` dentro de un bucle `for` que examine cada caso. Coloque un `break` luego de cada caso y pruébelo, luego quite los `breaks` y vea que sucede.

# 4: Inicialización y Limpieza

Así como la revolución de las computadoras progresó, la programación “insegura” se ha convertido en una de los mayores culpables que hace la programación costosa.

Dos de estos temas de seguridad son la *inicialización* y la *limpieza*. Mucho errores en C se suceden cuando el programador olvida inicializar una variable. Esto es especialmente verdadero con bibliotecas donde los usuarios no conocen cómo inicializar un componente de la biblioteca, o inclusive si deberían. La limpieza es un problema especial porque es fácil de olvidar un elemento cuando ha terminado con él, puesto que ya no interesa. De esta manera, los recursos utilizados por ese elemento son retenidos y se puede fácilmente quedarse sin recursos (el que más se hace notar, la memoria).

C++ introduce el concepto de *constructor*, un método especial que es automáticamente llamado cuando un objeto es creado. Java también ha adoptado el constructor, y además tiene un recolector de basura que automáticamente libera los recursos de memoria cuando no son utilizados más. Este capítulo examina los temas de inicialización y limpieza, y su soporte en Java.

## Inicialización garantida con el constructor

Podemos imaginarnos creando un método llamado **initialize()** para cada clase que escriba. El nombre es una insinuación debería ser llamado antes de utilizar un objeto. Desafortunadamente, esto significa que el usuario debe recordar llamar al método. En Java, el diseñador puede garantizar la inicialización de cada objeto proporcionando un método especial llamado *constructor*. Si una clase tiene un constructor, Java automáticamente llama el constructor cuando el objeto es creado, antes de que los usuarios puedan inclusive poner las manos en él. Así es que la inicialización es garantizada.

El siguiente desafío es cómo llamar a este método. Aquí hay dos temas. El primero es que cualquier nombre puede entrar en conflicto con un nombre

que se desee utilizar en un miembro de la clase. El segundo es que dado que el compilador es responsable por llamar el constructor, este debería siempre saber que método llamar. La solución de C++ parece la mas simple y lógica, así es realizado en Java también: el nombre del constructor es el mismo nombre que el de la clase. esto tiene sentido dado que el método será llamado automáticamente en la inicialización.

Aquí hay una clase simple con un constructor:

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.
class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}
public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} ///:~
```

Ahora cuando un objeto es creado:

```
| new Rock();
```

la memoria es asignada y el constructor es llamado. Esto garantiza que el objeto será propiamente inicializado antes que se pueda poner las manos encima de él.

Se puede ver que el estilo de codificación al colocar la primera letra de todos los métodos en minúscula no se aplica a los constructores, dado que el nombre del constructor debe corresponder con el nombre de la clase *exactamente*.

Como cualquier método, el constructor puede tener argumentos que permitan especificar *como* un objeto es creado. El siguiente ejemplo puede ser fácilmente cambiado para que el constructor tome un argumento.

```
//: c04:SimpleConstructor2.java
// Constructors can have arguments.
class Rock2 {
    Rock2(int i) {
        System.out.println(
            "Creating Rock number " + i);
    }
}
public class SimpleConstructor2 {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock2(i);
    }
} ///:~
```

Los argumentos del constructor proporcionan una manera de proporcionar parámetros a la inicialización de un objeto. Por ejemplo, si la clase **Tree** tiene un constructor que toma un único argumento entero indicando el tamaño del árbol, se puede crear objetos **Tree** como estos:

```
| Tree t = new Tree(12); // arbol de 12 pies  
Si Tree(int) es su único constructor, entonces el compilador no creará un objeto Tree de ninguna otra forma.
```

Los constructores eliminan una larga lista de problemas y hacen el código más fácil de leer. En el anterior fragmento de código, por ejemplo, no verá una llamada explícita a ningún método **initialize()** que esté conceptualmente separado de la definición. En Java, la definición y la inicialización son conceptos unificados-no se puede tener uno sin el otro.

El constructor es un tipo de método inusual porque no retorna ningún valor. Esto es claramente diferente de un valor de retorno **void**, en donde el método no retorna nada pero se sigue teniendo la opción de realizar algún tipo de retorno. Los constructores no retornan nada y no se tiene otra opción. Si hay un valor de retorno y se pudiera seleccionar uno propio, el compilador necesitaría de alguna forma saber qué hacer con ese valor de retorno.

## Sobrecarga de métodos

Una característica importante en cualquier lenguaje de programación es el uso de nombres. Cuando se crea un objeto, se le da un nombre a una región para almacenamiento. Un método es el nombre para una acción. Utilizando nombres para describir su sistema, se crea un programa que es fácil de entender y modificar para las personas. Es muy parecido a escribir en prosa-la meta es comunicarse con sus lectores.

Nos referimos a todos los objetos y métodos utilizando nombres. Nombres bien elegidos hacen más fácil para nosotros y otros entender el código.

Un problema se origina cuando se proyecta el concepto de matiz en el lenguaje humano en un lenguaje de programación. A menudo, la misma palabra tiene muchos significados-esto es *sobrecargar*. Esto es útil, especialmente cuando viene de diferencias triviales. Se dice “lava la camisa”, “lava el auto” y “lava el perro”. Sería tonto ser forzado a decir, “camizaLavar la camisa”, “autoLavar el auto” y “perroLavar el perro” simplemente porque el oyente no necesita realizar ninguna distinción acerca de la acción a realizar. La mayoría de los lenguajes humanos son redundantes, así es que si perdemos algunas palabras se seguirá determinando el significado. No necesitamos identificadores únicos-podemos deducir el significado del contexto.

Muchos lenguajes de programación (C en particular) requiere que tenga un único identificador para cada función. Así es que no se puede tener una función llamada **print()** para imprimir enteros y otra llamada **print()** para imprimir números de punto flotante-cada función requiere un único nombre.

En Jaca (y C++), otro factor fuerza la sobrecarga de nombres de métodos: el constructor. Dado que el nombre del constructor es predeterminado por el nombre de la clase, solo puede haber un nombre de constructor. ¿Pero que sucede si se quiere crear un objeto de mas de una forma? Por ejemplo, supongamos que se crea una clase que pueda inicializarse sola en una manera estándar o leyendo información desde un fichero. Necesitará dos constructores, uno que no tome argumentos (el constructor *por defecto*, también llamado constructor *no-arg*), y uno que tome un argumento del tipo **String**, que es el nombre del fichero con el cual se inicializará el objeto.

Ambos son constructores, así que deben tener el mismo nombre-el nombre de la clase. De esta manera, la *sobrecarga de métodos* esencial para permitir el mismo nombre de métodos para ser utilizado con diferentes tipos de argumentos. Y a pesar de que la sobrecarga de métodos es una condición necesaria para constructores, es de conveniencia general y puede ser utilizada con cualquier método.

He aquí un ejemplo que muestra dos constructores y dos métodos ordinarios sobrecargados:

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;
class Tree {
    int height;
    Tree() {
        prt("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        prt("Creating new Tree that is "
            + i + " feet tall");
        height = i;
    }
    void info() {
        prt("Tree is " + height
            + " feet tall");
    }
    void info(String s) {
        prt(s + ": Tree is "
            + height + " feet tall");
    }
    static void prt(String s) {
        System.out.println(s);
    }
}
```

```

    }
public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
} //:~

```

Un objeto **Tree** puede ser creado en un almácigo de una semilla, sin argumentos, o como una planta crecida en un criadero, con un tamaño. Para respaldar esto, hay dos constructores, uno que no toma argumentos (llamamos a los constructores que no toman argumentos *constructores por defecto*<sup>1</sup>) y uno que toma la altura existente.

Queremos también llamar al método **info()** en mas de una forma. Por ejemplo, con un argumento del tipo **String** si tenemos un mensaje extra que queremos imprimir, y sin necesidad de saber mas. Parece extraño dar dos nombres separados para lo que es obviamente el mismo concepto. Por fortuna, la sobrecarga de métodos permite utilizar el mismo nombre para ambos.

## Distinguiendo métodos sobrecargados

¿Si los métodos tienen el mismo nombre, como puede Java saber a que método nos referimos? Hay una regla simple: cada método sobrecargado debe tomar una única lista de tipos de argumentos.

Si pensamos en esto por unos segundos, tiene sentido: ¿como puede si no un programador indicar la diferencia entre dos métodos que tienen el mismo nombre que por el tipo de sus argumentos?

Incluso las diferencias en el orden de los argumentos son suficientes para distinguir dos métodos: (A pesar de que normalmente no se quiera tomar esta aproximación, dado que produce código difícil de mantener).

```

//: c04:OverloadingOrder.java
// Overloading based on the order of
// the arguments.
public class OverloadingOrder {
    static void print(String s, int i) {

```

---

<sup>1</sup> En alguna literatura de Java de Sun en lugar de llamarlos así se refieren con el desmañado pero descriptivo nombre de “constructores no-arg”. El término “constructor por defecto” ha estado en uso por muchos años y usaré ese.

```

        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("String first", 11);
        print(99, "Int first");
    }
} //:~

```

Los *dos* métodos **print()** tienen argumentos idénticos, pero el orden es diferente, y eso es lo que los hace distintos.

## Sobrecarga con primitivas

Una primitiva puede automáticamente promoverse de un tipo mas pequeño a uno mas grande y esto puede ser ligeramente confuso en combinación con la sobrecarga. El siguiente ejemplo muestra que sucede cuando una primitiva es manejada por un método sobrecargado:

```

//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.
public class PrimitiveOverloading {
    // boolean can't be automatically converted
    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 testConstVal() {
    prt("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}
void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
public static void main(String[] args) {
    PrimitiveOverloading p =
    new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
}

```

```

        p.testDouble();
    }
} //:~
```

Si vemos la salida de este programa, se puede ver que el valor constante 5 es tratado como un tipo **int**, así es que si un método sobrecargado esta disponible que tome un tipo **int** es utilizado. En los otros casos, si se tiene un tipo de datos que es mas pequeño que el argumento en el método, ese tipo de datos es promovido. El tipo **char** produce un efecto ligeramente diferente, dado que no encuentra una coincidencia exacta con un char, es promovido a el tipo **int**.

¿Que sucede si su argumento es *mayor* que el argumento esperado por el método sobrecargado? Una modificación al programa anterior nos da la respuesta:

```

//: c04:Demotion.java
// Demotion of primitives and overloading.
public class Demotion {
    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("double argument:");
        f1(x);f2((float)x);f3((long)x);f4((int)x);
        f5((short)x);f6((byte)x);f7((char)x);
```

```

    }
    public static void main(String[] args) {
        Demotion p = new Demotion();
        p.testDouble();
    }
} //:~

```

Aquí, los métodos toman valores primitivos más pequeños. Si el argumento es mayor entonces se deberá *convertir el tipo* a el necesario utilizando el nombre del tipo entre paréntesis. Si no hace esto, el compilador emitirá un mensaje de error.

Se debe ser consciente de que esto es una *conversión a menor*, que significa que se puede perder información durante la conversión. Esto es porque el compilador fuerza a hacerlo para indicar la conversión a menos.

## Sobrecarga de valores de retorno

Es común preguntarse “¿Por qué solo nombres de las clases y listas de argumentos de métodos? Por qué no distinguir métodos basándose en sus valores de retorno?” Por ejemplo, estos dos métodos, que tienen el mismo nombre y argumentos, son distinguidos fácilmente uno de otro:

```

void f() {}
int f() {}

```

Esto funciona correctamente cuando el compilador puede determinar sin equivocarse el significado del contexto, como en **int x = f()**. Sin embargo, se puede llamar a un método e ignorar el valor de retorno; esto es a menudo referido como *llamar a un método por su efecto secundario* (*calling a method for its side effect*) dado que no importa el valor de retorno pero en lugar de eso se quieren los otros efectos de la llamada al método. Así es que si se llama el método de esta forma:

```
f();
```

¿Cómo puede Java determinar que **f()** debe llamar? ¿Y cómo debería alguien que lee el código verlo? A causa de este tipo de problema, no se puede utilizar para distinguir métodos sobrecargados el tipo de valor de retorno.

## Constructores por defecto

Como se ha mencionado previamente, un constructor por defecto (también conocido como constructor “no-arg”) es el que no tiene argumentos, utilizado para crear un “objeto vainilla”. Si se crea una clase que no tiene constructores, el compilador automáticamente crea un constructor por defecto para usted. Por ejemplo:

```

//: c04:DefaultConstructor.java
class Bird {
    int i;
}

```

```

    }
public class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // default!
    }
} //:~

```

La línea

```
| new Bird();
```

crea un nuevo objeto y llama el constructor por defecto, aún cuando no haya uno definido explícitamente. Sin el no se tendrá método a llamar para crear el objeto. Sin embargo, si definimos alguno de los constructores (con o sin argumentos), el compilador *no* sintetizará por si solo.

```

class Bush {
    Bush(int i) {}
    Bush(double d) {}
}

```

Ahora si decimos:

```
| new Bush();
```

El compilador reclamará que no encuentra un constructor que corresponde. Es como cuando no se agrega ningún constructor, el compilador dice “*Esta obligado a tener algún constructor, así es que deje que yo haga uno por usted*”. Pero si escribe un constructor, el compilador dice “*Ha escrito un constructor así es que usted sabe lo que esta haciendo; si no coloca uno por defecto es porque da a entender que lo quiere afuera.*”

## La palabra clave **this**

Si se tiene dos objetos del mismo tipo llamados **a** y **b**, podemos preguntarnos como es que se puede llamar un método **f()** para ambos objetos:

```

class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);

```

¿Si hay solo un método llamado **f()**, como puede ese método saber de donde ha sido llamado, del objeto **a** o del **b**?

Para que se pueda escribir el código con una sintaxis orientada a objetos conveniente en donde se pueda “enviar un mensaje a un objeto”, el compilador hace algún trabajo clandestino para nosotros. Hay un primer argumento secreto pasado a el método **f()**, y este argumento es la referencia a el objeto que es manipulado. Así es que los dos métodos llamados anteriormente se transforman en algo como:

```

Banana.f(a,1);
Banana.f(b,2);

```

Esto es interno y no se puede escribir esta expresión y hacer que el compilador la acepte, pero da una idea de que es lo que está sucediendo.

Supongamos que está dentro de un método y queremos tomar la referencia del objeto actual. Dado que esta referencia es pasada *secretamente* por el compilador, no hay identificador para esta. Sin embargo, para este propósito hay una palabra clave **this**. Esta palabra clave -que puede ser utilizada solo dentro de un método- produce la referencia a el objeto que ha llamado al método. Se puede tratar esta referencia exactamente como otra referencia a objeto. Se debe tener presente que si estamos llamando a un método de la clase desde adentro de otro método de la clase, no necesita utilizar **this**; simplemente llamamos al método. La referencia actual **this** es automáticamente utilizada por el otro método. Entonces se puede decir:

```
class Apricot {  
    void pick() { /* ... */ }  
    void pit() { pick(); /* ... */ }  
}
```

Dentro de **pit()**, se puede decir **this.pick()** pero no es necesario. El compilador hace esto por nosotros automáticamente. La palabra clave **this** es utilizada solo para aquellos casos especiales en donde necesitamos explícitamente utilizar esa referencia para el objeto actual. Por ejemplo, a menudo es utilizada en instrucciones **return** cuando queremos retornar la referencia al objeto actual:

```
//: c04:Leaf.java  
// Simple use of the "this" keyword.  
public class Leaf {  
    int i = 0;  
    Leaf increment() {  
        i++;  
        return this;  
    }  
    void print() {  
        System.out.println("i = " + i);  
    }  
    public static void main(String[] args) {  
        Leaf x = new Leaf();  
        x.increment().increment().increment().print();  
    }  
} ///:~
```

Dado que **increment()** retorna la referencia al objeto actual mediante la palabra clave **this**, muchas operaciones pueden ser fácilmente realizadas en el mismo objeto.

## Llamando constructores desde constructores

Cuando escribimos varios constructores para una clase, hay veces que se quiere llamar un constructor desde otro para evitar código duplicado. Se puede hacer esto utilizando la palabra clave **this**.

Normalmente, cuando decimos **this**, es en el sentido de “este objeto” o de “el objeto actual”, y por si solo produce la referencia a el objeto actual. En un constructor, la palabra clave **this** tiene un significado diferente cuando le damos una lista de argumentos: este hace una llamada explícita a el constructor que se corresponde con la lista de argumentos. De esta manera se tiene una forma directa de llamar a otros constructores:

```
//: c04:Flower.java
// Calling constructors with "this."
public class Flower {
    int petalCount = 0;
    String s = new String("null");
    Flower(int petals) {
        petalCount = petals;
        System.out.println(
            "Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        System.out.println(
            "Constructor w/ String arg only, s=" + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
        //! this(s); // Can't call two!
        this.s = s; // Another use of "this"
        System.out.println("String & int args");
    }
    Flower() {
        this("hi", 47);
        System.out.println(
            "default constructor (no args)");
    }
    void print() {
        //! this(11); // Not inside non-constructor!
        System.out.println(
            "petalCount = " + petalCount + " s = " + s);
    }
    public static void main(String[] args) {
        Flower x = new Flower();
        x.print();
    }
} ///:~
```

El constructor **Flower(String s, int petals)** muestra, cuando mientras que se puede llamar un constructor utilizando **this**, no se pueden llamar dos. Además, la llamada al constructor debe ser la primer cosa que se haga o se obtendrá un mensaje de error del compilador.

Este ejemplo también muestra otra forma en que **this** es utilizado. Dado que el nombre del argumento **s** y el dato miembro **s** son el mismo, hay aquí una ambigüedad. Podemos resolverlo diciendo **this.s** para referirnos a el dato

miembro. A menudo se verá esta forma utilizada en código Java y en numerosos lugares en este libro.

En `print()` se puede ver que el compilador no dejara que se llame un constructor dentro de otro método que no sea el constructor.

### El significado de `static`

Con la palabra clave `this` en mente, puede entender completamente que significa hacer un método `static`. Esto significa que no hay `this` para un método particular. No puede llamar a un método que no es estático desde un método estático<sup>2</sup> (a pesar que la reversa es posible), y se puede llamar a un método estático para la clase misma, sin un objeto. De hecho, esto es para lo principal que un método estático es. Es como si se hubiera creado el equivalente a una función global (de C). Excepto que las funciones globales no son permitidas en Java, y colocar un método estático dentro de una clase permite acceder a otros métodos y campos estáticos.

Algunas personas discuten que los métodos estáticos no son orientados a objetos desde que tienen la semántica de una función global; con un método estático no se envía un mensaje a un objeto, dado que no hay `this`. Esto probablemente sea un argumento considerable, y si se encuentra que se está utilizando un *montón* de métodos estáticos probablemente haya que pensar una nueva estrategia. Sin embargo, `static` es pragmático y hay veces que se necesita verdaderamente, así es que si es o no “propriamente POO” se lo dejamos a los teóricos. Ciertamente, aún Smalltalk tiene el equivalente en sus “métodos de clases”.

## Limpieza: finalización y recolección de basura

Los programadores sabes acerca de la importancia de la inicialización, pero a menudo olvidan la importancia de la limpieza. ¿Después de todo, quien necesita limpiar un tipo `int`? Pero con las librerías, simplemente “dejar” un objeto una vez que ha terminado con él no es seguro. Claro, Java tiene el recolector de basura para reclamar la memoria de los objetos que ya no se utilizan. Ahora consideremos un caso muy inusual. Supóngase que un objeto asigna espacio de memoria “especial” sin utilizar `new`. El recolector de

---

<sup>2</sup> El único caso en que esto es posible ocurre cuando se pasa una referencia a un objeto dentro del método estático. Entonces, por medio de la referencia (que no es efectivamente `this`), se puede llamar métodos no estático y acceder a campos que no son estáticos. Pero típicamente si quiere hacer algo como eso simplemente cree un método no estático común.

basura solo sabe como liberar memoria asignada *con new*, así es que no sabe como liberar la memoria “especial” del objeto. Para manejar este caso, Java proporciona un método llamado **finalize()** que se puede definir en una clase. He aquí como *supuestamente* funciona. Cuando el recolector de basura está pronto para liberar el espacio asignado por el objeto, primero llamará a **finalize()**, y solo en la siguiente recolección reclamará la memoria del objeto. Así es que si se elige utilizar **finalize()**, dará la habilidad de realizar importantes tareas de limpieza *en el momento de la recolección de basura*.

Esto es una pequeña dificultad porque algunos programadores,, especialmente los programadores C++, pueden en un principio confundir **finalize()** con el *destructoren* C++, que es una función que es llamada siempre cuando un objeto es destruido. Pero es importante distinguir entre C++ y Java aquí, dado que en C++ *los objetos siempre son destruidos* (en un programa libre de errores), mientras que los no siempre se recoge la basura de los objetos Java. O, puesto de otra forma:

### La recolección de basura no es destrucción

Si se recuerda esto, se estará fuera de problema. Lo que esto significa es que si hay alguna actividad que deba ser realizada antes de que no se necesite mas un objeto, se debe realizar la actividad por uno mismo. Java no tiene destructores o conceptos similares, así es que debe crear un método común para realizar esta limpieza. Por ejemplo, supongamos que en el proceso de crear su objeto dibuja en la pantalla por si solo. Si no se borra esa imagen de la pantalla explícitamente, nunca lo borrará. Si se coloca algún tipo de funcionalidad dentro de **finalize()**, entonces si un objeto es recogido por el recolector de basura, la imagen será eliminada primero de la pantalla, pero si no se hace la imagen quedara. Así es que un segundo punto para recordar es:

### Sus objetos pueden no ser recogidos por el recolector.

Se puede que el espacio asignado a un objeto nunca es liberado porque el programa nunca se acerca a el punto de agotarse el almacenamiento. Si un programa termina y el recolector nunca llegó a recoger el resto del espacio para alguno de los objetos, esa memoria será retornada al sistema operativo *en grupo* cuando el programa termina. Esto es una cosa buena, porque la recolección de basura tiene costos operativos, y si nunca lo hacemos nunca incurrimos en ese gasto.

## ¿Para que es **finalize()**?

En este momento se puede pensar que nunca se utilizará **finalize()** como método de propósito general para limpieza. ¿Que tan bueno es esto?

Un tercer punto a recordar es este:

### La recolección de basura es solo acerca de memoria.

Esto es, la única razón para que exista el recolector de basura es para recuperar memoria que el programa no utilizará mas. Así es que cualquier actividad que este asociada con la recolección de basura, mucho mas el método **finalize()**, debe ser solo acerca de memoria y su eliminación del espacio asignado.

¿Esto no significa que si un objeto contiene otros objetos **finalize()** deberá explícitamente liberar estos objetos? Bueno, no-el recolector de basura se encarga de liberar todos los objetos de la memoria independientemente de como es creado el objeto. El limpia lo que necesita por lo que **finalize()** es limitado a casos especiales, en donde su objeto puede asignar algún espacio en otra forma que creando un objeto. ¿Pero como se puede observar, todo en Java es un objeto así es que como puede ser esto?

Parecería que **finalize()** esta en su sitio por la posibilidad de que se haga algo al estilo de C para asignar memoria utilizando otro mecanismo que el normal en Java. Esto puede en un principio suceder cuando se utilizan *métodos nativos* que son una forma de llamar código que no es Java desde Java (Los métodos nativos son discutidos en el apéndice B). C y C++ son los únicos lenguajes soportados actualmente por métodos nativos, pero dado que se puede llamar a subprogramas en otros lenguajes, puede efectivamente llamar cualquier cosa. Dentro de código que no es Java, la familia de funciones **malloc()** de C pueden ser llamadas para asignar memoria, y a no ser que llame a **free()** esa memoria asignada no será liberada, causando una laguna de memoria. Claro, **free()** es una función de C y C++, así es que necesita llamar un método nativo dentro de su método **finalize()**.

Luego de leer esto, probablemente se tenga la idea de que no se utilizará **finalize()** mucho. Y es cierto; no es el lugar para que se suceda una limpieza normal. ¿Así es que donde debe realizarse la limpieza normal?

## Se debe realizar limpieza

Para limpiar un objeto, el usuario de un objeto debe llamar un método para limpiarlo en el momento que se deseé. Esto suena muy directo, pero choca

un poco con el concepto de destructor de C++. En C++, todos los objetos son destruidos. O mejor dicho, todos los objetos *deberían de ser destruidos*. Si el objeto C++ es creado como local (i.e. en la pila-no es posible en Java), la destrucción sucede cuando se cierra la llave del alcance en donde el objeto fue creado. Si el objeto fue creado utilizando **new** (como en Java) el destructor es llamado cuando el programador llama a el operador **delete** de C++ (que no existe en Java). Si el programador C++ olvida llamar a **delete**, el destructor nunca es llamado y tenemos una laguna de memoria, mas la otra parte del objeto que nunca es limpiada. Este tipo de error puede ser muy difícil de encontrar.

En contraste, Java no permite que se creen objetos locales-siempre se tiene que utilizar **new**. Pero en Java, no hay "delete" a llamar para liberar el objeto dado que el recolector de basura libera la memoria asignada por nosotros. Así es que desde un punto de vista simplista puede decir que a causa del recolector de basura, Java no tiene destructor. Se verá cuando este libro progrese, sin embargo, que la presencia del recolector de basura no elimina la necesidad o utilidad de los destructores (y nunca se debería llamar **finalize()** directamente, así es que no es un camino apropiado para una solución). Si quiere realizar algún tipo de limpieza además que la de liberación de memoria debe llamar explícitamente a un método apropiado en Java, que es el equivalente a el destructor de C++ sin la comodidad.

Una de las cosas para las que **finalize()** puede ser útil es para observar el proceso de recolección de basura. El siguiente ejemplo muestra que está sucediendo y resume las descripciones previas de la recolección de basura:

```
//: c04:Garbage.java
// Demonstration of the garbage
// collector and finalization
class Chair {
    static boolean gcrun = false;
    static boolean f = false;
    static int created = 0;
    static int finalized = 0;
    int i;
    Chair() {
        i = ++created;
        if(created == 47)
            System.out.println("Creada la 47");
    }
    public void finalize() {
        if(!gcrun) {
            // La primera vez que finalize() es llamado:
            gcrun = true;
            System.out.println(
                "Comenzando a finalizar antes de que " +
                created + " Chairs hayan sido creadas");
        }
        if(i == 47) {
            System.out.println(
```

```

        "finalizando la silla #47, levantando una " +
        " bandera para parar la creación de sillas ");
        f = true;
    }
    finalized++;
    if(finalized >= created)
        System.out.println(
        "Todas las " + finalized + " finalizadas");
    }
}
public class Garbage {
    public static void main(String[] args) {
        // Si la bandera no ha sido configurada,
        // haga Chairs y Strings:
        while(!Chair.f) {
            new Chair();
            new String("Para tomar espacio ");
        }
        System.out.println(
        "Luego de que todas las Chairs han sido creadas:\n" +
        "total creadas = " + Chair.created +
        ", total terminadas = " + Chair.finalized);
        // Argumentos opcionales fuerzan la recolección
        // de basura y finalización:
        if(args.length > 0) {
            if(args[0].equals("gc") ||
            args[0].equals("all")) {
                System.out.println("gc():");
                System.gc();
            }
            if(args[0].equals("finalize") ||
            args[0].equals("all")) {
                System.out.println("runFinalization():");
                System.runFinalization();
            }
        }
        System.out.println("Chau! ");
    }
} //:~
}

```

El programa anterior crea muchos objetos **Chair**, y en algún punto luego de que el recolector de basura comienza a correr, el programa para de crear **Chairs**. Puesto que el recolector de basura no puede ejecutarse en cualquier momento, no se conoce exactamente cuando comenzará, así es que hay una bandera llamada **gcrun** para indicar cuando el recolector de basura ha comenzado a funcionar ya. Una segunda bandera **f** es una forma de que **Chair** le diga al bucle de **main()** que debe detenerse de crear objetos. Ambas banderas son configuradas con **finalize()**, que es llamada durante la recolección de basura.

Dos variables estáticas mas, **created** y **finalized**, mantienen el rastro del número de **Chairs** creados contra el número que ha sido terminado por el recolector de basura. Finalmente, cada **Chair** tiene su propia (no estático)

variable **i** de tipo **int** que indica que número es. Cuando el objeto **Chair** numero 47 es finalizado, la bandera es puesta en **true** para detener el proceso de creación de **Chair**.

Todo esto sucede en el **main()**, en el bucle

```
while(!Chair.f) {  
    new Chair();  
    new String("To take up space");  
}
```

Uno se puede preguntar como este bucle puede terminar, dado que no hay nada dentro del bucle que cambie el valor de **Chair.f**. Sin embargo el proceso **finalize()** puede, eventualmente, cuando termina con el número 47.

La creación de un objeto **String** durante cada iteración es simplemente un espacio extra asignado para estimular al recolector de basura a llevárselo, cuando comience a ponerse nervioso por la cantidad de memoria disponible.

Cuando se corra el programa, se debe proporcionar un argumento en la línea de programa que puede ser “gc”, “finalize”, o “all”. El argumento “gc” llamará a el método **System.gc()** (para forzar la ejecución del recolector de basura). Utilizando el argumento “finalize” se llama a

**System.runFinalization()** que-en teoría-que cualquier objeto no finalizado sea finalizado. Y “all” que produce que ambos métodos sean llamados.

El comportamiento de este program y de la versión de la primera edición de este libro muestra que todos los temas que involucran la recolección de basura y la finalización, con mucho de la evolución que sucede puertas adentro. De hecho, en el momento en que se lea esto, el comportamiento del programa puede haber cambiado una vez mas.

Si **System.gc()** es llamado, la finalización sucede para todos los objetos. Esto no es necesariamente el caso con implementaciones previas del JDK, a pesar que la documentación afirme otra cosa. Además, se verá que no parece hacer ninguna diferencia si **System.runFinalization()** es llamado.

Sin embargo, se verá que solo si **System.gc()** es llamado luego que todos los objetos son creados y descartados serán todos los **finalize()** llamados. Si no llamamos **System.gc()**, solo algunos de los objetos serán finalizados. En Java 1.1, un método llamado **System.runFinalizersOnExit()** fue introducido que causa que los programas ejecuten todos los **finalized()** cuando salen, pero el diseño lo descarto siendo desaprobado. Esto es otra pista de que los diseñadores de Java se están dando una paliza para solucionar el problema de recolección de basura y finalización. Solo podemos esperar que las cosas hayan sido terminadas en Java 2.

El siguiente programa muestra que la promesa que los **finalize()** se ejecutarán siempre, pero solo si explícitamente se fuerza a que suceda. Si no causamos que **System.gc()** sea llamado, obtendremos una salida como esta:

```

Created 47
Beginning to finalize after 3486 Chairs have been
created
Finalizing Chair #47, Setting flag to stop Chair
creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!

```

De esta forma, no todos los finalizadores son llamado en el momento que el programa se completa. Si **system.gc()** es llamado, finalizará y destruirá todos los objetos que no se utilizan mas en ese momento.

Se debe recordar que ni la recolección de basura ni la finalización es garantida. En la máquina virtual de Java (JVM) no esta cerca de quedarse sin memoria, entonces no desperdiciará (sabiamente) tiempo recuperando memoria a través del recolección de basura.

## La condición de muerto

En general, no se podrá confiar en que **finalize()** sea llamado, y se debe crear funciones de limpieza separadas y llamarlas explícitamente. Así es que parece que **finalize()** es solo para incomprensibles limpiezas de memoria que la mayoría de los programadores nunca utilizaran. Sin embargo, hay un uso muy interesante de **finalize()** en el que no hay que confiarse en que sea llamado cada vez. Esto es la verificación de *condición de muerto*<sup>3</sup> de un objeto.

En el momento que no estamos mas interesados en un objeto-cuando esta listo para ser limpiado-ese objeto estará en un estado donde su memoria puede ser liberada de forma segura. Por ejemplo, si el objeto representa un fichero abierto, ese fichero debe ser cerrado por el programador antes que el objeto sea recogido por el basurero. Si algunas partes del objeto no son limpiadas adecuadamente, entonces se tendrá un error en el programa que será muy difícil de encontrar. El valor de **finalize()** es que puede ser utilizado para descubrir esta condición, incluso si no es llamado siempre. Si una de las finalizaciones sucede para revelar el error, entonces se descubrirá el problema, que es lo que realmente importa.

Hay aquí un simple ejemplo de como se debe utilizar:

```

//: c04:DeathCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.
class Book {
    boolean checkedOut = false;
    Book(boolean checkOut) {

```

---

<sup>3</sup> Término acuñado por Bill Venners ([www.artima.com](http://www.artima.com)) durante un seminario que dimos juntos.

```

        checkedOut = checkOut;
    }
    void checkIn() {
        checkedOut = false;
    }
    public void finalize() {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}
public class DeathCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Proper cleanup:
        novel.checkIn();
        // Drop the reference, forget to clean up:
        new Book(true);
        // Force garbage collection & finalization:
        System.gc();
    }
} //:~

```

La condición de muerto es cuando todos los objetos **Book** han sido registrados luego de que son recolectados, pero en **main()** un error del programador no verifica en uno de los libros. Sin **finalize()** para verificar la condición de muerto, es un error muy difícil de encontrar.

Debe notarse que **System.gc()** es utilizado para forzar la finalización (y se debe hacer esto durante el desarrollo del programa para acelerar la depuración). Pero aun si no es muy probable que el **Book** errante sea eventualmente descubierto mediante repetidas ejecuciones del programa (asumiendo que el programa asigna suficiente espacio para causar una recolección de basura).

## Como funciona el recolector de basura

Si se esta acostumbrado a un lenguaje de programación donde ubicar objetos en el heap es costoso, se debe asumir naturalmente que el esquema de Java de ubicar todo (excepto primitivas) en el heap es costoso.

Sin embargo, el recolector de basura puede tener un impacto en el *aumento* de la velocidad de la creación de objetos. Esto puede sonar un poco curioso al principio-que la liberación de espacio afecte el asignar espacio-pero es la forma en que algunos JVMs trabajan y esto significa que asignar espacio para objetos en el heap en Java puede ser apenas mas rápido que crear espacio en la pila en otros lenguajes.

Por ejemplo, podemos pensar que el heap en C++ es un patio donde cada objeto vigila su propio pedazo de pasto. Este estado real puede comenzar a

ser abandonado un tiempo mas tarde y puede ser reutilizado. En algunas JVMs, el heap de Java es un poco diferente; es mas como una cinta transportadora que mueve hacia adelante cada ves que ubica un nuevo objeto. Esto significa que el espacio asignado es notablemente rápido. El puntero del heap es simplemente movido adelante en un territorio virgen, así es que efectivamente es lo mismo que la asignación de espacio en el stack de C++ (Claro que, hay un costo operativo extra para arreglar cuentas pero no es nada como realizar una búsqueda para asignar espacio).

Ahora se observa que el heap no es una cinta transportadora, y si se trata de esa forma, eventualmente se paginará un montón de memoria (que es un gran acierto de rendimiento) y luego se acabará. El truco es que mientras que el recolector de basura interviene y se compactan los objetos en el heap así que efectivamente se esta moviendo el puntero del heap mas cerca del comienzo de la cinta transportadora y lejos de una violación de paginación. El recolector de basura arregla las cosas y hace posible con altas velocidades, un modelo de heap infinitamente libre para ser utilizado para asignar memoria.

Para entender como trabaja esto, se necesita tener una pequeña mejor idea de la forma en que los distintos recolectores de basura trabajan. Una técnica simple pero lenta es la de conteo de referencias. Esto significa que cada objeto contiene un contador de referencias, y cada vez que una referencia es ligada a un objeto el contador de referencia es incrementado. Cada vez que una referencia cae fuera del alcance o es puesta a **null**, se el contador de referencias es disminuido. Este manejo de contadores de referencias es un pequeño pero constante costo que se sucede a lo largo del tiempo de vida del programa. La recolección de basura se mueve a través de la lista completa de objetos y cuando encuentra uno con un contador de referencia en cero libera la memoria. El inconveniente es que si los objetos se refieren de forma circular a cada uno de los otros estos pueden tener contadores de referencia que no sean cero aún siendo basura. Encontrar grupos que tengan referencias automáticas requiere trabajo extra para el recolector de basura. Los contadores de referencia son comúnmente utilizados para explicar un tipo de recolección de basura pero no parece ser utilizado en ninguna implementación de JVM.

En esquemas mas rápidos, la recolección de basura no esta basada en contadores de referencias. En lugar de eso, está basada en la idea de que cualquier objeto no muerto debe ser al fin de cuentas ser fácil de seguir para atrás hasta encontrar una referencia que viva en la pila o en una asignación estática. La cadena puede ir a través de varias capas de objetos. Esto, si se comienza en la pila y en el área de almacenamiento estático y se camina a través de las referencias se encontrarán todos los objetos vivos. Para cada referencia que se encuentre, debe trazar dentro de el objeto que apunta y seguir todas las referencias de ese objeto, buscando dentro de los objetos a

los cuales apunta, etc. hasta que se haya movido a través de toda la telaraña generada con las referencias a la pila o almacenamiento estático. Cada objeto por el que se mueve debe estar vivo. Se puede ver que no hay problemas con grupos desprendidos auto referenciados-estos simplemente no se encuentran, y con consiguiente son automáticamente recolectados.

En la estrategia descrita aquí, la JVM utiliza un esquema de recolección de basura *adaptable* y lo que hace con los objetos es localizarlos dependiendo de la variante utilizada en ese momento. Una de estas variantes es *stop-and-copy*. Esto significa que-por razones que se hacen evidentes mas adelante-el programa es primero detenido (esto no es un esquema recolección en segundo plano). Entonces, cada objeto vivo que es encontrado es copiado de un heap a otro, dejando atrás toda la basura. Además, como los objetos son copiados dentro del nuevo heap son empaquetados final con final, esto es compactados el nuevo heap (y permitiendo almacenar espacio simplemente juntándolo al final como fue descrito anteriormente).

Claro que, cuando un objeto es movido de un lugar a toro, todas las referencias que apuntan a el (i.e. esa *referencia*) objeto debe ser cambiada. La referencia que va desde el heap o desde el área de almacenamiento estático a el objeto pueden ser cambiadas correctamente, pero puede haber otras referencias que apuntan a este objeto que serán encontradas mas tarde durante la "marcha". Estas son corregidas cuando son encontradas (imaginemos una tabla que relaciona las viejas direcciones con las nuevas).

Hay aquí dos temas que hace estos "recolectores de copias" inefficientes. El primero es la idea de que se tienen dos heaps y se quita de un lugar toda la memoria adelante y atrás entre estos dos heap separados, manteniendo mucha mas memoria de la que realmente se necesita. Algunos JVMs negocian con esto asignando el heap en trozos mientras se necesita y simplemente copiando de un trozo a otro. El segundo tema es la copia. Una vez que el programa comienza a estabilizarse puede generar poco o nada de basura. A pesar de eso, un recolector que copie puede seguir copiando la memoria de un lugar a otro, lo que es un despilfarro. Para prevenir esto, algunos JVMs detectan que no hay basura que se este generando y cambian a otro esquema diferente (esta es la parte que se "adapta"). Este otro esquema es llamado *mark and sweep* y es lo que las primeras versiones de JVM de Sun utilizaban todo el tiempo. Para uso general, mark and sweep es bastante lento, pero cuando se sabe que estamos generando poco o nada de basura es rápido.

Este sigue la misma lógica de comenzar desde la pila y del almacenamiento estático y rastrear a través de todas las referencias para encontrar objetos vivos. Sin embargo, cada vez que se encuentra un objeto vivo es marcado para colocar una bandera en el, pero el objeto no es recolectado todavía. Solo cuando el proceso de marca es terminado es cuando la barrida ocurre. Durante la barrida, los objetos muertos son liberados. Sin embargo, no

sucede ninguna copia, así si el recolector elige compactar un heap fragmentado deshaciéndose de los objetos dispersos.

La “stop-and-copy” se refiere a la idea de que este tipo de recolección *no* se realiza en segundo plano; en lugar de esto, el programa es detenido cuando la recolección de basura se sucede. En la literatura de Sun se puede encontrar muchas referencias a la recolección de basura como un proceso en segundo plano con prioridad baja, pero resulta que la recolección de basura no esta implementada de esta forma, siquiera en las primeras versiones de JVM de Sun. En lugar de eso el recolector de basura de Sun se ejecuta cuando la memoria esta baja. Además, mark-and-sweep requiere que el programa sea detenido.

Como ha sido mencionado, en la JVM descrita aquí la memoria es asignada en grandes bloques. Si se ubica un objeto grande, tendrá su propio bloque. Un riguroso stop-and-copy se requiere para copiar todo objeto vivo del heap fuente a el nuevo heap después de que se pueda liberar el viejo, lo que traslada un montón de memoria. Con bloques, el recolector de basura puede utilizar bloques muertos para copiar objetos que recolecta. Cada bloque tiene un *contador de generación* para mantener la pista de si esta vivo. En un caso normal, solo los bloques creados desde la última recolección de basura son compactados; todos los demás bloques adquieren su contador de generación si tienen están referenciados desde algún lado. Esto maneja los casos normales de una gran cantidad de objetos temporales de vida corta. Periódicamente, un barrido total es hecho—los objetos grandes no se copian a pesar de todo (solo obtienen su contador de generación) y los bloques que contienen objetos pequeños son copiados y compactados. La JVM realiza un monitoreo de la eficiencia del recolector de basura y si comienza a gastar mucho tiempo porque todos los objetos tienen vidas largas se cambia a mark-and-sweep. De forma similar, la JVM mantiene un rastro de que tan exitoso es mark-and-sweep y si el heap comienza a estar fragmentado regresa a stop-and-copy. Esto es donde la parte que se “adapta” entra, así es que como para resumir; “generación que se adapta stop-and-copy mark-and-sweep”.

Hay un gran número de mejoras en la velocidad posibles en JVM. Una especialmente importante involucra la operación del cargador y el compilador Just-In-Time (JIT). Cuando una clase debe ser cargada (típicamente, la primera vez que se quiere crear un objeto de esa clase), el fichero **.class** es localizado y el código de bytes para esa clase es traído a memoria. En ese momento, una estrategia es simplemente compilar JIT todo, pero esto tiene dos inconvenientes: toma un poco mas de tiempo, que combinado a través de la vida del programa se puede sumar; y se incrementa el tamaño del ejecutable (los códigos de bytes son significativamente mas compactos que el código JIT) y esto puede causar paginado, lo que definitivamente hace el programa mas lento. Una

estrategia alternativa es una *lazy evaluation* (*evaluación perezosa*) que significa que el código no es compilado JIT hasta que es necesario. Este código, que nunca es ejecutado puede nunca ser compilado JIT.

# Inicialización de miembros

Java deja de utilizar esta forma para garantizar que las variables sean inicializadas antes de ser utilizadas. En el caso de las variables que son definidas localmente para un método, esta garantía se presenta de la forma de un error en tiempo de ejecución. Así es que se escribe:

```
void f() {  
    int i;  
    i++;  
}
```

se obtendrá un mensaje de error que indica que **i** puede no haber sido inicializada. Claro, el compilador puede darle a **i** un valor por defecto, pero es mas probable que sea un error del programador y un valor por defecto lo encubriría. Forzando al programador a proporcionar un valor de inicialización hace que sea mas probable de encontrar el error.

Sin embargo, si una primitiva es un dato miembro de una clase,, las cosas son un poco diferentes. Dado que cualquier método puede inicializar o utilizar ese dato, tal vez no sea práctico forzar a el usuario a inicializarlo a un valor apropiado antes de que el dato se utilizado. Sin embargo, no es seguro dejarlo con valor de basura, así es que se garantiza que cada dato miembro primitivo de una clase tenga un valor inicial. Esos valores se pueden entender aquí:

```
//: c04:InitialValues.java  
// Muestra los valores iniciales por defecto.  
class Measurement {  
    boolean t;  
    char c;  
    byte b;  
    short s;  
    int i;  
    long l;  
    float f;  
    double d;  
    void print() {  
        System.out.println(  
            "Valor inicial de los tipo de datos\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 InitialValues {
    public static void main(String[] args) {
        Measurement d = new Measurement();
        d.print();
        /* En este caso se puede también decir:
        new Measurement().print();
        */
    }
} //:~

```

La salida de este programa es:

```

Valor inicial de los tipo de datos
boolean false
char [ ] 0
byte 0
short 0
int 0
long 0
float 0.0
double 0.0

```

El valor de **char** es cero, lo que es impreso como un espacio.

Se verá mas adelante que cuando se define una referencia a un objeto dentro de una clase sin inicializar a un nuevo objeto, esa referencia se le da un valor especial **null** (que es una palabra clave de Java).

Se puede ver que a pesar que los valores no son especificados, automáticamente son inicializados. Así es que al menos aquí no hay amenaza de trabajar con variables sin iniciar.

## Inicialización específica

¿Que sucede si se quiere dar un valor inicial a una variable? Una forma directa de hacer esto es simplemente asignar el valor en el momento en que se define la variable en la clase (Obsérvese que esto no se puede hacer en C++, a pesar de que todos los principiantes de C++ lo intentan siempre). Aquí las definiciones de campos en la clase **Measurement** son cambiados para proporcionar valores iniciales:

```

class Measurement {
    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;
    ...
}

```

También se puede inicializar objetos no primitivos de la misma forma. Si **Depth** es una clase, se puede insertar una variable e inicializarla si se quiere:

```
class Measurement {  
    Depth o = new Depth();  
    boolean b = true;  
    // ...
```

Si no se da a o un valor inicial y se intenta utilizar igualmente, se obtendrá un error en tiempo de ejecución llamado *excepción* (Cubierto en el capítulo 10).

También se puede llamar a un método para proporcionar un valor de inicialización:

```
class CInit {  
    int i = f();  
    //...  
}
```

Este método puede tener argumentos, por su puesto, pero estos argumentos no pueden ser otra clase de miembros que no hayan sido inicializados todavía. De esta forma, se puede hacer esto:

```
class CInit {  
    int i = f();  
    int j = g(i);  
    //...  
}
```

Pero no se puede hacer esto:

```
class CInit {  
    int j = g(i);  
    int i = f();  
    //...  
}
```

Este es un lugar donde el compilador, apropiadamente *se queja* de una referencia inversa, dado que cuanta con el orden de inicialización y no con la forma en la cual el programa es compilado.

Esta estrategia de inicialización es simple y directa. Tiene la limitación de que *cada* objeto del tipo **Measurement** tendrá el mismo valor de inicialización. A veces esto es exactamente lo que se necesita, pero en otros momentos se necesita mas flexibilidad.

## Inicialización en constructor

El constructor puede ser utilizado para realizar inicializaciones, y esto nos da una gran flexibilidad es la programación dado que podemos llamar métodos y realizar acciones en tiempo de ejecución para determinar valores iniciales. Esto es algo para tener en mente, sin embargo: no se puede evitar la inicialización automática, que se sucede luego de que se entra en el constructor. Así es que, por ejemplo:

```

class Counter {
    int i;
    Counter() { i = 7; }
    // ...

```

entonces `i` primero será inicializado a 0, luego a 7. Esto es cierto con todos los tipos primitivos y las referencias a objetos, incluyendo aquellos a los que se les da una inicialización explícita en el momento de la definición. Por esta razón, el compilador no trata de forzar la inicialización de elementos en el constructor en ningún lugar en particular, o antes de ser utilizado - la inicialización ya está garantida<sup>4</sup>.

## Orden de inicialización

Dentro de una clase, el orden de inicialización está determinado por el orden en que las variables están definidas dentro de la clase. Las definiciones de variables pueden ser desparcados a través y entre las definiciones de métodos, pero las variables son inicializadas antes que cualquier método sea llamado - incluso el constructor. Por ejemplo:

```

//: c04:OrderOfInitialization.java
// Demuestra el orden de inicialización.
// Cuando el constructor es llamado a crear el
// objeto Tag, se verá un mensaje:
class Tag {
    Tag(int marker) {
        System.out.println("Tag(" + marker + ")");
    }
}
class Card {
    Tag t1 = new Tag(1); // Before constructor
    Card() {
        // Indicate we're in the constructor:
        System.out.println("Card()");
        t3 = new Tag(33); // Reinitialize t3
    }
    Tag t2 = new Tag(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Tag t3 = new Tag(3); // At end
}
public class OrderOfInitialization {
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Shows that construction is done
    }
}

```

---

<sup>4</sup> En contraste, C++ tiene la *lista de inicialización de constructor* que hace que la inicialización se suceda antes de entrar en el cuerpo del constructor, y es forzado para los objetos. Vea *Thinking in C++, 2<sup>da</sup> edición* (disponible en este CD ROM y en [www.BruceEckel.com](http://www.BruceEckel.com)).

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

En **Card**, las definiciones de los objetos **Tag** son intencionalmente desparramadas para probar que no son inicializadas después de que se entra al constructor o cualquier cosa se suceda. Además, **t3** es inicializada nuevamente dentro del constructor. La salida es:

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

De esta forma, la referencia **t3** es inicializada dos veces, una antes y una después de la llamada al constructor (El primer objeto es descartado, así puede ser recolectado mas tarde). Esto parece no ser muy eficiente al principio, pero garantiza la inicialización correcta - ¿Que sucedería si se definiera un constructor sobrecargado donde no se inicializa **t3** y donde no hay una inicialización por “defecto” para **t3** en su definición?

## Inicialización de datos estáticos

Cuando el dato es del tipo **static** la misma cosa sucede; si es una primitiva y no se inicializa, toma el valor inicial estándar para la primitiva. Si es una referencia a un objeto, toma el valor **null** hasta que se crea un nuevo objeto y la referencia se conecta a él.

Si queremos colocar la inicialización en el punto de la definición, se verá igual que para los tipos no estáticos. Hay una sola lugar para los datos estáticos, sin importar de cuantos objetos son creados. Pero la pregunta que surge es cuando el espacio asignado es inicializado. Un ejemplo aclara esta pregunta:

```
//: c04:StaticInitialization.java
// Especificando valores iniciales en una
// definición de clase.
class Bowl {
    Bowl(int marker) {
        System.out.println("Taza( " + marker + " )");
    }
    void f(int marker) {
        System.out.println("f( " + marker + " )");
    }
}
class Table {
    static Bowl b1 = new Bowl(1);
    Table() {
        System.out.println("Mesa()");
        b2.f(1);
    }
}
void f2(int marker) {
    System.out.println("f2( " + marker + " )");
```

```

        }
        static Bowl b2 = new Bowl(2);
    }
    class Cupboard {
        Bowl b3 = new Bowl(3);
        static Bowl b4 = new Bowl(4);
        Cupboard() {
            System.out.println("Aparador");
            b4.f(2);
        }
        void f3(int marker) {
            System.out.println("f3(" + marker + ")");
        }
        static Bowl b5 = new Bowl(5);
    }
    public class StaticInitialization {
        public static void main(String[] args) {
            System.out.println(
                "Creando un nuevo aparador en el principal");
            new Cupboard();
            System.out.println(
                "Creando un nuevo aparador en el principal");
            new Cupboard();
            t2.f2(1);
            t3.f3(1);
        }
        static Table t2 = new Table();
        static Cupboard t3 = new Cupboard();
    } //:~
}

```

**Bowl** permite ver la creación de la clase, y **Table** y **Cupboard** crea miembros estáticos de **Bowl** desparramados entre las definiciones de clase. Se puede ver que **Cupboard** crea un **Bowl** no estático **b3** antes de las definiciones estáticas. La salida muestra que sucede:

```

Taza(1)
Taza(2)
Mesa()
f(1)
Taza(4)
Taza(5)
Taza(3)
Aparador
f(2)
Creando un nuevo aparador en el principal
Taza(3)
Aparador
f(2)
Creando un nuevo aparador en el principal
Taza(3)
Aparador
f(2)
f2(1)
f3(1)

```

La inicialización estática ocurre solo si es necesaria. Si no se crea un objeto **Table** y nunca se hace referencia a **Table.b1** o a **Table.b2**, el objeto estático **Bowl b1** y **b2** nunca serán creados. Sin embargo, son inicializadas solo cuando el *primer objeto Table* es creado (o el primer acceso estático ocurre). Luego de eso, los objetos estáticos no son iniciados nuevamente.

Por orden de inicialización los estáticos son primero, si ya fueron definidos por un objeto creado previamente, y segundo los objetos que no son estáticos. Se puede ver la evidencia de esto en la salida.

Es útil realizar un resumen del proceso de creación de un objeto. Considerando una clase llamada **Dog**:

1. La primera vez que un objeto del tipo **Dog** es creado, o la primera vez que un método estático o campo estático de una clase **Dog** es accedido, el intérprete Java debe localizar el fichero **Dog.class**, lo que hace buscando a través de la ruta a las clases.
2. Inmediatamente que la clase **Dog.class** es leída (creando un objeto **Class**, del que se aprenderá más tarde), todos los inicializadores estáticos son ejecutados. De esta forma, la inicialización de los datos estáticos se realiza solo una vez, cuando el objeto **Class** es leído por primera vez.
3. Cuando se crea una clase **Dog** mediante la palabra clave **new**, el proceso de construcción para un objeto **Dog** primero asigna la memoria suficiente para un objeto **Dog** en el heap.
4. Este almacenamiento es limpiado a cero, automáticamente se configuran todas las primitivas en el objeto **Dog** a sus valores por defecto (cero para los números y sus equivalentes para **boolean** y **char**) y las referencias a **null**.
5. Las inicializaciones que ocurren en el punto de definición de campos son ejecutadas.
6. Los constructores son ejecutados. Como se podrá ver en el Capítulo 6, esto puede actualmente involucrar mucha actividad, especialmente cuando la herencia está involucrada.

## Inicialización estática explícita.

Java permite agrupar otras inicializaciones estáticas dentro de una “cláusula de construcción estática” (algunas veces llamada *bloque estático*) en una clase. Esta se ve de la siguiente forma:

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

```
|    }
|  // . . .
```

Esto parece ser un método, pero es solo una palabra clave **static** seguida por un cuerpo de método. Este código, como otras inicializaciones estáticas, es ejecutado solo una vez, la primera vez que creamos un objeto de la clase o la primer vez que tiene acceso a un miembro estático de esa clase (aún si nunca creo un objeto de esa clase). Por ejemplo:

```
///: c04:ExplicitStatic.java
// Explicit static initialization
// with the "static" clause.
class Cup {
    Cup(int marker) {
        System.out.println("Copa(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}
class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() {
        System.out.println("Copas");
    }
}
public class ExplicitStatic {
    public static void main(String[] args) {
        System.out.println("Dentro del principal()");
        Cups.c1.f(99); // (1)
    }
    // static Cups x = new Cups(); // (2)
    // static Cups y = new Cups(); // (2)
} //:~
```

Los inicializadores estáticos para **Cups** se ejecutan cuando ocurre un acceso a el objeto **c1** ocurre en la línea marcada (1), o si la línea (1) es comentada, la inicialización estática para Cups nunca ocurre. También, no importa si una de ambas líneas marcadas (2) se les quita el comentario; la inicialización estática para **Cups** se sucede solo una vez.

## Instancia de inicialización no estática

Java proporciona una sintaxis similar para inicializar variables no estáticas para cada objeto. He aquí un ejemplo:

```
///: c04:Mugs.java
// Java "Instance Initialization."
class Mug {
```

```

Mug(int marker) {
    System.out.println("Mug( " + marker + " )");
}
void f(int marker) {
    System.out.println("f( " + marker + " )");
}
}
public class Mugs {
    Mug c1;
    Mug c2;
    {
        c1 = new Mug(1);
        c2 = new Mug(2);
        System.out.println("c1 & c2 initialized");
    }
    Mugs() {
        System.out.println("Mugs( )");
    }
    public static void main(String[] args) {
        System.out.println("Adentro del método principal");
        Mugs x = new Mugs();
    }
} //:~

```

Se puede ver en la cláusula de inicialización:

```

{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
}

```

que se ve exactamente como una cláusula de inicialización estática exceptuando por la palabra clave **static** perdida. Esta sintaxis es necesaria para respaldar la inicialización de las *clases anónimas interiores*(vea Capítulo 8).

## Inicialización de arreglos

La inicialización de arreglos en C es propensa a errores y tediosa. C++ usa *aggregate initialization* para hacerla mucho mas segura<sup>5</sup>. Java no tiene “conglomerados” como C++, dado que todo es un objeto en Java. tiene arreglos y estos son soportados con inicialización de arreglos.

Un arreglo es simplemente una secuencia de objetos o primitivas, todos del mismo tipo y empaquetados juntos bajo un único nombre que lo identifica. Los arreglos son definidos y utilizados con paréntesis cuadrados [] llamados *operador de indexación* Para definir un arreglo simplemente colocamos a continuación del nombre de tipo unos paréntesis cuadrados.

---

<sup>5</sup> Véase *Thinking in C++*, 2<sup>da</sup> edición por una completa descripción de aggregate initialization.

```
| int[] a1;
```

Se puede colocar los paréntesis cuadrados luego del identificador para producir exactamente el mismo significado:

```
| int a1[];
```

Esto se adapta a lo esperado por los programadores de . El C y C++. El estilo anterior, sin embargo, es probablemente una sintaxis mas sensible, dado que el tipo es un “arreglo de enteros”. Este estilo será el utilizado en este libro.

El compilador no permite indicar que tan grande el arreglo es. Esto nos lleva nuevamente al tema de las “referencias”. Todo lo que tenemos en este punto es una referencia a un arreglo, y no hay espacio asignado para el arreglo. Para crear espacio de almacenamiento para el arreglo debemos escribir una expresión de inicialización. Para los arreglos, la inicialización pueden aparecer en cualquier lugar del código, pero se puede también utilizar un tipo especial de expresión de inicialización que debe ocurrir en el momento en que el arreglo es creado. Esta inicialización especial es un grupo de valores rodeados por llaves. Del espacio asignado (el equivalente a utilizar **new**) en este caso se ocupa el compilador. Por ejemplo:

```
| int[] a1 = { 1, 2, 3, 4, 5 };
```

¿Entonces, por que siempre se define una referencia a un arreglo sin un arreglo?

```
| int[] a2;
```

Bueno, es posible asignar un arreglo a otro en Java, así es que se puede decir:

```
| a2 = a1;
```

Lo que se esta haciendo realmente es copiando una referencia, como se demuestra aquí:

```
///: c04:Arrays.java
// Arrays of primitives.
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]);
    }
} ///:~
```

Se puede ver que **a1** se le da un valor de inicialización y a **a2** no; **a2** es asignado mas tarde-en este caso, a otro arreglo.

Hay algo nuevo aquí: todos los arreglos tienen un miembro intrínscico (tanto si son arreglos de objetos o arreglos de primitivas) al que puede escrutar-

pero no cambiar-para averiguar cuantos elementos hay en el arreglo. Este miembro es **length**. Dado que los arreglos en Java, como en C y C++, comienzan desde el elemento cero, el elemento mas grande que puede indexar es **length - 1**. Si se excede de esos tangos, C y C++ tranquilamente aceptan esto y permiten pasar por arriba toda la memoria, lo que es fuente de los mas infames errores. Sin embargo, Java protege contra este tipo de problemas causando un error en tiempo de ejecución (una *excepción*, el tema del Capítulo 10) si se sale de los límites. Claro, verificar cada arreglo tiene un costo en tiempo y en código y no hay forma de evitarlo, lo que significa que el acceso a arreglos puede ser una fuente de ineficiencia en un programa si ocurre en una coyuntura crítica. Los diseñadores de Java pensaron que valía la pena renunciar a una mejor eficiencia a cambio de seguridad en la Internet y una mejora en la productividad a la hora de programar.

¿Que sucede si no se conoce cuantos elementos necesitará en su arreglo cuando se esta escribiendo el programa? Simplemente se usa **new** para crear los elementos del arreglo. Aquí, **new** trabaja de la misma forma si se esta creando un arreglo de primitivas (**new** no creará un algo que no sea un arreglo de primitivas).

```
//: c04:ArrayNew.java
// Creating arrays with new.
import java.util.*;
public class ArrayNew {
    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(
            "length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            System.out.println(
                "a[" + i + "] = " + a[i]);
    }
} //:~
```

Puesto que el tamaño del arreglo es elegido de forma aleatoria (utilizando el método **pRand()**), es claro que la creación del arreglo sucede actualmente en tiempo de ejecución. Además, se podrá ver en la salida del programa que un arreglo de elementos de tipos primitivos son automáticamente inicializados a valores “vacíos” (para números y **char**, esto es cero, y para tipos **boolean**, es **false**).

Claro que, el arreglo puede también ser definido e inicializado en la misma instrucción.

```
| Int[] a = new int[pRand(20)];
```

Si estamos tratando con un arreglo de objetos de tipos que no son primitivos, debemos siempre utilizar **new**. Aquí, el tema de la referencia surge nuevamente porque lo que se esta creando es un arreglo de referencias. Considerando el tipo envoltura **Integer**, lo que es una clase y no una primitiva.

```
//: c04:ArrayClassObj.java
// Creating an array of nonprimitive objects.
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(
            "length of a = " + a.length);
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer(pRand(500));
            System.out.println(
                "a[" + i + "] = " + a[i]);
        }
    }
} //:~
```

Aquí, siempre que se llama a **new** para crear el arreglo:

```
| Integer[] a = new Integer[pRand(20)];  
se obtiene un solo arreglo de referencias, y hasta que la referencia por si sola  
no sea inicializada para crear un nuevo objeto Integer no tendremos una  
inicialización completa.
```

```
| a[i] = new Integer(pRand(500));  
Si se olvida crear el objeto, sin embargo, se tendrá una excepción en tiempo  
de ejecución cuando se intente leer un índice en un arreglo vacío.
```

Obsérvese en la formación del objeto **String** dentro de la instrucción para imprimir. Se puede ver que la referencia al objeto **Integer** es convertida automáticamente para producir una representación del valor del **String** dentro del objeto.

Es posible también inicializar arreglos de objetos utilizando llaves que encierran una lista. Hay dos formas:

```
//: c04:ArrayInit.java
// Array initialization.
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),
    };
}
} //:~

```

Esto es útil a veces, pero es más limitado dado que el tamaño del arreglo es determinado en tiempo de compilación. La coma final en la lista de inicializadores es opcional (Esta hace más fácil de mantener largas listas).

La segunda forma para inicializar arreglos proporciona una sintaxis conveniente para crear y llamar métodos que pueden producir el mismo efecto que una *lista de argumentos de variables* de C (conocidas como "varargs" en C). Estas pueden incluir cantidades de argumentos al igual que tipos desconocidos. Dado que todas las clases son al fin de cuentas heredadas de una clase base común **Object** (un tema que se aprenderá cuando este libro progrese), se puede crear un método que tome un arreglo de tipos **Object** y se llame de esta forma:

```

//: c04:VarArgs.java
// Using the array syntax to create
// variable argument lists.
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(11.11) });
        f(new Object[] {"one", "two", "three" });
        f(new Object[] {new A(), new A(), new A() });
    }
}
} //:~

```

En este punto no hay mucho que se pueda hacer con estos objetos desconocidos, y este programa utiliza conversión automática del tipo **String** para hacer algo útil con cada **Object**. En el Capítulo 12 que cubre la *identificación de tipo en tiempo de ejecución(RTTI run-time type identification)*, se aprenderá como descubrir el tipo exacto del objeto así se puede hacer algo interesante con ellos.

## Arreglos multidimensionales

Java permite crear fácilmente arreglos multidimensionales:

```

//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;

```

```

public class MultiDimArray {
    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]);
        // 3-D array with fixed length:
        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]);
        // 3-D array with varied-length vectors:
        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]);
        // Array of nonprimitive objects:
        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]);
    }
} //:~

```

El código utilizado para imprimir utiliza **length** así es que no depende de tamaños de arreglos fijos.

El primer ejemplo muestra un arreglo multidimensional de primitivas. Se delimita cada vector en el arreglo con llaves:

```

int[][] a1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
};

```

Cada grupo de paréntesis cuadrados nos posiciona dentro del siguiente nivel del arreglo.

El segundo ejemplo muestra un arreglo tridimensional asignado con **new**. Aquí, el arreglo entero es asignado de una sola vez.

```
| int[][][] a2 = new int[2][2][4];
```

El tercer ejemplo muestra cada vector en el arreglo que construye la matriz puede ser de cualquier largo:

```

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 **new** crea un arreglo con un primer elemento de largo aleatorio y el resto indeterminado. El segundo **new** dentro del bucle **for** completa los elementos pero deja el tercer índice indeterminado hasta que se ejecuta el tercer **new**.

Se puede ver en la salida que los valores de arreglos son automáticamente inicializados a cero si no se les da un valor de inicialización explícito.

Se puede tratar con arreglos de tipos no primitivos en una forma similar, lo que es mostrado en el cuarto ejemplo, demostrando la habilidad de juntar muchas expresiones **new** con llaves.

```

Integer[][] a4 = {
    { new Integer(1), new Integer(2) },
    { new Integer(3), new Integer(4) },
    { new Integer(5), new Integer(6) },
};

```

El quinto ejemplo muestra como un arreglo de objetos de tipos no primitivos pueden ser armados pieza por pieza:

```

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);
}

```

El  $i*j$  es simplemente para colocar un valor interesante dentro del **Integer**.

## Resumen

Este mecanismo aparentemente elaborado para la inicialización, el constructor, debe dar un duro golpe acerca de la importancia crítica apostada en la inicialización en el lenguaje. Como Stroustrup diseño C++, una de las primeras observaciones que hizo acerca de la productividad en C fue la impropia inicialización de variables causa una significante porción de los problemas de programación. Este tipo de errores son difíciles de encontrar, y temas similares se aplican a la limpieza inadecuada. Dado que los constructores permiten *garantizar la* inicialización y limpieza (el compilador no permitirá que un objeto sea creado sin la correspondiente llamada al constructor), se tiene un completo control y seguridad.

En C++, la destrucción es muy importante porque los objetos que son creados con **new** deben ser explícitamente destruidos. En Java, el recolector de basura automáticamente libera la memoria de todos los objetos, así es que el método de limpieza equivalente en Java no es necesario la mayoría del tiempo. En casos donde no se necesita un comportamiento como el del destructor, el recolector de basura de Java simplifica enormemente la programación, y agrega mucha de la seguridad necesitada en el manejo de memoria. Algunos recolectores de basura pueden a veces limpiar otros recursos como gráficos y manejadores de ficheros. Sin embargo, el recolector de basura agrega un costo de tiempo de ejecución agregado, el gasto es lo difícil de poner en perspectiva dado el enlentecimiento extra de los intérpretes de Java en el momento en que se escribió esto. Todos estos serán construidos, hay actualmente mas del constructor de lo que es mostrado aquí. En particular, cuando creamos nuevas clases utilizando herencia y composición la garantía de la construcción también se mantiene, y alguna sintaxis adicional es necesaria para soportar esto. Se aprenderá acerca de la composición, herencia y como afectan a sus constructores en capítulos futuros.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Créese una clase con un constructor por defecto (uno que no tome argumentos) que imprima un mensaje. Cree un objeto de esa clase.
2. Agregue un constructor sobrecargado al ejercicio 1 que tome un argumento del tipo **String** e imprima la cadena junto con su mensaje.
3. Créese un arreglo de referencias a objetos de la clase que se ha creado en el ejercicio 2, pero no cree los objetos para asignar dentro del arreglo. Cuando se ejecute el programa, observe cuando los mensajes de inicialización de la llamada al constructor son impresos.
4. Complete el ejercicio 3 creando objetos para atribuir a las referencias del arreglo.
5. Créese un arreglo de objetos **String** y asigne una cadena a cada elemento. Imprima el arreglo utilizando un bucle **for**.
6. Créese una clase llamada **Dog** con un método sobrecargado llamado **baark()**. Este método debe ser sobrecargado basándose en varios tipos de datos primitivos, e imprima diferentes tipos de ladridos, chillidos, etc., dependiendo de la versión del método sobrecargado se este llamado. Escríbase un **main()** que llame las diferentes versiones.
7. Modifíquese el ejercicio 6 para que dos de los métodos sobrecargados tengan dos argumentos (de dos tipos diferentes), pero en orden inverso en relación a cada uno de los otros. Verifíquese que esto funcione.
8. Cree una clase sin un constructor, y luego cree un objeto de esa clase en el **main()** para verificar que el constructor por defecto es automáticamente sintetizado.
9. Créese una clase con dos métodos. En el interior del primer método, llámese al segundo método dos veces: la primera vez sin utilizar **this** y la segunda vez utilizando **this**.
10. Créese una clase con dos constructores (sobrecargados). Utilizando **this**, llame el segundo constructor dentro del primero.
11. Créese una clase con un método **finalize()** que imprima un mensajes. En el **main()**, cree un objeto de su clase. Explíquese el comportamiento del programa.
12. Modifíquese el ejercicio 11 para que el **finalize()** sea llamado siempre.
13. Créese una clase llamada **Tank** que pueda ser llenada y vaciada, y con una *condición de muerto* que debe ser vaciada cuando el objeto es limpiado. Escríbase un **finalize()** que verifique esta condición de muerto. En el **main()**, verifíquese los posibles escenarios que pueden ocurrir cuando **Tank** es utilizado.

14. Créese una clase conteniendo un tipo **int** y un tipo **char** que no sea inicializado, imprímase esos valores para verificar que Java realiza una inicialización por defecto.
15. Créese una clase que contenga una referencia no inicializada a una cadena. Demuestre que esa referencia es inicializada por Java a **null**.
16. Cree una clase con un campo del tipo **String** que sea inicializado en el punto de la definición, y otro que sea inicializado por el constructor. ¿Cuál es la diferencia entre los dos métodos?
17. Créese una clase con una campo del tipo **static String** que sea inicializado por el bloque estático. Agréguese un método estático que imprima ambos campos y demuestre que son ambos inicializados antes de ser utilizados.
18. Cree una clase con un tipo **String** que sea inicializado utilizando "instancia de inicialización". Describase el uso de esta herramienta (otra mas que la especificada en este libro).
19. Escriba un método que cree e inicialice un arreglo de dos dimensiones de tipo **double**. El tamaño del arreglo es determinado por los argumentos del método, y los valores de inicialización son un rango determinado por los valores de comienzo y final que son también argumentos de estos métodos. Créese un segundo método que imprima el arreglo generado por el primer método. En el **main()** verifique que los métodos creando e imprimiendo varios tamaños diferentes de arreglos.
20. Repita el ejercicio 19 para un arreglo tridimensional.
21. Comente la línea marcada (1) en **ExplicitStatic.java** y verifique que la cláusula no es llamada. Ahora quite el comentario de la otra línea marcada (2) y verifique que la inicialización estática solo ocurre una vez.
22. Experimente con **Garbage.java** ejecutando el programa utilizando los argumentos "gc", "finalize", o "all". Repita el proceso y vea si se detecta cualquier patrón en la salida. Cambie el código para que sea llamado **System.runFinalization()** después que **System.gc()** y observe los resultados.

# 5: Escondiendo la implementación

Una consideración primaria en el diseño orientado a objetos es "separar las cosas que cambian de las cosas que se quedan iguales".

Esto es particularmente importante para librerías. El usuario (cliente programador) de esa librería debe ser capaz de confiar en la parte que utiliza, y saber que no necesita escribir nuevamente código si una nueva versión de la librería aparece. Por la otra parte, el creador de la librería debe tener la libertad de realizar modificaciones y mejoras con la certeza de que el código del cliente programador no será afectada por esos cambios.

Esto puede ser alcanzado mediante una convención. Por ejemplo, el programador de la librería debe estar de acuerdo a no quitar métodos existentes cuando se modifica una clase de la librería, dado que puede desbaratar el código del cliente programador.

La situación inversa es complicada, sin embargo. ¿En el caso de un miembro de datos, como puede el creador de la librería saber que datos miembros deben ser accedidos por el cliente programador? Esto es también verdad con métodos que solo son parte de la implementación de la una clase, y no significa que sean utilizados directamente por el cliente programador. ¿Pero que si el creador de la librería quiere deshacerse de la vieja implementación y colocar una nueva? Cambiar alguno de esos miembros pueden hacer que el código del cliente programador deje de funcionar. De esta forma el creador de la librería esta en un chaleco de fuerza y no puede cambiar nada.

Para solucionar este problema, Java proporciona *especificadores de acceso* para permitir al creador de la librería indicar que esta disponible para el cliente programador y que cosa no esta. Los niveles de control de acceso desde el "mas accesible" al "menos accesible" son **public**, **protected**, "friendly" (que no es una palabra clave), y **private**. Del anterior párrafo se puede pensar que, como diseñador de una librería se debe mantener todo lo que sea posible como "private", y exponer solo lo que se quiera que el cliente programador utilice. Esto es exactamente correcto, aún si a menudo existe una oposición intuitiva de las personas que programan otros lenguajes (especialmente C) y son utilizados para acceder a todo sin restricción. Al final de este capítulo se estará totalmente convencido del valor del control de acceso en Java.

El concepto de una librería de componentes y el control de quien puede acceder a los componentes de esa librería no es completo sin embargo. Se mantiene la pregunta de cuánto de los componentes son reunidos juntos en una unidad consistente de librería. Esto es controlado con la palabra clave **package** en Java, y el especificador de acceso son afectados si una clase está en el mismo paquete o en un paquete separado. Así es que para comenzar este capítulo, se aprenderá como los componentes de una librería son colocados dentro de paquetes. Luego, se será capaz de entender el significado completo de los especificadores de acceso.

# Paquete: la unidad de librería

Un paquete es lo que se obtiene cuando se utiliza la palabra clave **import** para traer una librería entera, como en:

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

Esto trae la librería de utilidades entera que es parte de la distribución estándar de Java. Puesto que, por ejemplo, la clase **ArrayList** es en **java.util**, se puede especificar el nombre completo **java.util.ArrayList** (lo cual se puede hacer sin la instrucción **import**), o simplemente se puede decir **ArrayList** gracias a el **import**.

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

Ahora se puede utilizar **ArrayList** sin calificarlo. Sin embargo, ninguna de las otras clases en **java.util**, están disponibles.

La razón para importar todo esto es proporcionar un mecanismo para manejar "espacios de nombres". Los nombres de todas las clases miembros son encapsulados de los demás. Un método **f()** dentro de una clase **A** que no entra en conflicto con un **f()** que tiene la misma firma (lista de argumentos) en la clase **B**. ¿Pero qué sucede con los nombres de clases? Supóngase que se está creando una clase **stack** que está instalada en una máquina que ya tiene una clase **stack** que ha sido creada por alguien más. Con Java en la Internet, esto puede suceder sin que el usuario lo sepa, dado que las clases pueden ser bajadas automáticamente en el proceso de ejecutar un programa.

Este potencial conflicto de nombres es donde es importante tener un completo control de los espacios de nombres en Java, y ser capaz de crear un nombre completo independientemente de las limitaciones de la Internet.

Hasta el momento, muchos de los ejemplos de este libro han existido en un único fichero y son diseñados para uso local, y no hay que molestarse con nombres de paquetes (en este caso el nombre de clase es colocado en el "paquete por defecto").. Esto es naturalmente una opinión, y para

simplificar este enfoque se usará donde sea posible en el resto del libro. Sin embargo, si se está planeando crear librerías o programas en la misma máquina, se deberá pensar acerca de prevenir conflictos con los nombres de clases.

Cuando se crea un fichero de código fuente para Java, es comúnmente llamado una *unidad de compilación* (a veces llamada *unidad de interpretación*). Cada unidad de compilación debe tener un nombre terminado en **.java**, y dentro la unidad de compilación puede haber una clase pública que debe tener el mismo nombre que el fichero (incluyendo las mayúsculas, pero excluyendo la extensión de fichero **.java**). Puede haber solo *una* clase public en cada unidad de compilación, de otra forma el compilador se quejará. El resto de las clases en la unidad de compilación, si hay alguna, están ocultas de el mundo fuera del paquete dado que *no son public*, y ellas son las clases de "soporte" para la clase **public** principal.

Cuando se compile un fichero **.java** se obtendrá un fichero de salida con exactamente el mismo nombre pero con la extensión **.class** para cada clase en el fichero **.java**. De esta forma se puede terminar con bastantes ficheros **.class** partiendo de un pequeño número de ficheros **.java**. Si se ha programado con un lenguaje compilado, podrían ser utilizado para el compilador para generar una forma intermedia (usualmente un fichero "obj") que es luego empaquetado junto con otros de su tipo utilizando un linker (para crear un fichero ejecutable) o un librarian (para crear una librería). Esta no es la forma en que Java trabaja. Un programa que trabaje es un montón de ficheros **.class**, que pueden ser empaquetados y comprimidos dentro de un fichero JAR (utilizando el archivador **jar** de Java). El intérprete de Java es responsable por encontrar, cargar, e interpretar estos ficheros<sup>1</sup>.

Una librería también es un grupo de estos ficheros clase. Cada fichero tiene una clase que es pública (no es obligatorio tener una clase pública, pero es típico), así es que hay un componente por cada fichero. Si se quiere indicar que todos esos componentes (que están en sus respectivos ficheros **.java** y **.class** separados) pertenecen a un solo sitio, hay es donde la palabra clave **package** aparece.

Cuando decimos:

```
| package mipaquete;
```

en el comienzo del fichero (si utiliza una instrucción **package**, *debe* estar en la primer línea distinta de un comentario en el fichero), se manifiesta que esta unidad de compilación es parte de una librería llamada **mipaquete**. O, dicho de otra forma, se esta diciendo que el nombre de clase público dentro de esta unidad de compilación esta bajo el paraguas con el nombre

---

<sup>1</sup> No hay nada en Java que fuerce la utilización de un interprete. Existen compiladores de código nativo Java que generan un fichero ejecutable que pueda trabajar solo.

**mipaquete**, y si alguien utilizar este nombre debe especificar el nombre completo o utilizar la palabra clave **import** en combinación con **mipaquete** (utilizando la elección dada previamente). Se puede ver que la convención para nombres de paquetes Java es utilizar letras minúsculas en todo el nombre, inclusive para las primeras letras de las palabras intermedias.

Por ejemplo, supóngase que el nombre del fichero es **MyClass.java**. Esto significa que puede ser una y solo una la clase pública en ese fichero, y el nombre de esta clase debe ser **MyClass** (incluyendo las mayúsculas):

```
| package mipaquete;
| public class MyClass {
| // . . .
```

Ahora, si alguien quiere utilizar **MyClass** o, por su importancia, alguna de las clases públicas en **mipaquete**, deben utilizar la palabra clave **import** para hacer el nombre o los nombres en **mipaquete** disponibles. La alternativa es dar el nombre totalmente calificado:

```
| mipaquete.MyClass m = new mipaquete.MyClass();
| La palabra clave import puede hacer esto mucho mas limpio.
```

```
| import mypackage.*;
| // . . .
| MyClass m = new MyClass();
```

Esto funciona teniendo en cuenta que lo que las palabras **package** e **import** le permiten hacer, como diseñador de una librería, es dividir el único espacio de nombres global así no hay conflictos de nombres, no importa cuantas personas se conectan a la Internet y escriben clases en Java.

## Creando nombres de paquetes únicos

Se debe observar que, dado que un paquete nunca realmente está "empaquetado" en un único fichero, un paquete puede ser creado por muchos ficheros **.class**, y las cosas pueden tornarse un poco desordenadas. Para prevenir esto, una cosa lógica para hacer es colocar todos los ficheros **.class** de un paquete particular dentro de un directorio individual; esto es, tomar ventaja de la estructura jerárquica del sistema operativo. Esta es una forma que Java recomienda el problema del desorden; se verá mas adelante la otra forma cuando la utilidad **jar** sea introducida.

Juntar los ficheros del paquete en un único directorio soluciona dos problemas mas; la creación de nombres de paquetes únicos, y encontrar esas clases que pueden estar enterradas en algún parte dentro de la estructura de directorios. Esto se logra, como se introduce en el Capítulo 2, codificando el camino de la ubicación del fichero **.class** dentro del nombre del paquete. El compilador fuerza esto, pero por convención, la primer parte

del nombre de paquete es el nombre de dominio del creador de la clase a la inversa. Dado que los nombres de la Internet se garantizan son únicos, *sí* se sigue esta convención se garantiza que el nombre del paquete será único y nunca habrá un conflicto de nombres (Esto es, hasta que pierda el nombre de dominio y lo tome alguien mas que comience a escribir código java con la misma ruta de nombres). Claro, si no se tiene un dominio propio entonces se deberá crear una combinación única (como su nombre y apellido) para crear un nombre de paquete único. Si decide publicar código Java vale la pena un pequeño esfuerzo para obtener un nombre de dominio.

La segunda parte de esta trampa es determinar el nombre del paquete dentro de un directorio en la máquina, así cuando el programa Java se ejecute y necesite cargar el fichero **.class** (lo que se hace de forma dinámica, en el alguna parte del programa que se necesite crear un objeto de esa clase en particular, o el primer momento en que se accede a un miembro estático de la clase), debe poder encontrar el directorio donde el fichero **.class** reside.

El intérprete Java procede de esta forma. Primero, encuentra la variable de ambiente CLASSPATH (la cual es definida por el sistema operativo, y a veces por el programa de instalación que instala Java o por alguna herramienta basada en Java de su máquina). CLASSPATH contiene uno o mas directorios que son utilizados como raíz para la búsqueda de ficheros **.class**. Comenzando de esa raíz, el intérprete tomara el nombre de paquete y remplazará cada punto con una barra para generar un nombre de ruta partiendo de la raíz en CLASSPATH (así es que el paquete **foo.bar.baz** se convierte en **foo\var\baz** o **foo/bar/baz** o posiblemente otra cosa dependiendo de sus sistema operativo. Esto es entonces concatenado a las varias entradas de la ruta en CLASSPATH. Ahí es donde buscará el fichero **.class** con el correspondiente nombre a la clase que esta tratando de crear (También busca algunos directorios estándares relativos a donde el interprete Java reside).

Para entender esto, consideremos mi nombre de dominio, que es **bruceeckel.com**. Si invertimos esto, **com.bruceeckel** establece mi nombre global para la clase. (La extensión com, edu, org, etc., fue formalmente llevada a mayúsculas en los paquetes Java, pero esto fue cambiado en Java 2 así es que el nombre del paquete entero es en minúsculas). Puedo además subdividir esto decidiendo que quiero crear una librería llamada **simple**, así es que terminaré el nombre de mi paquete:

```
| package com.bruceeckel.simple;  
Ahora este nombre de paquete puede ser utilizado como un espacio de  
nombres paraguas para los siguientes dos ficheros:
```

```
//: com:bruceeckel:simple:Vector.java  
// Creating a package.  
package com.bruceeckel.simple;  
public class Vector {
```

```

    public Vector() {
        System.out.println(
            "com.bruceeckel.util.Vector");
    }
} //:~

```

Cuando se crean paquetes propios, se descubre que la instrucción **package** debe ser el primer código que no sea un comentario en el fichero. El segundo fichero se ve en gran medida de la misma forma:

```

//: com:bruceeckel:simple>List.java
// Creating a package.
package com.bruceeckel.simple;
public class List {
    public List() {
        System.out.println(
            "com.bruceeckel.util.List");
    }
} //:~

```

Ambos ficheros fueron colocados en un subdirectorio en mi sistema:

```

| C:\DOC\JavaT\com\bruceeckel\simple
Si se quiere dar marcha atrás en esto, puede ver el nombre de paquete
com.bruceeckel.simple. ¿Pero qué sucede acerca de la primer parte de la
ruta? Se tiene cuidado de esto en la variable de ambiente CLASSPATH, la
que es, en mi máquina:

```

```

| CLASSPATH=. ;D:\JAVA\LIB;C:\DOC\JavaT
Se puede ver que CLASSPATH puede contener algunas alternativas de
caminos de búsquedas.

```

Hay aquí una variación cuando se utilizan ficheros JAR, sin embargo. Se debe colocar al nombre del fichero JAR en la ruta de clases, no solamente la ruta donde se puede localizar. Así es que para un JAR llamado **grape.jar** la ruta de clases debe incluir:

```

| CLASSPATH=. ;D:\JAVA\LIB;C:\flavors\grape.jar
Una vez que la ruta de clases es configurada adecuadamente, el siguiente
fichero puede ser colocado en cualquier directorio:

```

```

//: c05:LibTest.java
// Uses the library.
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 una instrucción **import**, comienza a buscar en los directorios especificados por CLASSPATH, buscando el subdirectorio **com\bruceeckel\simple**, entonces busca el fichero compilado con los nombres apropiados. (**Vector.class** para **Vector** y **List.class** para **List**). Vea

que ambas clases y los métodos deseados en **Vector** y **List** deben ser públicos.

Configurar la variable CLASSPATH ha sido una experiencia dura para usuarios Java principiantes (lo fue para mí, cuando comencé) que Sun hizo el JDK de Java 2 un poco más inteligente. Se puede encontrar que, cuando se instala, aún si no configura la variable CLASSPATH se es capaz de compilar y ejecutar programas básicos. Para compilar y ejecutar el código fuente de este libro (disponible en el paquete del CD ROM disponible en este libro, o en [www.BruceEckel.com](http://www.BruceEckel.com)), sin embargo, necesitará hacer algunas modificaciones en la ruta de clases (esto es explicado en el paquete del código fuente).

## Colisiones

¿Qué sucede si dos librerías son importadas mediante un \* y ellas incluyen los mismos nombres? Por ejemplo, supongamos que un programa hace esto:

```
| import com.bruceeckel.simple.*;
| import java.util.*;
```

Dado que **java.util.\*** también contiene la clase **Vector**, se producirá una potencial colisión. Sin embargo, hasta que no se escriba el código que cause una colisión, todo está bien—esto es bueno porque de otra forma terminaríamos escribir mucho para prevenir colisiones que nunca sucederían.

La colisión *se sucede* si ahora intentamos crear un **Vector**:

```
| Vector v = new Vector();
```

¿A qué clase **Vector** se refiere? El compilador no puede saberlo, y el lector puede no saberlo tampoco. Así es que el compilador se queja y obliga a ser explícito. Si se quiere un **Vector** estándar de Java, por ejemplo, se debe decir:

```
| java.util.Vector v = new java.util.Vector();
```

Dado que esto (junto con la ruta de clases) especifica completamente la localización de la clase **Vector**, aquí no hay necesidad de la instrucción **import java.util.\*** a no ser que este utilizando algo más de **java.util**.

## Una librería de herramientas a medida

Con este conocimiento, se puede crear una librerías propias de herramientas para eliminar el código duplicado. Se puede considerar, por ejemplo, crear un alias para **System.out.println()** para reducir la escritura. Esto puede ser parte de un paquete llamado **tools**:

```
| //: com:bruceeckel:tools:P.java
```

```
// The P.rint & P.println shorthand.
package com.bruceekel.tools;
public class P {
    public static void rint(String s) {
        System.out.print(s);
    }
    public static void println(String s) {
        System.out.println(s);
    }
} //:~
```

Se puede utilizar esta versión corta para imprimir un **String** con un fin de línea(**P.println()**) o sin un fin de línea (**P.rint()**).

Se puede adivinar que la ubicación de ese fichero debe ser en un directorio que comience con alguna de las rutas en CLASSPATH, luego continúe con **com/bruceekel/tools**. Luego de compilar, el fichero **P.class** puede ser utilizado en cualquier parte de su sistema con una instrucción **import**:

```
//: c05:ToolTest.java
// Uses the tools library.
import com.bruceekel.tools.*;
public class ToolTest {
    public static void main(String[] args) {
        P.println("Available from now on!");
        P.println("" + 100); // Obligarlo a ser una cadena
        P.println("" + 100L);
        P.println("" + 3.14159);
    }
} //:~
```

Debe notarse que todos los objetos pueden fácilmente ser obligados a formar parte de una representación del tipo **String** colocándolos en una expresión del tipo **String**; en el ejemplo anterior, si se comienza con una cadena vacía se logra el truco. Pero esto lleva a una interesante observación. Si se llama a **System.out.println(100)**, esto funciona sin convertir el tipo a **String**. Con alguna sobrecarga extra, se puede obtener la clase **P** para que realice esto de la misma forma (esto es un ejercicio del final de este capítulo).

Así es que desde ahora en adelante, cuando se aparezca con una nueva utilizad, puede agregarse a el directorio **tools** (o su directorio personal **util** o **tools**).

## Utilizando **import** para cambiar el comportamiento

Una característica que se pierde en Java es la *compilación condicional* de C, que permite cambiar un modificador y obtener diferentes comportamiento sin cambiar otro código. La razón de que esa característica fuera quitada de Java probablemente porque es a menudo mas utilizada en C para solucionar

temas de plataformas cruzadas: diferentes porciones de código son compilados dependiendo de la plataforma en la que el código es compilado. Dado que se pretende que Java sea automáticamente multiplataforma, este rasgo no es necesario.

Sin embargo, hay otros usos valiosos para la compilación condicional. Un uso muy común es para depurar código. Las herramientas de depuración son habilitadas durante el desarrollo y deshabilitadas cuando se distribuye el producto. Allen Holub ([www.holub.com](http://www.holub.com)) apareció con la idea de utilizar paquetes para imitar la compilación condicional. El usa esto para crear una versión Java de un muy útil *mecanismo de argumentación* de C, por lo cual se puede decir “esto debería ser verdadero” o “esto debería ser falso” y si la instrucción no está de acuerdo con su argumentación estará fuera. Igual que una herramienta es muy provechoso durante la depuración.

He aquí una clase que se puede utilizar para depuración:

```
///: com:bruceeckel:tools:debug:Assert.java
// Assertion tool for debugging.
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 pruebas booleanas, que imprimen mensajes de error si fallan. En el capítulo 10, se aprenderá acerca de herramientas más sofisticadas para tratar con errores llamadas *manejadores de error*, pero el método **perr()** trabajará bien por lo pronto.

La salida es impresa en el flujo de la consola de *error estándar* escribiendo en **System.err**.

Cuando se quiera utilizar esta clase, agregue una línea en su programa:

```
| import com.bruceeckel.tools.debug.*;
```

Para quitar la aseveración y así saltar el código, una segunda clase **Assert** es creada, pero de un paquete diferente:

```

//: com:bruceeckel:tools:Assert.java
// Turning off the assertion output
// so you can ship the program.
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 instrucción **import** a:

```

import com.bruceeckel.tools.*;
Ahora el programa no imprimirá mas argumentaciones. He aquí un
ejemplo:

```

```

//: c05:TestAssert.java
// Demonstrating the assertion tool.
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
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");
    }
}
} //:~

```

Cambiando el paquete que se importa, se cambia es código de la versión depurada a la versión productiva. Esta técnica puede ser utilizada por cualquier tipo de código condicional.

## Advertencia de paquete

Vale la pena recordar que cada vez que se cree un paquete, se debe implícitamente especificar una estructura de directorio cuando se de un nombre de paquete. El paquete debe vivir en el directorio indicado por este nombre, el cual debe ser localizado iniciando una búsqueda en las rutas de clases. Experimentando con la palabra clave **package** puede haber un poco de frustración al comienzo, porque a no ser que agregue a el nombre del paquete la ruta que lo contiene, se obtendrán un montón de misteriosos mensajes en tiempo de ejecución acerca de una clase particular imposible de encontrar, aún si esa clase esta ubicada en el mismo directorio. Si se obtiene un mensaje como este, se debe comentar las instrucciones **package**, y si corre se sabrá donde se encuentra el problema.

# Especificadores de acceso de Java

Cuando es utilizado, los especificadores de acceso de Java **public**, **protected**, y **private** son ubicados al frente de cada definición de cada miembro de su clase, tanto si es un campo o un método. Cada especificador de acceso controla el acceso para esa definición en particular solamente. Esto es un contraste distinto con C++, en donde el especificador de acceso controla todas las definiciones que siguen hasta el siguiente especificador de acceso.

De una forma u otra, todos tienen un tipo de acceso especificado. En las siguientes secciones, se aprenderá acerca de todos los tipos de accesos, comenzando con el acceso por defecto.

## “Amigable” (“Friendly”)

¿Qué si ni se indica para nada un especificador de acceso, como en los ejemplos anteriores de este capítulo? El acceso por defecto no tiene palabra clave, pero es comúnmente referido como “amigable”. Esto significa que todas las otras clases en el paquete actual tienen acceso a un miembro amigable, pero para todas las clases fuera de este paquete el miembro parece ser **private**. Dado que una unidad de compilación-un fichero- puede pertenecer solo a un solo paquete, todas las clases con una sola unidad de compilación es automáticamente amigable con cada una de las otras. De esta manera, se dice que los elementos amigables tienen *acceso al paquete*.

El acceso amigable permite agrupar clases relacionadas juntas en un paquete así pueden fácilmente interactuar con cada una de las otras.

Cuando se colocan clases juntas en un paquete (de esta forma garantizar un muto acceso a sus miembros amigables; e.g. hacerlos “amigos”) se “posee” el código en ese paquete. Esto tiene sentido de tal manera que solo el código que un programador posee tendrá acceso amigable a otro código que posea. Se podría decir que el acceso amigable le da un significado o una razón para agrupar clases juntas en un paquete. En muchos lenguajes la forma de organizar las definiciones en ficheros puede ser a la fuerza, pero en java se está coaccionado a organizar ellas en una forma sensible. Además, probablemente se quiera excluir clases que no deberían tener acceso a las clases que se encuentran en el paquete actual.

El control de clase cuyo código tiene acceso a sus miembros. No hay formas mágicas de “quebrarlo”. El código de otro paquete no puede mostrarse y decir, “Hola, soy un amigo de **Bob** y espero ver los miembros del tipo

**protected**, amigable y **private** de **Bob**. La única forma de otorgar acceso a un miembro es:

1. Hacer el miembro **public**. Entonces todos, en cualquier lugar, puede acceder a este.
2. Dejar el miembro sin especificador y hacerlo amigable, colocar la otra clase en el mismo paquete. Entonces la otra clase puede acceder al miembro.
3. Como se verá en el capítulo 6, cuando la herencia sea introducida, una clase heredada puede acceder a un miembro **protected** de la misma forma que un miembro público (pero no como miembros **private**). Estos pueden acceder a miembros amigables solo si las dos clases están en el mismo paquete. Pero no nos preocupemos de eso por ahora.
4. Proporcionar métodos “accessor/mutator” (también conocidos como métodos “get/set”) que leen y cambian el valor. Esto es la estrategia más civilizada en términos de POO, y es fundamental para JavaBeans, como se verá en el capítulo 13.

## **public**: acceso a la interfase

Cuando se utilice la palabra clave **public**, significa que la declaración de miembro que se encuentra inmediatamente después de **public** esta disponible para cualquiera, en particular para el cliente programador que utiliza la librería. Supongamos que se define un paquete **dessert** que contiene la siguiente unidad de compilación:

```
//: c05:dessert:Cookie.java
// Creates a library.
package c05.dessert;
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void bite() { System.out.println("bite"); }
} //:~
```

Recuerde, **Cookie.java** debe residir en un subdirectorio llamado **dessert**, en un directorio bajo **c05** (que indica el capítulo 5 de este libro) que debe encontrarse bajo un directorio de la ruta de clases. No debe cometerse el error de pensar que Java verá el directorio actual como un punto de partida para una búsqueda. Si no se tiene un ‘.’ como una de las rutas en la ruta de clases, Java no buscará allí.

Ahora si se crea un programa que utilice **Cookie**:

```
//: c05:Dinner.java
// Uses the library.
```

```

import c05.dessert.*;
public class Dinner {
    public Dinner() {
        System.out.println("Dinner constructor");
    }
    public static void main(String[] args) {
        Cookie x = new Cookie();
        //! x.bite(); // Can't access
    }
} //:~

```

se puede crear un objeto **Cookie**, dado que su constructor es público y la clase es del tipo **public** (Se verá más del concepto de clase pública más adelante). Sin embargo, el miembro **bite()** es inaccesible desde adentro de **Dinners.java** dado que **bite()** es amigable solo con el paquete **dessert**.

## El paquete por defecto

Es sorprendente descubrir que el siguiente código copila, aún cuando parece que quiebra las reglas:

```

//: c05:Cake.java
// Accesses a class in a
// separate compilation unit.
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
// The other class.
class Pie {
    void f() { System.out.println("Pie.f()"); }
} //:~

```

Se puede inicialmente ver estas como ficheros completamente ajenos, y con todo **Cake** es capaz de crear un objeto **Pie** y llamar su método **f()** ( Nótese que debe tener '.' en la ruta de clases para poder compilar estos ficheros)! Podría pensar que **Pie** y **f()** son amigables y por consiguiente no estar disponibles para **Cake**. Ellas son amigables-esa parte es correcta. La razón por la cual están disponibles en **Cake.java** es porque están en el mismo directorio y no tienen un nombre de paquete explícito. Java procesa los ficheros como este como partes implícitas de un paquete por defecto para ese directorio, y por lo tanto amigables para todos los otros ficheros en ese directorio.

## **private**: no se puede tocar eso!

La palabra clave **private** indica que nadie puede tener acceso a ese miembro exceptuando la clase en particular, los métodos internos de esa clase. Otras

clases en el mismo paquete no pueden acceder a miembros privados, así es que es como si aún se haya aislado la clase contra uno mismo. Por el otro lado, no es improbable que un paquete pueda ser creado por muchas personas que colaboran juntas, así es que **private** permite que se cambie libremente ese miembro sin preocuparse si afectará a otra clase en el mismo paquete.

El acceso “amigable” por defecto de paquete frecuentemente proporciona un nivel de encubrimiento adecuado; recuerde que un miembro “amigable” es inaccesible a el usuario de ese paquete. Esto es bueno, dado que el acceso por defecto es el único que normalmente es utilizado (y el único que se obtendrá si se olvida de agregar un control de acceso). De esta forma, normalmente solo se pensará en acerca del acceso para los miembros que explícitamente se quieren hacer públicos para el cliente programador, y como resultado, no se necesitará inicialmente pensar si se utilizará la palabra clave **private** a menudo dado que es tolerable salir del paso sin ella (Esto es un contraste distinto con C++). Sin embargo, el uso consistente de **private** es muy importante, especialmente en lo concerniente a hilos múltiples (Como se verá en el capítulo 14).

Aquí vemos un ejemplo del uso de **private**:

```
//: c05:IceCream.java
// Demonstrates "private" keyword.
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** es conveniente: se debe controlar como un objeto es creado y prevenir que alguien con acceso directo a un constructor en particular (o a todos ellos). En el ejemplo anterior, no se puede crear un objeto **Sundae** mediante su constructor, en lugar de eso se debe llamar a el método **makeASundae()** para que lo haga<sup>2</sup>.

Cualquier método del que se este seguro que es solo un método que “ayude” a esa clase puede hacerse privado, para asegurar que no se utilice accidentalmente en cualquier parte del paquete y prohibir de uno mismo de

---

<sup>2</sup> Hay otro efecto en este caso: dato que el constructor por defecto es el único definido, y es privado, previene la herencia de esta clase (un tema que introduciremos en el capítulo 6).

cambiar o quitar el método. Hacer un método privado garantiza que existe esa opción.

Lo mismo es cierto para campos privados dentro de una clase. A no ser que se desee exponer la implementación de capas mas bajas (la cual es una situación muy rara en la que se debe pensar), se debería hacer todos los campos privados. Sin embargo, solo porque una referencia a un objeto es privada dentro de una clase, no significa que algún otro objeto pueda tener una referencia pública a el mismo objeto (Véase el Apéndice A por temas de aliasing).

## **protected**: “un tipo de amistad”

El especificador de acceso **protected** requiere un salto adelante para ser entendido. Primero, se debe ser conciente que no se necesita entender esta sección para continuar este libro hasta herencia (Capítulo 6). Pero por un tema de amplitud, he aquí una breve descripción y un ejemplo de como utilizar **protected**.

La palabra clave **protected** trata con el concepto de herencia, que toma una clase existente y agrega nuevos miembros a esa clase sin tocar la clase existente, que se refiere como la *clase base*. Se puede también cambiar el comportamiento de los miembros existentes de la clase. Para heredar de una clase existente, se dice que la clase nueva se extiende de una clase existente utilizando **extends** de la siguiente forma:

```
| class Foo extends Bar {  
|     El resto de la definición de la clase se ve igual.
```

Si se crea un nuevo paquete y se hereda de una clase de otro paquete, los únicos miembros a los cuales se tiene acceso son los miembros públicos del paquete original (Claro, si se realiza una herencia en el *mismo* paquete, se tiene el acceso normal de paquete a todos los miembros “amigables”). A veces, el creador de la clase base desea tomar un miembro particular y conceder acceso a las clases derivadas pero no a el mundo en general. Esto es lo que **protected** hace. Si volvemos a referirnos a el fichero **Cookie.java**, la siguiente clase *no puede* acceder a el miembro “amigable”:

```
//: c05:ChocolateChip.java  
// Can't access friendly member  
// in another class.  
import c05.dessert.*;  
public class ChocolateChip extends Cookie {  
    public ChocolateChip() {  
        System.out.println(  
            "ChocolateChip constructor");  
    }  
    public static void main(String[] args) {  
        ChocolateChip x = new ChocolateChip();
```

```
    } //! x.bite(); // Can't access bite
}
} ///:~
```

Una de las cosas interesantes acerca de herencia es que si un método **bite()** existe en la clase **Cookie**, entonces también existe en la clase heredada de **Cookie**. Pero dado que **bite()** es “amigable” en un paquete del exterior, no está disponible en esta. Claro, se puede hacer este público, pero entonces cualquiera puede tener acceso y tal vez eso no es lo que se quiere. Si se cambia la clase **Cookie** como sigue:

```
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void bite() {
        System.out.println("bite");
    }
}
```

entonces **bite()** sigue con acceso “amigable” sin el paquete **dessert**, pero es también accesible para cualquier clase que sea heredada de **Cookie**. Sin embargo, *no* es público.

## Interfase e implementación

El control de acceso es a menudo referido como *implementación oculta*. La envoltura de datos y métodos dentro las clases en combinación con la implementación oculta es a menudo llamada *encapsulación*<sup>3</sup>. El resultado es un tipo de datos con determinadas características y comportamientos.

El control de acceso coloca límites dentro de un tipo de datos por dos razones importantes. La primera es para establecer lo que el cliente programador puede y no puede utilizar. Se pueden realizar mecanismos dentro de la estructura sin preocuparse que el cliente programador accidentalmente trate con los interiores como parte de la interfase que utilizará.

Esto nos lleva directamente a la segunda razón, que es separar la interfase de la implementación. Si la estructura es utilizada en un grupo de programas, pero el cliente programador no puede hacer nada más que enviar mensajes a la interfase pública, entonces se puede cambiar cualquier cosa que *no* sea pública (e.g. “amigable”, protegido o privado) sin necesitar modificaciones en el código del cliente.

Estamos ahora en el mundo de la programación orientada a objetos, donde una clase es actualmente descrita como “una clase de objetos”, como se

---

<sup>3</sup> Sin embargo, a menudo las personas se refieren a la implementación oculta sola como encapsulación.

describe una clase de peces o una clase de pájaros. Cualquier objeto que pertenece a esta clase compartirá estas características y comportamientos. La clase es una descripción de la forma en que todos los objetos se verán y actuarán.

En el lenguaje de POO original, Simula-67, la palabra clave **class** fue utilizada para describir un nuevo tipo de dato. La misma palabra clave ha sido utilizada por la mayoría de los lenguajes orientados a objetos. Esto es el punto focal en la totalidad del lenguaje: la creación de nuevos tipos de datos que son solo cajas conteniendo datos y métodos.

La clase es el concepto fundamental de POO en Java. Es una de las palabras claves que *no* serán colocadas en negrita en este libro-comienza a molestar en una palabra tan repetida como “class”.

Por un tema de claridad, se prefiere un estilo para crear clases que colocan los miembros del tipo **public** en el comienzo, seguidos por los del tipo **protected**, amigables y por último los del tipo **private**. La ventaja es que el usuario de una clase puede entonces leer comenzando desde arriba y ver primero que es importante para él (los miembros públicos, dado que a ellos puede acceder desde afuera del fichero), y parar de leer cuando se encuentra un miembro que no es público, lo que ya es 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 lo hará parcialmente fácil de leer porque la interfase y la implementación siguen mezcladas. Esto es, sigue viendo el código fuente-la implementación-dado que esta ahí en la clase. Además, los comentarios de documentación soportado por javadoc (descrito en el capítulo 2) reducen la importancia de la legibilidad del código por parte del cliente programador. Mostrar la interfase a el consumidor de una clase es realmente la tarea de el *navegador de clase(class browser)*, una herramienta cuyo trabajo es mostrar a todos las clases disponibles y mostrar lo que se puede hacer con ellas (i.e., que miembros están disponibles) en una forma útil. En el momento en que se lea esto, los navegadores serán una esperada parte de cualquier buena herramienta de desarrollo.

# Acceso a la clase

En Java, los especificadores de acceso pueden ser utilizados para determinar que clases *dentro* de una librería estarán disponibles para los usuarios de esa librería. Si se quiere que una clase este disponible para el cliente programador, se coloca la palabra clave **public** en algún lado antes de abrir la llave del cuerpo de la clase. Esto controla incluso si el cliente programador puede crear un objeto de la clase.

Para controlar el acceso a una clase, el especificador debe aparecer antes de la palabra clave **class**. Entonces se puede decir:

```
| public class Widget {  
| Ahora si el nombre de la librería es mylib cualquier cliente programador  
| puede acceder a Widget diciendo:
```

```
| import mylib.Widget;  
|  
| o
```

```
| import mylib.*;  
| Sin embargo, hay un grupo extra de restricciones:
```

1. Solo puede haber una clase pública por unidad de compilación (fichero). La idea es que cada unidad de compilación tenga una sola interfase pública representada por esa clase pública. Puede tener tantas clases “amigables” de soporte como se quiera. Si se tiene mas de una clase pública dentro de una unidad de compilación, el compilador dará un mensaje de error.
2. El nombre de la clase pública debe corresponder exactamente con el nombre del fichero que contiene la unidad de compilación, incluyendo las mayúsculas y las minúsculas. Así es que para **Widget**, el nombre del fichero debe ser **Widget.java**, y no **widget.java** o **WIDGET.java**. Nuevamente, se obtendrá un error en tiempo de compilación si no esta de acuerdo.
3. Si es posible, aunque no es típico, tener una unidad de compilación sin clase pública. En este caso, el nombre de fichero es el que se quiera.

¿Que sucede si se tiene una clase dentro de **mylib** que simplemente se esta utilizando para realizar las tareas ejecutadas por **Widget** o alguna otra clase pública en **mylib**? Si no se quiere tomar la molestia de crear la documentación para el cliente programador, y se tiene en mente que en algún momento se cambiarán totalmente las cosas incluyendo la clase en su totalidad, substituyéndola por una diferente. Si se desea dar esa flexibilidad, se necesita asegurar que ningún cliente programador se hizo dependiente de los detalles de implementación ocultos dentro de **mylib**. Para lograr esto se

deja la palabra clave **public** fuera de la clase, en cuyo caso se convierte en amigable (Esa clase solo puede ser utilizada dentro del paquete).

Se puede ver que una clase no puede ser **private** (esto la haría inaccesible para todos menos para la clase), o **protected**<sup>4</sup>. Así es que solo se tienen dos posibilidades para el acceso a la clase: “amigable” o **public**. Si no se desea que nadie pueda acceder a la clase, se puede hacer que todos los constructores sean del tipo **private**, de esta manera se previene que nadie excepto la persona que está escribiendo la clase, dentro de un miembro estático de la clase, de crear un objeto de esa clase<sup>5</sup>. He aquí un ejemplo:

```
///: c05:Lunch.java
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:
class Soup {
    private Soup() {}
    // (1) Allow creation via static method:
    public static Soup makeSoup() {
        return new Soup();
    }
    // (2) Create a static object and
    // return a reference upon request.
    // (The "Singleton" pattern):
    private static Soup ps1 = new Soup();
    public static Soup access() {
        return ps1;
    }
    public void f() {}
}
class Sandwich { // Uses Lunch
    void f() { new Lunch(); }
}
// Only one public class allowed per file:
public class Lunch {
    void test() {
        // Can't do this! Private constructor:
        // ! Soup priv1 = new Soup();
        Soup priv2 = Soup.makeSoup();
        Sandwich f1 = new Sandwich();
        Soup.access().f();
    }
}
} ///:~
```

Hasta ahora, muchos de los métodos han returned o **void** o un tipo primitivo, así es que la definición:

```
public static Soup access() {
    return ps1;
}
```

<sup>4</sup> Actualmente, una **clase interna** puede ser privada o protegida, pero esto es un caso especial. Esto será introducido en el capítulo 7.

<sup>5</sup> Se puede también hacer esto heredando (Capítulo 6) de esa clase.

Puede resultar un poco confusa al comienzo. La palabra antes que el nombre del método (**access**) indica lo que el método retorna. Hasta el momento lo mas común ha sido **void**, que significa que no retorna nada. Pero se puede retornar también una referencia a un objeto, que es lo que sucede aquí. Este método retorna una referencia a un objeto de la clase **Soup**.

La clase **Soup** muestra como prevenir la ceración directa de una clase haciendo que todos los constructores sean del tipo **private**. Recuerde que si no se crea explícitamente al menos un constructor, el constructor por defecto (un constructor sin argumentos) será creado. Escribir el constructor por defecto, no lo creará automáticamente. ¿Si se hace privado, nadie podrá crear un objeto de esa clase. Pero ahora, como se hace para que alguien utilice esa clase? El anterior ejemplo muestra dos opciones. Primero, un método estático es creado que para crear un nuevo **Soup** y retorna una referencia a este. Esto puede ser útil si se desea operaciones extras en el objeto **Soup** antes de retornar, o si se desea contar cuantos objetos **Soup** se han creado (quizás para restringir la población).

La segunda opción utiliza lo que llamamos *patrón de diseño*, que es cubierto en *Thinking in Patterns with Java* que se puede bajar en [www.BruceEckel.com](http://www.BruceEckel.com). Este patrón particular es llamado un “singleton” porque permite que un solo objeto sea creado. El objeto de la clase **Soup** es creado como un miembro del tipo **static private** de **Soup**, así es que hay uno y solo uno, y no se puede llegar a el a no ser a través de el método público **access()**.

Como se ha mencionado previamente, si no se coloca un especificador de acceso para el acceso a la clase esta es “amigable” por defecto. Esto significa que un objeto de esa clase puede ser creado por cualquier otra clase dentro del paquete, pero no fuera del paquete (Se debe recordar que, todos los ficheros dentro del mismo directorio que no tienen la declaración explícita del paquete son implícitamente parte del paquete por defecto de ese directorio). Sin embargo, si un miembro estático de la clase es público, el cliente programador puede acceder a ese miembro estático aún cuando no pueda crear un objeto de esa clase.

## Resumen

En cualquier relación es importante tener límites que sean respetados por todas las partes involucradas. Cuando se crea una librería, se establece una relación con el usuario de esa librería-el cliente programador-que es otro programador, pero uno armando una aplicación o utilizando la librería para armar otra librería mas grandes.

Sin reglas, los clientes programadores pueden hacer cualquier cosa que quieran con todos los miembros de una clase, aún si se prefiere que no se manipule algunos de los miembros. Todo desnudo ante el mundo.

En este capítulo se ve como las clases arman para formar librerías; primero, la forma como un grupo de clases es empaquetada dentro de una librería, y segundo, la forma en como la clase controla el acceso a sus miembros.

Se estima que un proyecto de programación C comienza a estropearse en algún lugar entre 50K y 100K líneas de código a causa de que C es un único “espacio de nombres” así es que los nombres comienzan a colisionar, produciendo un costo operativo extra de administración. En Java, la palabra clave **package**, el nombre del esquema de paquete, y la palabra clave **import** le da un control completo de los nombres así es que el tema de los conflictos de nombres es fácilmente solucionado.

Hay dos razones para controlar el acceso a los miembros, la primera es mantener las manos de los usuarios fuera de las herramientas que no deben tocar; herramientas que son necesarias para las maquinaciones internas de los tipos de datos, pero no son parte de la interfase que los usuarios necesitan para solucionar sus problemas en particular. Así es que el crear métodos y campos del tipo **private** es un servicio para los usuarios porque ellos pueden fácilmente ver que es importante para ellos y que deben ignorar. Esto simplifica el entendimiento de la clase.

La segunda y mas importante razón para controlar el acceso es permitir al diseñador de la librería cambiar el manejo interno de la clase sin preocuparse acerca de como afectará esto al cliente programador. Se puede crear una clase de una forma al inicio, y descubrir que reestructurando su código se proporcionará mucha mas velocidad. Si la interfase y la implementación son separadas claramente y protegidas, se puede lograr esto sin forzar al usuario a volver a escribir su código.

Los especificadores de acceso en Java les da un control de mucho valor a el creador de la clase. El usuario de la clase puede claramente ver exactamente que puede utilizar y que debe ignorar. Mas importante, sin embargo, es la habilidad de asegurar que ningún usuario se vuelva dependiente de alguna parte de la implementación de las capas mas bajas de una clase. Si se sabe esto, como creador de una clase, se puede cambiar la implementación de capas bajas con el conocimiento de que ningún cliente programador será afectado por los cambios porque no pueden acceder a esa parte de la clase.

Cuando se tiene la habilidad de cambiar la implementación de las capas bajas, no solo se puede mejorar el diseño mas adelante, pero también se tiene la libertad de cometer errores. No importa que cuidadoso sea el plan de diseño, se cometerán errores. Sabiendo que es relativamente seguro

cometer estos errores significa que se será mas experimental, se aprende rápido, y se terminan los proyectos mas rápido.

La interfase pública de una clase es lo que el usuario verá, así es que es la parte mas importante de la clase a tener “correcta” durante el análisis y el diseño. Aún cuando se permita alguna libertad de cambio. Si no se tiene la interfase correcta en un primer momento, se puede *agregámas* métodos, mientras que no se puede eliminar algo que los clientes programadores ya estén utilizando en su código.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Escriba un programa que cree un objeto **ArrayList** sin importar explícitamente `java.util.*`.
2. En la sección etiquetada “paquete: la unidad de librería”, coloque los fragmentos de código concernientes a **mypackage** en un grupo de ficheros Java que compilen y corran.
3. En la sección etiquetada “Colisiones”, tome los fragmentos de código, colóquelos dentro de un programa y verifíquese que las colisiones ocurren de hecho.
4. Generalice 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 instrucción `import` en **TestAsser.java** para habilitar y deshabilitar el mecanismo de argumentación.
6. Cree una clase con datos y métodos miembro **public**, **private**, **protected**, y “amigables”. Cree un objeto de esta clase y vea que tipo de mensajes de compilación se obtienen cuando se intenta acceder a los miembros de la clase. Se debe ser consciente de que las clases en el mismo directorio son parte del paquete por “defecto”.
7. Cree una clase con datos del tipo **protected**. Cree una segunda clase en el mismo fichero con un método que manipule los datos protegidos en la primera clase.
8. Cambie la clase **Cookie** como se especifica en la sección “**protected**: un tipo de amistad”. Verifique que **bite()** no es público.
9. En la sección marcada como “Acceso a la clase” se encuentran fragmentos de código descritos en **mylib** y **Widget**. Cree esta librería,

luego créese un **Widget** en una clase que no sea parte del paquete **mylib**.

10. Cree un nuevo directorio y edite la ruta de clases para incluir ese directorio nuevo. Copie el fichero **P.class** (producido compilando **com.bruceeckel.tools.P.java**) en su nuevo directorio y luego cambie los nombres del fichero, la clase **P** dentro y los nombres de métodos (Se debe también agregar salida adicional para ver como trabaja). Cree otro programa en un directorio diferente que utilice la nueva clase.
11. Siguiendo la forma del ejemplo **Lunch.java**, cree una clase llamada **ConnectionManager** que maneje un arreglo fijo de objetos **Connection**. El programador cliente no debe ser capas de crear explícitamente objetos **Connection**, y solo puede obtenerlos mediante un método estático en la clase **ConnectionManager**. Cuando **ConnectionManager** se ejecute sin objetos, debe retornar una referencia nula. Pruebe las clases con un **main()**.
12. Cree el siguiente fichero en el directorio **c05/local** (es de suponer en su ruta de clases):

```
///: c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
    public PackagedClass() {
        System.out.println(
            "Creating a packaged class");
    }
} ///:~
```

Luego créese el siguiente fichero 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();
    }
} ///:~
```

¿Explique por que el compilador genera un error. Hacer la clase **Foreign** parte del paquete **c05.local** cambiaría algo?

# 6: Reutilizando clases

Una de las mas convincentes características de Java es la reutilización de código. Pero para ser revolucionario, debemos ser capaces de hacer mas que copiar y cambiar código.

Esta es la estrategia utilizada en lenguajes procesales como C, y no funcionan muy bien. Como todo en Java, la solución gira alrededor de la clase. Se reutiliza código creando nuevas clases, pero en lugar de crearlas escarbando, se utilizan las clases existentes que alguien ya ha armado y depurado.

El truco es utilizar las clases sin ensuciar el código existente. En este capítulo se verán dos estrategias para lograr esto. La primera es muy directa: Simplemente se crean objetos de su clase existente dentro de la nueva clase. Esto es llamado *composición*, porque la nueva clase esta compuesta con clases existentes. Simplemente se reutiliza la funcionalidad del código, no su forma.

La segunda estrategia es mas sutil. Se crea una nueva clase como un *tipo de* una clase existente. Literalmente se toma la forma de la clase existente y se le agrega código sin modificar la clase existente. Este acto mágico es llamado *herencia*, y el compilador realiza la mayoría del trabajo. La herencia es una de las piedras angulares de la programación orientada a objetos, y tiene implicaciones adicionales que serán exploradas en el Capítulo 7.

Queda claro que la mayoría de la sintaxis y comportamiento son similares para la composición y la herencia (lo cual tiene sentido porque son dos formas de crear nuevos tipos partiendo de tipos existentes). En este capítulo, se aprenderá acerca de estos mecanismos de reutilización de código.

## Sintaxis de la composición

Hasta ahora, la composición ha sido utilizada muy frecuentemente. Simplemente colocamos referencias a objetos dentro de nuevas clases. Por ejemplo, supongamos que se quiere un objeto que almacene varios objetos

**String**, un par de primitivas y un objeto de otra clase. Para los objetos que no son primitivos, se colocan referencias dentro de la nueva clase, pero se pueden definir las primitivas directamente:

```
//: c06:SprinklerSystem.java
// Composition for code reuse.
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()**. Se aprenderá mas adelante que todo objeto no primitivo tiene un método **toString()**, y es llamado en situaciones especiales cuando el compilador quiere una tipo **String** pero tiene uno de estos objetos. Así es que la expresión:

| System.out.println("source = " + source);  
el compilador distingue que esta tratando de agregar un objeto **String** ("source = ") a un **WaterSource**. Esto no tiene sentido para el porque solo se puede “agregar” una cadena a otra cadena, así es que dice “convertiré **source** en una cadena llamando a **toString()**!”. Luego de hacer esto, se puede combinar las dos cadenas y pasar la cadena del tipo **String** a **System.out.println()**. En cualquier momento que se desee este comportamiento para una clase que se cree solo se debe escribir un método **toString()** para esa clase.

En un primer vistazo, se puede asumir-Java siendo tan seguro y cuidadoso como lo es-que el compilador puede automáticamente construir objetos para cada una de las referencias en el código mas arriba; por ejemplo,

llamando el constructor por defecto para **WaterSource** para inicializar **source**. La salida de la instrucción para imprimir es de hecho:

```
valve1 = null  
valve2 = null  
valve3 = null  
valve4 = null  
i = 0  
f = 0.0  
source = null
```

Las primitivas que son campos en una clases son automáticamente inicializados a cero, como se indicó en el capítulo 2. Pero las referencias a objetos son inicializados a **null**, y si se intenta llamar métodos para cualquiera de ellos se obtendrá una excepción. Si es actualmente muy bueno (y útil) que se puede imprimir igualmente sin lanzar una excepción.

Tiene sentido que el compilador no solamente cree un objeto por defecto para cada referencia porque se puede incurrir en una perdida de tiempo innecesaria en muchos casos. Si se quiere inicializar las referencia se puede hacer lo siguiente:

1. En el momento en que el objeto es definido. Esto significa que siempre serán inicializados antes que el constructor sea llamado.
2. En el constructor para esa clase.
3. Justo antes de que se necesite utilizar el objeto. Esto es también llamado *lazy initialization (inicialización perezosa)*. Esta puede reducir la sobrecarga en situaciones donde el objeto no necesita ser creado cada vez.

Todos estas estrategias son mostradas aquí:

```
//: c06: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();
    }
} //:~

```

Note que en el constructor **Bath** una instrucción es ejecutada antes que cualquier inicialización tenga lugar. Cuando no se inicializa en el punto de la definición, no se garantiza que se ejecute ninguna inicialización cuando se envíe un mensaje a una referencia a un objeto-exceptuando por la inevitable excepción en tiempo de ejecución.

He aquí la salida para el programa:

```

Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed

```

Cuando **print()** es llamado llena en **s4** así es que todos los campos son inicializados propiamente en el momento en que son utilizados.

## Sintaxis de la herencia

La herencia es una parte integral de Java (y de los lenguajes de POO en general). Esto quiere decir que siempre está heredando cuando se crea una clase, porque a no ser que usted explícitamente herede de alguna otra clase, implicitamente se hereda de la clase raíz estándar de Java llamada **Object**.

La sintaxis para la composición es obvia, pero para realizar herencia hay una forma inconfundiblemente diferente. Cuando se hereda, se dice “Esta nueva

clase es como esa vieja clase". Se enuncia esto en el código dando el nombre de la clase como es usual, pero antes de abrir las llaves del cuerpo de la clase, se coloca la palabra clave **extends** seguido por el nombre de la *clase base*. Cuando se realiza esto, automáticamente se obtienen todos los datos y métodos miembro de la clase base. He aquí un ejemplo:

```
//: c06: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 deja en evidencia algunas características. Primero, en el método **append()** en la clase **Cleanser**, las cadenas son concatenadas a **s** utilizando el operador **+=**, que es uno de los operadores (junto con '+') que los diseñadores de Java "sobrecargan" para trabajar con tipos **Strings**.

Segundo, ambos, **Cleanser** y **Detergent** contienen un método **main()**. Se puede crear un **main()** para cada una de sus clases, y esto es a menudo recomendado codificar de esta forma así su código de prueba es envuelto con la clase. Cada vez que tenga un montón de clases en un programa, solo el **main()** para la clase invocada en la línea de comandos será llamado (con tal de que **main()** sea público, no importa si la clase de la cual es parte es

pública). Así es que en este caso, cuando se diga **java Detergent**, **Detergent.main()** será llamado. Pero también se puede decir **java Cleanser** para invocar **Cleanser.main()**, aún si **Cleanser** no es una clase pública. Esta técnica de colocar un **main()** en cada clase permite probar unidades fácilmente para cada clase. Y si no se necesita quitar el **main()** cuando termina la prueba, se puede dejar para pruebas posteriores.

Aquí, se puede ver que **Detergent.main()** llama a **Cleanser.main()** explícitamente, pasándole los mismos argumentos de la línea de comando (sin embargo, se puede pasar los argumentos en un arreglo del tipo **String**).

Es importante que todos los métodos en **Cleanser** sean del tipo **public**. Recuerde que si deja de indicar algún especificador de acceso el miembro por defecto será “amigable”, que permite el acceso solo a los miembros del paquete. De esta manera, *dentro de este paquete*, cualquiera podría utilizar esos métodos sin tener acceso específico. **Detergent** no tendrá problemas, por ejemplo. Sin embargo, si una clase de otro paquete donde se herede de **Clenaser** podría acceder solo a los miembros públicos. Así es que para planear herencia, como regla general se debe hacer todos los campos del tipo **private** y todos los métodos del tipo **public** (los miembros protegidos también permiten acceso a las clases derivadas; se aprenderá acerca de esto mas tarde). Claro, en clases particulares se deberá hacer ajustes pero esto es en líneas generales.

Se puede ver que **Cleanser** tiene un grupo de métodos en su interfase: **append()**, **dilute()**, **apply()**, **scrub()** y **print()**. Dado que **Detergent** es *derivada de Cleanser* (mediante la palabra clave **extends**) automáticamente toma todos esos métodos en su interfase, incluso si se ven definidos explícitamente en **Detergent**. Se puede pensar en herencia, entonces, como una *reutilización de la interfase* (la implementación también viene con ella, pero esa parte no es el principal punto).

Como se ha visto en **scrub()**, es posible tomar un método que ha sido definido en la clase base y modificarlo. En este caso, de debe querer llamar el método de la clase base dentro de la nueva versión. Pero dentro de **scrub()** no puede simplemente llamar a **scrub()**, dado que producirá una llamada recursiva, algo que no se desea. Para solucionar este problema Java tiene la palabra clave **super** que se refiere a la “super clase” que es la clase de la cual se ha heredado. De esta forma la expresión **super.scrub()** llama la versión de la clase base del método **scrub()**.

Cuando se hereda no se está restringido a utilizar los métodos de la clase base. Se puede agregar nuevos métodos a la clase derivada exactamente de la misma forma en la que se colocan métodos en una clase: simplemente definiéndolos. El método **foam()** es un ejemplo de esto.

En **Detergent.main()** se puede ver que para un objeto **Detergent** se puede llamar todos los métodos que están disponibles en **Cleanser** de la misma forma que en **Detergent** (i.e., **foam()**).

## Inicializando la clase base

Dado que hay ahora dos clases involucradas-la clase base y la clase derivada-en lugar de simplemente una, puede ser un poco confuso intentar imaginar el objeto resultante producido de una clase derivada. De afuera, se ve como que la nueva clase tiene la misma interfase que la clase base y tal vez algunos métodos y campos adicionales. Pero la herencia no es solamente una copia de la interfase de la clase base. Cuando se crea un objeto de la clase derivada, esta contiene con ella un *subobjeto* de la clase base. Este subobjeto es el mismo que si se creara un objeto de la clase base. Es simplemente eso, desde afuera, el subobjeto de la clase base es envuelto con el objeto de la clase derivada.

Claro, es esencial que el subobjeto de la clase base sea inicializado correctamente y solo hay una manera de garantizar esto: realizar la inicialización en el constructor, llamando el constructor de la clase base, que tiene el conocimiento apropiado y los privilegios para realizar la inicialización de la clase base. Java automáticamente agrega llamadas a el constructor de la clase base. El siguiente ejemplo muestra este trabajo con tres niveles de herencia:

```
//: c06: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 para este programa muestra las llamadas automáticas:

```
Art constructor
Drawing constructor
Cartoon constructor
```

Se puede ver que la construcción se sucede de la base “hacia afuera”, así es que la clase base es inicializada antes que los constructores de la clase derivada puedan acceder a el.

Aún si no se crea un constructor para **Cartoon()**, el compilador sintetizará un constructor por defecto para que se llama a la clase base del constructor.

## Constructores con argumentos

El ejemplo anterior tiene constructores por defecto; esto es, no tienen argumentos. Es fácil para el compilador llamar estos porque no hay preguntas acerca de que argumentos pasar. Si su clase no tiene argumentos por defecto, o no se quiere llamar un constructor de la clase base que tenga argumentos, se debe explícitamente escribir las llamadas a el constructor de la clase base utilizando la palabra clave **super** y la lista de argumentos apropiada:

```
//: c06: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 se llama el constructor de la clase base en **BoardGame()**, el compilador se quejará de que no puede encontrar un constructor de la forma **Game()**. Además, la llamada a el constructor de la clase base *debe ser* la primer cosa que haga en el constructor de la clase derivada (El compilador lo recordará si esto se realiza mal).

## Capturando excepciones del constructor base

Como se habrá notado, el compilador fuerza que se coloque la llamada al constructor base primero en el cuerpo del constructor de la clase derivada. Esto significa que nada mas puede ir antes que esto. Como se verá en el

capítulo 10, esto impide que un constructor de una clase derivada capture cualquier excepción que provenga de la clase base. Esto puede ser inconveniente por momentos.

# Combinando composición y herencia

Es muy común utilizar la composición y la herencia juntos. el siguiente ejemplo muestra la creación de una clase mas compleja, utilizando herencia y composición, junto con la inicialización del constructor necesario:

```
//: c06: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);
    }
}
} //:~
```

Mientras que el compilador fuerza la inicialización de las clases bases, y que se haga correctamente al comienzo del constructor, no se asegura que inicialice los objetos miembros, así es que se debe recordar prestar atención a esto.

## Garantizando la limpieza adecuada

Java no tiene el concepto de C++ del *destructor*, un método que es automáticamente llamado cuando un objeto es destruido. La razón probablemente es que en Java la práctica es simplemente olvidar los objetos en lugar de destruirlos, permitiendo que el recolector de basura reclame la memoria cuando sea necesaria.

Muchas veces esto esta bien, pero hay momentos en que las clases deben realizar algunas actividades durante su vida que requieren limpieza. Como se ha mencionado en el capítulo 4, se puede conocer cuando el recolector de basura va a ser llamado, o si será llamado. Así es que si a veces se desea limpiar todo para una clase, se debe explícitamente escribir un método especial para hacerlo, y asegurarse que los clientes programadores sepan que deben llamar a este método. Además de todo esto -como se describe en el capítulo 10 ("Manejo de errores con excepciones")- se deberá cuidar contra una excepción colocando la limpieza en una cláusula **finally**.

Considere un ejemplo de un sistema de diseño asistido por computadora que dibuje dibujos en la pantalla:

```

//: c06:CADSystem.java
// Asegurando una limpieza adecuada.
import java.util.*;
```

```

class Shape {
    Shape(int i) {
        System.out.println("Constructor de formas");
    }
    void cleanup() {
        System.out.println("Limpieza de formas");
    }
}
class Circle extends Shape {
    Circle(int i) {
        super(i);
        System.out.println("Dibujando un círculo");
    }
    void cleanup() {
        System.out.println("Borrando un círculo");
        super.cleanup();
    }
}
class Triangle extends Shape {
    Triangle(int i) {
        super(i);
        System.out.println("Dibujando un triángulo");
    }
    void cleanup() {
        System.out.println("Borrando un triángulo");
        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("Dibujando una línea: " +
            start + ", " + end);
    }
    void cleanup() {
        System.out.println("Borrando una línea: " +
            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("Constructor combinado");
    }
}

```

```

void cleanup() {
    System.out.println("CADSystem.cleanup( )");
    // El orden de la limpieza es el reverso
    // del orden de la inicialización
    t.cleanup();
    c.cleanup();
    for(int i = lines.length - 1; i >= 0; i--)
        lines[i].cleanup();
    super.cleanup();
}
public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
        // Código para manejo de excepciones...
    } finally {
        x.cleanup();
    }
}
} //:~

```

Todo en este sistema es un tipo de **Shape** (que es simplemente un tipo de **Object** dado que es heredado implícitamente de la clase raíz). Cada clase redefine el método de limpieza de **Shape** además de llamar la versión de la clase base utilizando **super**. Las clases **Shape** específicas -**Circle**, **Triangle** y **Line**- todas tienen constructores que “dibujan”, a pesar que cualquier método llamado durante la vida del objeto puede ser responsable por hacer algo que necesite limpieza. Cada clase tiene su propio método **cleanup()** para restaurar las cosas a la forma que eran antes que el objeto existiera.

En el **main()** se puede ver dos palabras claves que son nuevas, y no serán introducidas oficialmente hasta el capítulo 10: **try** y **finally**. La palabra clave **try** indica que el bloque que sigue (delimitado por llaves) es una *región protegida* que significa que tiene un tratamiento especial. Uno de esos tratamientos especiales es el código en la cláusula **finally** seguido de la región seguro que *siempre* es ejecutada, no importa como se sale del bloque **try** (con manejo de excepción, es posible dejar un bloque **try** de varias formas no usuales). Aquí, la cláusula **finally** está diciendo “siempre se debe llamar **cleanup()** para **x**, no importa que suceda”. Estas palabras claves serán explicadas en el capítulo 10.

Se puede ver que en el método de limpieza se debe prestar atención a el orden de llamada para la clase base y los métodos para limpieza de los objetos miembros en caso de que un subobjeto dependa de otro. En general, se debe seguir la misma forma que se impone por un compilador C++ con sus destructores: Primero hay que realizar todos los trabajos de limpieza específicos en su clase, en el orden reverso de creación (En general, esto requiere que los elementos de la clase base sigan disponibles). Luego se llama a el método de limpieza de la clase base, como se demuestra aquí. Puede haber muchos casos en donde el tema de la limpieza no sea un problema; se deberá simplemente dejar al recolector de basura que realice el

trabajo. Pero cuando se deba realizar explícitamente, diligencia y atención son requeridos.

## Orden de la recolección de basura

No hay mucho en lo que confiarse cuando comienza la recolección de basura. El recolector de basura puede no ser llamado nunca. Se lo es, puede reclamar objetos en el orden que quiera. Lo mejor es no confiar en la recolección de basura para nada mas que el reclamo de memoria. Si se quiere que se suceda una limpieza, se debe programar métodos propios de limpieza y no confiar en **finalize()** (como ha sido mencionado en el capítulo 4, Java puede ser forzado a llamar a todos los finalizadores).

## Ocultar nombres

Solo los programadores C++ pueden sorprenderse del ocultar nombres, dado que ellos trabajan diferente en ese lenguaje. Si una clase base de Java tiene un método que ha sido sobrecargado varias veces, redefinir ese método en la clase derivada *no* ocultará nada de las versiones de la clase base. De esta manera la sobrecarga trabaja independientemente de si el método fue definido en este nivel o en la clase base:

```
//: c06:Hide.java
// Sobrecargar un nombre de método de una
// clase base en una clase derivada no
// oculta la versión de la clase base-
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 se verá en el siguiente capítulo, es mucho más común pasar por encima métodos utilizando la misma firma y tipo de retorno que la clase base. Esto puede ser confuso de otra forma (que es lo que C++ desaprueba para prevenir lo que probablemente es un error).

## Elegir composición o herencia

Ambas, composición y herencia permiten colocar subobjetos dentro de la nueva clase. Surge la pregunta acerca de la diferencia entre los dos, y cuando utilizar uno u otro.

La composición es generalmente utilizada cuando se quiere que las características de una clase existente dentro de su nueva clase, pero no su interfase. Esto es, incrustar un objeto así se puede utilizar para implementar funcionalidad en su nueva clase, pero el usuario de su nueva clase ve que la interfase que ha definido para la nueva clase en lugar de la interfase del objeto incrustado. Para realizar este objetivo, se incrustan objetos del tipo **private** de clases existentes dentro de su nueva clase.

A veces esto tiene sentido para permitir el usuario de la clase acceder directamente a la composición de su nueva clase; esto es, crear objetos miembro del tipo **public**. Los objetos miembro utilizan la implementación oculta ellos mismos, así es que es una cosa muy segura de hacer. Cuando el usuario sabe que esta ensamblando un manojo de partes, esto hace la interfase más fácil de entender. Un objeto **car** es un buen ejemplo:

```
//: c06: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
public Car() {
    for(int i = 0; i < 4; i++)
        wheel[i] = new Wheel();
}
public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
}
} //:~

```

Porque la composición de un auto es parte del análisis del problema (y no simplemente parte del diseño de capas bajas), hacer los miembros públicos ayuda al cliente programador a entender como utilizar la clase y requiere menos complejidad de código para el creador de la clase. Sin embargo, hay que tener en cuenta que esto es un caso especial y que en general se deberá hacer los campos privados.

Cuando se hereda, se toma una clase existente y se crea una versión especial de ella. En general, esto significa que esta tomando una clase de propósito general y especializándola para una necesidad particular. Con una pequeña reflexión, se podrá ver que no tiene sentido componer un auto utilizando un objeto vehículo -un auto no contiene un vehículo, *es* un vehículo. La relación *es un* es expresada con herencia, y la relación *tiene un* es expresada con composición.

## protected

Ahora que ha sido introducido en el concepto de herencia, la palabra clave **protected** finalmente tiene significado. En un mundo ideal, los miembros del tipo **private** siempre serán duros y rápidos, pero en proyectos reales hay veces que se quiere ocultar cosas de la mayoría del mundo y permitirle acceso a miembros de clases derivadas. La palabra clave **protected** es un cabecero hacia el pragmatismo. Ella dice “Esto es privado tanto como al usuario de la clase le importe, pero disponible para todo el que herede de esta clase o cualquiera en el mismo paquete”. Esto es, **protected** en Java es automáticamente “amigable”.

El mejor camino a tomar es dejar los datos miembros privados -se deberá siempre preservar su derecho de cambiar la implementación de las capas mas bajas. Se puede así conceder acceso controlado a los que hereden la clase a través de métodos protegidos:

```

//: c06: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); }
} //:~

```

Se puede ver que **change()** tiene acceso a **set()** porque esta es **protected**.

## Desarrollo creciente

Una de las ventajas de la herencia es que soporta desarrollo creciente permitiendo la introducción de nuevo código sin causar errores en el código existente. Esto también aísla nuevos errores dentro del nuevo código. Por herencia de una clase existente y funcional y agregando métodos y datos miembro (y redefiniendo métodos existentes), deja el código existente -que alguien más puede seguir utilizando- sin tocar y sin errores. Si se sucede un error, se sabe que es en el código nuevo, que es mucho mas corto y fácil de leer que si se ha modificado el cuerpo del código existente.

Es mas sorprendente como son separadas limpiamente las clases. No se necesita el código fuente de los métodos para reutilizar el código. A lo sumo, simplemente se importa el paquete (Esto es cierto para herencia y composición).

Es importante darse cuenta que el desarrollo de programas es un proceso creciente, tal como el aprendizaje humano. Se puede hacer todo el análisis que se deseé, pero se seguirá sin saber todas las respuestas cuando se comience con un proyecto. Se tendrá mucho mas éxito -y una realimentación inmediata- si se comienza a hacer “crecer” su proyecto como una criatura orgánica que evoluciona en lugar de construir todo de una como un rascacielos.

A pesar de que la herencia para experimentar puede ser una técnica útil, en algún punto después que las cosas se estabilizan se necesitará echar una nueva mirada a su jerarquía con un ojo puesto en colapsarla en una estructura razonable. Recuerde que por debajo de todo, la herencia es tratar de expresar una relación que dice “Esta nueva clase es un *tipo* de esa vieja clase”. El programa no deberá preocuparse de tirar bits por todos lados, en lugar de eso mediante la creación y manipulación de objetos de varios tipos expresar un modelo en términos que provengan del espacio del problema.

# Conversión ascendente (upcasting)

El aspecto mas importante de la herencia no es proporcionar métodos para una 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 clase existente”.

Esta descripción no es solo una forma caprichosa de explicar herencia -esta es soportada directamente por el lenguaje. Como un ejemplo, consideremos una clase base llamada **Instrument** que represente un instrumento musical, una clase derivada llamada **Wind**. Dado que herencia significa que todos los métodos en la clase base están disponibles también en la clase derivada, todos los mensajes que se puedan enviar a la clase base también pueden ser enviados a la clase derivada. Si la clase **Instrument** tiene un método **play()**, así será para los instrumentos **Wind**. Esto significa que puede decir de forma acertada que un objeto **Wind** es también un tipo de **Instrument**. El siguiente ejemplo muestra como el compilador soporta esta noción:

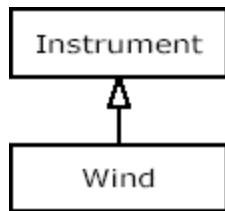
```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}
// Los objetos Wind son instrumentos
// porque ellos tienen la misma interfase:
class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} ///:~
```

Lo que es interesante en este ejemplo es el método **tune()**, que acepta una referencia a **Instrument**. Sin embargo en **Wind.main()** el método **tune()** es llamado dando una referencia a **Wind**. Dado que Java es determinado acerca de la prueba de tipos, suena extraño que un método que solo acepte un tipo acepte fácilmente otro tipo, hasta que se de cuenta que el objeto **Wind** es también un objeto **Instrument**, y no hay método para que **tune()** pueda llamar para un **Instrument** que esté en **Wind**. Dentro de **tune()**, el código trabaja para **Instrument** y cualquier cosa derivada de **Instrument**, y

el acto de convertir una referencia **Wind** en una referencia **Instrument** es llamada *upcasting (conversión ascendente)*

## ¿Por que “upcasting”?

La razón para el término es histórica, y basada en la forma en que los diagramas son dibujados tradicionalmente: con la raíz en la parte superior de la página creciendo hacia abajo (Claro, se puede dibujar los diagramas en la forma en que sean útiles). El diagrama para **Wind.java** es entonces:



Se realizar una conversión de una clase derivada a una clase base moviéndose arriba en el diagrama de herencia, así que comúnmente se dice que realizamos una conversión ascendente (upcasting). Realizar una conversión ascendente es siempre seguro dado que lo estamos haciendo desde un tipo mas específico a un tipo mas general. Esto es, la clase derivada es un subconjunto de la clase base. Esta puede contener mas métodos que la clase base, pero debe contener *al menos* los métodos de la clase base. Lo único que puede ocurrir a la interfase de clase durante la conversión es que pierda métodos, no ganarlos. Esta es por lo que el compilador permite la conversión ascendente sin una conversión explícita o alguna notación especial.

Se puede también realizar la inversa de la conversión ascendente, llamada conversión descendente (downcasting), pero esto involucra un dilema que es el objeto del capítulo 12.

## Revisión de composición contra herencia

En programación orientada a objetos, la mejor forma de crear y utilizar código es simplemente empaquetar datos y métodos juntos en una clase, y utilizar objetos de esa clase. Se utilizarán también clases existentes para crear nuevas clases con composición. Menos frecuentemente, se utilizará herencia. Así es que a pesar que la herencia consigue un gran énfasis cuando se aprende POO, no significa que se deba utilizar en todos lados donde sea posible. Por el contrario, se debe utilizar con moderación, solo cuando la herencia es útil. Una de las formas mas claras de determinar cuando debe utilizar composición o herencia es preguntar si siempre se realizará una

conversión ascendente de su nueva clase a la clase base. Si se debe realizar una conversión ascendente, entonces la herencia es necesaria, pero si no necesita realizar la conversión entonces se debe mirar mas de cerca si se necesita herencia. El siguiente capítulo (polimorfismo) proporciona una de las razones mas convincentes para realizar una conversión ascendente, pero si se recuerda preguntar “¿Necesito realizar una conversión?” tendrá una buena herramienta para decidir entre composición y herencia.

## La palabra clave **final**

La palabra clave final en Java tiene diferentes significados muy sutiles dependiendo del contexto, pero en general indica “Esto no puede ser cambiado”. Es posible que se quiera prevenir cambios por dos razones: diseño y eficiencia. Dado que estas dos razones son muy diferentes, es posible realizar un mal manejo de la palabra clave **final**.

Las siguientes secciones discuten los tres lugares donde la palabra clave **final** puede ser utilizada: para datos, métodos y clases.

### Datos del tipo final

Muchos lenguajes de programación tienen una forma de indicarle al compilador que ese dato es “constante”. Una constante es útil por dos razones:

1. Puede ser una constante *en tiempo de compilación* que se quiere que no cambie nunca.
2. Puede ser un valor inicializado en tiempo de ejecución que no se quiere cambiar.

En el caso de la constante en tiempo de compilación, se le esta permitido a el compilador “plegar” el valor constante en cualquier cálculo en el que sea utilizado; esto es, el calculo puede ser realizado en tiempo de compilación, eliminando algún costo operativo en tiempo de ejecución. En Java, estos tipos de constantes deben ser primitivas y son expresados utilizando la palabra clave **final**. Un valor debe ser dado en el momento en que es definido como constante.

Un campo que es al la ves del tipo **static** y **final** tiene solamente un lugar donde es almacenado así es que no puede ser cambiado.

Cuando se utiliza **final** con referencias a objetos en lugar de primitivas el significado se hace un poco confuso. Con una primitiva, **final** hace *referencia* a una constante. Una vez que la referencia es inicializada a un objeto, no puede ser nunca cambiada para apuntar a otro objeto. Sin embargo, el

objeto puede ser modificado a si mismo; Java no proporciona una forma para hacer cualquier objeto arbitrario una constante (Se puede, sin embargo, escribir una clase propia cuyos objetos tengan el efecto de ser constantes). Esta restricción incluye arreglos, que son también objetos.

He aquí un ejemplo que muestra los detalles de **final**:

```
//: c06: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 VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 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();
    // 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 reference
        //! fd1.a = new int[3];
        fd1.print("fd1");
        System.out.println("Creating new FinalData");
        FinalData fd2 = new FinalData();
        fd1.print("fd1");
        fd2.print("fd2");
    }
} ///:~
```

Dado que **i1** y **VAL\_TWO** son primitivas del tipo **final** con valores en tiempos de compilación, pueden ambos ser utilizados como constantes en tiempo de compilación y no son distintos de alguna forma importante. **VAL\_THREE** es la forma más común que se verán las constantes definidas: del tipo **public** así pueden ser utilizadas fuera del paquete, estáticas para

enfatizar que solo hay una, y **final** para indicar que es una constante. Debe notarse que las primitivas del tipo **final static** con valores constantes iniciales (esto es, constantes en tiempo de compilación) son nombradas con todas las letras mayúsculas por convención, con palabras separadas por infraguiones (exactamente igual que las constantes en C, que es donde la convención se origina). También debe notarse que **i5** no se conoce en tiempo de compilación, así es que no está en mayúsculas.

Solo porque algo es del tipo **final** no significa que su valor es conocido en tiempo de ejecución. Esto es demostrado inicializando **i4** y **i5** en tiempo de ejecución utilizando números generados de forma aleatoria. Esta parte del ejemplo muestra también la diferencia entre hacer un valor **final** del tipo estático o no estático. Esta diferencia muestra solo cuando los valores son inicializados en tiempo de ejecución, dado que los valores en tiempo de compilación son tratados igual por el compilador (Y presumiblemente la optimización no existe). La diferencia es mostrada en la salida de una ejecución:

```
| fd1: i4 = 15, i5 = 9  
| Creating new FinalData  
| fd1: i4 = 15, i5 = 9  
| fd2: i4 = 10, i5 = 9
```

Se puede ver que los valores de **i4** para **fd1** y **fd2** son únicos, pero el valor para **i5** no cambia cuando se crea el segundo objeto **FinalData**. Esto es porque es estático y es inicializado una vez cuando es cargado y no cada vez que un objeto es creado.

Las variables **v1** a **v4** demuestran el significado de una referencia del tipo **final**. Como se puede ver en el **main()**, solo porque **v2** es **final** no significa que se pueda cambiar este valor. Sin embargo, se puede volver a enlazar **v2** a un nuevo objeto, precisamente porque es **final**. Esto es lo que significa **final** para una referencia. Se puede ver el mismo significado es verdadero para un arreglo, que es solo otro tipo de referencias (Que yo sepa no hay forma para hacer a si mismas las referencias a arreglos del tipo **final**). Hacer las referencias del tipo **final** parece menos útil que hacer las primitivas del tipo **final**.

## Finales en blanco (blank finals)

Java permite la creación de tipos *finales en blanco (blank finals)*, que son campos que son declarados como **final** pero no se les da un valor de inicialización. En todos los casos, el final en blanco *debe* ser inicializado antes de utilizarse, y el compilador se asegura de esto. Sin embargo, los finales en blanco proporcionan más flexibilidad en el uso de la palabra clave **final** dado que, por ejemplo, un campo **final** dentro de una clase puede ahora ser diferente para cada objeto y a pesar de eso mantener su calidad de inmutable. He ahí un ejemplo:

```

///: c06: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 reference
    // 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();
    }
} //:~
```

Se está forzado a realizar asignaciones a campos con una expresión en el punto de la definición del campo o en cada constructor. Esta forma garantiza que el campo **final** sea inicializado antes de utilizarse.

## Argumentos finales

Java permite hacer los argumentos del tipo **final** declarándolos como en una lista de argumentos. Esto significa que dentro del método no se puede cambiar donde apunta el argumento con la referencia:

```

///: c06: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
    }
    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);
    }
} //:~
```

Se observa que se puede seguir asignando una referencia **null** a un argumento que es final sin que el compilador lo note, exactamente como se puede con un argumento que no es del tipo **final**.

El método **f()** y **g()** muestra que sucede cuando los argumentos de primitivas son del tipo **final**: se puede leer el argumento, pero no cambiarlo.

## Métodos finales

Hay dos razones para métodos del tipo **final**. La primera es colocar un “bloqueo” en el método para prevenir que cualquier clase heredada pueda cambiar su significado. Esto se realiza por razones de diseño cuando se quiere estar seguro que el comportamiento del método se mantendrá durante la herencia y no pueda ser sobreescrito.

La segunda razón para métodos del tipo **final** es la eficiencia. Si se define un método como **final**, se está permitiendo al compilador convertir todas las llamadas a el método en llamadas *en línea*. Cuando el compilador ve una llamada a método del tipo **final** puede (a su discreción) saltar la estrategia normal de insertar código para realizar el mecanismo de llamada al método (moviendo argumentos a la pila, saltar al código del método y ejecutarlo, regresar y limpiar todos los argumentos de la pila y luego ocuparse del valor de retorno) y en lugar de eso remplazar la llamada al método con una copia del código actual en el cuerpo del método. Claro, si un método es grande, cuando es código comienza a inflarse y probablemente no se vea ninguna ganancia en el rendimiento con la llamada en línea, dado que cualquier mejora será mermada por la cantidad de tiempo gastado dentro del método. Esto significa que el compilador de Java es capaz de detectar estas situaciones y elegir sabiamente cuando ejecutar en línea un método **final**. Sin embargo, es mejor no confiar en que el compilador es capaz de hacer esto y hacer un método **final** solo si es realmente pequeño o si se quiere evitar explícitamente la sobre escritura.

### **final** y **private**

Cualquier método del tipo **private** en una clase es implícitamente **final**. Porque no se puede acceder a un método del tipo **private**, no se puede sobreescribir (aún cuando el compilador no de un mensaje de error si se trata de sobreescribir el método, no se estará sobre escribiendo el método, solo se estará creando un nuevo método). Se puede agregar el especificador **final** a un método del tipo **private** pero no le dará al método significado extra.

Este tema puede causar confusión, dado que si trata de sobreescribir un método privado (que es implícitamente del tipo **final**) parecería funcionar:

```
| //: c06:FinalOverridingIllusion.java
```

```

// It only looks like you can override
// a private or private final method.
class WithFinals {
    // Identical to "private" alone:
    private final void f() {
        System.out.println("WithFinals.f()");
    }
    // Also automatically "final":
    private void g() {
        System.out.println("WithFinals.g()");
    }
}
class OverridingPrivate extends WithFinals {
    private final void f() {
        System.out.println("OverridingPrivate.f()");
    }
    private void g() {
        System.out.println("OverridingPrivate.g()");
    }
}
class OverridingPrivate2
extends OverridingPrivate {
    public final void f() {
        System.out.println("OverridingPrivate2.f()");
    }
    public void g() {
        System.out.println("OverridingPrivate2.g()");
    }
}
public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 =
            new OverridingPrivate2();
        op2.f();
        op2.g();
        // You can upcast:
        OverridingPrivate op = op2;
        // But you can't call the methods:
        //!! op.f();
        //!! op.g();
        // Same here:
        WithFinals wf = op2;
        //!! wf.f();
        //!! wf.g();
    }
} ///:~

```

La “sobre escritura” solo puede suceder si algo es parte de la interfase de la clase base. Esto es, se debe ser capaz de realizar una conversión ascendente a un objeto a su tipo base y llamar al mismo método (el punto de esto será mas claro en el siguiente capítulo). Si un método es privado, no es parte de la interfase de la clase base. Es simplemente código oculto fuera del interior de la clases, y solo sucede que tiene ese nombre, pero si crea un método del tipo **public**, **protected** o “amigable” en la clase derivada, no hay conexión con

el método que puede ser que tenga el nombre de la clase base. Dado que un método **private** es inalcanzable y en efecto invisible, no afecta en nada a no ser por la organización del código de la clase para la cual fue definido.

## Clases finales

Cuando se dice que una clase entera es **final** (a causa de que su definición es precedida por la palabra clave **final**), se declara que no se quiere heredar de esta clase o permitir que nadie mas pueda hacerlo. En otras palabras, por alguna razón el diseño de su clase en tal que nunca se necesitará realizar cambios, por seguridad o por razones de seguridad no se quiere que se puedan crear subclases. Alternativamente, habrá que manejar un tema de eficiencia, y si se desea asegurarse que alguna actividad relacionada con objetos de esa clase sean lo mas eficientes posibles.

```
//: c06: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++;
    }
} //:~
```

Se puede ver que los miembros pueden ser finales o no, como se elija. Las mismas reglas se aplican con **final** a los datos independientemente de si la clase es definida como final. Definiendo la clase como **final** simplemente previene la herencia-nada mas. Sin embargo, dado que previene la herencia todos los métodos en una clase **final** son implícitamente **final**, dado que no hay forma de sobrescribirlos. Así es que el compilador tiene las mismas opciones de eficiencia que si se declara un método **final**.

Se puede agregar el especificador **final** a un método en una clase **final**, pero no agrega ningún significado.

## Precauciones con final

Se puede ser propenso a hacer un método **final** cuando se esta diseñando una clase. Se puede sentir que la eficiencia es muy importante cuando se utiliza la clase y que nadie quiera sobreescribir sus métodos de cualquier forma. A veces esto es verdad.

Pero hay que ser cuidadoso con lo que se asume. En general, es difícil anticipar que clase pueda ser utilizada, especialmente en una clase de propósito general. Si se define un método como **final** se debe prevenir la posibilidad de reutilizar su clase a través de la herencia en proyectos de otros programadores simplemente porque uno no se imagina que pueda ser utilizada de esa forma.

La librería estándar de Java es un buen ejemplo de esto. En particular, la clase **Vector** de Java 1.0/1.1 fue comúnmente utilizada y parece haber tenido mas utilidades, si en nombre de la eficiencia, todos los métodos no hubieran sido declarados **final**. Es fácil de imaginar que se pueda querer heredar o sobreescribir esta clase en otra útil, pero los diseñadores de alguna forma decidieron que eso no era apropiado. Esto es irónico por dos razones.

Primero, **Stack** es heredada de **Vector**, que dice que **Stack** es un **Vector**, lo que realmente no es verdad desde un punto de vista lógico. Segundo, muchos de los métodos mas importantes de **Vector**, como lo es **addElement()** y **elementAt()** son **synchronized**. Como se verá en el capítulo 14, esto incurre en una sobrecarga importante en el rendimiento que probablemente anula cualquier ganancia proporcionada por **final**. Esto favorece a la teoría que los programadores son regularmente malos a la hora de adivinar donde se suelen suceder las optimizaciones. Esto simplemente es malo que un diseño defectuoso dentro de la librería estándar con la que debemos lidiar todos nosotros (Afortunadamente, Java 2 contiene librerías que remplazan **Vector** con **ArrayList**, donde el comportamiento es mucho mas civilizado. Desafortunadamente, hay una abundancia de código nuevo que fue estrieto que utiliza el viejo contenedor de la librería).

Es también interesante hacer notar que **Hashtable**, otra clase importante de la librería estándar, *no* tiene método finales. Como se ha mencionado en alguna parte de este libro, es obvio que algunas clases fueron diseñadas por personas totalmente diferentes (Se puede ver que los nombres de los métodos en **Hashtable** son mucho mas cortos comparados con los de la clase **Vector**, otra pieza de evidencia). Esto es precisamente el tipo de cosas que *no* suelen ser obvios para los consumidores de la librerías de clases. Cuando las cosas son inconsistentes simplemente crea mas trabajo para el usuario. sin embargo otra apología para el valor del diseño y al desarrollo del código (se puede ver que en Java 2 la librería **Hashtable** es reemplazada con **HashMap**).

# Inicialización y carga de clase

En los lenguajes mas tradicionales, los programas son cargados todos como una parte del proceso de carga. Esto es seguido por la inicialización, y luego el programa comienza. En el proceso de inicialización en estos lenguajes se debe ser cuidadoso controlando de tal forma que el orden de inicialización de las partes estáticas no causen problemas. En C++, por ejemplo, hay problemas si algo del tipo **static** espera otra cosa del tipo **static** para ser válido antes que el segundo sea inicializado.

Java no tiene este problema porque tiene otra estrategia para cargarse. Dado que todo en Java es un objeto, muchas actividades se hacer simples, y esta es una de ellas. Como se aprenderá mejor en el siguiente capítulo, el código compilado para cada clase existe en su propio fichero por separado.

Esto no es cargado hasta que el código es necesitado. En general, se puede decir que “El código de la clase es cargado en el punto en que se necesita por primera vez”. Esto a menudo no es hasta que el primer objeto de la clase es construido, pero la carga también ocurre cuando un campo estático o un método estático es accedido.

El punto en el que se utiliza por primera ves es también cuando una inicialización del tipo estático se sucede. Todos los objetos estáticos y el bloque de código estático será inicializado en el orden textual (esto es, el orden en el que se escribe en la definición de la clase) en el punto de carga. Las cosas del tipo **static**, claro, son inicializadas solo una vez.

## Inicialización con herencia

Es útil ver la totalidad del proceso de inicialización, incluida la herencia, para obtener una imagen completa de lo que sucede. Considerando el siguiente código:

```
//: c06:Beetle.java
// The full process of initialization.
class Insect {
    int i = 9;
    int j;
    Insect() {
        prt("i = " + i + ", j = " + j);
        j = 39;
    }
    static int xl =
        prt("static Insect.xl 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");
    public static void main(String[] args) {
        prt("Beetle constructor");
        Beetle b = new Beetle();
    }
} //:~

```

La salida de este programa es:

```

static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39

```

La primera cosa que sucede cuando se ejecuta Java en **Beetle** es que se intenta acceder **Beetle.main()** (un método estático), así es que primero se busca el código compilado para la clase **Beetle** (esto sucede que esta en un fichero llamado **Beetle.class**). En el procedo de cargarla, se advierte que tiene una clase base (esto es lo que la palabra clave **extends** indica), con lo cual es entonces cargada. Esto sucederá si se esta creando un objeto o no de la clase base (Se puede intente comentar la creación del objeto para probar esto).

En la clase base, la segunda clase base será cargada y así sucesivamente. Esto es importante porque la inicialización estática de la clase derivada puede depender de que un miembro de la clase base se inicializado propiamente.

En este punto, las clases necesarias han sido todas cargadas así es que el objeto puede ser creado. Primero, todas las primitivas en este objeto son configuradas a sus valores por defecto y las referencias a objetos con iniciadas a **null**-esto sucede con una violenta puesta a cero del área de memoria del objeto a cero. Entonces el constructor de la clase base será llamado. En este caso la llamada es automática, pero se puede también especificar la llamada al constructor de la clase base (como la primera operación del constructor **Beetle()**) utilizando **super**. La construcción de la clase base realiza el mismo proceso en el mismo orden que el constructor de la clase derivada. Después que el constructor de la clase base se completa,

las variables de la instancia son inicializadas en orden textual. Finalmente, el resto del cuerpo de constructor es ejecutado.

## Resumen

Herencia y composición permiten crear un nuevo tipo a partir de tipos existentes. Típicamente, sin embargo, se utiliza composición para reutilizar tipos existentes como parte de la implementación de las capas mas bajas del nuevo tipo, y herencia cuando se quiere reutilizar la interfase. Dado que la clase derivada tiene la interfase de la clase base, se puede realizar una conversión ascendente a la base, que es crítico para realizar polimorfismo, como se verá en el siguiente capítulo.

A pesar del fuerte énfasis de la herencia en la programación orientada a objetos, cuando se comienza a diseñar generalmente se prefiere la composición al principio y utilizar herencia solo cuando es claramente necesaria. Composición tiende a ser mas flexible. Además, utilizando el artificio de la herencia con el tipo de miembro, se puede cambiar el tipo exacto, y de esta manera el comportamiento, de aquellos objetos miembro en tiempo de ejecución. Por lo tanto, se puede cambiar el comportamiento del objeto compuesto en tiempo de ejecución.

A pesar de que la reutilización de código a través de la composición y la herencia es útil para un desarrollo rápido del proyecto, generalmente se quiere rediseñar la jerarquía de clases antes de permitir que otros programadores comiencen a depender de ella. La meta es una jerarquía en donde cada clase tenga un uso específico y no sean ni muy grandes (que abarquen mucha funcionalidad que es muy difícil de reutilizar) tampoco de manera irritante pequeña (no se puede utilizar por si mismo o sin agregar funcionalidad).

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Cree dos clases, **A** y **B**, con constructores por defecto (listas de argumentos vacías) que se anuncien a si mismos. Herede una nueva clase llamada **C** de **A**, y cree un objeto de clase **C** dentro de **C**. No cree un constructor para **C**. Cree un objeto de la clase **C** y observe los resultados.
2. Modifique el ejercicio 1 de tal forma que **A** y **B** tengan constructores con argumentos en lugar de los constructores por defecto. Escriba un

constructor para **C** y realizar todas las inicializaciones con el constructor de **C**.

3. Cree una clase simple. Dentro de una segunda clase, defina un campo para un objeto de la primer clase. Use inicialización “lazy” para crear una instancia de este objeto.
4. Herede una nueva clase a partir de la clase **Detergent**. Sobre escriba **scrub()** y agregue un nuevo método llamado **sterilize()**.
5. Tome el fichero **Cartoon.java** y comente el constructor para la clase **Cartoon**. Explique que sucede.
6. Tome el fichero **Chess.java** y comente el constructor para la clase **Chess**. Explique que sucede.
7. Pruebe que por defecto el compilador crea por usted constructores.
8. Pruebe que los constructores de la clase base son (a) siempre llamados, y (b) llamados antes del constructor de la clase derivada.
9. Cree una clase base con un solo constructor que no sea por defecto, y derive la clase con ambos, uno constructor por defecto y otro que no sea por defecto. En los constructores de la clase derivada, llama al constructor de la clase base.
10. Cree una clase llamada **Root** que contenga una instancia para cada una de las clases (las que se deben crear también) llamada **Component1**, **Component2**, y **Component3**. Derive de **Root** una clase **Stem** que contenga una instancia de cada “componente”. Todas las clases deben tener constructores por defecto que impriman un mensaje acerca de la clase.
11. Modifique el ejercicio 10 así cada clase solo tienen constructores que no sean por defecto.
12. Agregue una organización de métodos **cleanup()** adecuados para todas las clases del ejercicio 11.
13. Cree una clase con un método que sea sobrecargado tres veces. Herede una nueva clase, agréguese una nueva sobre escritura del método, y muestre cual de todos esos cuatro métodos está disponible en la clase derivada.
14. En los métodos **Car.java** agregue un método **service()** de **Engine** y llame a este método en **main()**.
15. Cree una clase dentro de un paquete. La clase debe contener un método protegido. Fuera del paquete, intente llamar el método protegido y explique los resultados. Ahora herede de la clase y llame a el método protegido desde adentro de un método de la clase derivada.

16. Cree una clase llamada **Amphibian**. Desde esta, herede una clase llamada **Frog**. Coloque métodos apropiados en la clase base. En el **main()**, cree un **Frog** y realice una conversión ascendente de este a **Anphibian**, y demuestre que todos los métodos siguen trabajando.
17. Modifique el ejercicio 16 de tal forma que **Frog** sobre escriba las definiciones de métodos de la clase base (proporcione nuevas definiciones utilizando las mismas firmas). Observe que sucede en **main()**.
18. Cree una clase con un campo estático **final** y un campo **final** y demuestre la diferencia entre los dos.
19. Cree una clase con una referencia **final** en blanco a un objeto. Realice la inicialización del **final** en blanco dentro de un método (no del constructor) justo antes de utilizarlo. Demuestre la garantía que la referencia **final** debe ser inicializada antes de utilizarse, y que no puede ser cambiada luego de que es inicializada.
20. Cree una clase con un método **final**. Herede desde esta clase y intente sobreescribir ese método.
21. Cree una clase **final** e intente heredar de ella.
22. Pruebe que la carga de una clase se realiza solo una vez. Pruebe que esa carga puede ser producida por la creación de una primera instancia de la clase o al acceder a un miembro estático.
23. En **Beetle.java**, herede un tipo específico de escarabajo (beetle) de una clase **Beetle**, seguido del mismo formato que la clase existente. Rastréela y explique la salida.

# 7: Polimorfismo

Polimorfismo es la tercera característica esencial de la programación orientada a objetos, luego de la abstracción de datos y la herencia.

Este promete otra dimensión de separación de la interfase de la implementación, para desacoplar *que* de *como*. Polimorfismo permite mejorar la organización del código y la legibilidad de la misma forma que facilita la creación de programas *extensibles* que pueden “crecer” no solo durante la creación original del proyecto sino que también cuando se desean nuevas herramientas.

La encapsulación crea nuevos tipos de datos combinando características y comportamientos. La implementación oculta separa la interfase de la implementación haciendo los detalles del tipo **private**. Este tipo de mecanismos de organización tiene sentido para alguien una base de programación procesal. Pero el polimorfismo maneja el desacople en términos de *tipos*. En el capítulo anterior, se puede ver como se permite el tratamiento del objeto por su tipo *o* por su tipo base. Esta habilidad es crítica porque permite que muchos tipos (derivados del mismo tipo base) sean tratados como si fueran de un solo tipo, y un mismo código trabajara todos esos diferentes tipos de la misma forma. El método polimórfico llamado permite un tipo para expresar su distinción de otro, tipo similar, siempre y cuando ambos sean derivados del mismo tipo base. Esta distinción es expresada a través de diferencias en el comportamiento de los métodos que de llaman a través de la clase base.

En este capítulo, se aprenderá acerca de polimorfismo (también llamado *enlazado dinámico* o *enlazado diferido* o *enlazado en tiempo de ejecución*) comenzando por lo básico, con ejemplos simples que aclararán todo menos el comportamiento polimórfico del programa.

## Revisión de conversión ascendente

En el capítulo 6 se pudo ver como un objeto puede ser utilizado como su propio tipo o como un objeto de su tipo base. Tomar una referencia a un objeto y tratarla como una referencia a su tipo base es llamado *conversión*

*ascendente*, porque la forma en que los árboles de herencia son dibujados es con la clase base en la parte más alta.

Se puede ver también que se origina un problema, que es mostrado en el siguiente código:

```
//: c07:music:Music.java
// Herencia & upcasting.
class Note {
    private int value;
    private Note(int val) { value = val; }
    public static final Note
        MIDDLE_C = new Note(0),
        C_SHARP = new Note(1),
        B_FLAT = new Note(2);
    } // Etc.
class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}
// Los objetos Wind son instrumentos
// porque ellos tienen la misma interfase:
class Wind extends Instrument {
    // Redefinición del método de la interfase:
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}
public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
    }
} ///:~
```

El método **Music.tune()** acepta una referencia a **Instrument**, pero también cualquier cosa derivada de **Instrument**. En el **main()**, se puede que sucede si una referencia a **Wind** es pasada a **tune()**, sin conversión necesaria. Esto es aceptable; la interfase en **Instrument** debe existir en **Wind**, porque **Wind** hereda de **Instrument**. El realizar una conversión ascendente de **Wind** a **Instrument** puede “estrechar” la interfase, pero no se puede tener menos que la interfase completa de **Instrument**.

## Olvidándose del tipo de objeto

¿Este programa puede ser extraño. Porque alguien intencionalmente se *olvida* del tipo de objeto? Esto es lo que sucede cuando se realiza una

conversión ascendente, y se ve mejor si **tune()** simplemente toma una referencia **Wind** como su argumento. Esto plantea un punto esencial: Si se hace esto, no se necesita escribir un nuevo **tune()** para cada tipo de **Instrumento** en su sistema. Si seguimos este razonamiento y agregamos instrumentos **Stringed** y **Brass**:

```
//: c07:music2:Music2.java
// Sobrecargar en lugar de realizar upcasting.
class Note {
    private int value;
    private Note(int val) { value = val; }
    public static final Note
        MIDDLE_C = new Note(0),
        C_SHARP = new Note(1),
        B_FLAT = new Note(2);
} // Etc.
class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}
class Wind extends Instrument {
    public void play(Note n) {
        System.out.println("Wind.play()");
    }
}
class Stringed extends Instrument {
    public void play(Note n) {
        System.out.println("Stringed.play()");
    }
}
class Brass extends Instrument {
    public void play(Note n) {
        System.out.println("Brass.play()");
    }
}
public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn);
    }
}
```

| } //:/~

Esto trabaja, pero tiene un gran inconveniente: Debe escribir métodos específicos para cada nueva clase **Instrument** que se agregue. Esto significa mucho mas programación en primer lugar, pero también significa que si quiere agregar un nuevo método como **tune()** o un nuevo tipo de **Instrumento**, se tiene un gran trabajo para hacer. Agregado el echo de que el compilador no dará ningún mensaje de error si se olvida sobrecargar algún método y la totalidad del trabajo con los tipos se vuelve inmanejable.

¿No sería mas agradable si se pudiera simplemente escribir un solo método que tome la clase base y su argumento, y nada de la clase derivada específica? Esto es, no sería agradable si se pudiera olvidar que hay clases derivadas y escribir el código para manejarse solo con la clase base?

Esto es exactamente lo que el polimorfismo permite hacer. Sin embargo, muchos programadores que tienen una base de programación procesal tienen pequeños problemas con la forma en que el polimorfismo trabaja.

## La vuelta

La dificultad con **Music.java** pueden verse ejecutando el programa. La salida es **Wind.play()**. Esto claramente es la salida deseada, pero esto no parece tener sentido que vaya a funcionar de esa forma. Vea el método **tune()**:

```
public static void tune(Instrument i) {  
    // ...  
    i.play(Note.MIDDLE_C);  
}
```

Este recibe una referencia **Instrument**. Así es que como es posible que el compilador conozca que esta referencia a **Instrument** apunta a **Wind** en este caso y no a **Brass** o **Stringed**? El compilador no puede hacerlo. Para lograr un entendimiento mas profundo de este tema, ayuda examinar el tema de enlazado (*binding*).

## Enlazado de llamadas a método

Conectar una llamada a un método a el cuerpo de un método es llamado enlazado. Cuando se realiza un enlazado antes que el programa se ejecute (por el compilador y el enlazador, si hay alguno), esto es llamado *enlazado con anticipación (early binding)*. Es posible que no se haya escuchado el término antes porque nunca ha sido una opción en lenguajes procesales. Los compiladores C solo tienen un tipo de llamada a métodos, y esto es enlazado con anticipación.

La parte confusa de los programas anteriores gira alrededor del enlazado con anticipación porque el compilador no puede saber el método correcto a llamar cuando es solo una referencia a **Instrument**.

La solución es llamada *enlazado a última hora(late binding)*, que significa que el enlace sucede en tiempo de ejecución basado en el tipo de objeto. Enlaces a última hora son llamados también *enlaces dinámicos (dynamic binding)* o *enlaces en tiempo de ejecución(run-time binding)*. Cuando un lenguaje implementa enlazado a última hora, debe tener algún mecanismo para determinar el tipo del objeto en tiempo de ejecución y llamar al método apropiado. Esto es, el compilador no sabe el tipo de objeto, pero el mecanismo de llamada a el método lo encuentra y llama al cuerpo del método apropiado. Los mecanismos de enlazado a última hora varían de lenguaje a lenguaje, pero se puede imaginar que algún tipo de información debe ser instalada en los objetos.

Todos los métodos de enlazado de Java utiliza enlazado a última hora a menos que sea declarado como **final**. Esto significa que por lo general no se necesita hacer ninguna decisión acerca de si se debe suceder un enlazado a última hora debe ocurrir -esto sucede automáticamente.

¿Por que se declararía un método final? Como se ha notado en el último capítulo, para prevenir que alguien sobre escriba el método. Tal vez mas importante, efectivamente “desactiva” el enlazado dinámico, o mas bien indica al compilador que el enlazado dinámico no es necesario. Esto permite al compilador generar al compilador general código ligeramente mas eficiente para las llamadas a métodos finales. Sin embargo, en muchos casos no se realizar ninguna diferencia de rendimientos global en el programa, así es que es mejor utilizar **final** como una decisión de diseño, y no como un intento de mejorar el rendimiento.

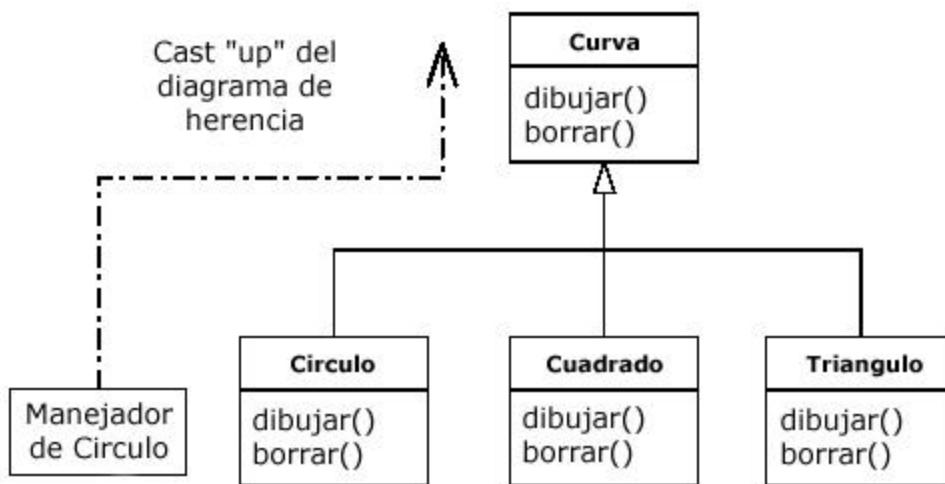
## Produciendo el comportamiento adecuado

Una vez que se conoce como todos los métodos se enlazan en Java sucede de forma polimórfica mediante enlazado a última hora, se puede escribir el código para comunicarse con la clase base y saber que todas las clases derivadas trabajaran correctamente utilizando el mismo código. O puesto de otra forma, se “envía un mensaje a un objeto y se deja que el objeto se figure que es lo correcto para hacer”.

El ejemplo de programación orientada a objetos clásico es el de la “curva”. Esto es comúnmente utilizado porque es fácil de visualizar, pero desafortunadamente puede confundir a los programadores principiantes y

hacer que piensen que la programación orientada a objetos es para programación gráfica el cual no es claro el caso.

El ejemplo de las curvas es la clase base llamada **Curva** y varios tipos derivados: **Círculo**, **Cuadrado**, **Triangulo**, etc. La razón de que el ejemplo trabaje tan bien es la facilidad para decir “un círculo es un tipo de curva” y entender los diagramas que muestran las relaciones:



La conversión puede ocurrir en una instrucción tan simple como:

```
| Curva s = new Círculo();
```

Aquí, un objeto **Círculo** es creado y la referencia que resulta es inmediatamente asignada a **Curva**, que parecería ser un error (asignación de un tipo a otro); y aún está bien porque un **Círculo** es una **Curva** por herencia. Así es que el compilador está de acuerdo con la instrucción y no emite un mensaje de error.

Supóngase que se llama un método de la clase base (que han sido sobreescrito en la clase derivada):

```
| s.dibujar();
```

Nuevamente, se debe esperar que el método **dibujar()** de **Curva** sea llamado porque esto es, después de todo, una referencia a una **Curva** -así es que ¿cómo puede el compilador saber hacer otra cosa? Y a pesar de todo el **Círculo.dibujar()** adecuado es llamado gracias a el enlazado a última hora (polimorfismo).

El siguiente ejemplo lo coloca de una forma ligeramente diferente:

```
//: c07: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:
                    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 la interfase común para todas las cosas heredadas de **Shape** -esto es, todas las curvas pueden ser dibujadas y borradas. Las clases derivadas sobre escriben estas definiciones para proporcionar un único comportamiento para cada tipo específico de forma.

La clase principal **Shape** contiene un método estático llamado **randShape()** que produce una referencia a un objeto del tipo **Shape** seleccionado de forma aleatoria cada vez que se lo llama. Se puede ver que la conversión

ascendente se sucede en cada una de las instrucciones **return**, que toman una referencia a un **Circle**, **Square**, o **Triangle** y la envía fuera de los métodos como tipo de retorno, **Shape**. Así es que cada vez que se llama a este método nunca se tendrá la posibilidad de saber que tipo específico es, dado que siempre se entrega una referencia **Shape** plana.

**main()** contiene un arreglo de referencias **Shape** que es llenado mediante llamadas a **randShape()**. En este punto se debe conocer que se tienen Curvas, pero no sabe nada más específico acerca de ellas (y tampoco el compilador). Sin embargo, cuando se mueve a través del arreglo y se llama a **draw()** para cada uno, el comportamiento correcto del tipo mágicamente se sucede, como se puede ver en el ejemplo de salida:

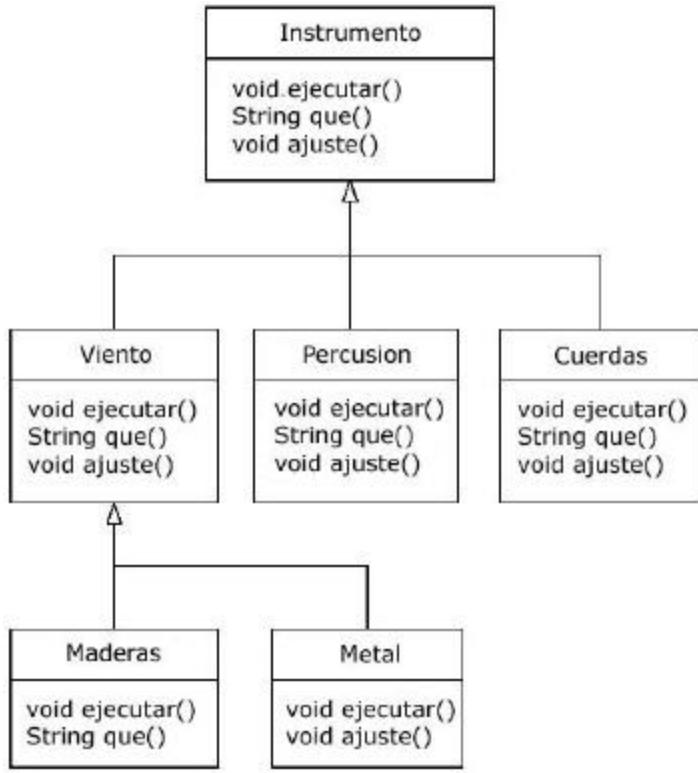
```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

Claro que, dado que las curvas son elegidas de forma aleatoria cada vez, cada vez que se ejecute se tendrán resultados diferentes. El punto de elegir las curvas de forma aleatoria es persuadir a entender que el compilador puede no tener especial conocimiento que le permita hacer la llamada correcta en tiempo de ejecución. Todas las llamadas a **draw()** son echas a través de enlazado dinámico.

## Extensibilidad

Ahora vamos a regresar a el ejemplo de los instrumentos musicales. Gracias al polimorfismo, se pueden agregar la cantidad de tipos que se deseen a el sistema sin cambiar el método **tune()**. En un programa orientado a objetos bien diseñado, muchos de todos los métodos seguirán el modelo de **tune()** y se comunicarán solamente con la interfase de la clase base. Como un programa es *extensible* porque se puede agregar nuevas funcionalidades heredando nuevos tipos de datos de una clase base en común. Los métodos que manipulan la interfase de la clase base no necesitarán ser cambiados en absoluto para acomodarse a las nuevas clases.

Considere que sucede si se toma el ejemplo de los instrumentos y se agrega mas métodos en la clase base y en algunos de las clases nuevas. He aquí un diagrama:



Todas estas nuevas clases funcionan correctamente con el viejo, método **ajuste()** sin modificar. Aún si **ajuste()** se encuentra en un fichero separado y nuevos métodos son agregados a la interfase de **Instrumento**, **ajuste()** trabaja correctamente sin compilar nuevamente. He aquí una implementación del siguiente diagrama:

```

//: c07:music3:Music3.java
// An extensible program.
import java.util.*;
class Instrumento {
    public void ejecutar() {
        System.out.println("Instrumento.ejecutar()");
    }
    public String que() {
        return "Instrumento";
    }
    public void ajuste() {}
}
class Viento extends Instrumento {
    public void ejecutar() {
        System.out.println("Viento.ejecutar()");
    }
    public String que() { return "Viento"; }
    public void ajuste() {}
}

```

```

class Percusion extends Instrumento {
    public void ejecutar() {
        System.out.println("Percusion.ejecutar()");
    }
    public String que() { return "Percusion"; }
    public void ajuste() {}
}
class Stringed extends Instrumento {
    public void ejecutar() {
        System.out.println("Cuerdas.ejecutar()");
    }
    public String que() { return "Cuerdas"; }
    public void ajuste() {}
}
class Metal extends Viento {
    public void ejecutar() {
        System.out.println("Metal.ejecutar()");
    }
    public void ajuste() {
        System.out.println("Metal.ajustar()");
    }
}
class Madera extends Viento {
    public void ejecutar() {
        System.out.println("Viento.ejectuar()");
    }
    public String que() { return "Madera"; }
}
public class Music3 {
    // No importa el tipo, así es que los nuevos
    // tipos agregados al sistema seguirán trabajando:
    static void ajustar(Instrumento i) {
        // ...
        i.ejecutar();
    }
    static void afinarTodo (Instrumento[] e) {
        for(int i = 0; i < e.length; i++)
            afinar(e[i]);
    }
    public static void main(String[] args) {
        Instrumento[] orquesta = new Instrumento[5];
        int i = 0;
        // Upcasting mientras se agrega al arreglo:
        orquesta[i++] = new Viento();
        orquesta[i++] = new Percusion();
        orquesta[i++] = new Cuerdas();
        orquesta[i++] = new Metal();
        orquesta[i++] = new Madera();
        afinarTodo(orquesta);
    }
} ///:~

```

Los nuevos métodos son **que()**, que retorna una referencia a una cadena con una descripción de la clase, y **ajustar()** que proporciona alguna forma de ajustar cada instrumento.

En el **main()**, cuando se coloca algo dentro del arreglo de **Instrumento** se está realizando automáticamente una conversión ascendente a **Instrumento**.

Se puede ver que el método **ajuste()** es felizmente ignorante de todos los cambios de código que se suceden a su alrededor, y de la misma forma trabaja correctamente. Esto es exactamente lo que se supone que el polimorfismo proporciona. Los cambios den código no causan daños a partes del programa que suelen no ser afectadas. Puesto de otra forma, el polimorfismo es una de las técnicas más importantes para permitir al programador “separar las cosas que cambian de las cosas que permanecen iguales”.

## Sobrescribir frente a sobrecargar

Si observamos de forma diferente el primer ejemplo de este capítulo. En el siguiente programa, la interfase del método **ejecutar()** es cambiada en el proceso de sobre escritura, lo que significa que no se está sobrescribiendo el método, en lugar de esto se está sobrecargando. El compilador permite sobrecargar métodos sin quejarse. Pero el comportamiento probablemente no es lo que se desea. He aquí un ejemplo:

```
//: c07:WindError.java
// Accidentalmente cambia la interfase.
class NotaX {
    public static final int
        C_MEDIO = 0, C_SOSTENIDO = 1, C_BEMOL = 2;
}
class InstrumentoX {
    public void ejecutar(int NotaX) {
        System.out.println("InstrumentoX.ejecutar()");
    }
}
class VientoX extends InstrumentoX {
    // OOPS! Se cambia la interfase del método:
    public void ejecutar(NotaX n) {
        System.out.println("VientoX.ejecutar(NotaX n)");
    }
}
public class VientoError {
    public static void afinar(InstrumentoX i) {
        // ...
        i.ejecutar(NotaX.C_MEDIO);
    }
    public static void main(String[] args) {
        VientoX flauta = new VientoX();
        afinar(flauta); // No es el comportamiento correcto!
    }
}
```

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

Aquí hay otro aspecto confuso presentado aquí. En **InstrumentoX**, el método **ejecutar()** toma un **int** que identifica la **NotaX**. Esto es, aun cuando **NotaX** es un nombre de clase, esta puede ser utilizada como identificador sin problema. Pero en **VientoX**, **ejecutar()** se toma una referencia a **NotaX** que es un identificador **n** (A pesar de que se podría decir **ejecutar(NotaX NotaX)** sin un error). De esta manera esto aparece como que el programador intenta sobreescribir **ejecutar()** pero equivocándose un poco al escribirlo. El compilador, sin embargo, asume que se esta sobrecargando y no sobreescibiendo que era lo que se pretendía. Debe notarse que si se sigue la convención de nombres del estándar de Java, el identificador del argumento será **notaX** (con 'n' minúscula), que lo distinguiría del nombre de clase.

En **ajustar**, el **Instrumento i** envía el mensaje **ejecutar()**, que será diferenciado del nombre de clase.

En **ajustar**, el **InstrumentoX i**, sin uno de los miembros **NotaX (C\_MEDIO)** como argumento. Puesto que **NotaX** contiene definiciones del tipo **int**, esto significa que la versión **int** del ahora sobrecargado método **play()** es llamada, y dado que *no* ha sido sobrecargada la versión de la clase base es utilizada.

La salida es:

```
| InstrumentoX.ejecutar()
```

Esto, ciertamente parece ser una llamada a un método polimórfico. Una vez que se entienda que esta sucediendo, se puede solucionar el problema de forma bastante simple, pero hay que imaginarse la dificultad de puede ser encontrar el error si esta sepultado en un programa de considerable tamaño.

## Clases y métodos abstractos

En todos los ejemplos de instrumentos, los métodos en la clase base **Instrumento** eran siempre métodos “ficticios”. Si estos métodos son alguna vez llamados, se estará haciendo algo mal. Esto es porque el intento de **Instrumento** es crear una *interfase común* para todas las clases derivadas de esto.

La única razón para establecer esta interfase común es que pudo haberse expresado diferente para cada subtipo. Este establece una forma básica, así es que se puede decir que es común con todas las clases derivadas.

Otra forma de decir esto es llamar a **Instrumento** una *clase base abstracta* o simplemente una *clase abstracta*. Se puede crear una clase abstracta

cualquier momento se quiera manipular un grupo de clases a través de esta interfase común. Todos los métodos de la clase derivada que se corresponden con la firma de la declaración de la clase base serán llamados mediante el mecanismo de enlazado dinámico (sin embargo, como se ha visto en la última sección, si el nombre del método es el mismo que la clase base pero el argumento es diferente, se tiene que sobrecargar, lo que probablemente no es lo que se quiera).

Si se tiene una clase abstracta como **Instrumento**, los objetos de esa clase nunca tendrán significado. Esto es, **Instrumento** trata de expresar solo la interfase, y no una implementación en particular, así es que crear un objeto **Instrumento** no tiene sentido, y probablemente se quiera prevenir a el usuario de hacer esto. Esto se puede lograr haciendo que todos los métodos en **Instrument** impriman errores, pero esto retrasa la información hasta el momento de ejecutar y requiere una prueba exhaustiva y confiable por parte del usuario. Es siempre mejor capturar los problemas en tiempo de compilación.

Java proporciona un mecanismo para hacer esto llamado el *método abstracto*. Esto es un método que está incompleto; solo tiene una declaración y no un cuerpo de método. Ha aquí la sintaxis para la declaración de métodos abstractos:

```
| abstract void f();
```

Una clase que contiene un método abstracto es llamada una *clase abstracta*. Si una clase contiene uno o más métodos abstractos, la clase debe ser calificada como **abstract** (De otra forma, el compilador entrega un mensaje de error).

¿Si una clase abstracta está incompleta, qué es lo que el compilador supuestamente hace cuando alguien intenta hacer un objeto de esa clase? No se puede crear de forma segura un objeto de una clase abstracta, así es que se obtendrá un mensaje de error del compilador. De esta forma el compilador se asegura de la pureza de la clase abstracta, y no se necesitamos preocuparnos de desprovecharla.

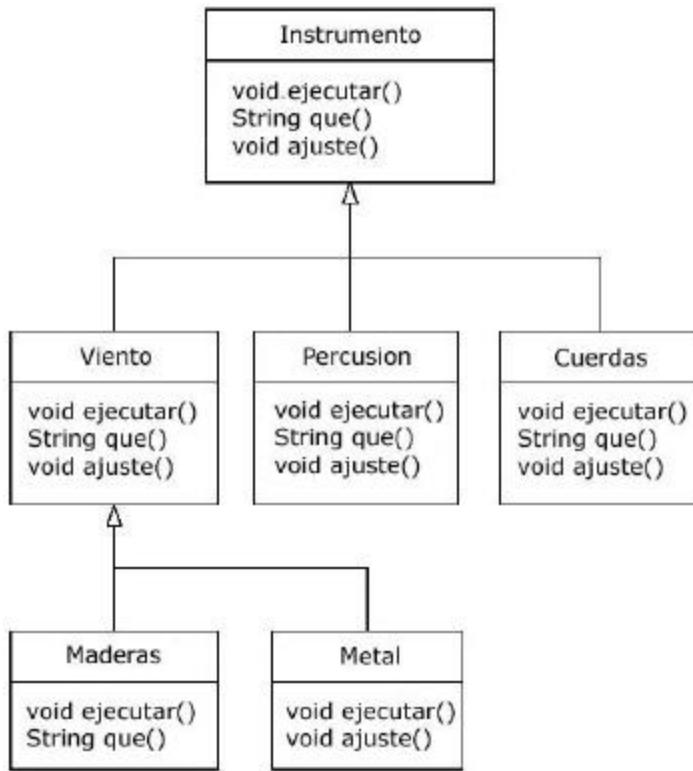
Si se hereda de una clase abstracta y se quiere crear objetos del nuevo tipo, se deben proporcionar definiciones de métodos para todos los métodos abstractos de la clase base. Si no se hace (y se puede elegir no hacerlo), entonces la clase derivada es también abstracta y el compilador lo forzará a calificar esa clase con la palabra clave **abstract**.

Es posible crear una clase como **abstract** sin incluir ningún método abstracto. Esto es útil cuando se tiene una clase que no tiene sentido que tenga algún método abstracto, y se quiera prevenir alguna instancia de la clase.

---

<sup>1</sup> Para los programadores C++, esto es el análogo de las *funciones virtuales puras*.

La clase **Instrument** puede fácilmente ser convertida en una clase abstracta. Solo algunos de los métodos serán abstractos, dado que hacer una clase abstracta no fuerza a hacer todos los métodos abstractos. Así es como se vería:



Vemos aquí el ejemplo orquesta modificado para utilizar clase y métodos abstractos:

```

//: c07:music4:Music4.java
// Clases abstractas y métodos.
import java.util.*;
abstract class Instrumento {
    int i; // espacio asignada para cada uno
    public abstract void ejecutar();
    public String que() {
        return "Instrumento";
    }
    public abstract void ajuste();
}
class Viento extends Instrumento {
    public void ejecutar() {
        System.out.println("Viento.ejecutar ()");
    }
    public String que() { return "Viento"; }
    public void ajuste() {}
}
  
```

```

    }
    class Percusion extends Instrumento {
        public void ejecutar() {
            System.out.println("Percusion.ejecutar()");
        }
        public String que() { return "Percusion"; }
        public void ajuste() {}
    }
    class Cuerdas extends Instrumento {
        public void ejecutar() {
            System.out.println("Cuerdas.ejecutar()");
        }
        public String que() { return "Stringed"; }
        public void ajuste() {}
    }
    class Metal extends Viento {
        public void ejecutar() {
            System.out.println("Metal.ejecutar()");
        }
        public void ajuste() {
            System.out.println("Metal.ajuste()");
        }
    }
    class Madera extends Viento {
        public void ejecutar() {
            System.out.println("Madera.ejecutar()");
        }
        public String que() { return "Madera"; }
    }
    public class Music4 {
        // No importan los tipos, así es que los nuevos
        // tipos agregados al sistema seguirán trabajando bien:
        static void afinar(Instrumento i) {
            // ...
            i.ejecutar();
        }
        static void afinarTodo(Instrumento[] e) {
            for(int i = 0; i < e.length; i++)
                afinar(e[i]);
        }
        public static void main(String[] args) {
            Instrumento[] orquesta = new Instrumento[5];
            int i = 0;
            // Upcasting mientras se agrega al arreglo:
            orquesta[i++] = new Viento();
            orquesta[i++] = new Percusion();
            orquesta[i++] = new Cuerdas();
            orquesta[i++] = new Metal();
            orquesta[i++] = new Madera();
            afinarTodo(orquesta);
        }
    } ///:~
}

```

Se puede ver que no hay realmente cambios a no ser en la clase base.

Es útil crear clases abstractas y métodos porque ellos hacen la abstracción de la clase explícita, y le indican al usuario y al compilador como se pretende que sea utilizada.

# Constructores y polimorfismo

Como es usual, los constructores son diferentes de otro tipo de métodos. Esto es también verdadero cuando se involucra polimorfismo. Aún cuando los constructores no son polimórficos (a pesar que se pueda tener un tipo de “constructor virtual”, como se verá en el capítulo 12), es importante entender la forma en que los constructores trabajan en jerarquías complejas y con polimorfismo. Entender esto ayudará a evitar enredos desagradables.

## Orden de llamada a constructores

El orden de llamada a constructores fue brevemente discutido en el capítulo 4 y nuevamente en el capítulo 6, pero esto fue antes que el polimorfismo fuera introducido.

Un constructor para la clase base es siempre llamado en el constructor para la clase derivada, encadenando hacia arriba la jerarquía de herencia así es que un constructor para cada clase base es llamado. Eso tiene sentido porque el constructor tiene un trabajo especial: ver que el objeto sea creado adecuadamente. Una clase derivada tiene acceso a sus propios miembros solamente, y no a aquellos de la clase base (cuyos miembros son típicamente privados). Solo el constructor de la clase base tiene el conocimiento adecuado y accede a inicializar sus propios elementos. Por consiguiente es esencial que todos los constructores sean llamados, de otra forma el objeto no será construido en su totalidad. Esto es porque el compilador fuerza la llamada del constructor para cada porción de la clase derivada. Será silenciosamente llamado el constructor por defecto si no se llama explícitamente un constructor de la clase base en el cuerpo de la clase derivada. Si no hay constructores por defecto, el compilador se quejará (En el caso donde una clase no tiene constructores, el compilador automáticamente sintetizará un constructor por defecto).

Echemos un vistazo en un ejemplo que nos muestra los efectos de la composición, herencia, y polimorfismo en el orden de construcción:

```
//: c07:Sandwich.java
// Orden de llamadas a constructor.
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 compleja clase por afuera de otras clases y cada clase tiene un constructor que se anuncia a si mismo. La clase importante es **Sandwich**, que rechaza tres niveles de herencia (cuatro, si cuenta la herencia implícita de **Object**) y tres objetos miembro. Cuando un objeto **Sandwich** es creado en el **main()**, la salida es:

```

Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()

```

Esto significa que el orden de las llamadas a el constructor para un objeto complejo es el siguiente:

1. El constructor de la clase base es llamado. Este paso es repetido de forma recursiva de tal forma que la raíz de la jerarquía es construido primero, seguido de la siguiente clase derivada, etc., hasta que la última clase derivada es alcanzada.
2. Los inicializadores de miembros son llamados en el orden de su declaración.

### 3. El cuerpo del constructor de la clase derivada es llamado.

El orden de las llamadas a los constructores es importante. Cuando se hereda, se conoce todo acerca de la clase base y se puede acceder a cualquier miembro sea público o protegido de la clase base. Esto significa que se deba ser capaz de asumir que todos los miembros de la clase base son válidos cuando se esta en la clase derivada. En un método normal, la construcción ya se ha realizado, así es que todos los miembros de todas las partes del objeto han sido construidos. Dentro del constructor, sin embargo, se debe ser capaz de asumir que todos los miembros que se están utilizando han sido armados. La única forma de garantizar esto es que el constructor de la clase base sea llamado primero. Entonces cuando se esta en el constructor de la clase derivada, todos los miembros a los cuales se puede acceder en la clase base son inicializados. “El conocimiento de que todos los miembros son válidos” dentro del constructor es también la razón de que, cuando quiera que sea posible, debe inicializar todos los objetos miembro (esto es, objetos colocados en la clase utilizando composición) en el punto de definición en la clase (por ejemplo, **b**, **c**, y **I** en el ejemplo anterior). Si se sigue esta práctica, ayudará a estar seguro de que todas los miembros clase y objetos del objeto actual han sido inicializados. Desafortunadamente, esto no se puede manejar en todos los casos, como se verá en la siguiente sección.

## Herencia y **finalize()**

Cuando se utiliza composición para crear una nueva clase, uno no se preocupa nunca de finalizar los objetos miembro de esa clase. Cada miembro es un objeto independiente, y de esta manera es recolectada su basura y el finalizado independiente de que sea un miembro de la clase o no. Con la herencia, sin embargo, se debe sobrescribir **finalize()** en la clase derivada si una limpieza especial debe suceder como parte de la recolección de basura. Cuando se sobrescribe **finalize()** en una clase heredada, es importante recordar llamar a la versión de **finalize()** de la clase base, dado que de otra forma la finalización de la clase base no se sucederá. El siguiente ejemplo prueba esto:

```
//: c07:Frog.java
// Prueba de finalización con herencia.
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() throws Throwable {
    System.out.println(
        "LivingCreature finalize");
    // Llama a la versión LAST de la clase base!
    if(DoBaseFinalization.flag)
        super.finalize();
}
}
class Animal extends LivingCreature {
    Characteristic p =
        new Characteristic("has heart");
    Animal() {
        System.out.println("Animal()");
    }
protected void finalize() throws Throwable {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
        super.finalize();
}
}
class Amphibian extends Animal {
    Characteristic p =
        new Characteristic("can live in water");
    Amphibian() {
        System.out.println("Amphibian()");
    }
protected void finalize() throws Throwable {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
        super.finalize();
}
}
public class Frog extends Amphibian {
    Frog() {
        System.out.println("Frog()");
    }
protected void finalize() throws Throwable {
    System.out.println("Frog finalize");
    if(DoBaseFinalization.flag)
        super.finalize();
}
}
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!");
        // Fuerza la llamada de los finalizadores:
        System.gc();
    }
} //:~
}

```

La clase **DoBaseFinalization** simplemente mantiene una bandera que le indica a clase en la jerarquía donde llamar a **super.finalize()**. Esta bandera es fijada basada en un argumento colocado en la línea de comandos, así se puede ver el comportamiento con y sin la finalización de la clase base.

Cada clase en la jerarquía también contiene un objeto miembro de la clase **Characteristic**. Se puede ver que independientemente de donde los finalizadores del la clase base sean llamados, los objetos miembro de **Characteristic** son siempre finalizados.

Cada método **finalize()** sobrescrito debe tener acceso como mínimo a los miembros protegidos dado que el método **finalize()** en la clase **Object** es del tipo **protected** y el compilador no permitirá reducir el acceso durante la herencia (“amigable” es menos accesible que **protected**).

En **Frog.main()**, el la bandera **DoBaseFinalization** es fijada y un único objeto **Frog** es creado. Recuerde que la recolección de basura -y en particular la finalización- puede no sucederse para cualquier objeto en particular, así es que hay que hacer cumplir esto, la llamada a **System.gc()** dispara la recolección de basura y de ese modo la finalización. Sin una 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

```

Se puede ver, ciertamente, que ningún finalizador es llamado para la clase base de **Frog** (el objeto miembro es finalizado, como se esperaba). Pero si agrega el argumento “finalize” en la línea de comandos, se obtiene:

```

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

```

A pesar de que el orden de los objetos miembro y finalizadores es el mismo orden en que han sido creados, técnicamente el orden de finalización de los objetos no esta especificado. Con la clase base, sin embargo, se tiene control sobre el orden de finalización. El mejor orden para utilizar es el mostrado aquí, que es el orden inverso de inicialización. Siguiendo la forma utilizada en C++ por los destructores, se debe realizar la finalización de la clase derivada primer, luego la finalización de la clase base. Esto es porque la finalización de la clase base puede llamar algunos métodos en la clase base que requiere que los componentes de la clase base sigan vivos, así es que no se deben destruir prematuramente.

## Comportamiento de los métodos polimórficos dentro de los constructores

La jerarquía de llamadas a constructores plantea un dilema interesante. ¿Qué sucede si se esta dentro de un constructor y se llama a un método comprometido dinámicamente del objeto que se construye? Dentro de un método común se puede imaginar que sucederá -la llamada comprometida dinámicamente es resuelta en tiempo de ejecución porque el objeto no puede conocer cuando pertenece a la clase donde esta el método o a alguna clase derivada de esta. Para ser consistente se puede pensar que esto sucederá dentro de los constructores.

Ese no es el caso exactamente. Si se llama a un método comprometido dinámicamente dentro de un constructor, la definición sobrescrita para ese método es utilizada. Sin embargo, el *efecto* puede ser mas bien inesperado, y puede esconder alguna dificultad para encontrar errores.

Conceptualmente, el trabajo del constructor es traer a el objeto a la existencia (lo que es apenas una hazaña mediocre). Dentro de cualquier constructor, el objeto entero puede ser parcialmente formado -se puede conocer solo que objetos de la clase base han sido inicializados, pero no se puede saber que clases han sido heredadas. Una llamada a un método comprometido dinámicamente, sin embargo, llega "hasta el exterior" de la

jerarquía de herencia. Este llama a un método en una clase derivada. Si se hace esto dentro de un constructor, la llamada a un método que puede manipular miembros que no han sido inicializados todavía -una receta segura para desastres.

Se puede ver el problema en el siguiente ejemplo:

```
//: c07: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, así es que esta diseñado para ser sobrescrito. Claro que, se esta forzado a sobrescribirlo en **RoundGlyph**. Pero el constructor de **Glyph** llama este método, y la llamada para en **RoundGlyph.draw()**, que parece ser el objetivo. Pero se debe ver 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 es si quiera el valor por defecto 1. Es 0. Esto probablemente puede resultar en un punto o nada de nada dibujado en la pantalla, y se estará abandonado con la mirada fija, tratando de entender por que el programa no funciona.

El orden de inicialización descrito en la sección previa no es bastante completo, y esto es la clave para solucionar el misterio. El proceso actual de inicialización es:

1. El espacio asignad para el objeto es inicializado al binario cero antes de que nada mas suceda.
2. Los constructores de la clase base son llamados como se ha descrito anteriormente. En este punto, el método sobrescrito **draw()** es llamado (si, *antes* que el constructor de **RoundGlyph** sea llamado), quien encuentra un valor de **radius** cero, debido a el paso 1.
3. Los inicializadores de miembros son llamados en el orden de la declaración.
4. El cuerpo del constructor de la clase derivada es llamado.

Hay una razón para esto, que es que todo es al menos inicializado a cero (o lo que sea que cero signifique para un tipo particular de dato) y no solamente queda como basura. Esto incluye referencias a objetos que son incrustadas dentro de una clase mediante composición, el cual comienza en **null**. Así es que si se olvida inicializar esa referencia se obtendrá un error en tiempo de ejecución. Todo lo demás se pone a cero, que es usualmente un valor revelador cuando observamos en la salida.

Por otro lado, se debería estar un poco horrorizado del resultado de este programa. Se ha realizado una cosa lógicamente correcta, y aun el comportamiento es misteriosamente equivocado, sin quejas del compilador (C++ produce un comportamiento mas racional en esta situación). Los errores como este pueden ser fácilmente enterrados y se toma un largo tiempo descubrirlos.

Como resultado, una buena guía para los constructores es, “Hágase lo menos posible para llevar a un objeto en un estado correcto, y si es posible evitarlo, no llame ningún método”. Los únicos método seguros para llamar dentro de un constructor son aquellos que son finales en la clase base (Esto también se aplica a métodos privados, que son automáticamente finales). Estos no pueden ser sobrescritos y no pueden producir este tipo de sorpresa.

## Diseñando con herencia

Una vez que se ha aprendido polimorfismo, se puede ver que todo debe ser heredado porque polimorfismo es como una herramienta ingeniosa. Esto puede agobiar los diseños; de hecho si se elige herencia primero cuando se esta utilizando una clase para hacer una nueva clase, las cosas comienzan a complicarse innecesariamente.

Una mejor estrategia es elegir composición primero, cuando no es tan obvio lo que se utilizará. La composición no fuerza un diseño a una jerarquía de herencia. Pero la composición es también mas flexible dado que es posible elegir dinámicamente un tipo (y su comportamiento) cuando utilizamos

composición, mientras que la herencia requiere un tipo exacto que debe ser conocido en tiempo de compilación. El siguiente ejemplo ilustra esto:

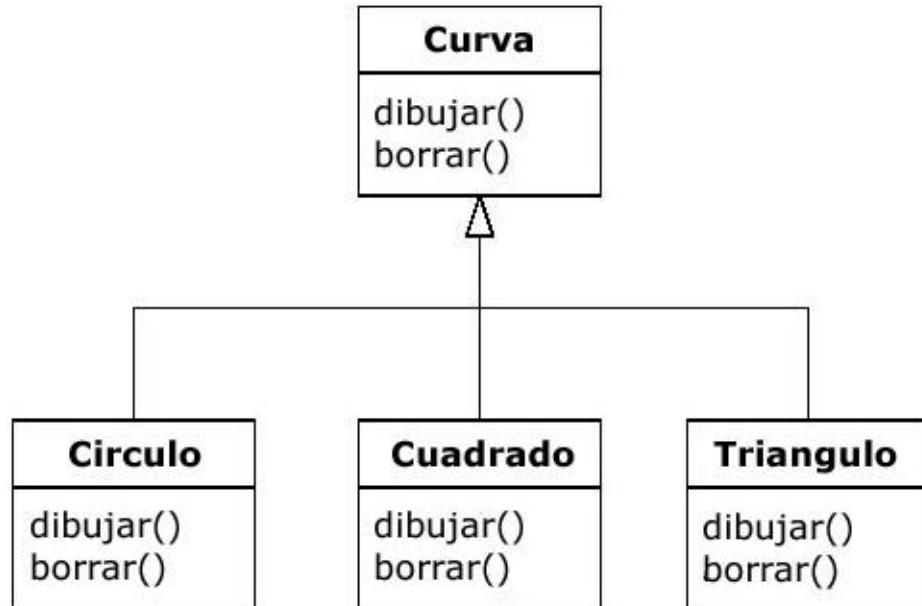
```
//: c07:Transmogrify.java
// Dinámicamente cambia el comportamiento de
// un objeto mediante composición.
abstract class Actor {
    abstract void act();
}
class HappyActor extends Actor {
    public void act() {
        System.out.println("HappyActor");
    }
}
class SadActor extends Actor {
    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 una referencia a un **Actor**, que es inicializado como un objeto **HappyActor**. Esto significa que **go()** produce un comportamiento particular. Pero dado que una referencia puede ser enlazado nuevamente a un objeto diferente en tiempo de ejecución, una referencia para el objeto **SadActor** puede ser sustituida en **a** y entonces el comportamiento producido por **go()** cambia. De esta manera se gana flexibilidad dinámica en tiempo de ejecución (Esto es llamado el *Patrón Estado* Vea *Pensando en Patrones con Java* el cual es posible bajar de [www.BruceEckel.com](http://www.BruceEckel.com)). En contraste, no se puede decidir heredar de forma distinta en tiempo de ejecución; eso debe ser completamente determinado en tiempo de compilación.

Una línea general es “Use herencia para expresar diferencias en el comportamiento, y campos para expresar variaciones de estado”. En el ejemplo anterior, ambos son utilizados: dos diferentes clases son heredadas para expresar la diferencia en el método **act()**, y **Stage** utiliza composición para permitir que el estado sea cambiado. En este caso, ese cambio en el estado se sucede para producir un cambio en el comportamiento.

## Herencia pura contra extensión

Cuando se estudió herencia, se pudo ver que la forma mas ordenada de crear una jerarquía de herencia es tomar una aproximación “pura”. Esto es, solo los método se han establecido en la clase base o interfase serán sobrescritos en la clase derivada, como se ve en el diagrama:



Esto puede ser llamado una relación pura “es un” dado que la interfase de la clase establece que es. La herencia garantiza que cualquier clase derivada tendrá la interfase de la clase base y nada mas. Si se sigue el siguiente diagrama, las clases derivadas también *no tendrán mas que* la interfase de la clase base.

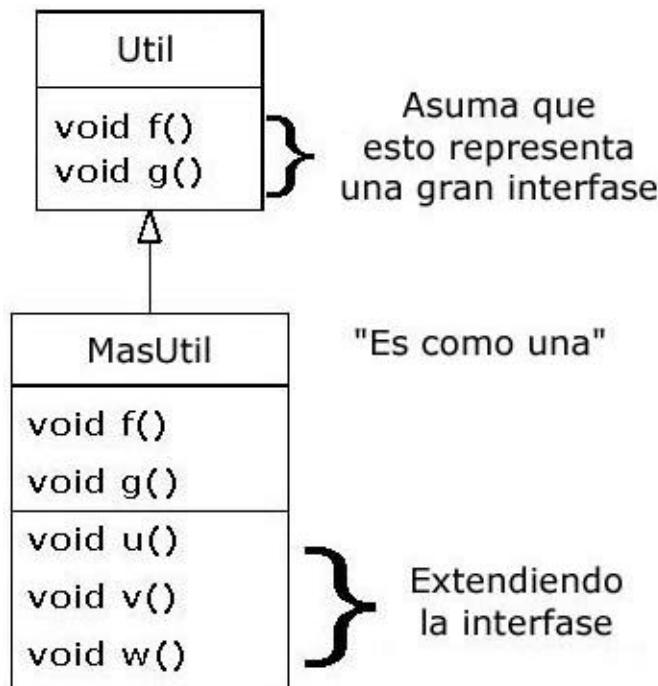
Esto puede pensado como una *substitución pura*, dado que los objetos de la clase derivada pueden ser perfectamente substituidos por la clase base, y nunca se necesitará conocer ningún tipo de información extra acerca de las subclases cuando se están utilizando:



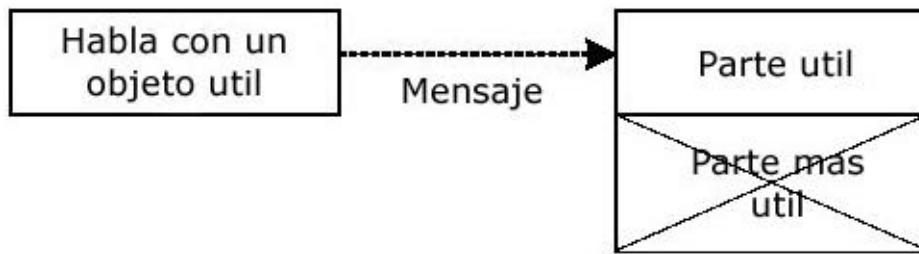
Esto es, la clase base puede recibir cualquier mensajes que se pueda enviar a la clase derivada dado que las dos tienen la misma interfase. Todo lo que

necesita realizar una conversión ascendente de la clase derivada y nunca mirar atrás para fijarse cual es el tipo exacto del objeto que está manejando. Todo es manejado mediante polimorfismo.

Cuando se vea de esta manera, se verá como que una relación pura "es una" es la única forma acertada de hacer las cosas, y cualquier otro diseño confunde y es por definición inservible. Esto es una trampa también. Tan pronto como se comience a pensar de esta forma, se dará un giro y se descubrirá que extender la interfase (lo cual, desafortunadamente, la palabra clave **extends** parece animar) es la solución perfecta para un problema particular. Esto puede ser nombrado como una relación "es como una" dado que la clase derivada es *como* la clase base -tiene la misma interfase fundamental- pero tiene otras herramientas que requieren métodos adicionales para implementar:



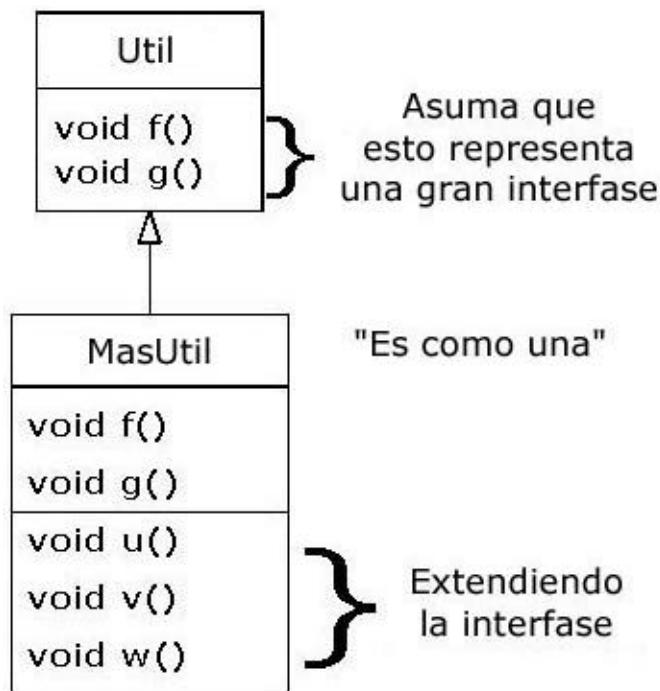
A pesar que esto es también una estrategia razonable (dependiendo de la situación) es inconveniente. La parte extendida de la interfase en la clase derivada no está disponible desde la clase base, así es que una vez que se realiza una conversión ascendente no se puede llamar a los nuevos métodos:



Si no se está realizando una conversión ascendente en este caso, no molesta, pero a menudo se tendrá una situación en donde necesitará redescubrir el tipo exacto del objeto para acceder a los métodos extendidos de este tipo. La siguiente sección muestra como se hace esto.

## Conversión descendente e identificación de tipo en tiempo de ejecución

Dado que se pierde la información del tipo específico con una conversión ascendente (moviéndose hacia arriba en la jerarquía de herencias), tiene sentido recuperar el tipo de información -esto es, mover hacia abajo en la jerarquía de herencia- se utiliza una conversión descendente. Sin embargo, se sabe que una conversión ascendente es siempre seguro; la clase base no puede tener una interfase más grande que la clase derivada, sin embargo cada mensaje que se envía a través de la interfase de la clase base es garantido que será aceptada. Pero con una conversión descendente, no se sabe realmente que una curva (por ejemplo) es actualmente un círculo. En lugar de eso podría ser un triángulo o un cuadrado o algún otro tipo.



Para solucionar este problema debe haber una forma de garantizar que un conversión descendente es correcto, dado que no queremos accidentalmente convertir a el tipo equivocado y enviar un mensaje que el objeto no pueda aceptar. Esto sería bastante peligroso.

En algunos lenguajes (como C++) se debe realizar una operación especial para obtener de una conversión descendente un tipo seguro, pero en Java *cada conversión* es verificada! Así es que aún cuando se parezca que se está realizando una conversión mediante paréntesis común y corriente, en tiempo de ejecución esta conversión es verificada para asegurar que es de hecho el tipo que se piensa que es. Si no lo es, se obtiene una **ClassCastException**. Este acto de verificar tipos en tiempo de ejecución es llamado *identificación de tipo en tiempo de ejecución* (RTTI). El siguiente ejemplo demuestra el comportamiento de RTTI;

```

//: c07:RTTI.java
// Conversión descendente & e identificación de
// tipo en tiempo de ejecución (RTTI).
import java.util.*;
class Util {
    public void f() {}
    public void g() {}
}
class MasUtil extends Util {
    public void f() {}
    public void g() {}
    public void u() {}
}

```

```

        public void v() {}
        public void w() {}
    }
public class RTTI {
    public static void main(String[] args) {
        Util[] x = {
            new Util(),
            new MasUtil()
        };
        x[0].f();
        x[1].g();
        // En tiempo de compilación: método no encontrado en Util:
        //!! x[1].u();
        ((MasUtil)x[1]).u(); // Conversión descendente/RTTI
        ((MasUtil)x[0]).u(); // Lanza una excepción
    }
} //:~

```

Como en el diagrama, **MasUtil** extiende la interfase de **Util**. Pero dado que es heredado, se puede también realizar una conversión ascendente a **Util**. Se puede ver que esto sucede en la inicialización del arreglo **x** en **main()**. Dado que ambos objetos en el arreglo son de la clase **Util**, se puede enviar el método **f()** y **g()** a ambos, y si se intenta llamar a **u()** (el cual solamente existe en **MasUtil**) se obtendrá un error en tiempo de compilación.

Si se desea acceder a la interfase extendida de un objeto **MasUtil**, se podrá realizar una conversión descendente. Si es el tipo correcto, será exitoso. De otra forma se obtendrá una **ClassCastException**. No necesita escribir código especial para esta excepción, dado que indica un error de programación que puede suceder en cualquier parte de un programa.

Hay mas de RTTI que la simple conversión. Por ejemplo, hay una forma de ver con que tipo se está tratando *antes* de que se intente realizar una conversión descendente. Todo el capítulo 12 es devoto del estudio de los diferentes aspectos de la identificación de tipo en tiempo de ejecución.

## Resumen

Polimorfismo significa “formas diferentes”. En la programación orientada a objetos, se tiene la misma cara (la interfase común en la clase base) y diferentes formas de utilizar esa cara: las diferentes versiones de los métodos comprometidos dinámicamente.

Se ha visto en este capítulo que es imposible entender, o aún crear un ejemplo de polimorfismo sin utilizar abstracción de datos y herencia.

Polimorfismo es una característica que no puede ser vista aisladamente (como se puede hacer con una instrucción **switch**, por ejemplo), en lugar de eso se trabaja solamente en armonía como parte de una “gran pintura” de relaciones de clases. Las personas a menudo se confunden con otras,

características no orientadas a objetos de Java, como sobrecarga de métodos, que es a veces presentada como orientada a objetos. No hay que dejarse embaucar: si no es enlazado a última hora, no es polimorfismo.

Para utilizar polimorfismo -y de esta manera técnicas de programación orientada a objetos- efectivamente en los programas se debe expandir su visión de programación para incluir no solamente miembros y mensajes de una clase individual, también se debe incluir el populacho de clases que están en medio y sus relaciones con cada una de las otras. A pesar que esto requiere un esfuerzo significante, es una lucha digna, porque los resultados son un desarrollo de programas mas rápido y un código mas fácil de mantener.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Agregue un nuevo método a la clase base de **Shapes.java** que imprime un mensaje, pero no lo sobrescriba en la clase derivada. Explique que sucede. Ahora sobrescríbala en una de las clases derivadas pero no en las otras, y vea que sucede. Finalmente, sobrescríbala en todas las clases derivadas.
2. Agregue un nuevo tipo de **Shape** a **Shapes.java** y verifique en el **main()** que el polimorfismo trabaja para su nuevo tipo como lo hace para los viejos tipos.
3. Cambie **Music3.java** de tal forma que **que()** se convierta en el método **toString()** de **Object**. Trate de imprimir el objeto **Instrumento** utilizando **System.out.println()** (sin ninguna conversión).
4. Agregue un nuevo tipo de **Instrumento** a **Music3.java** y verifique que el polimorfismo trabaja para su nuevo tipo.
5. Modifique **Music3.java** de tal forma que cree de forma aleatoria objetos **Instrumento** de la forma en que **Shapes.java** lo hace.
6. Cree una jerarquía de herencia de **Roedores: Raton, Gerbo, Hamster**, etc. En la clase base, proporcione métodos que son comunes a todos los **Roedores** y proporcione estos en las clases derivadas para que realicen distintos comportamientos dependiendo del tipo específico de **Roedor**. Cree un arreglo de **Roedores**, llénelo con diferentes tipos específicos de **Roedores** y llame a los métodos de la clase base para ver que sucede.

7. Modifique el ejercicio 6 para que **Roedor** sea una clase abstracta. Hágase los método de **Roedor** abstracto cuando sea posible.
8. Cree una clase como abstracta sin incluir ningún método abstracto, y verifique que no se pueda crear una instancia de esa clase.
9. Agregue una clase **Picke** a **Sandwich.java**.
10. Modifique el ejercicio 6 para demostrar el orden de inicialización de la clase base y las clases derivadas. Ahora agréguese objetos miembros a ambas, la clase base y las clases derivadas, y muéstrese el orden en que sus inicializaciones ocurren durante la construcción.
11. Cree una jerarquía de herencia de tres niveles. Cada clase en la jerarquía debe tener un método **finalize()**, y debe llamar de forma adecuada la versión de **finalize()** de la clase base. Demuestre que su jerarquía trabaja propiamente.
12. Cree una clase con dos métodos. En el primer método, llámeselos el segundo método. Herede una clase y sobrescriba el segundo método. Cree un objeto en la clase derivada, realice una conversión ascendente a el tipo base, y llame el primer método. Explique que sucede.
13. Cree una clase base con un método **abstract print()** que sobrescribe en la clase derivada. La versión sobrescrita del método imprime el valor de una variable **int** definida en la clase derivada. En el punto de la definición de esta variable, asígnale un valor que no sea cero. En el constructor de la clase base, llame este método. En el **main()** cree un objeto del tipo derivado, y luego llame a el método **print()**. Explique los resultados.
14. Siguiendo el ejemplo en **Transmogrify.java**, cree una clase **Starship** que contenga una referencia **AlertStatus** que pueda indicar tres diferentes estados. Incluya los métodos para cambiar esos estados.
15. Cree una clase abstracta sin método. Derive la clase y agregue un método. Cree un método estático que tome una referencia a la clase base, realice una conversión descendente a la clase derivada, y llame el método. En el **main()**, demuestre que esto funciona en la clase base, de esta manera elimine la necesidad del realizar una conversión descendente.

# 8: Interfases y clases internas

Las interfases y las clases internas proporcionan los métodos mas sofisticados de organización y control de los objetos en su sistema.

C++, por ejemplo, no contiene este tipo de mecanismos, a pesar que el programador listo puede simularlas. El echo de que existan en Java indica que lo han considerado suficientemente importante para proporcionar soporte directo a través de palabras claves del lenguaje.

En el capítulo 7, se aprendió acerca de la palabra clave **abstracta**, que permite crear uno o mas métodos en una clase que no tenga definiciones -se proporciona una parte de la interfase sin proporcionar la implementación correspondiente, que es creada por herederos. La palabra clave **interface** produce una clase completamente abstracta, una que no proporciona ningún tipo de implementación. Se aprenderá que la interfase es mas que simplemente una clase abstracta tomada al extremo, dado que esta permite realizar una variación sobre “herencias múltiples” de C++, creando una clase en la que se pueda realizar una conversión ascendente a mas de un tipo base.

En un principio, las clases internas de ven mas como una mecanismo simple de ocultar código: se colocan clases dentro de otras clases. Se aprenderá, sin embargo, que las clases internas hacen mas que eso -conocen y pueden comunicarse con las clases circundantes- y entonces el tipo de código que se puede escribir con clase interiores es mas elegante y claro, a pesar de que es un nuevo concepto para la mayoría. Toma un tiempo sentirse confortable con el diseño utilizando clases internas.

## Interfases

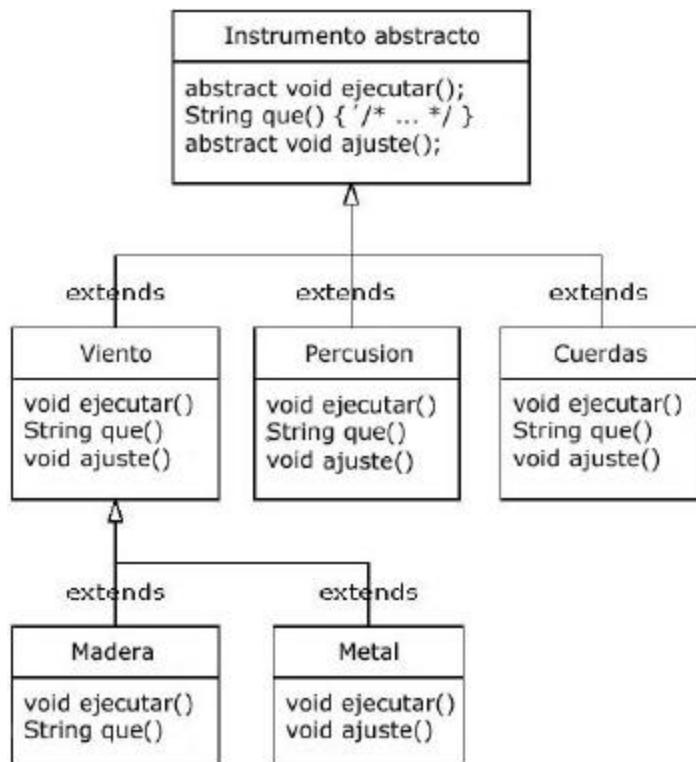
La palabra clave **interface** toma el concepto de abstracción un paso mas adelante. Puede pensar que es como una clase abstracta “pura”. Esto permite al creador establecer la forma de una clase: nombres de métodos, listas de argumentos, y tipo de retornos, pero no el cuerpo de los métodos. Una interfase puede almacenar también campos, pero estos son

implícitamente estáticos y finales. Una interfase proporciona solo una forma, pero no la implementación.

Una interfase dice: “Esto es como todas las clases que *implementen* esta interfase particular se verán”. De esta forma, cualquier código que utilice una interfase particular sabrá que métodos deben ser llamados para esa interfase, y esto es todo. Así es que la interfase es utilizada para establecer un “protocolo” entre clases (Algunos lenguajes de programación orientada a objetos tienen una palabra clave llamada *protocol* para hacer la misma cosa).

Para crear una interfase, se utiliza la palabra clave **interface** en lugar de la palabra clave **class**. Como una clase, se puede agregar la palabra clave **public** antes de la palabra clave **interface** (pero solo si la interfase es definida en un fichero con el mismo nombre) o no colocarla para darle la el estado de “amigable” así solo puede ser utilizable dentro del mismo paquete.

Para hacer que una clase se ajuste a una interfase en particular (o a un grupo de interfaces) se debe utilizar la palabra clave **implements**. Se estará diciendo “La interfase es como se ve pero ahora especificaremos como ella *trabaja*”. Aparte de eso, se ve como una herencia. El diagrama para el ejemplo de los instrumentos se muestra aquí:



Una vez que se ha implementado una interfase, esa implementación se convierte en una clase común que puede ser extendida de una forma regular.

Se puede elegir explícitamente hacer las declaraciones de métodos en una interfase como públicos. Pero estos, son públicos aún si no se indica. Así es que cuando se implemente una interfase, los métodos de la interfase deben ser definidos como públicos. De otra forma serán por defecto “amigables”, y se reducirá la accesibilidad de un método durante la herencia, lo que no es permitido por el compilador Java.

Esto se puede ver en esta versión modificada del ejemplo de **Instrumento**. Se puede ver que cada método en la interfase es estrictamente una declaración, que es la única cosa que el compilador permite. Además, ninguno de los métodos en **Instrumento** es declarada como público, pero son automáticamente públicos de todas formas:

```
//: c08:music5:Music5.java
// Interfaces.
import java.util.*;
interface Instrumento {
    // Constante en tiempo de compilación:
    int i = 5; // static & final
    // No se puede tener definiciones de métodos:
    void ejecutar(); // Automáticamente público
    String que();
    void ajustar();
}
class Viento implements Instrumento {
    public void ejecutar() {
        System.out.println("Viento.ejecutar()");
    }
    public String que() { return "Viento"; }
    public void ajustar() {}
}
class Percusion implements Instrumento {
    public void ejecutar() {
        System.out.println("Percusion.ejecutar()");
    }
    public String que() { return "Percusión"; }
    public void ajustar() {}
}
class Cuerdas implements Instrumento {
    public void ejecutar() {
        System.out.println("Cuerdas.ejecutar()");
    }
    public String que() { return "Cuerdas"; }
    public void ajustar() {}
}
class Metal extends Viento {
    public void ejecutar() {
        System.out.println("Metal.ejecutar()");
    }
}
```

```

        public void ajustar() {
            System.out.println("Metal.ajustar() ");
        }
    }
    class Madera extends Viento {
        public void ejecutar() {
            System.out.println("Madera.ejecutar() ");
        }
        public String que() { return "Madera"; }
    }
    public class Music5 {
        // No importa el tipo, así es que los nuevos
        // tipos agregados al sistema funcionaran igual:
        static void afinar(Instrumento i) {
            // ...
            i.ejecutar();
        }
        static void afinarTodo(Instrumento[] e) {
            for(int i = 0; i < e.length; i++)
                afinar(e[i]);
        }
        public static void main(String[] args) {
            Instrumento[] orquesta = new Instrumento[5];
            int i = 0;
            // Upcasting durante la asignación a el arreglo:
            orquesta[i++] = new Viento();
            orquesta[i++] = new Percusion();
            orquesta[i++] = new Cuerdas();
            orquesta[i++] = new Metal();
            orquesta[i++] = new Madera();
            afinarTodo(orquesta);
        }
    } ///:~
}

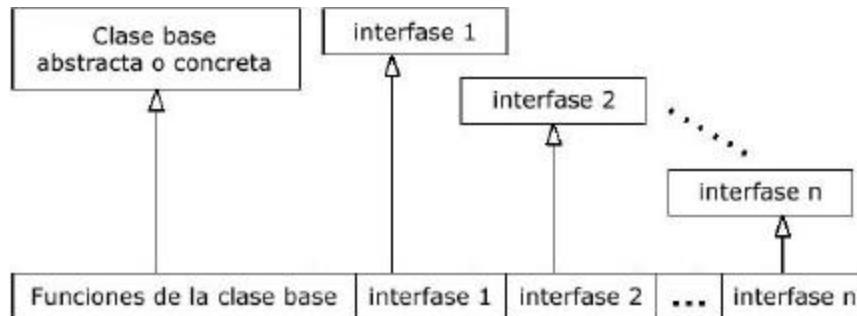
```

El resto del código trabaja de la misma forma, no importa si se esta realizando un upcast a una clase “común” llamada **Instrumento**, a una clase abstracta llamada **Instrument** o a una interfase llamada **Instrument**. El comportamiento es el mismo. De hecho, se puede ver en el método **afinar()** que no hay ninguna evidencia acerca de si **Instrumento** es una clase “común”, una clase abstracta o una interfase. Este el objetivo: cada estrategia le da a el programador un control diferente sobre la forma en que los objetos son creados y utilizados.

## “Herencias múltiples” en Java

La interfase no es simplemente una forma “mas pura” de una clase abstracta. Tiene un propósito mas elevado que ese. Puesto que una **interfase** no tiene ningún tipo de implementación -esto es, no tiene espacio asociado con la interfase- no hay nada que impida que muchas interfaces sean combinadas. Esto es de mucha importancia dado que hay veces en donde se necesita decir “Una **x** es una **a** y una **b** y una **c**”. En C++, este acto de

combinar múltiples interfaces de clases es llamado *herencia múltiple* y mas bien acarrea algún tipo de equipaje incómodo dado que cada clase puede tener una implementación. En Java, se puede realizar el mismo acto, pero solo una de las clases puede tener implementación, así los problemas que se ven en C++ no se suceden con Java cuando se combinan múltiples interfaces:



En una clase derivada, no se está forzado a tener una clase base que sea abstracta o “concreta” (una sin métodos abstractos). Si se realiza una herencia de una clase que no es una interfaz, solo se puede heredar una. Todo el resto de los elementos base deben ser interfaces. Se coloca todos los nombres de interfase luego de la palabra clave **implements** y separados con comas. Se puede tener tantas interfaces como se desee -cada una se convierte en un tipo independiente al que se pueda realizar un upcast. El siguiente ejemplo muestra una clase concreta combinada con muchas 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
}
} //:~

```

Se puede ver que **Hero** combina la clase concreta **ActionCharacter** con las interfaces **CanFight**, **CanSwim**, y **CanFly**. Donde se combina una clase concreta con interfaces de esta forma, la clase concreta debe ir primera, luego las interfaces (El compilador da un error de otra forma).

Se puede ver que la firma para **fight()** es la misma en la interfase **CanFight** y la clase **ActionCharacter**, y que **fight()** no es proporcionada con una definición en **Hero**. Esta regla para una interfase es que se debe heredar de ella (como se verá en breve), pero entonces no tiene otra interfase. Si se quiere crear un objeto con el nuevo tipo, debe ser una clase con todas las definiciones proporcionadas. Aún cuando **Hero** no proporcione explícitamente una definición para **fight()**, la definición viene con **ActionCharacter** así es que es automáticamente proporcionada y es posible crear un objeto de **Hero**.

En la clase **Adventure**, se puede ver que hay cuatro métodos que toma como argumentos las diversas interfaces y la clase concreta. Cuando un objeto **Hero** es creado, este puede ser pasado por cualquiera de estos métodos, lo que significa que se realiza un upcast para cada interfase a su vez. Dado que la forma en que las interfaces son planeadas en Java, esto trabaja sin dificultades y sin un esfuerzo particular por parte del programados.

Se debe tener presente que la razón principal de las interfaces es mostrada en el ejemplo mas arriba: ser capaz de realizar un upcast a mas de un tipo base. Sin embargo, una segunda razón para utilizar interfaces es la misma que utilizar una clase abstracta: prevenir que el cliente programador de crear un objeto de esta clase y establecer que solo es una interfase. Esto trae una pregunta: ¿Se debe utilizar una interfase o una clase abstracta? Una interfase nos da los beneficios de una clase abstracta y los beneficios de una interfase, así es que es posible crear una clase base sin ninguna definición de método o variables miembro así es que es preferible siempre utilizar una interfase que una clase abstracta. De hecho, si se sabe que algo va a ser una clase base, la primer elección es hacer una interfase y solo si se esta forzado a tener definiciones de métodos o variables miembro se debe cambiar a una clase abstracta, o si es necesario a una clase concreta.

## Colisión de nombres cuando se combinan interfaces

Se puede encontrar una pequeña dificultad cuando se implementan múltiples interfaces. En el ejemplo anterior, **CanFight** y **ActionCharacter** tienen un método exactamente igual llamado **void fight()**. Esto no es un problema porque el método es idéntico en ambos casos. ¿Pero qué sucede si no lo es? He aquí 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 C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}
class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 1; }
}
// Methods differ only by return type:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} //://:~
```

La dificultad se sucede a causa de que la sobreescritura, implementación y sobrecarga están desagradablemente mezcladas juntas, y las funciones sobrecargadas no pueden diferenciarse solo por el tipo de retorno. Cuando se les quita el comentario a las dos últimas líneas, el mensaje de error lo dice todo:

```
InterfaceCollision.java:23: f() en C no puede
implementar f() en I1; intento de utilizar
tipos de retorno incompatibles
encontrado : int
requerido: void
InterfaceCollision.java:24: interfaces I3 y I1 son
incompatibles; ambas definidas en f
(), pero con diferentes tipos de retorno
```

Utilizando los mismos nombres de métodos en interfaces diferentes las cuales se intenta combinar generalmente causan confusión en la legibilidad del código. Hay que esforzarse por evitarlos.

# Extendiendo una interfase con herencia

Se puede fácilmente agregar declaraciones de métodos a una interfase utilizando herencia, y se puede también combinar muchas interfaces en una sola con herencia. En ambos casos se obtiene una nueva interfase como se puede ver en este ejemplo:

```
//: c08: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);
    }
} ///:~
```

**DangerousMonster** es una simple extensión de **Monster** que produce una nueva interfase. Esta es implementada en **DragonZilla**.

La sintaxis utilizada en **Vampire** trabaja *solo* cuando las interfaces son heredadas. Normalmente, se puede utilizar **extends** con una sola clase, pero dado que una interfase puede ser creada con muchas otras interfaces, **extends** se puede referir a muchas interfaces base cuando se está creando una nueva **interfase**. Como se puede ver, los nombres de las interfaces son simplemente separados con comas.

## Agrupación de constantes

Dado que los campos que se colocan dentro de una interfase son automáticamente estáticos y finales, la interfase es una herramienta conveniente para crear grupos de valores constantes, tanto como se haría con un tipo **enum** en C o en C++. Por ejemplo:

```
//: c08:Months.java
// Using interfaces to create groups of constants.
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;
} //:~
```

Se debe advertir el estilo de Java de utilizar todos las letras en mayúsculas (con infracciones para separar múltiples palabras en un mismo identificador) para estáticas finales que tienen inicializadores constantes.

El campo en una interfase es público automáticamente, así es que no es necesario especificarlo.

Ahora se puede utilizar las constantes fuera del paquete importando **c08.\*** o **c08.Months** simplemente de la misma forma en que se haría con otro paquete, y referenciar los valores con expresiones como **Months.JANUARY**. Claro, lo se obtendrá simplemente será un entero, así es que no hay la seguridad de tipo que las enumeraciones de C++ tiene, pero esta (comúnmente utilizada) técnica es ciertamente una mejora sobre los números muy codificados en los programas (cuya estrategia es referida a menudo como utilizar “números mágicos” y produce código muy difícil de mantener).

Si se quiere seguridad de tipo extra, se puede crear una clase como esta<sup>1</sup>:

```
//: c08:Month2.java
// A more robust enumeration system.
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"),
```

---

<sup>1</sup> Esta estrategia fue inspirada en un correo electrónico de Rich Hoffarth

```

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 es llamada **Month2**, dado que ya hay una clase **Month** en la librería estándar de Java. Es una clase final con un constructor privada así es que no se puede heredar o crear alguna instancia de ella. Las únicas instancias son la estática final creada en la clase misma: **JAN**, **FEB**, **MAR**, etc. Estos objetos son también utilizados en el arreglo **month**, que deja elegir meses por números en lugar de nombrándolos (Se debe ver que el **JAN** extra en el arreglo para proporcionar un desplazamiento en uno, así el mes de diciembre es el número 12). En el **main()** se puede ver la seguridad del tipo: **m** es un objeto **Month2** así es que este puede ser asignado solo a **Month2**. En el ejemplo anterior **Months2.java** proporciona solo números enteros, así es que a una variable entera que pretende representar a un mes se le puede actualmente dar cualquier valor entero, lo cual no es muy seguro.

Esta estrategia también permite utilizar **==** o **equals()** de forma intercambiable, como es mostrado en el final del **main()**.

## Inicializando cambios en interfaces

Los campos definidos en interfaces son automáticamente estáticos y finales. Estos no pueden ser “finales en blanco”, pero pueden ser inicializados con expresiones no constantes. Por ejemplo:

```

//: c08: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;
}

```

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

Dado que los campos son estáticos, son inicializados cuando la clase es cargada por primera vez, lo que sucede cuando alguno de los campos es accedido la primera vez. Ha aquí 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);
    }
} //:/~
```

El campo, clase, no es parte de la interfase pero en lugar de eso es almacenado en un área estática para la interfase.

## Interfases anidadas

<sup>2</sup>Las interfaces pueden ser anidadas con clases y con otras interfaces. Esto deja al descubierto una gran cantidad de características muy interesantes:

```
class A {
    interface B {
        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
    public interface C {
        void f();
    }
    class CImp implements C {
        public void f() {}
    }
    private class CImp2 implements C {
        public void f() {}
    }
    private interface D {
        void f();
    }
    private class DImp implements D {
        public void f() {}
    }
    public class DImp2 implements D {
        public void f() {}
    }
    public D getD() { return new DImp2(); }
}
```

---

<sup>2</sup> Gracias a Martin Danner por hacer esta pregunta durante un seminario

```

private D dRef;
public void receiveD(D d) {
    dRef = d;
    dRef.f();
}
}
interface E {
    interface G {
        void f();
    }
    // Redundant "public":
    public interface H {
        void f();
    }
    void g();
    // Cannot be private within an interface:
    !!! private interface I {}
}
public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }
    class CImp implements A.C {
        public void f() {}
    }
    // Cannot implement a private interface except
    // within that interface's defining class:
    !!! class DImp implements A.D {
        !!! public void f() {}
        !!!
    }
    class EIImp implements E {
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
    class EIImp2 implements E {
        public void g() {}
        class EG implements E.G {
            public void f() {}
        }
    }
    public static void main(String[] args) {
        A a = new A();
        // Can't access A.D:
        !!! A.D ad = a.getD();
        // Doesn't return anything but A.D:
        !!! A.DImp2 di2 = a.getD();
        // Cannot access a member of the interface:
        !!! a.getD().f();
        // Only another A can do anything with getD():
        A a2 = new A();
        a2.receiveD(a.getD());
    }
} //!

```

La sintaxis para anidar una interfase sin una clase es razonablemente obvia y de la misma forma que con las interfaces no anidadas estas pueden tener visibilidad pública o “amigable”. Se puede también ver que las interfaces anidadas públicas y “amigables” pueden ser implementadas como clases anidadas públicas, “amigables” y privadas.

Como un nuevo giro imprevisto, las interfaces pueden también ser privadas como se ha visto en **A.D** (el mismo requisito de sintaxis es utilizado por las interfaces anidadas por lo que se refiere para clases anidadas).

¿Que tan bueno es una interfase anidada privada? Se puede adivinar que solo puede ser implementada como clase anidada privada como en **DImp**, pero **A.DImp2** muestra que también puede ser implementada como una clase pública. Sin embargo, **A.DImp2** puede solo ser utilizada como esta. No se está permitido mencionar el hecho de que se implementa la interfase privada, así es que implementar una interfase privada es la forma de forzar la definición de los métodos en esa interfase sin agregar ningún tipo de información (esto es, sin permitir el realizar una conversión ascendente).

El método **getD()** produce un futuro dilema concerniente a la interfase privada: es un método público que retorna una referencia a una interfase privada. ¿Que se puede hacer con el valor de retorno de este método? En el **main()**, se puede ver muchos intentos de utilizar este valor de retorno, todos los cuales fallan. La única cosas que funciona es si el valor de retorno es manejado por un objeto que tenga permiso de utilizarlo -en este caso, otro **A**, mediante el método **received()**.

La interfase **E** muestra que las interfaces pueden ser anidadas con cada una de las otras. Sin embargo, las reglas acerca de interfaces -en particular, que todos los elementos de las interfaces deben ser públicos- son estrictamente implementadas aquí, así es que una interfase anidada con otra interfase es automáticamente pública y no puede ser hecha privada.

**NestingInterfaces** muestra las distintas formas en que se pueden implementar las interfaces anidadas. En particular, se debe notar que cuando se implementa una interfase, no se requiere implementar ninguna interfase anidada con ella. También, las interfaces privadas no pueden ser implementadas fuera de su definición de clases.

Inicialmente, estas características pueden verse como que son agregadas estrictamente por un tema de consistencia sintáctica, pero generalmente encuentro que una vez que se conoce acerca de la característica, se descubre frecuentemente lugares donde es útil.

# Clases internas

Es posible colocar la definición de una clase dentro de otra definición de clase. esto es llamado una *clase interna*. La clase interna es una característica valiosa porque permite agrupar clases que lógicamente permanecen juntas y para controlar la visibilidad de una dentro de otra. Sin embargo, es importante entender que las clases internas son claramente diferentes de la composición.

A menudo, cuando ese esta aprendiendo acerca de ellas, la necesidad de clases interna es inmediatamente obvia. En el final de esta sección, después de toda la sintaxis y la semántica de las clases internas sea descrita, se podrán encontrar ejemplos que hacen claro los beneficios de las clases internas.

Se crean las clases internas exactamente como se puede imaginar - colocando la definición de clase dentro de una clase que la rodee:

```
//: c08:Parcel1.java
// Creating inner classes.
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);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
} ///:~
```

Las clases internas, cuando son utilizadas dentro de **ship()**, se ven igual que cuando utilizamos cualquiera de las otras clases. Aquí, la única diferencia práctica es que los nombres son anidados dentro de **Parcel1**. Se podrá ver en un momento que esto no es únicamente la diferencia.

Más típicamente, una clase externa puede tener un método que retorne una referencia a una clase interna, como esta:

```
//: c08:Parcel2.java
// Returning a reference to an inner class.
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);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("Tanzania");
        Parcel2 q = new Parcel2();
        // Defining references to inner classes:
        Parcel2.Contents c = q.cont();
        Parcel2.Destination d = q.to("Borneo");
    }
} ///:~
```

Si se quiere hacer un objeto de la clase interna en cualquier lado excepto desde adentro de un método no estático de la clase externa, se debe especificar que el tipo de ese objeto es

*NombreDeLaClaseExterna.NombreDeLaClaseInterná* como se ve en el **main()**.

## Clases internas y conversión ascendente

Hasta el momento, las clases internas no parecen tan dramáticas. Después de todo, si esta escondido, se está detrás, Java ya tiene un mecanismo para ocultar perfectamente bueno -simplemente permite a la clase ser “amigable” (visible solo dentro del paquete) en lugar de crearla como una clase interna.

Sin embargo, las clases internas realmente se heredan en si mismas cuando se comienza a realizar un upcast a una clase base, y en particular a una interfase (El efecto de producir una referencia a una interfase desde un objeto que la implemente es esencialmente lo mismo que realizar un upcast a la clase base). Esto es porque la clase interna -la implementación de la interfase- puede ser completamente desconocida y no disponible para cualquiera, lo que es conveniente para ocultar la implementación. Todo lo que se obtiene es la referencia a la clase base de la interfase.

Primer, las interfaces comunes serán definidas en sus propias ficheros así es que ellas pueden ser utilizadas 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 para el cliente programador (Se debe recordar, que la interfase automáticamente hace todos sus miembros públicos).

Cuando se obtiene una referencia a la clase base o la interfase, es posible que no se pueda incluso encontrar el tipo exacto, como se muestra aquí:

```
//: c08:Parcel3.java
// Returning a reference to an inner class.
public class Parcel3 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination
        implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination 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 pc = p.new PContents();
    }
} //:~

```

Se debe ver que dado que **main()** esta en **Test**, cuando se quiera ejecutar este programa no se ejecutará **Parcel3**, en lugar de eso:

```
| java Test
```

En el ejemplo, **main()** debe estar en una clase separada para demostrar lo privado de la clase interna **PContents**.

En **Parcel3**, algo nuevo ha sido agregado: la clase interior **PContents** es privada así es que ninguna excepto **Parcel3** puede ser accedida.

**PDestination** es protegida, así es que ninguna excepto **Parcel3**, las clases en el paquete **Parcel3** (dado que son protegidos también conceden acceso al paquete -esto es, protegido es también “amigable”), y los herederos de **Parcel3** pueden acceder a **PDestination**. Esto significa que el cliente programador tiene conocimiento y acceso restringido a estos miembros. De hecho, incluso no se puede realizar un conversión descendente a una clase privada interna (o a una clase interna protegida si es una heredera), porque no se puede acceder al nombre, como se puede ver en **class Test**. De esta forma, la clase privada interna le proporciona a el diseñador de la clase una forma para prevenir completamente dependencias de escritura de código y una forma de ocultar completamente los detalles de la implementación. Además, extender una interfase es inútil desde la perspectiva del cliente programador dado que este no puede acceder a ningún método adicional que no sean parte de la clase de la interfase pública. Esto también proporciona una oportunidad para el compilador Java de generar código mas eficiente.

Clases normales (no internas) no pueden ser hechas privadas o públicas - solo públicas o “amigables”.

## Clases internas en métodos y alcances

Lo que se ha visto hasta ahora engloba el uso típico de las clases internas. En general, el código que se escribirá y leerá acerca de clases internas será de clases internas “planas” que son simples y fáciles de entender. Sin embargo, el diseño para las clases internas es bastante completo y hay una gran cantidad de otras, mas confusas formas en las que se pueden utilizar si se desea: las clases internas pueden ser creadas sin un método o incluso con un alcance arbitrario. Aquí hay dos razones para hacer esto:

1. Como se ha mostrado previamente, se esta implementando una interfase de algún tipo así es que se esta creando y retornando una referencia.

2. Se está resolviendo un problema complicado y se quiere crear una clase que asista en su solución, pero no quiere que este públicamente disponible.
- En los siguientes ejemplos, el código anterior será modificado para utilizar:
1. Una definición de clase sin un método.
  2. Una clase definida dentro del alcance en un método.
  3. Una clase anónima extendiendo una clase que tiene un constructor que no es el constructor por defecto.
  4. Una clase anónima que realiza inicialización de campos.
  5. Una clase anónima que realiza una construcción utilizando inicialización de instancias (las clases anónimas internas no pueden tener constructores)

A pesar de que es una clase común con una implementación, **Wrapping** es también utilizado como una “interfase” común para su clase derivada:

```
//: c08:Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
} //:~
```

Se puede advertir acerca de que **Wrapping** tiene un constructor que requiere un argumento, para hacer las cosas un poco más interesantes.

El primer ejemplo muestra la creación de una clase entera sin el alcance de un método (en lugar del alcance de otra clase):

```
// Nesting a class within a method.
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()** en lugar que sea parte de **Parcel4** (También se advierte que se puede utilizar el identificador de clase **PDestination** para una clase interna adentro de cada clase en el mismo subdirectorio sin una colisión de nombres). Por consiguiente, **PDestination**

no puede ser accedido desde afuera de **dest()**. De advierte que la conversión ascendente que ocurre en la instrucción de retorno -nada se perfila de **dest()** excepto una referencia a **Destination**, la clase base. Claro, el hecho que el nombre de la clase **PDestination** sea colocado dentro de **dest()** no significa que **PDestination** no sea un objeto válido una vez que retorne **dest()**.

El siguiente ejemplo muestra como se puede anidar una clase interna con cualquier alcance arbitrario:

```
//: c08:Parcel5.java
// Nesting a class within a scope.
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();
        }
        // No se puede utilizar aquí! Fuera del alcance:
        // 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** está anidada dentro del alcance de una instrucción **if**. Esto no significa que la clase es condicionalmente creada -es compilada junto con todo lo demás. Sin embargo, no esta disponible fuera del alcance en el cual esta definida. Aparte de esto, se ve exactamente como una clase común.

## Clases internas anónimas

El siguiente ejemplo se ve un poco extraño:

```
//: c08:Parcel6.java
// Un método que retorna una clase interna anónima.
public class Parcel6 {
    public Contents cont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        }; // Punto y coma requerido en este caso
    }
}
```

```

    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
} //:~

```

¡El método **cont()** combina la creación del valor de retorno con la definición de la clase que representa ese valor de retorno! Además, la clase es anónima -no tiene nombre. Para hacer las cosas un poco peor, se ve como si se estuviera comenzando a crear un objeto **Contents**:

```
| return new Contents()
```

Pero entonces, antes de llegar al punto y coma, se dice, “Pero espere, creo que me deslizaré en una definición de clase”:

```

| return new Contents() {
|     private int i = 11;
|     public int value() { return i; }
| };

```

Lo que esta extraña sintaxis es; “Cree un objeto de una clase anónima que es heredada de **Contents**”. A la referencia retornada por la expresión **new** se le realiza un upcast automáticamente a una referencia **Contents**. La clase interna anónima es un atajo para:

```

class MyContents implements Contents {
    private int i = 11;
    public int value() { return i; }
}
return new MyContents();

```

En la clase interna anónima, **Contents** es creado utilizando un constructor por defecto. El siguiente código muestra que hacer si su clase base necesita un constructor con un argumento.

```

//: c08:Parcel7.java
// An anonymous inner class that calls
// the base-class constructor.
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 se pasa el argumento a el constructor de la clase base, visto aquí como la **x** pasada en **new Wrapping(x)**. Una clase anónima no puede tener un constructor el cual se llamaría normalmente **super()**.

En los dos ejemplos anteriores, el punto y coma no marca el final del cuerpo de la clase (como se hace en C++). En lugar de eso, indica el final de la expresión que sucede contiene la clase anónima. Esto, es idéntico a el uso de un punto y coma en cualquier lado.

¿Que sucede si necesita ejecutar algún tipo de inicialización para un objeto de una clase interna anónima? Dado que es anónima, no tiene nombre para darle al constructor -así es que no puede tener un constructor. Se puede, sin embargo, realizar una inicialización en el punto de la definición de sus campos:

```
///: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
// of Parcel5.java.
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 se esta definiendo una clase interna anónima y se quiere utilizar un objeto que esta definido fuera de la clase anónima interna, el compilador requiere que el objeto de afuera sea **final**. Por esto es que el argumento de **dest()** es **final**. Si se olvida, se obtendrá un mensaje de error en tiempo de compilación.

Siempre y cuando se este asignando simplemente un campo, el siguiente enfoque esta correcto. ¿Pero que si se necesita realizar alguna actividad propia de un constructor? Con la *inicialización de instancia* se puede, en efecto, crear un constructor para una clase interna anónima:

```
///: c08:Parcel9.java
// Utilización de "inicialización de instancia" para realizar
// la construcción de una clase interna anónima.
public class Parcel9 {
    public Destination
    dest(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Inicializador de instancia para cada objeto:
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Over budget!");
            }
        };
    }
}
```

```

        }
        private String label = dest;
        public String readLabel() { return label; }
    };
}
public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
}
} //:~

```

Dentro de la inicialización de instancia se puede ver código que puede no ser ejecutado como parte de un inicializador de campo (esto es, la instrucción **if**). Así es que en efecto, una instancia de inicialización es el constructor para una clase anónima interna. Claro, es limitada; no se puede sobrecargar inicializadores de instancia así es que puede tener solo uno de estos constructores.

## La unión a la clase externa

Hasta el momento, parece que una clase interna simplemente es una forma de ocultar nombres y un esquema de organización de código, lo que es útil pero no totalmente convincente. Sin embargo, hay otra peculiaridad.

Cuando se crea una clase interna, un objeto de esa clase tiene un enlace con el objeto encerrado que lo crea, y se esta manera este puede acceder a los miembros de ese objeto que lo encierra -*sin* ningún requisito especial.

Además, las clases internas tienen derechos de acceso a todos los elementos en la clase que lo encierra<sup>3</sup>. El siguiente ejemplo demuestra esto:

```

//: c08:Sequence.java
// Holds a sequence of Objects.
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++;
        }
    }
}

```

---

<sup>3</sup> Esto es muy diferente a el diseño de *clases anidadas* en C++, que es simplemente un mecanismo para ocultar nombres. No hay unión a un objeto que lo encierra y no hay permisos implicados en C++.

```

    }
    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++;
        }
    }
    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(sl.current());
            sl.next();
        }
    }
}
//:~
```

La **Sequence** es simplemente un arreglo de tamaño fijo de **Object** con una clase que lo envuelve. Se puede llamar a **add()** para agregar un nuevo **Object** al final de la secuencia (si hay espacio sobrante). Para traer cada uno de los objetos en una secuencia, hay una interfase llamada **Selector**, que permite ver si está al final mediante **end()**, para ver el **Object** actual con **current()**, y moverse al siguiente **Object** mediante **next()** en **Sequence**. Dado que **Selector** es una interfase, muchas otras clases pueden implementar la interfase en sus propias formas, y muchos métodos pueden tomar la interfase como argumentos, para crear código genérico.

Aquí, el **SSelector** es una clase privada que proporciona funcionalidad a **Selector**. En el **main()**, se puede ver la creación de una **Sequence**, y luego el agregado de un cierto número de objetos **String**. Entonces, un **Selector** es producido por una llamada a **getSelector()** y esto es utilizado para moverse a través de **Sequence** y seleccionar cada ítem.

Al comienzo, la creación de **SSelector** se ve simplemente como otra clase interna. Pero si la examinamos mas detalladamente. Se puede ver que cada uno de los métodos **end()**, **current()**, y **next()** se refieren a **obs**, que es una referencia que no es parte de **SSelector**, pero en lugar de eso es un campo **private** en la clase que lo encierra. Sin embargo, la clase interna puede acceder a métodos y campos de la clase circundante como si fuera su dueña. Esto vuelve a ser muy conveniente, como se pudo ver en el ejemplo mas arriba.

Así es que una clase interna tiene acceso automático a los miembros de la clase que la encierra. ¿Como puede esto suceder? La clase interior debe mantener una referencia a un objeto particular de la clase circundante que es responsable por crearlo. Entonces cuando se refiere a un miembro de la clase circundante, esta referencia (oculta) es utilizada para seleccionar ese miembro. Afortunadamente, el compilador tiene cuidado de todos esos detalles, pero se puede entender ahora que un objeto de una clase interna puede ser creada solo en asociación con un objeto de la clase circundante. La construcción del objeto de la clase interna requiere la referencia a el objeto de la clase circundante, y el compilador se quejará si no se tiene acceso a esa referencia. La mayoría de las veces esto sucede sin ninguna intervención por parte del programador.

## Clases internas estáticas

Si no se necesita una conexión entre el objeto de la clase interna y el objeto externo, entonces se puede hacer la clase interna del tipo **static**. Para entender el significado de **static** cuando se aplica a una clase interna, se debe recordar que el objeto de una clase interna común implícitamente mantiene una referencia a el objeto de la clase circundante que lo creó. Esto no es verdad, sin embargo, cuando se dice que una clase interna es estática. Una clase estática interna significa:

1. Que no se necesita un objeto de una clase externa para crear un objeto de una clase interna estática.
2. Que no se puede acceder a un objeto de una clase externa desde un objeto de una clase interna estática.

Las clases internas estáticas son diferentes que las clases internas no estáticas de otra forma, igualmente. Los campos y métodos en una clase interna no estática solo pueden estar en el nivel exterior de la clase, así es que las clases no estáticas no pueden tener datos estáticos, campos estáticos, o clases internas estática. Sin embargo, las clases estáticas internas pueden tener todos estos:

```
//: c08:Parcel10.java
// Static inner classes.
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 whereTo) {
```

```

        label = whereTo;
    }
    public String readLabel() { return label; }
    // Static inner classes can contain
    // other static elements:
    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()**, ningún objeto **Parcel10** es necesario; en lugar de eso se utiliza la sintaxis normal para seleccionar miembros estáticos para llamar a esos métodos que retornan referencias a **Contents** y **Destination**.

Como se verá en breve, en una clase interna (no estática) común, el enlace a el objeto de la clase externa es alcanzado con una referencia especial **this**. Una clase interna estática no tiene esta referencia especial **this**, la que la hace análoga a un método estático.

Normalmente se puede colocar cualquier código dentro de una interfase, pero una clase estática interna puede ser parte de una interfase. Dado que la clase es estática no viola las reglas para interfaces -la clase interna estática es solo colocada dentro de los espacios de nombres de la interfase:

```

//: c08:IInterface.java
// Static inner classes inside interfaces.
interface IInterface {
    static class Inner {
        int i, j, k;
        public Inner() {}
        void f() {}
    }
} //:~

```

Al comienzo de este libro, sugerí colocar un **main()** en cada clase para que actúe como dispositivo de pruebas para esa clase. Un inconveniente para esto es la cantidad de código extra compilado que debe cargar por esto. Si esto es un problema, se puede utilizar una clase interna estática para almacenar el 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 llamada **TestBed\$Tester** (para ejecutar el programa, se dice **java TestBed\$Tester**). Se puede utilizar esta clase para pruebas, pero no necesita incluirla en el producto terminado.

## Haciendo referencia al objeto clase externo

Si necesita producir la referencia para el objeto de la clase externo, se nombra el objeto de clase externo seguido por un punto y **this**. Por ejemplo, en la clase **Sequence.SSelector**, alguno de estos métodos pueden producir la referencia almacenada para la clase externa **Sequence** indicándola como **Sequence.this**. El resultado de la referencia es automáticamente el tipo correcto (Esto es conocido y verificada en tiempo de compilación, así es que no hay sobrecarga en tiempo de ejecución).

A veces se quiere indicar algún otro objeto para crear un objeto de uno de su clase interna. Para hacer esto se debe proporcionar una referencia a el otro objeto de la clase interna. Para hacer esto se debe proporcionar una referencia a el otro objeto de la clase externa en la expresión **new**, como esta:

```

//: c08:Parcel11.java
// Creating instances of inner classes.
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 una clase interna directamente, no se debe seguir la misma forma y referenciar el nombre de la clase externa **Parcel11** como se espera, pero en lugar de eso se debe utilizar un *objeto* de la clase externa para hacer un objeto de la clase interna:

```

| Parcel11.Contents c = p.new Contents();
Esto, no es posible para crear un objeto de una clase interna a no ser que ya
tenga un objeto de la clase externa. Esto es porque el objeto de la clase
interna es silenciosamente conectado a el objeto de la clase externa que es
de donde se ha creado. Sin embargo, si se crea una clase interna estática,
entonces no se necesita una referencia a un objeto de la clase externa.

```

## Alcanzando el exterior desde una clase anidada de forma múltiple

<sup>4</sup>No importa cuan profunda esté anidada una clase interna -puede acceder de forma transparente a todos los miembros de todas las clases con las cuales está anidada, como se ve aquí:

```

//: c08:MultiNestingAccess.java
// Las clases anidadas pueden acceder a todos los miembros de todos
// los niveles de las clases con las que son anidadas.
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();
    }
} //:~

```

---

<sup>4</sup> Gracias de nuevo a Martin Danner

Se puede ver que en **MNA.A.B**, los métodos **g()** y **f()** se pueden llamar sin ninguna calificación (a pesar del hecho de que ellos son privados). Este ejemplo también demuestra que la sintaxis necesaria para crear objetos de clases internas anidadas de forma múltiple cuando se crean los objetos en una clase diferente. La sintaxis “**.new**” produce el alcance así es que no se tiene que calificar el nombre de la clase en la llamada del constructor.

## Heredando de una clase interna

Dado que el constructor de la clase interna debe enganchar a una referencia del objeto de la clase que la encierra, las cosas son ligeramente complicadas cuando se hereda de una clase interna. El problema es que la referencia “secreta” a el objeto de la clase que lo encierra *debe* ser inicializado, y aún en la clase derivada ya no hay un objeto por defecto a donde engancharse. La respuesta es utilizar una sintaxis proporcionada para hacer la asociación explícita:

```
//: c08: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);
    }
} ///:~
```

Se puede ver que **InheritInner** se extiende solo en la clase interna, no a la externa. Pero cuando es tiempo de crear un constructor, el constructor por defecto no es bueno y no se puede simplemente pasar una referencia al el objeto que lo envuelve. Además, se debe utilizar la sintaxis

| referenciaALaClaseQueEnvuelve.super();  
dentro del constructor. Esto proporciona la referencia necesaria y el programa compilará.

## ¿Pueden ser las clases internas ser sobrescritas?

¿Que sucede cuando se crea una clase interna, luego se hereda de la clase que la envuelve y se define nuevamente la clase interna? Esto es. ¿Es posible

sobrescribir una clase interna? Esto parece como que sería un concepto poderoso, pero “sobrescribir” una clase interna como si fuera otro método de la clase externa no hace nada realmente:

```
//: c08:BigEgg.java
// An inner class cannot be overridden
// 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();
    }
} ///:~
```

Los constructores por defecto son sintetizados automáticamente por el compilador, y estas llamadas a el constructor por defecto de la clase base. Se puede pensar que puesto que **BigEgg** se esta creando, la versión “sobrescrita” de **Tolk** será utilizada, pero este no es el caso. La salida es:

```
New Egg()
Egg.Yolk()
```

Este ejemplo simplemente muestra que no hay ninguna clase interna mágica extra continuando cuando se hereda de una clase exterior. Las dos clases internas son entidades completamente separadas, cada una en su propio espacio de nombres. Sin embargo, sigue siendo posible explícitamente heredar de 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 **BifEgg2.Yolk** explícitamente extiende **Egg2.Yolk** y sobrescribe sus métodos. El método **insertYolk()** permite que **BifEgg2** realice un upcast de sus propios objetos **Yolk** dentro de su referencia **y** en **Egg2**, así cuando **g()** llama **y.f()** la versión sobrescrita de **f()** es utilizada. La salida es:

```

Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()

```

La segunda llamada a **Egg2.Yolk()** es la llamada a el constructor de la clase base del constructor de **BigEgg2.Yolk**. Se puede ver que la versión sobrescrita de **f()** es utilizada cuando **g()** es llamada.

## Identificadores de clases internas

Dado que cada clase produce un fichero **.class** que contiene toda la información acerca de como crear objetos de este tipo (esta información produce una “meta clase” llamada el objeto **Class**), se puede adivinar que es clase interna debe producir un fichero **.class** para almacenar la información de *sus* objetos **Class**. Los nombres de esos ficheros/clases tienen una fórmula estricta: el nombre de la clase que lo contiene, seguido por un ‘\$’, seguido por el nombre de la clase interna. Por ejemplo, los ficheros **.class** creados por **InheritUnner.java** incluyen:

```

InheritInner.class
WithInner$Inner.class
WithInner.class

```

Si las clases internas son anónimas, el compilador simplemente genera números como identificadores de la clase interna. Si la clase interna es anidada con clases internas, sus nombres son simplemente anexados luego de un '\$' y el(los) identificador(es) de la clase externa.

A pesar que este esquema de generar nombres internos es simple y directo, es también robusto y maneja la mayoría de las situaciones<sup>5</sup>. Dado que es el esquema de nombres estándar de Java, los ficheros generados son automáticamente independientes de la plataforma (Véase que el compilador Java cambia sus clases internas en todo tipo de otras formas para hacer su trabajo).

## ¿Por que clases internas?

En este punto se ha visto un montón de sintaxis y semántica que describen la forma en que las clases internas trabajan, pero esto no responde la pregunta de por que existen. ¿Por que Sun se tomo tanto trabajo para agregar este rasgo fundamental del lenguaje?

Típicamente, las clases internas heredan de una clase o implementan una interfase, y el código en la clase interna manipula el objeto de la clase externa en cuyo interior fue creada. Así es que se puede decir que una clase interna proporciona un tipo de ventana en la clase externa.

Una pregunta que directa al corazón de las clases internas es esta: ¿Si simplemente necesito una referencia a una interfase, por que no simplemente hago que la clase externa implemente esa interfase? La respuesta es que no se puede siempre tener la conveniencia de las interfaces -a veces se está trabajando con implementaciones. Así es que la razón mas convincente para las clases internas es:

Cada clase interna puede independientemente heredar de una implementación. De esta manera, la clase interna no esta limitada por el hecho de que la clase externa ya este heredada de una implementación.

Sin la habilidad que esa clase interna proporciona al heredar -en efecto- de mas de una clase concreta o abstracta, algunos diseños y problemas de programación pueden ser intratables. Así es que una forma de ver la clase interna es como la solución total a el problema de la herencia múltiple. Las interfaces solucionan parte del problema, pero las clases internas efectivamente permiten "herencia de múltiple implementación". Esto es, las clases internas efectivamente permiten heredar de mas de una no interfase.

---

<sup>5</sup> Por el otro lado, '\$' es un meta carácter para el shell de Unix y por esto a veces se tienen problemas cuando se listan ficheros .class. Esto es un poco extraño por parte de Sun, una compañía basada en Unix. Adivino que no consideraron este tema, pero en lugar de eso el programador se había naturalmente enfocado en los ficheros de los fuentes.

Para ver esto en mas detalle, considere una situación donde se tiene dos interfaces que deben de alguna manera ser implementadas sin una clase. Dado la flexibilidad de las interfaces, tiene dos alternativas: una clase simple o una clase interna:

```
//: c08:MultiInterfaces.java
// Dos formas en la que una clase puede
// implementar multiples interfaces.
interface A {}
interface B {}
class X implements A, B {}
class Y implements A {
    B makeB() {
        // Anonymous inner class:
        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());
    }
} ///:~
```

Claro que, esto asume que para la estructura del código las dos formas tienen sentido. Sin embargo, normalmente se tendrá algún tipo de orientación por parte de la naturaleza del problema acerca de si utilizar una sola clase o una clase interna. Pero sin otras limitaciones, en el siguiente ejemplo la estrategia que se toma no hace mucha diferencia realmente desde un punto de vista de la implementación. Ambos trabajan.

Sin embargo, si se tiene una clase abstracta o concreta en lugar de interfaces, se esta repentinamente limitado a utilizar clases internas si su clase debe de alguna forma implementar ambas otras:

```
//: c08:MultiImplementation.java
// Con clases concretas o abstractas, las clases
// internas son solo la forma para producir el
// efecto de "herencia de múltiple implementación."
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 se necesita implementar un problema de “herencia de múltiple implementación”, se puede concebir código acerca de todo sin la necesidad de una clase interna. Pero con las clases internas se tiene estas características adicionales:

1. La clase interna puede tener múltiples instancias, cada una con su propio estado de información que es independiente de la información en el objeto de la otra clase.
2. En una clase externa simple, se puede tener muchas clases internas, cada una que implemente la misma interfase o herede de la misma clase en una forma diferente. Un ejemplo de esto será mostrado en breve.
3. El punto de creación del objeto de la clase interna no está ligado a la creación del objeto de la clase externa.
4. No hay una relación posiblemente confusa de “es un” con la clase interna; es una entidad separada.

Como un ejemplo, si **Sequence.java** no utilizara clases internas, no se tendría que decir “una **Sequence** es un **Selector**”, y no solo se sería capaz de tener un **Selector** en existencia para una particular **Sequence**. También, se puede tener un segundo método, **getRSelector()**, esto produce un **Selector** que mueve para atrás la secuencia. Este tipo de flexibilidad es solo capaz con las clases internas.

## Cierres y callbacks

Un *cierre* es un objeto al que se puede llamar que retiene información acerca del alcance en el cual fue creado. Partiendo de esta definición, se puede ver que una clase interna es un objeto orientado a cierre, porque no solo contiene cada parte de la información del objeto de la clase externa (“el alcance en el cual fue creado”), si no que automáticamente mantiene una referencia de retorno a la totalidad del objeto de la clase externa, donde tiene permiso de manipular todos los miembros, inclusive los privados.

Uno de los argumentos más convincentes hechos para incluir algún tipo de mecanismo de puntero en Java fue permitir *callbacks*. Con una callback, algún otro objeto entrega un pedazo de información que le permite realizar en algún punto más adelante llamar al objeto que lo originó. Esto es un concepto muy poderoso, como se podrá ver en el capítulo 13 y 16. Si una callback es implementada utilizando un puntero, sin embargo, se debe confiar en que el programador se comporte y no desaproveche el puntero.

Como se ha visto por ahora, Java tiene la tendencia de ser mas cuidadoso que eso, así es que los punteros no son incluidos en el lenguaje.

El cierre proporcionado por la clase interna es una solución perfecta; mas flexible y segura que un puntero. He aquí un ejemplo simple:

```
//: c08:Callbacks.java
// Utilizando clases internas para callbacks
interface Incrementable {
    void increment();
}
// Muy simple para implementar solo la interfase::
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();
    }
}
// La clase debe implementar increment() de
// alguna otra forma, se debe utilizar 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 proporciona también una distinción adicional entre implementar una interfase mediante una clase externa o hacerlo con una clase interna. **Callee1** es claramente la solución mas simple en términos del código. **Callee2** hereda de **MyIncrement** que ya tiene un método **increment()** diferente que hace algo no relacionado con eso que se espera de la interfase **Incrementable**. Cuando **MyIncrement** es heredado dentro de **Callee2**, **increment()** no puede ser sobrescrito para utilizar por **Incrementable**, así es que se esta forzado a proporcionar una implementación por separado utilizando una clase interna. Debe verse también que cuando se crea una clase interna no se agrega o modifica la interfase de la clase externa.

Debe notarse que todo exceptuando **getCallbackReference()** en **Callee2** es privado. Para permitir *cualquier* conexión al mundo externo, la interfase **Incrementable** es esencial. Aquí se puede ver como las interfaces permiten una completa separación de la interfase con la implementación.

La clase interna **Closure** simplemente implementa **Incrementable** para proporcionar un gancho de retroceso dentro de **Callee2** -pero un gancho seguro. Quienquiera que tenga la referencia **Incrementable** puede, claro, solo llamar a **increment()** y no tiene otras habilidades (a diferencia de un puntero, que permitirá correr descontroladamente).

**Caller** toma una referencia **Incrementable** en su constructor (a pesar que la captura de la referencia callback puede suceder en cualquier momento) y entonces, en algún momento mas adelante, se utiliza la referencia para llamar dentro de la clase **Callee**.

El valor de la callback esta en su flexibilidad -se puede dinámicamente decidir que funciones serán llamadas en tiempo de ejecución. El beneficio de esto se hace mas evidente en el capítulo 13, donde las callbacks son utilizadas en todos lados para implementar funcionalidades en interfaces gráficas de usuario (GUI).

# Clases internas y frameworks de control

Un ejemplo mas complejo del uso de las clases internas pueden ser encontradas en algo a lo que llamará aquí como *framework de control*

Un *framework de aplicaciones* una clase o un grupo de clases que están diseñadas para solucionar un tipo particular de problema. Para poner en práctica un framework de aplicación, debe heredar de una o mas clases y sobrescribir algunos de los métodos. El código que se escribe en los métodos sobrescritos construyen a la medida la solución general proporcionadas por un framework de aplicación, para solucionar su problema específico. El framework de control es un tipo de aplicación particular de framework de aplicación dominado por la necesidad de responder a eventos; un sistema que responde primariamente a eventos es llamado un *sistema accionado por eventos*. Uno de los problemas mas importantes en la programación de aplicaciones es la interfase gráfica de usuario (GUI), que es poco menos que enteramente accionado por eventos. Como se verá en el capítulo 13, la librería Swing de Java es un framework de control que de forma elegante soluciona el problema de la GUI y que utiliza mucho las clases internas.

Para ver como las clases internas permiten la simple creación y uso de los frameworks de control, considere un framework de control cuyo trabajo es ejecutar eventos cuando quiera que esos eventos estén “listos”. A pesar que “listo” puede significar cualquier cosa, en este caso el incumplimiento esta fundamentado en tiempo de reloj. Lo que sigue es un framework de control que contiene información que no es específica acerca de que es lo que esta controlando. Primero, aquí está la interfase que describe cualquier evento de control. Es una clase abstracta en lugar de la interfase actual porque el comportamiento por defecto es realizar el control fundamentado en tiempo, así es que algo de la implementación puede estar incluida aquí:

```
//: c08:controller:Event.java
// The common methods for any control event.
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 en que se quiere que **Event** ejecute, mientras que **ready()** indica cuando es tiempo de ejecutarlo. Claro, **ready()** puede ser sobreescrito en una clase derivada para basar **Event** en alguna otra cosa que tiempo.

**action()** es el método que es llamado cuando el **Event** esta **ready()**, y **description()** da una información textual acerca de **Event**.

El siguiente fichero contiene el framework de control actual que maneja y dispara los eventos. La primer clase es realmente solo una clase de “ayuda” cuyo trabajo es guardar los objetos **Event**. Se puede remplazar con un container apropiado, y en el capítulo 9 de descubrirá otros contenedores que harán el truco sin que se requiera la escritura de este código extra:

```
///: c08:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c08.controller;
// Esto es solo una forma de mantener objetos Event.
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; // (En la vida real, lanza una excepción)
        events[index++] = e;
    }
    public Event getNext() {
        boolean looped = false;
        int start = next;
        do {
            next = (next + 1) % events.length;
            // Verifica que el bucle se realice desde el comienzo:
            if(start == next) looped = true;
            // Si se pasa del comienzo, la lista
            // es vacía:
            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** arbitrariamente mantiene 100 objetos **Event** (Si un contenedor “real” del capítulo 8 es utilizado aquí no se necesita preocuparse del tamaño máximo, dado que cambiará el tamaño solo). **index** es utilizado para mantener la pista del siguiente espacio disponible, y **next** es utilizado cuando se esta observando por el siguiente **Event** en la lista, para ver si se encuentra dentro del bucle. Esto es importante durante la llamada a **getNext()**, porque los objetos **Event** son quitados de la lista (utilizando **removeCurrent()**) una vez que se ejecute, así es que **getNext()** encontrará agujeros en la lista y se moverá a través de ellos.

Debe verse que **removeCurrent()** no configura simplemente alguna bandera indicando que el objeto no esta mas en uso. En lugar de eso, coloca la referencia en **null**. Esto es importante porque si el recolector de basura ve que la referencia sigue en uso no limpiará el objeto. Si se piensa que las referencias pueden colgarse (como ellas podrían aquí), entonces es una buena idea colocarlas a **null** para darle permiso al recolector de basura de limpiarlas.

**Controller** es donde se sucede el trabajo actual. Utiliza un **EventSet** para mantener sus objetos **Event**, y **addEvent()** permite agregar nuevos eventos a esta lista. Pero el método mas importante es **run()**. Este método realiza un bucle a través de **EventSet**, buscando un objeto **Event** que este listo para ejecutarse. Si mediante el método **ready()** se encuentra listo, se llama al método **action()**, imprime la descripción mediante **description()**, y luego se quita el objeto **Event** de la lista.

Debe verse que hasta el momento en este diseño no se sabe nada acerca de *que* exactamente hace un **Event**. Y este es el punto crucial del diseño; como “separa las cosas que cambian de las cosas que siguen siendo las mismas”. O, para utilizar mis términos, el “vector de cambio” son las diferentes acciones de los distintos tipos de objetos **Event**, y se expresa las diferentes acciones creando diferentes subclases **Event**.

Esto es donde las clases internas comienzan a jugar. Ellas permiten dos cosas:

1. Para crear la implementación completa de un framework de control de aplicación en una sola clase, en consecuencia encapsular todo es único en esta implementación. Las clases internas son utilizadas para expresar las muchas diferentes tipos de **action()** necesaria para solucionar el problema. Además, el siguiente ejemplo usa clases

internas privadas así es que la implementación esta oculta completamente y puede ser cambiada con impunidad.

2. Las clases internas controlan esta implementación de volverse torpes, dado que se es capas de acceder fácilmente a cualquiera de los miembros en la clase externa. Sin esta habilidad el código puede convertirse en suficientemente desagradable a el punto terminar buscando una alternativa.

Considerando una implementación particular del marco de trabajo de control diseñado para controlar las funciones de un invernadero<sup>6</sup>. Cada acción es completamente diferente: encender y apagar luces, agua y termostatos, sonar campanas, y reiniciar el sistema. Pero el marco de trabajo de control es diseñado para fácilmente aislar estos códigos diferentes. Las clases internas permiten tener muchas versiones derivadas de la clase base, **Event**, dentro de una única clase. Por cada tipo de acción se hereda una nueva clase interna **Event**, y escribir el código de control dentro de **accion()**.

Como es típico en un marco de trabajo de aplicación, la clase **GreenhouseControls** es heredada de **Controller**:

```
//: c08:GreenhouseControls.java
// Esto produce una aplicación específica de
// sistema de control, todo en una sola clase. Las
// clases internas 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() {
            // Coloque el código de control de hardware
            // aquí para físicamente prender la luz.
            light = true;
        }
        public String description() {
            return "Light is on";
        }
    }
    private class LightOff extends Event {
        public LightOff(long eventTime) {
            super(eventTime);
```

---

<sup>6</sup> Por alguna razón esto es siempre un problema que me satisface solucionar; viene de mi temprano libro de C++ *Inside & Out*, pero Java permite una solución mucho mas elegante.

```

    }
    public void action() {
        // Coloque el código de control de hardware
        // aquí para físicamente apagar la luz.
        light = false;
    }
    public String description() {
        return "Light is off";
    }
}
private class WaterOn extends Event {
    public WaterOn(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Coloque el control de hardware 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() {
        // Coloque el control de hardware 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() {
        // Coloque el control de hardware 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() {
        // Coloque el control de hardware aquí
        thermostat = "Day";
    }
    public String description() {

```

```

        return "Thermostat on day setting";
    }
}
// Un ejemplo de una action() que inserta una
// nueva de si misma en la lista de eventos:
private int rings;
private class Bell extends Event {
    public Bell(long eventTime) {
        super(eventTime);
    }
    public void action() {
        // Ring 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();
        // En lugar de codificación, se puede
        // colocar información de configuración
        // en un fichero 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));
        // Puede inclusive agregar 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();
}
} ///:~

```

Debe notarse que **light**, **water**, **thermostat**, y **rings** pertenecen a la clase externa **GreenhouseControls**, y aún las clases internas pueden acceder a esos campos sin ningún requisito o permiso especial. También, la mayoría de los métodos **action()** involucra algún tipo de control de hardware, que puede, en el mejor de los casos involucrar código que no es Java.

Muchas de las clases **Event** se ven similares, pero **Bell** y **Restart** son especiales. **Bell** suena, y si todavía no ha sonado suficiente veces agrega un nuevo objeto **Bell** a la lista de eventos, así es que sonara nuevamente más tarde. Debe notarse como las clases internas *al menos* se ven como herencia múltiple: **Bell** tiene todos los método de **Event** y estos también parece tener todos los métodos de la clase externa **GreenhouseControls**.

**Restart** es responsable por la inicialización del sistema, así es que agrega todos los métodos apropiados. Claro que una forma mas sencilla de lograr esto es evitar el codificar los eventos y en lugar de eso leerlos de un fichero (Un ejercicio en el capítulo 11 pregunta como modificar este ejemplo para hacer exactamente eso). Dado que **Restart()** es simplemente otro objeto **Event**, se puede también agregar un objeto sin **Restart.action()** así es que el sistema regularmente reinicia solo. Y todo lo que necesita para hacer el **main()** es crear un objeto **GreenhouseControls** y agregar un objeto **Restart** para ponerlo en marcha.

Este ejemplo nos guía por un largo camino para apreciar el valor de las clases internas, especialmente cuando son utilizadas con un marco de trabajo de control. Sin embargo, en el capítulo 13 se verá de que forma elegante las clases internas son utilizadas para describir las acciones de una interfaces grafica de usuario. Para el momento en que terminemos el capítulo se deberá estar totalmente convencido.

## Resumen

Las interfaces y las clases internas son los conceptos mas sofisticadas que se encontrarán en muchos leguajes de POO. Por ejemplo, no hay nada como ellos en C++. Juntos, resuelven el mismo problema que C++ intente resolver con herencia múltiple (HM). Sin embargo, HM en C++ se vuelve complicado de utilizar, donde las interfaces y clases internas de Java son, en comparación mas accesibles.

A pesar que estas características por si solas son directas, el uso de estas características es un tema de diseño, en gran medida igual que el polimorfismo. Todo el tiempo, se transformará en reconocer situaciones donde se deberá utilizar una interfase, o una clase interna, o ambas. Pero en este punto en este libro se deberá al menos sentir confortable con la sintaxis y la semántica. Cuando se vean estas características del lenguaje en uso eventualmente se interiorizarán.

# Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Pruebe que los campos en una interfase son implícitamente estáticas y finales.
2. Cree una interfase que tenga tres métodos, en su propio paquete. Implemente la interfase un paquete diferente.
3. Pruebe que todos los métodos en una interfase son automáticamente públicos.
4. En **co7:Sandwich.java**, cree una interfase llamada **FastFood** (con los métodos apropiados) y cambie **Sandwich** de tal forma que también incremente **FastFood**.
5. Cree tres interfaces, cada una con dos métodos. Herede una nueva interfase de las tres, agregue un nuevo método. Cree una clase implementando la nueva interfase y también heredando de una clase bien establecida. Ahora escriba cuatro métodos, cada uno de los cuales toma uno de las cuatro interfaces como argumento. En el **main()**, cree un objeto de su clase y páselos a cada uno de los métodos.
6. Modifique el ejercicio 5 creando una clase abstracta y herédelo dentro de la clase derivada.
7. Modifique **Music5.java** agregando una interfase **Playable**. Elimine la declaración **play()** de **Instrument**. Agregue **Playable** a la clase derivada incluyéndola en la lista de implementaciones. Cambie **tune()** para que tome un **Playable** en lugar de un **Instrument**.
8. Cambie el ejercicio 6 en el capítulo 7 para que **Rodent** sea una interfase.
9. En **Adventure.java**, agregue una interfase llamada **CanClimb**, siguiendo la forma de las otras interfaces.
10. Escriba un programa que importe y utilice **Month2.java**.
11. Siguiendo el ejemplo dado en **Month2.java**, cree una enumeración de días de la semana.
12. Cree una interfase con al menos un método, en su propio paquete. Cree una clase en un paquete separado. Agregue una clase interna protegida que implemente la interfase. En un tercer paquete, herede de su clase y, dentro de un método, retorne un objeto de la clase interna protegida, conversión ascendente a la interfase durante el retorno.

13. Cree una interfase con al menos un método, implemente esa interfase definiendo una clase interna dentro de un método, que retorne una referencia a su interfase.
14. Repita el ejercicio 13 pero defina la clase interna sin un alcance dentro del método.
15. Repita el ejercicio 13 utilizando una clase interna anónima.
16. Cree una clase interna privada que implemente una interfase publica. Escriba un método que retorne una referencia a una instancia de la clase interna privada, realice una conversión ascendente a la interfase. Muestre que la clase interna esta completamente oculta intentando realizar una conversión descendente a ella.
17. Cree una clase con un constructor que no sea por defecto y sin constructor por defecto. Cree una segunda clase que tenga un método que retorne una referencia a la primera clase. Cree una segunda clase que tenga un método que retorne una referencia a la primer clase. Cree el objeto para que retorne creando una clase interna anónima que herede de la primer clase.
18. Cree una clase con un campo privado y un método privado. Cree una clase interna con un método que modifique el campo de la clase externa y llame el método de la clase externa. En un segundo método de clase externa, cree un objeto de la clase interna y llame su método, luego muestre el efecto en el objeto de la clase externa.
19. Repita el ejercicio 18 utilizando una clase interna anónima.
20. Cree una clase conteniendo una clase interna estática. En el **main()**, cree una instancia de la clase interna.
21. Cree una interfase conteniendo una clase interna estática. Implemente esta interfase y cree una instancia de la clase interna.
22. Cree una clase conteniendo una clase interna que contenga en si misma una clase interna. Repita esto utilizando una clase interna estática. Deben verse con cuidado los nombres de los ficheros **.class** producidos por el compilador.
23. Cree una clase con una clase interna. En una clase separada, cree una instancia de la clase interna.
24. Cree una clase con una clase interna que tenga un constructor que no sea por defecto. Cree una segunda clase con una clase interna que herede de la primera clase interna.
25. Repare el problema en **WindError.java**.

26. Modifique **Sequence.java** agregando un método **getRSelector()** que produzca una implementación diferente de la interfase **Selector** que mueve un paso atrás a través de la secuencia desde el final al comienzo.
27. Cree una interfase **U** con tres métodos. Cree una clase **A** con un método que produzca una referencia a un **U** creando una clase interna anónima. Cree una segunda clase **B** que contenga un arreglo de **U**. **B** debe tener un método que acepte y almacene una referencia a **U** en el arreglo, un segundo método que configure una referencia en el arreglo (especificado por el argumento del método) a **null** y un tercer método que mueva a través del arreglo y llame los métodos en **U**. En el **main()**, cree un grupo de objetos **A** y un solo objeto **B**. Llene el **B** con referencias **U** producidas por los objetos **A**. Utilice el **B** para realizar llamadas para atrás dentro de todos los objetos **A**. Quite algunas de las referencias **U** de **B**.
28. En **GreenhouseControls.java**, agregue una clase interna **Event** que prenda y apague ventiladores.
29. Muestre que una clase interna tiene acceso a los elementos privados de su clase externa.- Determine si la inversa es verdadera.

# 9: Almacenando objetos

Es un programa bastante simple si solo tiene una cantidad fija de objetos con tiempos de vida conocidos.

En general, los programas estarán creando nuevos objetos basados en algún criterio que será conocido solo en el momento que el programa se está ejecutando. No se conocerá hasta el momento de la ejecución la cantidad o incluso el tipo exacto del objeto que se necesita. Para solucionar el problema general de programación, se necesita ser capaz de crear cualquier número de objetos, en cualquier momento, en cualquier lugar. Así es que no se puede confiar en la creación de un nombre de referencia para almacenar cada uno de los objetos:

```
| MyObject myReference;
```

Para resolver este problema esencial de la mejor forma, Java tiene muchas formas de almacenar objetos (o mejor, referencias a objetos). El tipo incorporado es el arreglo, que ha sido discutido anteriormente. También, la librería de utilitarios de Java tiene un razonablemente completo grupo de clases contenedoras (también conocidas como compendio de clases o *collection classes*, pero dado que las librerías de Java 2 usan el nombre **Collection** para referirse a un subgrupo especial de la librería, usaré el término mas inclusivo término "contenedor"). Los contenedores proporcionan formas sofisticadas de almacenar e incluso manipular los objetos.

## Arreglos

La mayoría de la introducción necesaria a los arreglos está en la última sección del capítulo 4, donde es mostrado como se define e inicializa un arreglo. Retener los objetos es el tema de este capítulo, y un arreglo es simplemente una forma de almacenar objetos. ¿Entonces, qué es lo que hace un arreglo especial?

Hay dos cosas que distinguen los arreglos de otros tipos de contenedores: eficiencia y tipo. El arreglo es la forma mas eficiente que proporciona Java para almacenar y acceder de forma aleatoria a una secuencias de objetos (actualmente, referencias a objetos) El arreglo es una simple secuencia

lineal, que hace el acceso a los objetos rápidos, pero se paga por esta velocidad: cuando se crea un arreglo, su tamaño es fijo y no puede ser cambiado durante la vida del objeto. Se puede sugerir la creación de un arreglo de un tamaño particular y entonces, si se agota el espacio, crear uno nuevo y mover todas las referencias del viejo al nuevo. Esto es el comportamiento de la clase **ArrayList**, que será estudiada mas adelante en este capítulo. Sin embargo, dado el gasto de esta flexibilidad en el tamaño, una **ArrayList** es sensiblemente menos eficiente que un arreglo.

La clase contenedor **vector** en C++ *conoce* el tipo de los objetos que contiene, pero tiene un inconveniente diferente cuando se compara con arreglos en Java: el vector **operator[]** de C++ no chequea los límites, así es que se puede pasar el final<sup>1</sup>. En Java, se realiza una verificación de los límites independientemente de si se esta utilizando un arreglo o un contenedor -se obtendrá una **RuntimeException** si se excede los límites. Como se aprenderá en el capítulo 10, este tipo de excepción indica un error de programación, y de esta manera no se necesita verificarlo en el código. Aparte, la razón para que C++ no verifique los límites con cada acceso es la velocidad -en Java se tiene una se tiene un gasto de tiempo de computación constante por chequeos de limites todo el tiempo para arreglos y contenedores.

La otra clase contenedora genérica que será estudiada en este capítulo, **List**, **Set**, y **Map**, todas tratan con objetos que no tienen un tipo específico. Esto es, ellos son tratados como objetos del tipo **Object**, la clase raíz de todas las clases en Java. Esto funciona bien desde un punto de vista necesario para crear solo un contenedor, y cualquier objeto Java podrá ir en el contenedor (excepto primitivas-estas pueden ser colocadas en contenedores como constantes utilizando envoltorios para primitivas de Java, o como valores permutables envolviéndolos en sus propias clases). Esto es el segundo lugar donde un arreglo es superior a un contenedor genérico: cuando se crea un arreglo, se crea para almacenar un tipo específico. Esto significa que el tipo se obtiene en tiempo de compilación chequeando para prevenir que no se coloque el tipo equivocado, o equivocarse cuando se esta extrayendo. Claro, Java previene de enviar un mensaje inapropiado a un objeto, en tiempo de compilación o en tiempo de ejecución. Así es que no es mucho riesgo de una forma u otra, es simplemente amable si el compilador lo indica, mas rápido en tiempo de ejecución, y hay menos probabilidades de que el usuario final tenga una sorpresa con una excepción.

Por un tema de eficiencia y verificación de tipo vale la pena siempre tratar de utilizar un arreglo si es posible. Sin embargo, cuando se trate de solucionar un problema mas general los arreglos pueden ser muy

---

<sup>1</sup> Es posible sin embargo, preguntar que tan grande es el vector, y en el método **at()** realizar la verificación de límites.

restrictivos. Luego de analizar la utilización de arreglos este capítulo será totalmente consagrado a las clases contenedoras proporcionadas por Java.

## Los arreglos son objetos de primera calidad

Independientemente del tipo de arreglo con el que se este trabajando, el identificador de arreglo es actualmente una referencia a un objeto verdadero que es creado en el heap. Este es el objeto que contiene las referencias a los otros objetos y puede ser creado implícitamente como parte de la sintaxis de inicialización del arreglo o explícitamente con una expresión **new**. Parte del objeto arreglo (de echo, el único campo o método al cual se puede acceder) es el miembro de solo lectura **length** que indica cuantos elementos pueden ser almacenados en el objeto arreglo.

El siguiente ejemplo muestra las distintas formas en las que un arreglo puede ser inicializado, y como las referencias a arreglos pueden ser asignadas a los diferentes objetos arreglo. Muestra también los arreglos de objetos y de primitivas son al menos idénticos en su uso. La única diferencia es que ese arreglo de objetos contiene referencias, mientras que los arreglos de primitivas contienen los valores primitivos directamente.

```
//: c09:ArraySize.java
// Initialization & re-assignment of arrays.
class Weeble {} // A small mythical creature
public class ArraySize {
    public static void main(String[] args) {
        // Arrays of objects:
        Weeble[] a; // Null reference
        Weeble[] b = new Weeble[5]; // Null references
        Weeble[] c = new Weeble[4];
        for(int i = 0; i < c.length; i++)
            c[i] = new Weeble();
        // Aggregate initialization:
        Weeble[] d = {
            new Weeble(), new Weeble(), new Weeble()
        };
        // Dynamic aggregate initialization:
        a = new Weeble[] {
            new Weeble(), new Weeble()
        };
        System.out.println("a.length=" + a.length);
        System.out.println("b.length = " + b.length);
        // The references 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);
// Arrays of primitives:
int[] e; // Null reference
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);
e = new int[] { 1, 2 };
System.out.println("e.length = " + e.length);
}
} //:~

```

He aquí 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 arreglo **a** es inicialmente una referencia a **null**, y el compilador previene de hacer algo con esta referencia hasta que haya sido inicializada propiamente. El arreglo **b** es inicializado para apuntar a un arreglo de referencias **Weeble**, pero actualmente ningún objeto **Weeble** son colocados en el arreglo. Sin embargo, se puede seguir preguntando cual es el tamaño del arreglo, dado que **length** solo dice cuantos elementos *pueden* ser colocados en el arreglo, esto es el tamaño del objeto arreglo, no el número de elementos que actualmente contiene. sin embargo, cuando un objeto

arreglo es creado sus referencias son automáticamente inicializadas a **null**, así es que se puede ver cuando en una ubicación en particular del arreglo hay un objeto verificando si es o no **null**. De forma similar, un arreglo de primitivas es automáticamente inicializado a cero para tipos numéricos, **(char)0** para **char**, y **false** para **boolean**.

Al arreglo **c** muestra la creación de un objeto arreglo seguido por la asignación de objetos **Weeble** para todas las ubicaciones en el arreglo. El arreglo **d** muestra la sintaxis de la "inicialización en conjunto" que produce que un objeto del tipo arreglo sea creado (implícitamente con **new** en el heap, exactamente como para el arreglo **c**) y inicializado con objetos **Weeble**, todo en una sola instrucción.

La siguiente inicialización de arreglo puede ser pensado como "inicialización dinámica en conjunto". La inicialización en conjunto utilizada por **d** debe ser utilizada en el punto de la definición de **d**, pero con la segunda sintaxis se puede crear y inicializar un arreglo de objetos en cualquier lugar. Por ejemplo, supóngase que **hide()** es un método que toma un arreglo de objetos **Weeble**. Se puede llamar a este diciendo:

```
| hide(d);  
pero también se puede crear también dinámicamente el arreglo pasándolo  
como argumento:  
| hide(new Weeble[] { new Weeble(), new Weeble() });  
En algunas situaciones esta nueva sintaxis proporciona una forma mas  
conveniente de escribir código.
```

La expresión

```
| a = d;  
muestra como se puede tomar una referencia que esta enganchada a un  
objeto arreglo y asignarla a otro objeto arreglo, exactamente como se podría  
hacer con otro tipo de referencia a objeto. Ahora a y b apuntan a el mismo  
objeto arreglo en el heap.
```

La segunda parte de **ArraySize.java** muestra que los arreglo de primitivas trabajan exactamente como arreglos *exceptuando* que los arreglos de primitivas contienen los valores primitivos directamente.

## Contenedores de primitivas

Las clases contendores pueden almacenar solo referencias a objetos. Un arreglo, sin embargo, puede ser creado para almacenar primitivas directamente, de la misma forma que referencias a objetos. Es posible utilizar las clases "envoltura" como **Integer**, **Double**, etc. para colocar valores primitivos dentro de un contenedor, pero las clases envoltura para primitivas pueden ser complicadas de utilizar. Además, es mucho mas

eficiente crear y acceder a un arreglo de primitivas que a un contenedor de primitivas envueltas.

Claro, si se esta utilizando un tipo primitivo y necesita la flexibilidad de un contenedor que puede ser expandido automáticamente cuando se necesita mas espacio, el arreglo no funciona y se esta forzado a utilizar un contenedor para envolver primitivas. Se puede pensar que debería haber un tipo especializado de **ArrayList** para cada uno de los datos primitivos, pero Java no proporciona esto. Algún tipo de mecanismo mediante plantillas puede algún día soportar una mejor forma para que Java maneje este problema<sup>2</sup>.

## Retornando un arreglo

Supongamos que se esta escribiendo un método y no se quiere retornar solamente una cosa, se quiere retornar un montón de cosas. Lenguajes como C y C++ hacen esto dificultoso porque no se puede simplemente retornar un arreglo, solo un puntero a un arreglo. Esto introduce problemas porque hace desordenado el control del tiempo de vida del arreglo, el cual fácilmente conduce a agujeros en la memoria. Java tiene una estrategia similar, pero solo se "retorna un arreglo". Actualmente, claro, se esta retornando una referencia a un arreglo, pero con Java nunca se preocupa acerca de la responsabilidad de ese arreglo -este estará en el momento en que se lo necesite y el recolector de basura lo limpiará cuando se haya terminado.

Como ejemplo, considere retornar un arreglo de Cadenas:

```
//: c09: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];
        boolean[] picked =
            new boolean[flav.length];
        for (int i = 0; i < n; i++) {
            int t;
            do
                t = (int)(Math.random() * flav.length);
            while (picked[t]);
```

---

<sup>2</sup> Este es uno de los lugares donde C++ es claramente superior a Java, dado que C++ soporta *tipos parametrizados* con la palabra clave **template**.

```

        results[i] = flav[t];
        picked[t] = true;
    }
    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 arreglo de cadenas llamado **result**. El tamaño de este arreglo es n, determinado por el argumento que se pasa al método. Entonces este comienza a elegir sabores de forma aleatoria del arreglo **flav** y a colocarlos en **result**, el cual finalmente se retorna. Retornar un arreglo es simplemente como retornar cualquier otro objeto -es una referencia. No es importante que el arreglo fuera creado con **flavorSet()**, o que el arreglo fuera creado en cualquier lado, para este problema. El recolector de basura tiene cuidado de limpiar el arreglo cuando se haya terminado con él, y el arreglo persistirá mientras se necesite.

Aparte, debe notarse que cuando **flavorSet()** elige de forma aleatoria, se asegura que esa elección no ha sido realizada. Esto es realizado con un bucle **do** que realiza elecciones hasta que encuentra uno que no este en el arreglo **picked** (claro que, una comparación de cadenas puede también realizarse para ver si la elección aleatoria ya esta entre el arreglo con los resultados, pero las comparaciones de **String** son ineficientes). Si es exitoso, agrega la entrada y encuentra el siguiente (**i** se incrementa).

**main()** imprime 20 grupos de sabores, así es que se puede ver que **flavorSet()** selecciona los sabores en orden aleatorio cada vez. Es fácil ver esto si se realiza una redirección de la salida a un fichero. Y cada vez que se vea el contenido del fichero, se debe recordar, que solo se quería el helado, no se necesitaba.

## La clase **Array**

En **java.util**, se encontrara la clase **Array**, que contiene un grupo de métodos estáticos que realizan funciones utilitarias para arreglos. Estas son cuatro funciones básicas: **equals()**, para comparar la igualdad de dos arreglos; **fill()**, para llenar un arreglo con un valor; **sort()**, para ordenar el arreglo; y **binarySearch()**, para encontrar un elemento en un arreglo ordenado. Todos estos métodos son sobrecargados para todos los tipos primitivos y **Objects**. Además, hay un método individual llamado **asList()** que toma un arreglo y

lo coloca dentro de un contenedor **List** -del cual se aprenderá mas tarde en este capítulo.

A pesar de que es útil, la clase **Array** no llega a ser totalmente funcional. Por ejemplo, sería bueno ser capas de imprimir los elementos de un arreglo sin tener que codificar un bucle **for** a mano cada vez. Y como se habrá visto, el método **fill()** solo toma un solo valor y lo coloca en el arreglo, así es que si se quiere -por ejemplo- llenar el arreglo con un grupo de números generados de forma aleatoria, **fill()** no ayudará.

Así es que tiene sentido suplementar la clase **Array** con algunas utilidades adicionales, se estarán ubicadas en el paquete **com.bruceeckel.util** por conveniencia. Estos imprimirán un arreglo de cualquier tipo, y llenaran un arreglo con valores de objetos que son creados por un objeto llamado *generador*que se puede definir.

Dado que el código necesita ser creado para cada tipo primitivo de la misma forma que para **Object**, hay aquí una gran cantidad de código duplicado<sup>3</sup>. Por ejemplo, una interfase “generadora” se requiere para cada tipo dado que el tipo de retorno de **next()** debe ser diferente en cada caso:

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator {
    Object next();
} //:~
//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator {
    boolean next();
} //:~
//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator {
    byte next();
} //:~
//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator {
    char next();
} //:~
//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
public interface ShortGenerator {
    short next();
} //:~
//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
```

---

<sup>3</sup> El programador C++ notará cuanto código puede derrumbarse con el uso de argumentos por defecto y plantillas. El programador Python verá que la totalidad de la librería será en su mayoría innecesaria en este lenguaje.

```

public interface IntGenerator {
    int next();
} //:~
//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator {
    long next();
} //:~
//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator {
    float next();
} //:~
//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator {
    double next();
} //:~

```

**Arrays2** contiene una variedad de funciones **print()**, sobrecargadas para cada tipo. Se puede simplemente imprimir un arreglo, se puede agregar un mensaje antes que el arreglo sea impreso, o se puede imprimir un rango de elementos dentro del arreglo. El código de **print()** se explica solo:

```

//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide
// additional useful functionality when working
// with arrays. Allows any array to be printed,
// and to be filled via a user-defined
// "generator" object.
package com.bruceeckel.util;
import java.util.*;
public class Arrays2 {
    private static void
    start(int from, int to, int length) {
        if(from != 0 || to != length)
            System.out.print("[ " + from + ":" + to + " ] ");
        System.out.print("(");
    }
    private static void end() {
        System.out.println(")");
    }
    public static void print(Object[] a) {
        print(a, 0, a.length);
    }
    public static void
    print(String msg, Object[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }
    public static void
    print(Object[] a, int from, int to){
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
        }
    }
}

```

```

        if(i < to -1)
            System.out.print(", ");
    }
    end();
}
public static void print(boolean[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, boolean[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(boolean[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to -1)
            System.out.print(", ");
    }
    end();
}
public static void print(byte[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, byte[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(byte[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to -1)
            System.out.print(", ");
    }
    end();
}
public static void print(char[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, char[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(char[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to -1)

```

```

        System.out.print(", ");
    }
    end();
}
public static void print(short[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, short[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(short[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}
public static void print(int[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, int[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(int[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}
public static void print(long[] a) {
    print(a, 0, a.length);
}
public static void
print(String msg, long[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}
public static void
print(long[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}

```

```

        }
        end();
    }
    public static void print(float[] a) {
        print(a, 0, a.length);
    }
    public static void
    print(String msg, float[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }
    public static void
    print(float[] a, int from, int to) {
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to - 1)
                System.out.print(", ");
        }
        end();
    }
    public static void print(double[] a) {
        print(a, 0, a.length);
    }
    public static void
    print(String msg, double[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }
    public static void
    print(double[] a, int from, int to){
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to - 1)
                System.out.print(", ");
        }
        end();
    }
    // Fill an array using a generator:
    public static void
    fill(Object[] a, Generator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void
    fill(Object[] a, int from, int to,
    Generator gen){
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
    public static void
    fill(boolean[] a, BooleanGenerator gen) {
        fill(a, 0, a.length, gen);
    }
    public static void

```

```

fill(boolean[] a, int from, int to,
BooleanGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(byte[] a, ByteGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(byte[] a, int from, int to,
ByteGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(char[] a, CharGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(char[] a, int from, int to,
CharGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(short[] a, ShortGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(short[] a, int from, int to,
ShortGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(int[] a, IntGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(int[] a, int from, int to,
IntGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(long[] a, LongGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(long[] a, int from, int to,
LongGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

```

```

public static void
fill(float[] a, FloatGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(float[] a, int from, int to,
FloatGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
public static void
fill(double[] a, DoubleGenerator gen) {
    fill(a, 0, a.length, gen);
}
public static void
fill(double[] a, int from, int to,
DoubleGenerator gen){
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
private static Random r = new Random();
public static class RandBooleanGenerator
implements BooleanGenerator {
    public boolean next() {
        return r.nextBoolean();
    }
}
public static class RandByteGenerator
implements ByteGenerator {
    public byte next() {
        return (byte)r.nextInt();
    }
}
static String ssource =
"ABCDEFGHIJKLMNPQRSTUVWXYZ" +
"abcdefghijklmnopqrstuvwxyz";
static char[] src = ssource.toCharArray();
public static class RandCharGenerator
implements CharGenerator {
    public char next() {
        int pos = Math.abs(r.nextInt());
        return src[pos % src.length];
    }
}
public static class RandStringGenerator
implements Generator {
    private int len;
    private RandCharGenerator cg =
    new RandCharGenerator();
    public RandStringGenerator(int length) {
        len = length;
    }
    public Object next() {
        char[] buf = new char[len];
        for(int i = 0; i < len; i++)

```

```

        buf[i] = cg.next();
        return new String(buf);
    }
}
public static class RandShortGenerator
implements ShortGenerator {
    public short next() {
        return (short)r.nextInt();
    }
}
public static class RandIntGenerator
implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) {
        mod = modulo;
    }
    public int next() {
        return r.nextInt() % mod;
    }
}
public static class RandLongGenerator
implements LongGenerator {
    public long next() { return r.nextLong(); }
}
public static class RandFloatGenerator
implements FloatGenerator {
    public float next() { return r.nextFloat(); }
}
public static class RandDoubleGenerator
implements DoubleGenerator {
    public double next() {return r.nextDouble();}
}
} //:~

```

Para llenar un arreglo de elementos utilizando un generador, el método **fill()** toma una referencia a una interfase generadora apropiada, que tiene un método **next()** que de alguna manera producirá un objeto del tipo adecuado (dependiendo de como la interfase esté implementada). El método **fill()** simplemente llama al método **next()** hasta que el rango deseado ha sido llenado. Ahora se puede crear cualquier generador implementando la interfase apropiada, y utilizar su generador con **fill()**.

Los generadores de datos aleatorios son útiles para pruebas, así es que un grupo de clases internas es creada para implementar todas las interfaces generadoras primitivas, de la misma forma que un generador **String** para representar **Object**. Se puede ver que **RandStringGenerator** utiliza **RandCharGenerator** para llenar un arreglo de caracteres, que es colocado dentro de un **String**. El tamaño del arreglo esta determinado por el argumento del constructor.

Para generar números que no sean muy grandes, por defecto **RandIntGenerator** a un módulo de 10,000, pero el constructor sobrecargado permite elegir un valor más pequeño.

He aquí un programa para verificar la librería, y para demostrar como es utilizada:

```
//: c09:TestArrays2.java
// Test and demonstrate Arrays2 utilities
import com.bruceekel.util.*;
public class TestArrays2 {
    public static void main(String[] args) {
        int size = 6;
        // Or get the size from the command line:
        if(args.length != 0)
            size = Integer.parseInt(args[0]);
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays2.fill(a1,
                    new Arrays2.RandBooleanGenerator());
        Arrays2.print(a1);
        Arrays2.print("a1 = ", a1);
        Arrays2.print(a1, size/3, size/3 + size/3);
        Arrays2.fill(a2,
                    new Arrays2.RandByteGenerator());
        Arrays2.print(a2);
        Arrays2.print("a2 = ", a2);
        Arrays2.print(a2, size/3, size/3 + size/3);
        Arrays2.fill(a3,
                    new Arrays2.RandCharGenerator());
        Arrays2.print(a3);
        Arrays2.print("a3 = ", a3);
        Arrays2.print(a3, size/3, size/3 + size/3);
        Arrays2.fill(a4,
                    new Arrays2.RandShortGenerator());
        Arrays2.print(a4);
        Arrays2.print("a4 = ", a4);
        Arrays2.print(a4, size/3, size/3 + size/3);
        Arrays2.fill(a5,
                    new Arrays2.RandIntGenerator());
        Arrays2.print(a5);
        Arrays2.print("a5 = ", a5);
        Arrays2.print(a5, size/3, size/3 + size/3);
        Arrays2.fill(a6,
                    new Arrays2.RandLongGenerator());
        Arrays2.print(a6);
        Arrays2.print("a6 = ", a6);
        Arrays2.print(a6, size/3, size/3 + size/3);
```

```

        Arrays2.fill(a7,
            new Arrays2.RandFloatGenerator());
        Arrays2.print(a7);
        Arrays2.print("a7 = ", a7);
        Arrays2.print(a7, size/3, size/3 + size/3);
        Arrays2.fill(a8,
            new Arrays2.RandDoubleGenerator());
        Arrays2.print(a8);
        Arrays2.print("a8 = ", a8);
        Arrays2.print(a8, size/3, size/3 + size/3);
        Arrays2.fill(a9,
            new Arrays2.RandStringGenerator(7));
        Arrays2.print(a9);
        Arrays2.print("a9 = ", a9);
        Arrays2.print(a9, size/3, size/3 + size/3);
    }
} //:~

```

El parámetro **size** tiene un valor por defecto, pero se puede también colocar en la línea de comandos.

## Llenando un arreglo

La librería estándar **Arrays** también tiene un método **fill()**, pero este es más bien trivial -solo duplica un único valor dentro de cada ubicación, o en el caso de los objetos, copia la misma referencia dentro de cada ubicación. Utilizando **Array2.print()**, el método **Arrays.fill()** puede ser fácilmente demostrado:

```

//: c09:FillingArrays.java
// Using Arrays.fill()
import com.bruceeeckel.util.*;
import java.util.*;
public class FillingArrays {
    public static void main(String[] args) {
        int size = 6;
        // Or get the size from the command line:
        if(args.length != 0)
            size = Integer.parseInt(args[0]);
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays.fill(a1, true);
        Arrays2.print("a1 = ", a1);
        Arrays.fill(a2, (byte)11);
        Arrays2.print("a2 = ", a2);
        Arrays.fill(a3, 'x');
        Arrays2.print("a3 = ", a3);
    }
}

```

```

        Arrays.fill(a4, (short)17);
        Arrays2.print("a4 = ", a4);
        Arrays.fill(a5, 19);
        Arrays2.print("a5 = ", a5);
        Arrays.fill(a6, 23);
        Arrays2.print("a6 = ", a6);
        Arrays.fill(a7, 29);
        Arrays2.print("a7 = ", a7);
        Arrays.fill(a8, 47);
        Arrays2.print("a8 = ", a8);
        Arrays.fill(a9, "Hello");
        Arrays2.print("a9 = ", a9);
        // Manipulating ranges:
        Arrays.fill(a9, 3, 5, "World");
        Arrays2.print("a9 = ", a9);
    }
} //:~

```

Se puede también llenar el arreglo entero, o -como se muestra en las dos últimas instrucciones- un rango de elementos. Pero dado que se puede proporcionar un solo valor a utilizar para llenar utilizando **Arrays.fill()**, el método **Arrays2.fill()** produce resultados mucho mas interesantes.

## Copiando un arreglo

La librería estándar de Java proporciona un método estático, **System.arraycopy()**, que puede hacer mucho mas rápido la copia de un arreglo que si se utiliza un bucle for para realizar la copia a mano. **System.arraycopy()** se sobrecarga para manejar todos los tipos. Aquí hay un ejemplo para manipular arreglos de enteros.

```

//: c09:CopyingArrays.java
// Using System.arraycopy()
import com.bruceekel.util.*;
import java.util.*;
public class CopyingArrays {
    public static void main(String[] args) {
        int[] i = new int[25];
        int[] j = new int[25];
        Arrays.fill(i, 47);
        Arrays.fill(j, 99);
        Arrays2.print("i = ", i);
        Arrays2.print("j = ", j);
        System.arraycopy(i, 0, j, 0, i.length);
        Arrays2.print("j = ", j);
        int[] k = new int[10];
        Arrays.fill(k, 103);
        System.arraycopy(i, 0, k, 0, k.length);
        Arrays2.print("k = ", k);
        Arrays.fill(k, 103);
        System.arraycopy(k, 0, i, 0, k.length);
        Arrays2.print("i = ", i);
        // Objects:
    }
}

```

```

        Integer[] u = new Integer[10];
        Integer[] v = new Integer[5];
        Arrays.fill(u, new Integer(47));
        Arrays.fill(v, new Integer(99));
        Arrays2.print("u = ", u);
        Arrays2.print("v = ", v);
        System.arraycopy(v, 0,
        u, u.length/2, v.length);
        Arrays2.print("u = ", u);
    }
} //:~

```

Los argumentos de **arraycopy()** son el arreglo fuente, el desplazamiento dentro del arreglo fuente de donde se comenzará a copiar, el arreglo destino, el desplazamiento dentro del arreglo destino donde la copia comienza, y el número de elementos a copiar. Naturalmente, cualquier violación de los límites del arreglo producirá una excepción.

El ejemplo muestra que los arreglos de primitivas y los arreglos de objetos pueden ser copiados. Sin embargo, si se copia arreglos de objetos solo las referencias son copiadas -no hay duplicación de los objetos. Esto es llamado una *copia superficial*(vea el Apéndice A).

## Comparando arreglos

Los arreglos proporcionan la sobrecarga del método **equals()** para comparar la igualdad de arreglos enteros. Nuevamente, esta es sobrecargada para todas las primitivas, y para **Object**. Para ser igual, los arreglos deben tener el mismo número de elementos y cada elemento debe ser equivalente a cada elemento correspondiente en el otro arreglo, utilizando **equals()** para cada elemento (Para primitivas, el **equal()** de la clase envolvente de la primitiva es utilizada; por ejemplo, **Integer.equals()** para **int**). He aquí un ejemplo:

```

//: c09:ComparingArrays.java
// Using Arrays.equals()
import java.util.*;
public class ComparingArrays {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        System.out.println(Arrays.equals(a1, a2));
        a2[3] = 11;
        System.out.println(Arrays.equals(a1, a2));
        String[] s1 = new String[5];
        Arrays.fill(s1, "Hi");
        String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
        System.out.println(Arrays.equals(s1, s2));
    }
} //:~

```

Originalmente, **a1** y **a2** son exactamente iguales, así es que la salida es “verdadera”, pero entonces uno de los elementos es cambiado así es que la segunda línea para la salida es “falso”. En el último caso, todos los elementos de **s1** apuntan a el mismo objeto, pero **s2** tiene cinco objetos que son uno mismo. Sin embargo, la igualdad esta basada en el contenido (mediante **Object.equals()**) así es que el resultado es “verdadero”.

## Comparaciones de elementos de un arreglo

Una de las características ausentes en las librerías Java 1.0 y 1.1 son las operaciones algorítmicas -incluso la simple ordenación. Esto fue una situación confuso para alguien que esperaba una librería estándar adecuada. Afortunadamente, Java 2 soluciona la situación, al menos para el problema del ordenamiento.

Un problema con escribir código genérico para ordenar es que se deben realizar comparaciones basadas en el tipo actual de objeto. Claro, una estrategia es escribir un método diferente de ordenamiento para cada tipo diferente, pero se debe ser capaz de reconocer que esto no produce un código que sea fácil de reutilizar para los nuevos tipos.

Una meta primaria en el diseño de programación es “separar cosas que cambian de cosas que se quedan iguales”, y aquí, el código que se queda igual es el algoritmo de ordenación general, y las cosas que cambian de un uso al otro es la forma en que los objetos son comparados. Así es que en lugar de escribir todo el código de comparación en muchas rutinas de formas diferentes, la técnica de *llamada de retorno (callback)*es utilizada. Con una llamada de retorno, la parte del código que varía de caso a caso es encapsulada dentro de su propia clase, y la parte del código que siempre es la misma siempre realizara una llamada de retorno al código que cambia.

En Java 2, hay dos formas de proporcionar funcionalidad de comparación. La primera es con el *método de comparación natural*que es concedido a una clases implementando la interfase **java.lang.Comparable**. Esta es una interfase muy simple con un solo método, **compareTo()**. Este método toma otro objeto **Object** como argumento, y produce un valor negativo si el argumento es menor que el objeto actual, cero si el argumento es igual, y un valor positivo si el argumento es mayor que el objeto actual.

He aquí una clase que implementa **Comparable** y demuestra la forma de comparar que utiliza el método de librería estándar de Java **Array.sort()**:

```
//: c09:CompType.java
// Implementing Comparable in a class.
import com.bruceeeckel.util.*;
import java.util.*;
```

```

public class CompType implements Comparable {
    int i;
    int j;
    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }
    public String toString() {
        return "[i = " + i + ", j = " + j + "]";
    }
    public int compareTo(Object rv) {
        int rvi = ((CompType)rv).i;
        return (i < rvi ? -1 : (i == rvi ? 0 : 1));
    }
    private static Random r = new Random();
    private static int randInt() {
        return Math.abs(r.nextInt()) % 100;
    }
    public static Generator generator() {
        return new Generator() {
            public Object next() {
                return new CompType(randInt(),randInt());
            }
        };
    }
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, generator());
        Arrays2.print("before sorting, a = ", a);
        Arrays.sort(a);
        Arrays2.print("after sorting, a = ", a);
    }
} //:~

```

Cuando se define la función de comparación, se es responsable por decidir que significa comparar uno de sus objetos con otros. Aquí, solo los valores **i** son utilizados en la comparación, y los valores **j** son ignorados.

El método **randInt()** estático produce valores positivos entre cero y 100, y el método **generator()** produce un objeto que implementa la interfase **Generator**, creando una clase interna anónima (vea el capítulo 8). Esto crea objetos **CompType** inicializándolos con valores aleatorios. En el **main()**, el generador es utilizado para llenar un arreglo de **CompType**, que luego es ordenado. Si **Comparable** no fue implementado, se obtiene un mensaje de error en tiempo de compilación cuando se intenta llamar a **sort()**.

Ahora supongamos que alguien entrega una clase que no implementa **Comparable**, o se entrega esta clase *que implementa Comparable*, pero se decide que no gusta la forma en que trabaja y preferiblemente se tenga una función de comparación diferente para ese tipo. Para hacer esto, se debe utilizar una segunda estrategia para comparar objetos, creando una clase separada que implemente una interfase llamada **Comparator**. Esto tiene dos métodos, **compare()** y **equals()**. Sin embargo, no se tiene que implementar

**equals()** a no ser por necesidades especiales de rendimiento, dado que en cualquier momento se puede crear una clase que herede implícitamente de **Object**, que tiene un **equals()**. Así es que se puede simplemente utilizar el **equals()** por defecto de **Objects** y satisfacer el contrato impuesto por la interfase.

La clase **Collections** (que veremos mas adelante) contiene un solo **Comparator** que invierte el ordenamiento natural. Esto puede ser fácilmente aplicado a el **CompType**:

```
//: c09:Reverse.java
// The Collections.reverseOrder() Comparator.
import com.bruceekel.util.*;
import java.util.*;
public class Reverse {
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, CompType.generator());
        Arrays2.print("before sorting, a = ", a);
        Arrays.sort(a, Collections.reverseOrder());
        Arrays2.print("after sorting, a = ", a);
    }
} ///:~
```

La llamada a **Collections.reverseOrder()** produce la referencia a el **Comparator**.

Como segundo ejemplo, el siguiente **Comparator** compara objetos **CompType** basados en sus valores **j** en lugar de sus valores **i**:

```
//: c09:ComparatorTest.java
// Implementing a Comparator for a class.
import com.bruceekel.util.*;
import java.util.*;
class CompTypeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        int j1 = ((CompType)o1).j;
        int j2 = ((CompType)o2).j;
        return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
    }
}
public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, CompType.generator());
        Arrays2.print("before sorting, a = ", a);
        Arrays.sort(a, new CompTypeComparator());
        Arrays2.print("after sorting, a = ", a);
    }
} ///:~
```

El método **compare()** debe retornar un entero negativo, cero o un entero positivo si el primer argumento es menor, igual o mayor que el segundo respectivamente.

## Ordenando un arreglo

Con los métodos de ordenamiento incluidos, se puede ordenar cualquier arreglo de primitivas, y cualquier arreglo de objetos que implemente **Comparable** o tenga un **Comparator** asociativo. Esto cubre un gran agujero en las librerías de Java lo crea o no, no hay soporte en Java 1.0 o 1.1 para ordenar cadenas! Aquí hay un ejemplo que genera objetos **String** de forma aleatoria y luego los ordena:

```
//: c09:StringSorting.java
// Sorting an array of Strings.
import com.bruceeeckel.util.*;
import java.util.*;
public class StringSorting {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa,
            new Arrays2.RandStringGenerator(5));
        Arrays2.print("Before sorting: ", sa);
        Arrays.sort(sa);
        Arrays2.print("After sorting: ", sa);
    }
} ///:~
```

Algo que se notará acerca de la salida en el algoritmo de ordenamiento de cadenas es la lexicografía, ya que colocas todas las palabras comenzando con letras mayúsculas primero, seguido por todas las palabras que comienzan con minúscula. Las guías telefónicas típicamente son ordenadas de esta forma). Se puede querer agrupar las letras juntas sin importar si son mayúsculas o minúsculas, y se puede hacer esto definiendo una clase **Comparator**, de esta manera sobrescribir el comportamiento por defecto de **Comparable** para las cadenas. Para reutilizar, esto debe ser agregado al paquete "util":

```
//: com:bruceeeckel:util:AlphabeticComparator.java
// Keeping upper and lowercase letters together.
package com.bruceeeckel.util;
import java.util.*;
public class AlphabeticComparator
    implements Comparator{
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.toLowerCase().compareTo(
            s2.toLowerCase());
    }
} ///:~
```

Cada cadena es convertida a minúsculas antes de la comparación. El método **compareTo()** original para cadenas proporciona la funcionalidad deseada.

He aquí una prueba utilizando **AlphabeticComparator()**:

```
//: c09:AlphabeticSorting.java
```

```

// Keeping upper and lowercase letters together.
import com.bruceekel.util.*;
import java.util.*;
public class AlphabeticSorting {
public static void main(String[] args) {
String[] sa = new String[30];
Arrays2.fill(sa,
new Arrays2.RandStringGenerator(5));
Arrays2.print("Before sorting: ", sa);
Arrays.sort(sa, new AlphabeticComparator());
Arrays2.print("After sorting: ", sa);
}
} //:~

```

El algoritmo de ordenamiento que es utilizado en la librería estándar de Java esta diseñado para ser óptimo para un tipo particular que se esté ordenando -un Quicksort para primitivas, y un ordenamiento de mezcla estable para objetos. Así es que no necesita perder tiempo preocupándose por rendimiento a no ser que su herramienta de perfilado indique que el proceso de ordenamiento es un cuello de botella.

## Búsqueda en un arreglo ordenado

Una vez que un arreglo es ordenado, se puede realizar una búsqueda rápida de un ítem en particular utilizando **Arrays.binarySearch()**. Sin embargo, es muy importante que no intente utilizar **binarySearch()** en un arreglo sin ordenar; los resultados pueden ser impredecibles. Los siguientes ejemplos utilizan un **RandIntGenerator** para llenar un arreglo, luego produce valores para buscar:

```

//: c09:ArraySearching.java
// Using Arrays.binarySearch().
import com.bruceekel.util.*;
import java.util.*;
public class ArraySearching {
    public static void main(String[] args) {
        int[] a = new int[100];
        Arrays2.RandIntGenerator gen =
            new Arrays2.RandIntGenerator(1000);
        Arrays2.fill(a, gen);
        Arrays.sort(a);
        Arrays2.print("Sorted array: ", a);
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);
            if(location >= 0) {
                System.out.println("Location of " + r +
                    " is " + location + ", a[" +
                    location + "] = " + a[location]);
                break; // Out of while loop
            }
        }
    }
}

```

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

En el bucle **while**, los valores aleatorios son generados como ítems de búsqueda, hasta que uno de ellos es encontrado.

**Arrays.binarySearch()** produce un valor mayor o igual a cero si el ítem de búsqueda es encontrado. De otra forma, produce un valor negativo representando el lugar donde el elemento debe ser insertado si se quiere mantener un arreglo ordenado a mano. El valor producido es

-(punto de inserción) - 1

El punto de inserción es el índice a el primer elemento mayor que la clave, o **a.size()** si todos los elementos son menores que la clave especificada.

Si el arreglo contiene elementos duplicados, no hay garantía que uno sea encontrado. El algoritmo no esta diseñado para soportar elementos duplicados, tanto como tolerarlos. Si necesita una lista ordenada de elementos no duplicados, sin embargo, utilice un **TreeSet**, que será introducido mas tarde en este capítulo. Este cuida los detalles automáticamente. Solo en casos de cuellos de botella de rendimiento debe remplazar el **TreeSet** con un arreglo mantenido a mano.

Si ordena un arreglo utilizando un **Comparator** (arreglos de primitivas no permiten ordenar con un **Comparator**), de debe incluir el mismo **Comparator** cuando se realice una **binarySearch()** (utilizando la versión sobrecargada de la función que proporciona). Por ejemplo, el programa **AlphabeticSorting.java** puede ser modificado para realizar una búsqueda:

```
//: c09:AlphabeticSearch.java
// Searching with a Comparator.
import com.bruceeckel.util.*;
import java.util.*;
public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa,
            new Arrays2.RandStringGenerator(5));
        AlphabeticComparator comp =
            new AlphabeticComparator();
        Arrays.sort(sa, comp);
        int index =
            Arrays.binarySearch(sa, sa[10], comp);
        System.out.println("Index = " + index);
    }
} //:/~
```

El **Comparator** debe ser pasado a la **binarySearch()** sobrecargada como el tercer argumento. En el ejemplo mas arriba, el éxito es garantido porque el ítem de búsqueda es extraído del arreglo a si mismo.

## Resumen de arreglos

Para resumir lo que se ha visto hasta ahora, la primera y mas eficiente elección para almacenar un grupo de objetos debería ser un arreglo, y si se está forzado a esta elección si se quiere almacenar un grupo de primitivas. En el resto de este capítulo veremos un caso mas genera, donde no se conoce en el momento en que se esta escribiendo en programa cuantos objetos se van a necesitar, o si se necesita una forma mas sofisticada de almacenar sus objetos. Java proporciona una librería de *clases contenedoras* para solucionar este problema, los tipos básicos son **List**, **Set** y **Map**. Se puede solucionar un sorprendente número de problemas utilizando estas herramientas.

Entre sus otras características -**Set** por ejemplo, almacena solo un objeto de cada valor, y **Maps** un *arreglo asociativo* que deja que se asocie cada objeto con otro objeto -Las clases contenedoras de Java automáticamente cambiarian de tamaño a ellas mismas. Así es que, a diferencia de los arreglos se puede colocar cualquier cantidad de objetos sin preocuparse acerca de que tan grande hacer el contenedor cuando de esta escribiendo el programa.

## Introducción a los contenedores

Para mi, las clases contenedoras son una de las mas poderosas herramientas para programar en bruto porque incrementa significativamente el músculo de programación. Los contenedores de Java 2 representan un cuidadoso rediseño<sup>4</sup> de mas bien pobres mostrados en Java 1.0 y 1.1. Algunos de los rediseños hacen cosas herméticas y mas acertadas. También completan la funcionalidad de la librería de contenedores, proporcionando el comportamiento de listas enlazadas, colas y deques (colas con doble final, que se pronuncian "decks").

El diseño de una librería de contenedores es difícil (cierto en la mayoría de los problemas de diseño en librerías). En C++, las clases contenedoras cubren las bases con muchas clases diferentes. Esto fue mejor de lo que estaba disponible antes de las clases contenedoras de C++ (nada), pero no fueron bien trasladadas en Java. En el otro extremo, he visto librerías de contenedores que consisten en una sola clase, "contenedora", que actúa como una secuencia lineal y un arreglo asociativo al mismo tiempo. La librería de contenedores de Java 2 logra un balance: la total funcionalidad

---

<sup>4</sup> Por Joshua Bloch en Sun.

que se espera de una librería de contendores madura y la facilidad para aprender y utilizar de las clases contenedoras de C++ y otras librerías de contendores similares. El resultado puede verse como un poco estrañafalario en sus sitios.

La librería de contendores de Java 2 toma el tema de "almacenar los objetos" y lo divide en dos conceptos distintos:

1. **Colección**: un grupo de elementos individuales, a menudo con alguna regla aplicada. Una **List** debe almacenar los elementos en una secuencia particular y un **Set** no puede tener elementos duplicados (una bolsa, que no está implementada en la librería de contendores de Java -dado que la **List** proporciona suficiente de esa funcionalidad- no tiene ese tipo de reglas).
2. **Map**: un grupo de pares de objetos del tipo clave valor. A primera vista, esto puede parecer como que debe ser una **Collection** de pares, pero cuando se trata de implementar de esta forma el diseño se vuelve complicado, así que es más claro tener otro concepto. Por el otro lado, es conveniente ver partes de un **Map** creando una **Collection** para representar esa parte. De esta forma un **Map** puede retornar un **Set** de estas claves, una **Collection** de sus valores, o un **Set** de sus pares. Los **Map**, como los arreglos, pueden ser fácilmente expandidos a múltiples dimensiones sin agregar nuevos conceptos; simplemente se crea un **Map** cuyos valores sean **Maps** (y los valores de aquellos **Maps** pueden ser **Maps**, etc).

Veremos primero las características de los contendores, luego iremos por los detalles, y finalmente aprenderemos por qué hay versiones diferentes de algunos contendores, y cómo elegirlos.

## Desplegando contendores

A diferencia de los arreglos, los contendores se despliegan bien sin ninguna ayuda. Aquí hay un ejemplo que también introduce a los tipos básicos de contendores:

```
public class PrintingContainers {  
    static Collection fill(Collection c) {  
        c.add("dog");  
        c.add("dog");  
        c.add("cat");  
        return c;  
    }  
    static Map fill(Map m) {  
        m.put("dog", "Bosco");  
        m.put("dog", "Spot");  
        m.put("cat", "Rags");  
        return m;  
    }  
}
```

```

    }
    public static void main(String[] args) {
        System.out.println(fill(new ArrayList()));
        System.out.println(fill(new HashSet()));
        System.out.println(fill(new HashMap()));
    }
} //:~
```

Como se ha mencionado antes, hay dos categorías básicas en las librerías de contenedores. La distinción esta basada en el número de ítems que son almacenados en cada ubicación del contenedor. La categoría **Collection** solo almacena un ítem en cada ubicación (El nombre confunde un poco dado que la librería entera de contenedores es a menudo llamada "colección"). Esto incluye la **List**, que almacena un grupo de ítems en una secuencia específica, y **Set**, que solo permite la suma de un ítem de cada tipo. La **ArrayList** es un tipo de **List**, y **HashSet** es un tipo de **Set**. Para agregar ítems a cualquier **Collection**, hay un método **add()**.

La **Map** almacena pares de claves y valores, mas bien como una pequeña base de datos. El programa mas arriba utiliza un estilo de **Map**, el **HashMap**. Si se tiene un **Map** que asocia estados con sus capitales y se quiere saber la capital de Ohio, se busca -mas o menos como se estuviera indexado dentro de un arreglo (**Maps** son también llamados *arreglos asociativos*). Para agregar elementos a un **Map** hay un método **put()** que toma una clave y un valor como argumentos. El ejemplo anterior solo muestra como se agregan elementos y no muestra los elementos almacenados luego que se agregaron. Esto será mostrado mas tarde.

El método sobrecargado **fill()** rellena **Collections** y **Maps**, respectivamente. Si se mira la salida, se puede ver que el comportamiento por defecto cuando se despliega en pantalla (proporcionado por diversos métodos **toString()**) producen resultados bastante legibles, así es que no es necesario soporte adicional para el despliegue como con los arreglos.

```

[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```

Una **Collection** es desplegada rodeada por paréntesis cuadrados, con cada elemento separado con una coma. Un **Map** esta rodeado por llaves, con cada clave y valor asociado con un signo de igual (claves en la izquierda, valores en la derecha).

Se puede inmediatamente ver el comportamiento básico de los diferentes contenedores. La **List** almacena los objetos exactamente como fueron entrados, sin ningún ordenamiento o edición. El **Set**, sin embargo, solo acepta uno de cada uno de los objetos y usa su propio método interno para ordenarlo (en general, se esta preocupado solo por si alguno es o no es miembro del **Set**, no por el orden en el cual aparecen -para eso se debe utilizar una **List**). Y el **Map** solo acepta uno de cada tipo de ítem, basado en

la clave, y también tiene su propio ordenamiento interno y no importa el orden en el cual se entran los ítems.

## Cargando contenedores

A pesar de que el problema de desplegar contenedores ya está solucionado, el cargar los contenedores padece de la misma deficiencia que **java.util.Arrays**. Exactamente como los arreglos, hay una clase amiga llamada **Collections** que contiene métodos utilitarios estáticos incluyendo uno llamado **fill()**. Este **fill()** también duplica una sola referencia a objeto por todo el contenedor, y también solo trabaja para objetos **List** y no para **Sets** o **Maps**:

```
//: c09:FillingLists.java
// The Collections.fill() method.
import java.util.*;
public class FillingLists {
    public static void main(String[] args) {
        List list = new ArrayList();
        for(int i = 0; i < 10; i++)
            list.add("");
        Collections.fill(list, "Hello");
        System.out.println(list);
    }
} ///:~
```

Este método es aún menos útil por el hecho de que solo pueden remplazar elementos que ya se encuentran en la **List**, y no pueden agregar nuevos elementos.

Para ser capaz de crear ejemplos interesantes, he aquí una librería complementaria llamada **Collections2** (parte de **com.bruceeckel.util** por conveniencia) con un método **fill()** que utiliza un generador para agregar elementos, y permite especificar el número de elementos que se quiere agregar. La interfase generadora definida previamente trabajara para **Collections**, pero **Map** requiere su propia interfase generadora dado que un par de objetos (una clave y un valor) deben ser producidos por cada llamada a **next()**. Aquí tenemos la clase **Pair**:

```
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;
public class Pair {
    public Object key, value;
    Pair(Object k, Object v) {
        key = k;
        value = v;
    }
} ///:~
```

Además, tenemos la interfase generadora que produce el **Pair**:

```
//: com:bruceeckel:util:Collections2.java
// Para llenar cualquier tipo de contenedor
```

```

// utilizando un objeto generador.
package com.bruceeckel.util;
import java.util.*;
public class Collections2 {
    // Llenar un arreglo utilizando un generador:
    public static void
    fill(Collection c, Generator gen, int count) {
        for(int i = 0; i < count; i++)
            c.add(gen.next());
    }
    public static void
    fill(Map m, MapGenerator gen, int count) {
        for(int i = 0; i < count; i++) {
            Pair p = gen.next();
            m.put(p.key, p.value);
        }
    }
    public static class RandStringPairGenerator
        implements MapGenerator {
        private Arrays2.RandStringGenerator gen;
        public RandStringPairGenerator(int len) {
            gen = new Arrays2.RandStringGenerator(len);
        }
        public Pair next() {
            return new Pair(gen.next(), gen.next());
        }
    }
    // Objeto por defecto así es que no se tiene
    // que crear el propio:
    public static RandStringPairGenerator rsp =
        new RandStringPairGenerator(10);
    public static class StringPairGenerator
        implements MapGenerator {
        private int index = -1;
        private String[][] d;
        public StringPairGenerator(String[][] data) {
            d = data;
        }
        public Pair next() {
            // Fuerza a que el índice se pliegue:
            index = (index + 1) % d.length;
            return new Pair(d[index][0], d[index][1]);
        }
        public StringPairGenerator reset() {
            index = -1;
            return this;
        }
    }
    // Utilice un grupo de datos predefinido:
    public static StringPairGenerator geography =
        new StringPairGenerator(
        CountryCapitals.pairs);
    // Produce una secuencia de arreglos en 2D:
    public static class StringGenerator
        implements Generator {

```

```

private String[][] d;
private int position;
private int index = -1;
public
    StringGenerator(String[][] data, int pos) {
    d = data;
    position = pos;
}
public Object next() {
    // Fuerza a el índice a plegarse:
    index = (index + 1) % d.length;
    return d[index][position];
}
public StringGenerator reset() {
    index = -1;
    return this;
}
}
// Use un grupo de datos predefinido:
public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs,0);
public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs,1);
} //:~

```

Las dos versiones de **fill()** toman un argumento que determina en número de ítems para agregar a el contenedor. Además, hay dos generadores para el **Map**: **RandStringPairGenerator**, que crea cualquier cantidad de pares de cadenas con galimatías cuyo largo esta determinado por el argumento del constructor; y **StringPairGenerator**, que produce pares de **String** dando un arreglo **String** de dos dimensiones. El **StringGenerator** también toma un arreglo **String** de dos dimensiones pero genera un solo ítems en lugar del **Pairs**. Los objetos estáticos **rsp**, **geography**, **countries**, y **capitals** proporcionan generadores previamente creados, los últimos tres utilizando todos los países del mundo y sus capitales. Debe notarse que si se crean mas pares de los que están disponibles, los generadores comenzarán nuevamente desde el principio, y su se colocan los pares dentro del **Map**, los duplicados simplemente serán ignorados.

Aquí tenemos un grupo de datos predefinidos, que consisten en nombres de países y sus capitales. Este grupo esta en fuente pequeña para prevenir que ocupe espacio innecesario:

```

//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;
public class CountryCapitals {
    public static final String[][] pairs = {
        // Africa
        {"ALGERIA", "Algiers"}, {"ANGOLA", "Luanda"}, ,
        {"BENIN", "Porto-Novo"}, {"BOTSWANA", "Gaberone"}, ,
        {"BURKINA FASO", "Ouagadougou"}, {"BURUNDI", "Bujumbura"}, ,
        {"CAMEROON", "Yaounde"}, {"CAPE VERDE", "Praia"}, ,
        {"CENTRAL AFRICAN REPUBLIC", "Bangui"}, ,
    }
}

```

```

    {"CHAD", "N'djamena"}, {"COMOROS", "Moroni"},  

    {"CONGO", "Brazzaville"}, {"DJIBOUTI", "Djibouti"},  

    {"EGYPT", "Cairo"}, {"EQUATORIAL GUINEA", "Malabo"},  

    {"ERITREA", "Asmara"}, {"ETHIOPIA", "Addis Ababa"},  

    {"GABON", "Libreville"}, {"THE GAMBIA", "Banjul"},  

    {"GHANA", "Accra"}, {"GUINEA", "Conakry"},  

    {"GUINEA", "-"}, {"BISSAU", "Bissau"},  

    {"CETE D'IVOIR (IVORY COAST)", "Yamoussoukro"},  

    {"KENYA", "Nairobi"}, {"LESOTHO", "Maseru"},  

    {"LIBERIA", "Monrovia"}, {"LIBYA", "Tripoli"},  

    {"MADAGASCAR", "Antananarivo"}, {"MALAWI", "Lilongwe"},  

    {"MALI", "Bamako"}, {"MAURITANIA", "Nouakchott"},  

    {"MAURITIUS", "Port Louis"}, {"MOROCCO", "Rabat"},  

    {"MOZAMBIQUE", "Maputo"}, {"NAMIBIA", "Windhoek"},  

    {"NIGER", "Niamey"}, {"NIGERIA", "Abuja"},  

    {"RWANDA", "Kigali"}, {"SAO TOME E PRINCIPE", "Sao Tome"},  

    {"SENEGAL", "Dakar"}, {"SEYCHELLES", "Victoria"},  

    {"SIERRA LEONE", "Freetown"}, {"SOMALIA", "Mogadishu"},  

    {"SOUTH AFRICA", "Pretoria/Cape Town"}, {"SUDAN", "Khartoum"},  

    {"SWAZILAND", "Mbabane"}, {"TANZANIA", "Dodoma"},  

    {"TOGO", "Lome"}, {"TUNISIA", "Tunis"},  

    {"UGANDA", "Kampala"},  

    {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)", "Kinshasa"},  

    {"ZAMBIA", "Lusaka"}, {"ZIMBABWE", "Harare"},  

    // Asia  

    {"AFGHANISTAN", "Kabul"}, {"BAHRAIN", "Manama"},  

    {"BANGLADESH", "Dhaka"}, {"BHUTAN", "Thimphu"},  

    {"BRUNEI", "Bandar Seri Begawan"}, {"CAMBODIA", "Phnom Penh"},  

    {"CHINA", "Beijing"}, {"CYPRUS", "Nicosia"},  

    {"INDIA", "New Delhi"}, {"INDONESIA", "Jakarta"},  

    {"IRAN", "Tehran"}, {"IRAQ", "Baghdad"},  

    {"ISRAEL", "Jerusalem"}, {"JAPAN", "Tokyo"},  

    {"JORDAN", "Amman"}, {"KUWAIT", "Kuwait City"},  

    {"LAOS", "Vientiane"}, {"LEBANON", "Beirut"},  

    {"MALAYSIA", "Kuala Lumpur"}, {"THE MALDIVES", "Male"},  

    {"MONGOLIA", "Ulan Bator"}, {"MYANMAR (BURMA)", "Rangoon"},  

    {"NEPAL", "Katmandu"}, {"NORTH KOREA", "P'yongyang"},  

    {"OMAN", "Muscat"}, {"PAKISTAN", "Islamabad"},  

    {"PHILIPPINES", "Manila"}, {"QATAR", "Doha"},  

    {"SAUDI ARABIA", "Riyadh"}, {"SINGAPORE", "Singapore"},  

    {"SOUTH KOREA", "Seoul"}, {"SRI LANKA", "Colombo"},  

    {"SYRIA", "Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)", "Taipei"},  

    {"THAILAND", "Bangkok"}, {"TURKEY", "Ankara"},  

    {"UNITED ARAB EMIRATES", "Abu Dhabi"}, {"VIETNAM", "Hanoi"},  

    {"YEMEN", "Sana'a"},  

    // Australia and Oceania  

    {"AUSTRALIA", "Canberra"}, {"FIJI", "Suva"},  

    {"KIRIBATI", "Bairiki"},  

    {"MARSHALL ISLANDS", "Dalap-Uliga-Darrit"},  

    {"MICRONESIA", "Palikir"}, {"NAURU", "Yaren"},  

    {"NEW ZEALAND", "Wellington"}, {"PALAU", "Koror"},  

    {"PAPUA NEW GUINEA", "Port Moresby"},  

    {"SOLOMON ISLANDS", "Honaira"}, {"TONGA", "Nuku'alofa"},  

    {"TUVALU", "Fongafale"}, {"VANUATU", "< Port-Vila"},  

    {"WESTERN SAMOA", "Apia"},
```

```

// Eastern Europe and former USSR
{ "ARMENIA", "Yerevan" }, { "AZERBAIJAN", "Baku" },
{ "BELARUS (BYELORUSSIA)", "Minsk" }, { "GEORGIA", "Tbilisi" },
{ "KAZAKSTAN", "Almaty" }, { "KYRGYZSTAN", "Alma-Ata" },
{ "MOLDOVA", "Chisinau" }, { "RUSSIA", "Moscow" },
{ "TAJIKISTAN", "Dushanbe" }, { "TURKMENISTAN", "Ashkabad" },
{ "UKRAINE", "Kyiv" }, { "UZBEKISTAN", "Tashkent" },
// Europe
{ "ALBANIA", "Tirana" }, { "ANDORRA", "Andorra la Vella" },
{ "AUSTRIA", "Vienna" }, { "BELGIUM", "Brussels" },
{ "BOSNIA", "—" }, { "HERZEGOVINA", "Sarajevo" },
{ "CROATIA", "Zagreb" }, { "CZECH REPUBLIC", "Prague" },
{ "DENMARK", "Copenhagen" }, { "ESTONIA", "Tallinn" },
{ "FINLAND", "Helsinki" }, { "FRANCE", "Paris" },
{ "GERMANY", "Berlin" }, { "GREECE", "Athens" },
{ "HUNGARY", "Budapest" }, { "ICELAND", "Reykjavik" },
{ "IRELAND", "Dublin" }, { "ITALY", "Rome" },
{ "LATVIA", "Riga" }, { "LIECHTENSTEIN", "Vaduz" },
{ "LITHUANIA", "Vilnius" }, { "LUXEMBOURG", "Luxembourg" },
{ "MACEDONIA", "Skopje" }, { "MALTA", "Valletta" },
{ "MONACO", "Monaco" }, { "MONTENEGRO", "Podgorica" },
{ "THE NETHERLANDS", "Amsterdam" }, { "NORWAY", "Oslo" },
{ "POLAND", "Warsaw" }, { "PORTUGAL", "Lisbon" },
{ "ROMANIA", "Bucharest" }, { "SAN MARINO", "San Marino" },
{ "SERBIA", "Belgrade" }, { "SLOVAKIA", "Bratislava" },
{ "SLOVENIA", "Ljubljana" }, { "SPAIN", "Madrid" },
{ "SWEDEN", "Stockholm" }, { "SWITZERLAND", "Berne" },
{ "UNITED KINGDOM", "London" }, { "VATICAN CITY", "---" },
// North and Central America
{ "ANTIGUA AND BARBUDA", "Saint John's" }, { "BAHAMAS", "Nassau" },
{ "BARBADOS", "Bridgetown" }, { "BELIZE", "Belmopan" },
{ "CANADA", "Ottawa" }, { "COSTA RICA", "San Jose" },
{ "CUBA", "Havana" }, { "DOMINICA", "Roseau" },
{ "DOMINICAN REPUBLIC", "Santo Domingo" },
{ "EL SALVADOR", "San Salvador" }, { "GRENADA", "Saint George's" },
{ "GUATEMALA", "Guatemala City" }, { "HAITI", "Port-au-Prince" },
{ "HONDURAS", "Tegucigalpa" }, { "JAMAICA", "Kingston" },
{ "MEXICO", "Mexico City" }, { "NICARAGUA", "Managua" },
{ "PANAMA", "Panama City" }, { "ST. KITTS", "—" },
{ "NEVIS", "Basseterre" }, { "ST. LUCIA", "Castries" },
{ "ST. VINCENT AND THE GRENADINES", "Kingstown" },
{ "UNITED STATES OF AMERICA", "Washington, D.C." },
// South America
{ "ARGENTINA", "Buenos Aires" },
{ "BOLIVIA", "Sucre (legal)/La Paz(administrative)" },
{ "BRAZIL", "Brasilia" }, { "CHILE", "Santiago" },
{ "COLOMBIA", "Bogota" }, { "ECUADOR", "Quito" },
{ "GUYANA", "Georgetown" }, { "PARAGUAY", "Asuncion" },
{ "PERU", "Lima" }, { "SURINAME", "Paramaribo" },
{ "TRINIDAD AND TOBAGO", "Port of Spain" },
{ "URUGUAY", "Montevideo" }, { "VENEZUELA", "Caracas" },
};

} ///:~

```

Esto es simplemente un arreglo de dos dimensiones de datos **String**<sup>5</sup>. Hay una simple prueba utilizando el método **fill()** y generadores:

```
//: c09:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;
public class FillTest {
    static Generator sg =
        new Arrays2.RandStringGenerator(7);
    public static void main(String[] args) {
        List list = new ArrayList();
        Collections2.fill(list, sg, 25);
        System.out.println(list + "\n");
        List list2 = new ArrayList();
        Collections2.fill(list2,
            Collections2.capitals, 25);
        System.out.println(list2 + "\n");
        Set set = new HashSet();
        Collections2.fill(set, sg, 25);
        System.out.println(set + "\n");
        Map m = new HashMap();
        Collections2.fill(m, Collections2.rsp, 25);
        System.out.println(m + "\n");
        Map m2 = new HashMap();
        Collections2.fill(m2,
            Collections2.geography, 25);
        System.out.println(m2);
    }
} ///:~
```

Con estas herramientas se puede fácilmente probar los contenedores llenándolos con datos interesantes.

## Desventaja de los contenedores: tipos desconocidos

La “desventajas” de utilizar los contenedores de Java es que se pierde la información de tipo cuando se coloca un objeto en un contenedor. Esto sucede porque el programador de la clase contenedor no tiene idea de qué tipo específicamente se va a colocar en el contenedor y haciendo que el contenedor almacene solo un tipo impide que se convierta en una herramienta de propósito general. Así es que en lugar de eso, el contenedor

---

<sup>5</sup> Este dato fue encontrado en la Internet, luego procesado para crear un programa Python (vea [www.Python.org](http://www.Python.org)).

almacena referencias de cualquier tipo (Claro, esto no incluye primitivas, dado que no son heredadas de nada). Esto es una gran solución, excepto porque:

1. Dado que la información de tipo es desechada cuando se coloca una referencia a un objeto en un contenedor, no hay restricciones del tipo de objeto que se va a colocar en el contenedor, aún si se piensa almacenar, digamos, gatos. Cualquiera podría simplemente colocar perros en el contenedor.
2. Dado que la información de tipo se pierde, la única cosa que el contenedor conoce que almacena es una referencia a un objetos. Se debe realizar una conversión para corregir el tipo antes de utilizarlo.

El lado bueno de esto es que Java no deja que se *desaprovechelos* objetos que se coloca en un contenedor. Si coloca un perro en un contenedor de gatos y luego intenta tratar todo en el contenedor como gatos, se obtendrá una excepción en tiempo de ejecución cuando se intente sacar la referencia del contenedor y convertirla a gato.

Aquí hay un ejemplo utilizando el típico caballo de tiro de los contenedores, **ArrayList**. Para las personas que comienzan, se puede pensar un **ArrayList** como “un arreglo que automáticamente se expande a si mismo”. Utilizando un **ArrayList** es directo: crear uno, colocar objetos utilizando **add()**, y mas tarde sacarlos con **get()** utilizando un índice -exactamente igual que con un arreglo pero sin los paréntesis cuadrados<sup>6</sup>.

**ArrayList** también tiene un método **size()** para saber cuantos elementos fueron agregados para de esta forma no salirse del arreglo causando una excepción.

Primero, las clases **Cat** y **Dog** son creadas:

```
//: c09:Cat.java
public class Cat {
    private int catNumber;
    Cat(int i) { catNumber = i; }
    void print() {
        System.out.println("Cat #" + catNumber);
    }
} ///:~

//: c09:Dog.java
public class Dog {
    private int dogNumber;
    Dog(int i) { dogNumber = i; }
    void print() {
        System.out.println("Dog #" + dogNumber);
    }
}
```

---

<sup>6</sup> Este es el lugar donde la sobrecarga de operadores sería bueno.

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

**Cats** y **Dogs** son colocados dentro del contenedor y luego extraídos:

```
//: c09:CatsAndDogs.java
// Simple container example.
import java.util.*;
public class CatsAndDogs {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        // Not a problem to add a dog to cats:
        cats.add(new Dog(7));
        for(int i = 0; i < cats.size(); i++)
            ((Cat)cats.get(i)).print();
        // Dog is detected only at run-time
    }
} //:/~
```

Las clases **Cat** y **Dog** son distintas -no tienen nada en común excepto que son **Objects** (Si no se indica explícitamente de que clase esta heredando, se heredará automáticamente de **Object**). Dado que **ArrayList** almacena **Objects**, no solo se pueden colocar objetos **Cat** dentro de este contenedor utilizando el método **add()** de **ArrayList**, también se pueden agregar objetos **Dog** sin quejas en tiempo de compilación o en tiempo de ejecución. Cuando extraiga lo que se piensa que es un objeto **Cat** utilizando el método **get()**, se traerá la referencia a un objeto que se debe convertir a **Cat**. Luego se necesita encerrar la expresión entera entre paréntesis para forzar la evaluación de esta conversión cuando llamamos el método **print()** para **Cat**, de otra forma se obtendrá un error de sintaxis. Entonces, en tiempo de ejecución, cuando se intente convertir el objeto **Dog** en **Cat** se obtendrá una excepción.

Esto es mas que una molestia. Es algo que puede crear dificultades para encontrar errores de programación. Si una parte (o varias partes) de un programa insertan objetos en un contenedor, y se descubre solo en una parte alejada del programa una excepción de que un objeto equivocado fue colocado en el contenedor, entonces se debe encontrar donde la inserción sucedió. Por otro lado, es conveniente comenzar con algunas clases contenedoras estandarizadas para programar, a pesar de la carencia y la dificultad.

## Algunas veces funciona de todas formas

Dejamos de lado que en algunos casos las cosas parecen trabajar correctamente sin conversiones inversas del tipo original. Un caso es bastante especial: la clase **String** algo de ayuda extra del compilador para

hacer que trabaje como una seda. Donde quiera que el compilador espere un objeto del tipo **String** y no tenga uno, automáticamente llamara el método **toString()** que esta definido en **Object** y puede ser sobrecargado por cualquier clase Java. Este método produce el objeto **String** deseado, que luego es utilizado donde quiera que se quiera.

De esta forma, todo lo que se necesita para hacer que se imprima objetos de su clase es sobrecargar el método **toString()**, como se muestra en el siguiente ejemplo:

```
//: c09:Mouse.java
// Overriding toString().
public class Mouse {
    private int mouseNumber;
    Mouse(int i) { mouseNumber = i; }
    // Override Object.toString():
    public String toString() {
        return "This is Mouse #" + mouseNumber;
    }
    public int getNumber() {
        return mouseNumber;
    }
} ///:~

//: c09:WorksAnyway.java
// In special cases, things just
// seem to work correctly.
import java.util.*;
class MouseTrap {
    static void caughtYa(Object m) {
        Mouse mouse = (Mouse)m; // Cast from Object
        System.out.println("Mouse: " +
            mouse.getNumber());
    }
}
public class WorksAnyway {
    public static void main(String[] args) {
        ArrayList mice = new ArrayList();
        for(int i = 0; i < 3; i++)
            mice.add(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.get(i));
            MouseTrap.caughtYa(mice.get(i));
        }
    }
} ///:~
```

Se puede ver que **toString()** esta sobrecargada en **Mouse**. El segundo bucle **for** en el **main()** se puede encontrar la instrucción:

```
| System.out.println("Free mouse: " + mice.get(i));
```

Después del signo ‘+’ el compilador espera un objeto **String**. **get()** produce un **Object**, así es que el para obtener el **String** deseado el compilador llama implícitamente a **toString()**. Desafortunadamente, se puede trabajar este tipo de magia con **String**: no esta disponible para ninguno otro tipo.

Una segunda estrategia para ocultar la conversión ha sido colocada dentro de **MouseTrap**. El método **caughtYa()** no acepta un **Mouse**, acepta un **Object**, que es entonces convertido a **Mouse**. Esto es bastante presuntuoso, claro, dado que aceptando un **Object** todo puede ser pasado a el método. Sin embargo, la conversión es incorrecta -si se pasa el tipo equivocado- se tendrá una excepción en tiempo de ejecución. Esto no es bueno si hablamos de verificaciones en tiempo de compilación pero sigue siendo robusto. Se debe notar en la utilización de este método:

```
| MouseTrap.caughtYa(mice.get(i));  
Ninguna conversión es necesaria.
```

## Haciendo una **ArrayList** consciente del tipo

Puede que uno no quiera rendirse a este tema todavía. Una solución mas guerrera es crear una nueva clase utilizando la clase **ArrayList**, tal que acepte solo un tipo y produzca solo el tipo deseado:

```
//: c09:MouseList.java  
// A type-conscious ArrayList.  
import java.util.*;  
public class MouseList {  
    private ArrayList list = new ArrayList();  
    public void add(Mouse m) {  
        list.add(m);  
    }  
    public Mouse get(int index) {  
        return (Mouse)list.get(index);  
    }  
    public int size() { return list.size(); }  
} //:~
```

Aquí hay una prueba para el nuevo contenedor:

```
//: c09:MouseListTest.java  
public class MouseListTest {  
    public static void main(String[] args) {  
        MouseList mice = new MouseList();  
        for(int i = 0; i < 3; i++)  
            mice.add(new Mouse(i));  
        for(int i = 0; i < mice.size(); i++)  
            MouseTrap.caughtYa(mice.get(i));  
    }  
} //:~
```

Esto es similar a el ejemplo anterior, excepto que la nueva clase **MouseList** tiene un miembro privado del tipo **ArrayList**, y métodos exactamente igual a una **ArrayList**. Sin embargo, no acepta y produce objetos genéricos del tipo **Object**, solo objetos **Mouse**.

Debe notarse que si **MouseListener** en lugar de haber sido *heredado* de **ArrayList**, el método **add(Mouse)** no podría simplemente sobrecargar el método existente **add(Object)** y entonces seguiría no habiendo restricción en el tipo de objeto que podría ser agregado. De esta forma, el **MouseListener** comienza a *reemplazar* el **ArrayList**, realizando algunas actividades antes de pasar la responsabilidad (vea *Thinking in Patterns with Java* el cual se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com)).

Dado que **MouseListener** aceptará solo un **Mouse**, si se indica:

```
| mice.add(new Pigeon());
```

Se obtendrá un mensaje de error en *tiempo de compilación*. Esta estrategia, que es mas tediosa desde el punto de vista de la codificación, indicara inmediatamente si esta utilizando un tipo inapropiado.

Note que no es necesaria la conversión cuando se utiliza **get()** -siempre es un **Mouse**,

### Tipos parametrizados

Este tipo de problema no es aislado -hay varios caso en los cuales se necesita crear nuevos tipos basados en otros tipos, y en el cual es útil tener especificación del tipo en tiempo de compilación. Este es el concepto de *tipo parametrizado*. En C++, esto es directamente soportado por el lenguaje mediante *plantillas*. Sería bueno que en una futura versión de Java soporte algunas variaciones de tipos parametrizados; el actual candidato a ganar automáticamente crea clases similares a **MouseListener**.

## Iteradores

En una clase contenedora, se debe tener una forma de colocar cosas dentro y una forma de sacarlas. Después de todo, esta es la tarea primaria de un contenedor -almacenar cosas. En el **ArrayList**, **add()** es la forma en la que se insertan objetos, y **get()** es *una* forma de sacarlas. **ArrayList** es flexible -se puede seleccionar cualquier cosa en cualquier momento, y seleccionar múltiples elementos a la vez utilizando diferentes índices.

Si se quiere comenzar a pensar en un nivel mas alto, hay un inconveniente: se necesita saber el tipo exacto del contenedor para utilizarlo. Esto parece no ser malo en un comienzo. ¿Pero cuando se comienza a utilizar **ArrayList**, y mas tarde en su programa se descubre que dado que la forma en que se esta

utilizando el contenedor podría ser mucho mas eficiente utilizar **LinkedList** en lugar de **ArrayList**? ¿O supongamos que se quiere escribir un fragmento de código genérico que no se sabe o no importa con que tipo de contenedor se esta trabajando, así es que puede ser utilizado en diferentes tipos de contenedores sin volver a escribir ese código?

El concepto de *iterator* puede ser utilizado para alcanzar esta abstracción. Un iterator es un objeto cuyo trabajo es moverse a través de una secuencia de objetos y seleccionar cada objeto en esa secuencia sin que el cliente programador sepa o le importe acerca de la estructura de las capas mas bajas de esa secuencia. Además, el iterator es usualmente lo que es llamado una objeto “peso ligero”: uno que es económico de crear. Por esta razón, a menudo se encuentra aparentemente extrañas restricciones para iteradores; por ejemplo, algunos iteradores pueden moverse en una sola dirección.

El **Iterator** de Java es un ejemplo de iterator con este tipo de restricciones. No hay mucho que se pueda hacer con uno excepto:

1. Preguntarle al contenedor para manejar un **Iterator** utilizando un método llamado **iterator()**. Este **iterator** estará listo para retorna el primer elemento en la secuencia de en su primer llamada al método **next()**.
2. Obtener el siguiente objeto en la secuencia con **next()**.
3. Ver si *hay mas* elementos en la secuencia con **hasNext()**.
4. Quitar el último elemento retornado por el iterator con **remove()**.

Esto es solo, es una simple implementación de un iterator, pero sigue siendo poderoso (y hay un mas sofisticado **ListIterator** para **Lists**). Para ver como trabajan, volvamos a visitar el programa **CatsAndDogs.java** atrás en este capítulo. En la versión original, el método **get()** fue utilizado para seleccionar cada elemento, pero en la siguiente versión modificada un **Iterator** es utilizado.

```
//: c09:CatsAndDogs2.java
// Simple container with Iterator.
import java.util.*;
public class CatsAndDogs2 {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();
        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));
        Iterator e = cats.iterator();
        while(e.hasNext())
            ((Cat)e.next()).print();
    }
} ///:~
```

Se puede ver que en las dos últimas líneas ahora se utiliza un **Iterator** para moverse dentro de la secuencia en lugar de un bucle **for**. Con el **Iterator**, no

hay que preocuparse por el número de elementos en el contenedor. **hasNext()** y **next()** tienen cuidado de esto por nosotros.

Otro ejemplo, considerando la creación de un método de impresión de propósito general:

```
//: c09:HamsterMaze.java
// Using an Iterator.
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(Iterator e) {
        while(e.hasNext())
            System.out.println(e.next());
    }
}
public class HamsterMaze {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 3; i++)
            v.add(new Hamster(i));
        Printer.printAll(v.iterator());
    }
} ///:~
```

Miremos de cerca **printAll()**. Se puede notar que no hay información acerca del tipo de sentencia. Todo lo que tiene es un **Iterator**, y eso es todo lo que se necesita conocer acerca de la secuencia: que se puede obtener el siguiente objeto, y que se puede saber cuando se está en el final. Esta idea de tomar un contenedor de objetos y pasar a través de él para realizar una operación en cada uno es poderosa, y será vista a través de este libro.

El ejemplo es aún más genérico, dado que implícitamente utiliza el método **Object.toString()**. El método **println()** es sobrecargado para todos los tipos primitivos de la misma forma que **Object**; en cada caso una cadena es automáticamente producida llamando el método **toString()** apropiado.

A pesar de que es innecesario, se puede ser más explícito utilizando una conversión, que tiene el efecto de llamar a **toString()**.

```
| System.out.println((String)e.next());
```

En general, sin embargo, se querrá hacer algo más que llamar a los métodos de **Object**, así es que se incurrirá contra el tema de la conversión de tipo nuevamente. Se debe asumir que se ha tenido un **Iterator** en una secuencia de un tipo particular en la cual se estaba interesado, y se han convertido los objetos resultantes en ese tipo (obteniéndose una excepción en tiempo de ejecución si se está equivocado).

## Recursión no intencionada

Dado que (como en cada una de las otras clases), los contenedores estándares de Java son heredados de **Object**, estos contienen un método **toString()**. Este ha sido sobreescrito así es que pueden producir un **String** que los represente, incluyendo los objetos que almacenan. Dentro de **ArrayList** por ejemplo, el **toString()** camina sobre los elementos de **ArrayList** y llama a **toString()** para cada uno. Supongamos que se quiere imprimir la dirección de su clase. Parece tener sentido simplemente referirse a **this** (en particular, los programadores C++ son propensos a esta estrategia):

```
//: c09:InfiniteRecursion.java
// Accidental recursion.
import java.util.*;
public class InfiniteRecursion {
    public String toString() {
        return " InfiniteRecursion address: "
            + this + "\n";
    }
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} ///:~
```

Si simplemente se crea un objeto **InfiniteRecursion** y luego se imprime, se tendrá una interminable secuencia de excepciones. esto es verdadero si se colocan los objetos **InfiniteRecursion** en un **ArrayList** y se imprime el **ArrayList** como se muestra aquí. Lo que está sucediendo es la conversión de tipo automático para **Strings**. Cuando se dice:

```
| "InfiniteRecursion address: " + this
```

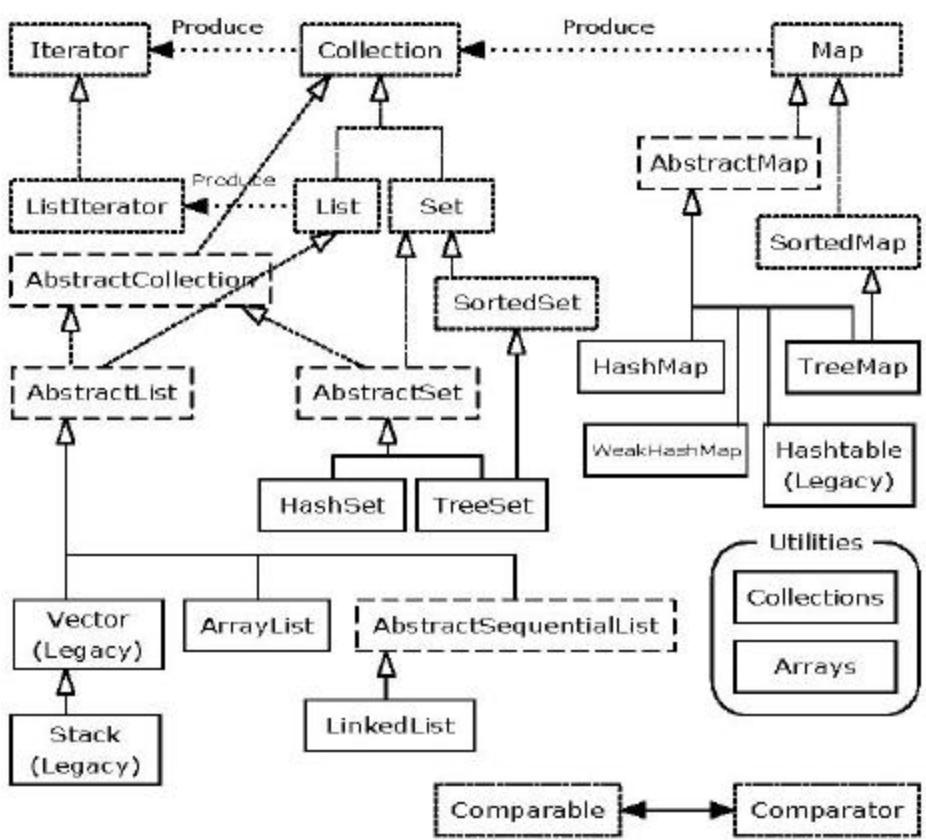
El compilador ve un **String** seguido por un '+' y algo que no es una cadena, así es que trata de convertir **this** a **String**. Se hace la conversión llamando a **toString()**, lo que produce una llamada recursiva.

Si realmente se quiere imprimir la dirección del objeto en este caso, la solución es llamar el método **toString()** de **Object**, que hace exactamente eso.

Así es que en lugar de decir **this**, se debe decir **super.toString()** (Esto solo trabaja si se está directamente heredando de **Object**, o si ninguna de sus clases padres son sobrecargadas a el método **toString()**).

# Clasificación de contenedores

**Collections** y **Maps** pueden ser implementados en diferentes formas, de acuerdo a las necesidades del programa. Ayuda mucho ver un diagrama de los contenedores de Java 2:



El diagrama puede ser un poco abrumador al principio, pero se verá que solo hay tres componentes contenedores: **Map**, **List** y **Set**, y solo dos o tres implementaciones de cada uno (con, típicamente, una versión preferida). Cuando se vea esto, los contenedores no son tan desalentadores.

Los cuadros punteados representan interfaces, los cuadros hechos con rayas representan clases abstractas, y los cuadros sólidos con clases reales (concretas). Las flechas con línea punteada indican que una clase particular implementa una interfase (o en el caso de una clase abstracta, parcialmente implementa esa interfase). Las flechas sólidas muestran que una clase puede

producir objetos de la clase a la cual apunta la flecha. Por ejemplo, cualquier **Collection** puede producir un **Iterator**, donde una **List** puede producir un **ListIterator** (de la misma forma que un **Iterator** común y corriente, dado que **List** es heredada de **Collection**).

Las interfaces que están interesadas en almacenar objetos son **Collection**, **List**, **Set** y **Map**. Idealmente, se escribirá más de su código para hablar de estas interfaces, y el único lugar donde se especificará el tipo preciso que se está utilizando es en el punto de creación. Así es que se puede crear una **List** de esta forma:

```
| List x = new LinkedList();
```

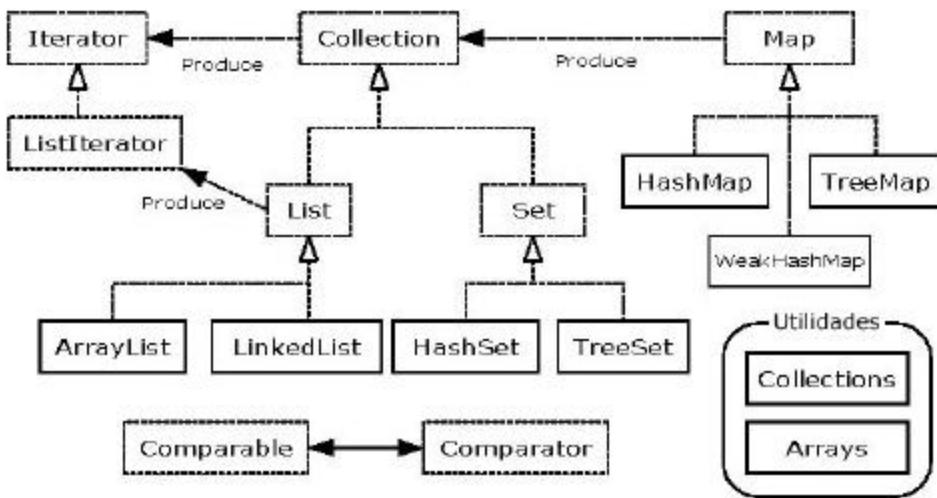
Claro, se puede también decidir hacer **x** una **LinkedList** (en lugar de una **List** genérica) y cargar la información de tipo exacta con **x**. La belleza (y la ambición) de utilizar la interfase es que si se decide cambiar su implementación, todo lo que necesita es cambiarla en el punto de creación, de la siguiente forma:

```
| List x = new ArrayList();
```

El resto de su código puede permanecer sin tocar (algunas de estas generalidades pueden ser también alcanzada con iteradores).

En la jerarquía de clases, se puede ver unas clases cuyos nombres comienzan con “**Abstract**”, y estos pueden ser un poco confusos al principio. Son simples herramientas que implementan parcialmente una interfase particular. Si se está creando un **Set** propio, por ejemplo, no se comenzaría con la interfase **Set** y e implementar todos los métodos, en lugar de eso se hereda de **AbstractSet** y se hace el trabajo mínimo necesario para hacer la nueva clase. Sin embargo, la librería de contenedores tiene suficiente funcionalidad para satisfacer sus necesidades virtualmente todo el tiempo. Así es que para sus propósitos, se puede ignorar cualquier clase que comience con “**Abstract**”.

Por consiguiente, cuando se ve el diagrama, lo que realmente debe preocupar es solo aquellas interfaces en la parte superior del diagrama y las clases bien establecidas (aquellas que tienen un cuadro sólido rodeándolas). Típicamente se creará un objeto de una clase bien establecida, se realizará una conversión ascendente a la interfase correspondiente, y luego se usará la interfase a través del resto del código. Además, no se necesitará considerar los elementos heredados cuando se escriba el nuevo código. Por lo tanto, el diagrama puede ser enormemente simplificado para verse de esta forma:



Ahora solo incluye las interfaces y clases que se encontrarán en los ejemplos básicos, y también los elementos en los cuales nos enfocaremos en este capítulo.

He aquí un ejemplo simple, que llena una **Collection** (representada aquí con un **ArrayList**) con objetos **String**, y que luego imprime cada elemento de la **Collection**:

```

//: c09:SimpleCollection.java
// A simple example using Java 2 Collections.
import java.util.*;
public class SimpleCollection {
    public static void main(String[] args) {
        // Upcast because we just want to
        // work with Collection features
        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());
    }
} ///:~

```

La primer línea en el **main()** crea un objeto **ArrayList** y luego realiza una conversión ascendente a una **Collection**, dado que este ejemplo utiliza solo los métodos de **Collection**, cualquier objeto de la clase heredado de **Collection** trabajará, pero **ArrayList** es el típico caballo de tiro de **Collection**.

El método **add()**, como su nombre lo sugiere, coloca un nuevo elemento en la **Collection**. Sin embargo, la documentación cuidadosamente declara que **add()** “se asegura que ese contenedor contiene el elemento específico”. Esto es para tolerar el significado de **Set**, que agrega un elemento solo si ya no

esta allí. Con una **ArrayList**, o cualquier tipo de **List**, **add()** siempre significa “ponlo aquí”, porque en las **Lists** no importa si hay elementos duplicados.

Todas las **Collections** pueden producir un **Iterator** con el método **iterator()**. Aquí, un **Iterator** es creado y utilizado para atravesar la **Collection**, imprimiendo cada elemento.

## Funcionalidad de Collection

La siguiente tabla muestra todo lo que se puede hacer con una **Collection** no se incluyen aquellos método que automáticamente vienen con **Object**), y de esta forma, todo lo que se puede hacer con una **Set** o una **List** (**List** también tiene funcionalidades adicionales). **Maps** no son heredadas de **Collection**, y serán tratados separados.

boolean add(Object)	Se asegura que el contenedor almacena el argumento. Retorna falso si no se agrega el argumento (Esto es un método “opcional”, descrito mas adelante en este capítulo).
boolean addAll(Collection)	Agrega todos los elementos en el argumento. Retorna <b>true</b> si alguno de los elementos fue agregado (“Opcional”).
void clear()	Quita todos los elementos en el contenedor (“Opcional”).
boolean contains(Object)	<b>true</b> si el contenedor almacena el argumento.
boolean containsAll(Collection)	<b>true</b> si el contenedor almacena el elemento en el argumento.
boolean isEmpty()	<b>true</b> si el contenedor no tienen elementos
Iterator iterator()	Retorna un <b>Iterator</b> que se puede utilizar para moverse a través de los elementos en el contenedor
boolean remove(Object)	Si el argumento es un contenedor, una instancia de

	contenedor, una instancia de ese elemento es eliminado. Retorna <b>true</b> si la eliminación se sucede (“Opcional”).
boolean removeAll(Collection)	Elimina todos los elementos que están contenidos en el argumento. Retorna <b>true</b> si alguna eliminación se sucede (“Opcional”).
boolean retainAll(Collection)	Retiene solo los elementos que están contenidos en el argumentos (una “intersección” teóricamente). Regresa <b>true</b> si algún cambio se sucede (“Opcional”).
int size()	Retorna el número de elementos en el contenedor.
Object[] toArray()	Retorna un arreglo contenido todos los elementos en el contenedor.
Object toArray(Object[] a)	Retorna un arreglo contenido todos los elementos en el contenedor, cuyo tipo es aquel del arreglo antes que un <b>Object</b> plano (se debe realizar una conversión del arreglo a el tipo correcto).

Note que no hay función **get()** para acceder de forma aleatoria a un elemento. Esto es porque **Collection** también incluye **Set**, que mantiene su propio orden interno (y esto hace la búsqueda del acceso aleatorio absurda). De esta manera, si se quiere examinar todos los elementos de una **Collection** se debe utilizar un iterator; esta es la única forma de ir a buscar las cosas de vuelta.

El siguiente ejemplo muestra todos estos métodos. Nuevamente, esto trabaja con todo lo que se herede de **Collection**, pero una **ArrayList** es utilizada como un tipo de “menor común denominador”.

```
//: c09:Collection1.java
// Cosas que se pueden hacer con todas las Collections.
import java.util.*;
import com.bruceeckel.util.*;
public class Collection1 {
    public static void main(String[] args) {
```

```

Collection c = new ArrayList();
Collections2.fill(c,
    Collections2.countries, 10);
c.add("ten");
c.add("eleven");
System.out.println(c);
// Cree un arreglo de la lista:
Object[] array = c.toArray();
// Cree un arreglo de cadenas de la lista:
String[] str =
    (String[])c.toArray(new String[1]);
// Encuentre los elementos máximo y mínimo; esto significa
// cosas diferentes dependiendo de la forma en que
// la interfase Comparable interfase es implementada:
System.out.println("Collections.max(c) = " +
    Collections.max(c));
System.out.println("Collections.min(c) = " +
    Collections.min(c));
// Agregue una Collection a otra Collection
Collection c2 = new ArrayList();
Collections2.fill(c2,
    Collections2.countries, 10);
c.addAll(c2);
System.out.println(c);
c.remove(CountryCapitals.pairs[0][0]);
System.out.println(c);
c.remove(CountryCapitals.pairs[1][0]);
System.out.println(c);
// Elimine todos los componentes que están en la
// colección de argumentos:
c.removeAll(c2);
System.out.println(c);
c.addAll(c2);
System.out.println(c);
// ¿Hay algún elemento en esta Collection?
String val = CountryCapitals.pairs[3][0];
System.out.println(
    "c.contains(" + val + ") = "
    + c.contains(val));
// ¿Hay una Collection en esta Collection?
System.out.println(
    "c.containsAll(c2) = " + c.containsAll(c2));
Collection c3 = ((List)c).subList(3, 5);
// Mantiene todos los elementos que están
// en c2 y c3 (una intersección de grupos):
c2.retainAll(c3);
System.out.println(c);
// Se deshace de todos los elementos
// en c2 que también aparecen en c3:
c2.removeAll(c3);
System.out.println("c.isEmpty() = " +
    c.isEmpty());
c = new ArrayList();
Collections2.fill(c,
    Collections2.countries, 10);

```

```

        System.out.println(c);
        c.clear(); // Remove all elements
        System.out.println("after c.clear():");
        System.out.println(c);
    }
} //:~
```

Las **ArrayLists** son creadas conteniendo diferentes grupos de datos y se realizan conversiones ascendentes a objetos **Collection**, así es que es claro que ningún otro mas que la interfase **Collection** es utilizada. **main()** utiliza simples ejercicios para mostrar todos los métodos en **Collection**.

La siguiente sección describe las distintas implementaciones de **List**, **Set**, y **Map** e indica en cada caso (con un asterisco) cual podría ser la elección por defecto. Se notará que las clases heredadas **Vector**, **Stack**, y **Hashtable** no están incluidas porque en todos los casos son clases preferidas dentro de los contenedores de Java 2.

## Funcionalidad de **List**

La **List** básica es bastante simple de utilizar, como se ha visto hasta ahora con **ArrayList**. A pesar de la mayor parte del tiempo solo se utilizará **add()** para insertar objetos, **get()** para obtenerlos uno por vez, un **iterator()** para obtener un **Iterator** de la secuencia, hay también un grupo de otros método que pueden ser útiles.

Además, hay actualmente dos tipos de **List**: la básica y la **ArrayList**, que se distinguen en el acceso aleatorio a los elementos, y la mucho mas poderosa **LinkedList** (que no esta diseñada para accesos rápidos, pero tiene un grupo mas general de métodos).

<b>List</b> (interfase)	El orden es la característica mas importante de una <b>List</b> ; esta promete mantener los elementos en una secuencia particular. La <b>List</b> agrega una cantidad de métodos a la <b>Collection</b> que permite insertar y eliminar elementos del medio de una <b>List</b> (Esto es recomendable solo para una <b>LinkedList</b> ). Una <b>List</b> producirá un <b>ListIterator</b> , y se puede utilizar para atravesar la lista en ambas direcciones, de la misma forma que insertar y eliminar elementos en el medio de la <b>List</b> .
<b>ArrayList*</b>	Una <b>List</b> implementada con un arreglo. Permite un rápido acceso aleatorio a los elementos, pero es lento cuando se inserta o se eliminan elementos en el medio de la lista. <b>ListIterator</b> debe ser utilizado solo para atravesar para adelante y para atrás en una <b>ArrayList</b> , pero no para insertar y remover

	elementos, lo que es costoso comparado con una <b>LinkedList</b> .
LinkedList	Proporciona un óptimo acceso secuencial, con inserciones de bajo costo y eliminaciones del medio de la <b>List</b> . Relativamente lento para acceso aleatorio. (Se debe utilizar <b>ArrayList</b> en su lugar). También tiene <b>addFirst()</b> , <b>addLast()</b> , <b>getFirst()</b> , <b>getLast()</b> , <b>removeFirst()</b> , y <b>removeLast()</b> (que no está definido en ninguna interfase o clase base) para permitir que se utilice como pila, cola o deque.

Los métodos en el siguiente ejemplo cubren diferentes grupos de actividades: cosas que cada lista puede hacer (**basicTest()**), moviéndose mediante un **Iterator** (**iterMotion()**) contra cambiar las cosas con un **Iterator** (**iterManipulation()**), viendo los efectos de la manipulación de **List** (**testVisual()**), y operaciones disponibles solo para **LinkedLists**,

```
//: c09:List1.java
// Cosas que se pueden hacer con listas.
import java.util.*;
import com.bruceekel.util.*;
public class List1 {
    public static List fill(List a) {
        Collections2.countries.reset();
        Collections2.fill(a,
            Collections2.countries, 10);
        return a;
    }
    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"); // Agregar al final
        // Agregar una colección:
        a.addAll(fill(new ArrayList()));
        // Agregar una colección comenzando por la posición 3:
        a.addAll(3, fill(new ArrayList()));
        b = a.contains("1"); // ¿Esta ahí?
        // ¿Esta la colección entera ahí?
        b = a.containsAll(fill(new ArrayList()));
        // Las listas permiten el acceso aleatorio, lo que no
        // tiene mucho costo para ArrayList, y es caro
        // para las LinkedList:
        o = a.get(1); // Obtiene el objeto en la posición 1
        i = a.indexOf("1"); // Indica el índice del objeto
        b = a.isEmpty(); // ¿Algún elemento dentro?
        it = a.iterator(); // Iterator común
        lit = a.listIterator(); // ListIterator
        lit = a.listIterator(3); // Comienza en la posición 3
```

```

        i = a.lastIndexOf("1"); // Última coincidencia
        a.remove(1); // Elimina la posición 1
        a.remove("3"); // Elimina este objeto
        a.set(1, "y"); // Especifica la posición 1 para "y"
        // Mantiene todo lo que esta en el argumento
        // (la intersección de los dos grupos):
        a retainAll(fill(new ArrayList()));
        // Remueve todo lo que se encuentra en el argumento:
        a.removeAll(fill(new ArrayList()));
        i = a.size(); // ¿Qué tan grande es?
        a.clear(); // Elimina todos los elementos
    }
    public static void iterMotion(List a) {
        ListIterator it = a.listIterator();
        b = it.hasNext();
        b = it.hasPrevious();
        o = it.next();
        i = it.nextInt();
        o = it.previous();
        i = it.previousIndex();
    }
    public static void iterManipulation(List a) {
        ListIterator it = a.listIterator();
        it.add("47");
        // De debe mover a un elemento luego de add():
        it.next();
        // Elimina el elemento que simplemente fue generado:
        it.remove();
        // Se debe mover a un elemento luego de remove():
        it.next();
        // Cambie el elemento que simplemente fue producido:
        it.set("47");
    }
    public static void testVisual(List a) {
        System.out.println(a);
        List b = new ArrayList();
        fill(b);
        System.out.print("b = ");
        System.out.println(b);
        a.addAll(b);
        a.addAll(fill(new ArrayList()));
        System.out.println(a);
        // Insertar, eliminar, y remplazar elementos
        // utilizando un ListIterator:
        ListIterator x = a.listIterator(a.size()/2);
        x.add("one");
        System.out.println(a);
        System.out.println(x.next());
        x.remove();
        System.out.println(x.next());
        x.set("47");
        System.out.println(a);
        // Atravesar la lista para atrás:
        x = a.listIterator(a.size());
        while(x.hasPrevious())
    }
}

```

```
System.out.print(x.previous() + " ");
System.out.println();
System.out.println("testVisual finished");
}
// Aquí hay algunas cosas que solo las
// LinkedLists pueden hacer:
public static void testLinkedList() {
    LinkedList ll = new LinkedList();
    fill(ll);
    System.out.println(ll);
    // Tratarla como una pila, empujando:
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Como "fisgongear" en lo mas alto de la pila:
    System.out.println(ll.getFirst());
    // Como sacar de una pila:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Tratarla como una cola, jalando elementos
    // del final de la cola:
    System.out.println(ll.removeLast());
    // Con las operaciones anteriores es una dequeue!
    System.out.println(ll);
}
public static void main(String[] args) {
    // Crea y llena una nueva lista cada vez:
    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 hechas para mostrar la sintaxis correcta, y mientras que el valor de retorno es capturado, no es utilizado. En algunos casos, el valor de retorno no es capturado dado que no es típicamente utilizado. Se debe encontrar todas las utilizaciones de cada uno de estos métodos en la documentación en línea en [java.sun.com](http://java.sun.com) antes de utilizarlas.

# Creando una pila de una **LinkedList**

Una pila es algo referido como un contenedor “ultimo en entrar, primero en salir” (LIFO). Esto es, cualquier cosa que “empuje” en la pila último es lo primero que se puede “extraer”. Como todos los otros contenedores en Java, lo que se empuja y se extrae son **Objects**, así es que se debe realizar una

conversión lo que se extrae, a no ser que solo se utilice el comportamiento del **Object**.

La **LinkedList** tiene métodos que directamente implementan funcionalidad de pila, así es que simplemente se puede utilizar una **LinkedList** en lugar de crear una clases pila.

```
//: c09:StackL.java
// Making a stack from a LinkedList.
import java.util.*;
import com.bruceekel.util.*;
public class StackL {
    private LinkedList list = new LinkedList();
    public void push(Object v) {
        list.addFirst(v);
    }
    public Object top() { return list.getFirst(); }
    public Object pop() {
        return list.removeFirst();
    }
    public static void main(String[] args) {
        StackL stack = new StackL();
        for(int i = 0; i < 10; i++)
            stack.push(Collections2.countries.next());
        System.out.println(stack.top());
        System.out.println(stack.top());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
} ///:~
```

Si se quiere el comportamiento de pila solamente, heredar es inapropiado aquí porque producirá una clase con todos los métodos de una **LinkedList** (se verá mas tarde que este gran error fue cometido por los diseñadores de la librería de Java 1.0 con **Stack**).

## Creando una cola con una **LinkedList**

Una *cola* es un contendedor “*primero en entrar, primero en salir*” (FIFO). Esto es, se colocan cosas en un final, y se empujan hacia el otro. Así es que el orden en que se colocan es el mismo orden en el que salen. La **LinkedList** tiene métodos para dar soporte a el comportamiento de una cola, así es que se puede utilizar en una clase **Cola**.

```
//: c09:Queue.java
// Making a queue from a LinkedList.
import java.util.*;
public class Queue {
    private LinkedList list = new LinkedList();
```

```

public void put(Object v) { list.addFirst(v); }
public Object get() {
    return list.removeLast();
}
public boolean isEmpty() {
    return list.isEmpty();
}
public static void main(String[] args) {
    Queue queue = new Queue();
    for(int i = 0; i < 10; i++)
        queue.put(Integer.toString(i));
    while(!queue.isEmpty())
        System.out.println(queue.get());
}
} //:~

```

De puede también crear una deque (cola con doble final) de una **LinkedList**. Esto es como una cola, pero se puede agregar y borrar elementos de ambos finales.

## Funcionalidad de **Set**

**Set** tiene la misma interfase que **Collection**, así es que no hay funcionalidad extra como la hay con las dos diferentes **Lists**. En lugar de eso, **Set** es exactamente una **Collection**, solo tiene diferente comportamiento (Esto es el uso ideal de herencia y polimorfismo: expresar diferente comportamiento). Un **Set** rechaza almacenar mas de una instancia de cada valor de objeto (lo que constituye el “valor” de un objeto es mas complejo, como se verá).

<b>Set</b> (interfase )	Cada elemento que se agrega a <b>Set</b> debe ser único; de otra forma <b>Set</b> no agregará el elemento duplicado. Los <b>Objects</b> agregados a <b>Set</b> deben definir un <b>equals()</b> para establecer la unicidad del objeto. <b>Set</b> tiene exactamente la misma interfase que <b>Collection</b> . La interfase de <b>Set</b> no garantiza que mantendrá los elementos en un orden particular.
<b>HashSet*</b>	Para <b>Sets</b> donde el tiempo de búsqueda es importante los <b>Objects</b> deben también definir <b>hashCode()</b> .
<b>TreeSet</b>	Un <b>Set</b> ordenado respaldado por un árbol. De esta forma, se puede extraer una secuencia ordenada de un <b>Set</b> .

El siguiente ejemplo *no* muestra todo lo que se puede hacer con un **Set**, dado que la interfase es la misma que **Collection**, y se practicó en el ejemplo anterior. En lugar de eso, esto demuestra el comportamiento que hace un **Set** único:

```

//: c09:Set1.java
// Cosas que se pueden hacer con Sets.
import java.util.*;
import com.bruceekel.util.*;
public class Set1 {
    static Collections2.StringGenerator gen =
        Collections2.countries;
    public static void testVisual(Set a) {
        Collections2.fill(a, gen.reset(), 10);
        Collections2.fill(a, gen.reset(), 10);
        Collections2.fill(a, gen.reset(), 10);
        System.out.println(a); // Duplicados no!
        // Agregue otro set para este:
        a.addAll(a);
        a.add("one");
        a.add("one");
        a.add("one");
        System.out.println(a);
        // Busca algo:
        System.out.println("a.contains(\"one\"): " +
            a.contains("one")));
    }
    public static void main(String[] args) {
        System.out.println("HashSet");
        testVisual(new HashSet());
        System.out.println("TreeSet");
        testVisual(new TreeSet());
    }
} //:~

```

Los valores duplicados son agregados a **Set**, pero cuando son impresos se vera que **Set** ha aceptados solo una instancia de cada valor.

Cuando se ejecute este programa se verá que el orden mantenido por el **HashSet** es diferente de **TreeSet**, dado que cada uno tiene una forma diferente de ordenar elementos para poder ser localizados mas tarde (**TreeSet** los mantiene ordenados, mientras que **HashSet** utiliza una función de hash, la que es diseñada para una rápida búsqueda). Cuando se crean sus propios tipos, se debe tener cuidado porque un **Set** necesita una forma de mantener un orden de almacenamiento, lo que significa que se debe implementar la interfase **Comparable** y definir el método **CompareTo()**. He aquí un ejemplo:

```

//: c09:Set2.java
// Putting your own type in a Set.
import java.util.*;
class MyType implements Comparable {
    private int i;
    public MyType(int n) { i = n; }
    public boolean equals(Object o) {
        return
            (o instanceof MyType)
            && (i == ((MyType)o).i);
    }
}

```

```

public int hashCode() { return i; }
public String toString() { return i + " "; }
public int compareTo(Object o) {
    int i2 = ((MyType)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
}
}
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 void test(Set a) {
        fill(a, 10);
        fill(a, 10); // Try to add duplicates
        fill(a, 10);
        a.addAll(fill(new TreeSet(), 10));
        System.out.println(a);
    }
    public static void main(String[] args) {
        test(new HashSet());
        test(new TreeSet());
    }
} ///:~

```

La forma para definir **equals()** y **hashCode()** será descrita mas tarde en este capítulo. Se debe definir un **equals()** en ambos casos, pero el **hashCode()** es absolutamente necesario solo si la clase será colocada en un **HashSet** (lo que es bueno, dado que generalmente será su primera elección como una implementación **Set**). Sin embargo, como estilo de programación se debería siempre sobrecargar **hashCode()** cuando se sobrecargue **equals()**. Este proceso será detallado totalmente mas tarde en este capítulo.

En el **compareTo()**, debe notarse que *no* se utiliza la “simple y obvia” forma **return i-i2**. A pesar de que esto es un error común de programación, solo trabajará propiamente si **i** e **i2** son enteros “sin signo” (Si Java *tuviera* una palabra clave “*unsigned*”, cosa que no es). Esto rompe el entero con signo de Java que no es suficientemente grande para representar la diferencia entre dos enteros con signo. Si **i** es un entero positivo grande y **j** es un entero negativo grande, **i-j** produce un desbordamiento y retorna un valor negativo, por lo que no funcionará.

## SortedSet

Si se tiene un **SortedSet** (de los cuales **TreeSet** es el único disponible), los elementos están garantizados de estar en orden lo que permite funcionalidad adicional para proporcionar estos métodos en la interfase de **SortedSet**:

**Comparator comparator()**: Produce el **comparator** utilizado por este **Set**, o **null** para ordenamiento natural.

**Object first()**: Produce el menor elemento.

**Object last()**: Produce el mayor elemento.

**SortedSet subSet(fromElement, toElement)**: Produce una vista de este **Set** con elementos de **fromElement**, inclusive, hasta **toElement**, exclusivo.

**SortedSet tailSet(fromElement)**: Produce una vista de este **Set** con los elementos mayores o iguales a **toElement**.

## Funcionalidad de Map

Una **ArrayList** permite seleccionar de una secuencia de objetos utilizando un número, así es que en cierto sentido asocia números a objetos. ¿Pero que si se quiere seleccionar de una secuencia de objetos utilizando algún otro criterio? Una pila es un ejemplo: el criterio de selección es “la última cosa empujada en la pila”. Un poderoso giro de esta idea de “seleccionar de una secuencia” es llamada alternativamente un *map*, un *diccionario*, o un *arreglo asociativo*. Conceptualmente, parece una **ArrayList**, pero en lugar de buscar objetos utilizando un número. ¡Se buscan utilizando *otro objeto!* Esto es a menudo un proceso clave en un programa.

El concepto muestra en Java como la interfase **Map**. El método **put(Object key, Object value)** agrega un valor (lo que se quiera), y lo asocia con una clave (lo que se va a buscar). **get(Object key)** produce el valor dando la clave correspondiente. Se puede también probar un **Map** para ver si contiene la clave o si un valor con **containsKey()** y **containsValue()**.

La librería estándar de Java contiene dos tipos diferentes tipos de **Maps**:

**HashMap** y **TreeMap**. Ambos tienen la misma interfase (dado que ambos implementan **Map**), pero difieren en una forma clara: eficiencia. Si se observa lo que hace **get()**, este parece mas lento de buscar a través de (por ejemplo) una **ArrayList** por una clave. Aquí es donde **HashMap** acelera las cosas. En lugar de una búsqueda lenta por una clave, esta utiliza un valor especial llamado *código hash*. Es código hash es una forma de tomar cierta información en el objeto en cuestión y convertirlo en “relativamente el único” entero para ese objeto. Todos los objetos Java pueden producir un código hash, y **hashCode()** es un método en la clase raíz **Object**. Un **HashMap** toma el **hashCode()** de un objeto u lo usa para rápidamente buscar la clave. Este resulta en una dramática mejora en el rendimiento<sup>7</sup>.

---

<sup>7</sup> Si este aumento de velocidad no es apropiado para sus necesidades de rendimiento, se puede acelerar escribiendo su propio **Map** y hacerlo a la medida para sus tipos particulares

<b>Map</b> (interfase)	Mantiene una asociación clave-valor (pares) , así es que se puede buscar un valor utilizando una clave.
HashMap*	Implementación basada en una tabla hash (Utilice esta en lugar de una <b>Hashtable</b> ). Proporciona rendimiento constante en tiempo para insertar y encontrar pares. El rendimiento puede ser ajustado mediante constructores que permita configurar la <i>capacidad y el factor de carga</i> de la tabla hash.
TreeMap	Implementación basada en un árbol rojo-negro. Cuando se ven las claves o los pares, estos serán ordenados (determinado por <b>Comparable</b> o por <b>comparator</b> , discutidos mas adelante). El punto de <b>TreeMap</b> es que se obtienen los resultados ordenados. <b>TreeMap</b> es el único <b>Map</b> con el método <b>subMap()</b> , que permite retornar una parte del árbol.

A veces se necesita conocer los detalles de como el hashing trabaja, así veremos esto un poco mas adelante.

El siguiente ejemplo utiliza el método **Collection2.fill()** y la prueba de grupos de datos que fuera previamente definida.

```
//: c09:Map1.java
// Things you can do with Maps.
import java.util.*;
import com.bruceekel.util.*;
public class Map1 {
    static Collections2.StringPairGenerator geo =
        Collections2.geography;
    static Collections2.RandStringPairGenerator
        rsp = Collections2.rsp;
    // Producing a Set of the keys:
    public static void printKeys(Map m) {
        System.out.print("Size = " + m.size() + ", ");
        System.out.print("Keys: ");
        System.out.println(m.keySet());
    }
    // Producing a Collection of the values:
```

---

para evitar retrasos de conversiones de y hacia **Objects**. Para alcanzar niveles mas altos aún de rendimiento, los entusiastas de la velocidad pueden utilizar la segunda edición de Donald Knuth's *The art of computer programming, Volume 3: Shorting and Searching*. para reemplazar las listas con arreglos que tienen dos beneficios adicionales: pueden ser optimizados con características que permitan un óptimo almacenamiento en disco y pueden ahorrar la mayoría del tiempo de creación y de recolección de basura de cada registro individual.

```

public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
}
public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Map has 'Set' behavior for keys:
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
    printValues(m);
    System.out.println(m);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[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 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: "+key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): " +
        + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // 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 HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
}
} //:~

```

Los métodos **printKeys()** y **printValues()** no solo son utilitarios útiles, estos demuestran también como producir vistas **Collection** de un **Map**. El método **keySet()** produce un **Set** respaldado por las claves en el **Map**. Tratamiento similar es dado a **values()**, que produce una **Collection** conteniendo todos los valores en el **Map** (Debe notarse que las claves deben ser únicas, mientras que los valores pueden contener duplicados). Dado que estas **Collection**s son respaldadas por **Map**, cualquier cambios en una **Collection** será reflejado en el **Map** asociado.

El resto del programa proporciona un ejemplo simple de cada operación **Map**, y probar cada tipo de **Map**.

Como ejemplo de uso de un **HashMap**, considere un programa que verifique la aleatoriedad del método **Math.random()**. Idealmente, este produciría una perfecta distribución de números aleatorios, pero para verificar esto se deberá generar un montón de números aleatorios y contarlos para colocarlos en varios rangos. Un **HashMap** es perfecto para esto, dado que asocia objetos con objetos (en este caso, el objeto valor contiene el número producido por **Math.random()** junto con el número de veces que el número aparece):

```
//: c09:Statistics.java
// Simple demostración de HashMap.
import java.util.*;
class Counter {
    int i = 1;
    public String toString() {
        return Integer.toString(i);
    }
}
class Statistics {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            Integer r =
                new Integer((int)(Math.random() * 20));
            if(hm.containsKey(r))
                ((Counter)hm.get(r)).i++;
            else
                hm.put(r, new Counter());
        }
        System.out.println(hm);
    }
} ///:~
```

En el **main()**, cada vez que un número es generado es envuelto dentro de un objeto **Integer** de tal manera de que se pueda utilizar la referencia con el **HashMap** (no se puede utilizar una primitiva con un contenedor, solo una referencia a un objeto). El método **containsKey()** verifica para ver si la clave ya se encuentra en el contenedor (Esto es. ¿El número ya ha sido encontrado?). Si ya se encuentra, el método **get()** produce el valor asociado a la clave, el que en este caso es un objeto **Counter**. El valor **i** dentro del contador es incrementado para indicar que un número aleatorio mas de este ha sido encontrado.

Si la clave no ha sido encontrada todavía, el método **put()** colocara un nuevo par clave-valor dentro del **HashMap**. Dado que **Counter** automáticamente inicia la variable **i** a uno cuando es creada, esta indica la primer ocurrencia de este número aleatorio particular.

Para mostrar el **HashMap**, este es impreso simplemente. El método de **HashMap toString()** se mueve a través de todos los pares clave-valor llamando **toString()** para cada uno. El **Integer.toString()** es predefinido, y se puede ver el **toString()** para **Counter**. La salida para una corrida (con algunas finales de línea agregados 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}
```

Se puede preguntar por la necesidad de la clase **Counter**, que parece no tener la funcionalidad de la clase envoltura **Integer**. ¿Por qué no utilizar **int** o **Integer**? Bueno, no se puede utilizar un **int** porque todos los contenedores solo pueden almacenar solo referencias a **Object**.

Luego de ver contenedores la clases envolturas pueden tener un poco mas de sentido, dado que no se puede colocar ninguno de los tipos primitivos en los contenedores. Sin embargo, la única cosa que se *puede* hacer con las envolturas de Java es inicializarlas a un valor particular y leer ese valor. Esto es, no hay forma de cambiar el valor una vez que se ha un objeto envoltura ha sido creado. Esto hace que la envoltura **Integer** sea inútil inmediatamente para solucionar nuestro problema, así es que estamos forzados a crear una nueva clase que satisfaga la necesidad.

## SortedMap

Si tiene un **SortedMap** (de los cuales **TreeMap** es el único disponible), hay una garantía de que las claves están en orden lo que permite funcionalidad adicional proporcionada con estos métodos en la interfase **SortedMap**:

**Comparator comparator()**: Produce la comparación utilizada para este **Map**, o **null** para un ordenamiento natural.

**Object firstKey()**: Produce la clave mas baja.

**Object lastKey()**: Produce la clave mas alta.

**SortedMap subMap(fromKey, toKey)**: Produce una vista de este **Map** con claves de **fromKey**, inclusive, hasta **toKey**.

**SortedMap headMap(toKey)**: Produce una vista de este **Map** con claves menores que **toKey**.

**SortedMap tailMap(fromKey)**: Produce una vista de este **Map** con claves mayores o iguales que **fromKey**.

Hashing y códigos hash

En el ejemplo previo, una clase de la librería estándar (**Integer**) fue utilizada como clave para un **HashMap**. Esta trabaja bien como clave, porque tiene

todas las cosas necesarias para hacer que trabaje correctamente como clave. Pero una dificultad común sucede con **HashMaps** cuando se crean clases propias para ser utilizadas como claves. Por ejemplo, considere un sistema para predecir el tiempo que hace corresponder objetos **Groundhog** con objetos **Prediction**. Esto parece bastante directo -se crean las dos clases, y se utiliza **Groundhog** como la clave y **Prediction** como valor.

```
//: c09:SpringDetector.java
// Looks plausible, but doesn't work.
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) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10; i++)
            hm.put(new Groundhog(i), new Prediction());
        System.out.println("hm = " + hm + "\n");
        System.out.println(
            "Looking up prediction for Groundhog #3:");
        Groundhog gh = new Groundhog(3);
        if(hm.containsKey(gh))
            System.out.println((Prediction)hm.get(gh));
        else
            System.out.println("Key not found: " + gh);
    }
} //:~
```

Cada **Groundhog** esta dando un valor idéntico, así es que se puede encontrar un **Prediction** en el **HashMap** diciendo. “Dame la **Prediction** asociado con el número de **Groundhog** 3”. La clase **Prediction** contiene un **boolean** que es inicializado utilizando **Math.random()**, y un **toString()** que interpreta el resultado. En **main()**, un **HashMap** es llenado con **Groundhogs** y sus **Predictions** asociadas. El **HashMap** es impreso así es que se puede ver que ha sido llenado. Entonces un **Groundhog** con un número idéntico de 3 es utilizado como clave para buscar la predicción para la **Groundhog** numero 3 (que como se puede ver esta en el **Map**),

Se ve suficientemente simple, pero no trabaja. El problema es que **Groundhog** es heredado de la clase raíz en común **Object** (que es lo que sucede si no especifica una clase base, de esta manera todas las clases son a

fin de cuentas heredadas de **Object**). El método **hashCode()** de **Object** es utilizado para generar el código hash para cada objeto, y por defecto este utiliza la dirección de ese objeto. De esta forma, la primer instancia de **Groundhog(3)** no produce un código hash igual a el código hash para la segunda instancia de **Groundhog(3)** que ha sido utilizada para realizar la búsqueda.

Se puede pensar que todo lo que necesita hacer es volver a escribir una función apropiada para **hashCode()**. Pero seguiría sin funcionar hasta que se haya hecho una cosa mas: sobrescribir **equals()** que es también parte de **Object**. Este método es utilizado por **HashMap** cuando se trata de determinar si su clave es igual a cualquiera de las claves de la tabla. Nuevamente, el **Object.equals()** simplemente compara direcciones de objetos, así es que **Groundhog(3)** no es igual a otro **Groundhog(3)**.

De esta forma, para utilizar su propia clase como clave en un **HashMap**, se debe sobrescribir ambas funciones, **hashCode()** y **equals()**, como se muestra en la siguiente solución a el problema anterior:

```
//: c09:SpringDetector2.java
// Una clase que es utilizada como clave en un HashMap
// debe sobrescribir hashCode() y equals().
import java.util.*;
class Groundhog2 {
    int ghNumber;
    Groundhog2(int n) { ghNumber = n; }
    public int hashCode() { return ghNumber; }
    public boolean equals(Object o) {
        return (o instanceof Groundhog2)
            && (ghNumber == ((Groundhog2)o).ghNumber);
    }
}
public class SpringDetector2 {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        for(int i = 0; i < 10; i++)
            hm.put(new Groundhog2(i),new Prediction());
        System.out.println("hm = " + hm + "\n");
        System.out.println(
            "Looking up prediction for groundhog #3:");
        Groundhog2 gh = new Groundhog2(3);
        if(hm.containsKey(gh))
            System.out.println((Prediction)hm.get(gh));
    }
} ///:~
```

Debe notarse que se utiliza la clase **Prediction** del ejemplo anterior, así es que **SpringDetector.java** debe ser compilado primero o se obtendrá un error en tiempo de compilación cuando se trate de compilar **SpringDetector2.java**.

**Groundhog2.hashCode()** retorna el número de marmota<sup>8</sup> como un identificador. En este ejemplo, el programador es responsable por asegurarse que no existan dos marmotas con el mismo número ID. El **hashCode()** no es necesario para retornar un identificador único (algo que se entenderá mejor mas tarde en este capítulo), pero el método igual debe ser capas de determinar estrictamente cuando dos objetos son equivalentes.

Aun cuando pareciera ser que el método **equals()** es solo una prueba para ver cuando el argumento es una instancia **Groundhog2** (utilizando la palabra clave **instanceof**, que será explicada totalmente en el capítulo 12), **instanceof** actualmente silenciosamente realiza una segunda verificación de cordura, para ver si el objeto en null. Asumiendo que es el tipo correcto y no null, la comparación esta basada en la actual **ghNumbers**. Esta vez, cuando se ejecute el programa, se podrá ver que produce la salida correcta.

Cuando se cree una clase propia para utilizar en **HashSet**, se debe prestar atención en los mismo temas que cuando se utiliza una clave en un **HashMap**.

## Entendiendo hashCode()

El ejemplo mas arriba es solo el inicio de a través de una solución correcta del problema. Este muestra que si no sobrescribe **hashCode()** y **equals()** de acuerdo a su clave, la estructura de datos hash (**HashSet** o **HashMap**) no será capaz de tratar con sus claves propiamente. Sin embargo, para tener una buena solución para el problema se necesita entender que sucede dentro de la estructura de datos hash.

Primero, considere la motivación detrás del hashing: se quiere encontrar un objeto utilizando otro objeto. Pero se puede lograr esto con un **treeSet** o un **treeMap** también. Es posible implementar su propio **Map**. Para hacerlo de esta forma, el método **Map.entrySet()** debe ser alimentado para producir un grupo de objetos **Map.Entry**. **MPair** será definido como el nuevo tipo de **Map.Entry**. Para que sea colocado en un **TreeSet** debe implementarse **equals()** y ser **Comparable**:

```
//: c09:MPair.java
// A Map implemented with ArrayLists.
import java.util.*;
public class MPair
implements Map.Entry, Comparable {
    Object key, value;
    MPair(Object k, Object v) {
        key = k;
        value = v;
    }
```

---

<sup>8</sup> N.T. Groundhog significa marmota. El autor se esta refiriendo tal vez a la propiedad de predecir el tiempo que se le atribuye a la marmota.

```

public Object getKey() { return key; }
public Object getValue() { return value; }
public Object setValue(Object v){
    Object result = value;
    value = v;
    return result;
}
public boolean equals(Object o) {
    return key.equals(((MPair)o).key);
}
public int compareTo(Object rv) {
    return ((Comparable)key).compareTo(
        ((MPair)rv).key);
}
} //:~

```

Debe notarse que las comparaciones solo se interesan en las claves, así es que los valores duplicados son perfectamente aceptables.

El siguiente ejemplo implementa un **Map** utilizando un par de **ArrayLists**:

```

//: c09:SlowMap.java
// Un Map implementado con ArrayLists.
import java.util.*;
import com.bruceeckel.util.*;
public class SlowMap extends AbstractMap {
    private ArrayList
    keys = new ArrayList(),
    values = new ArrayList();
    public Object put(Object key, Object value) {
        Object result = get(key);
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
            values.set(keys.indexOf(key), value);
        return result;
    }
    public Object get(Object key) {
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    public Set entrySet() {
        Set entries = new HashSet();
        Iterator
        ki = keys.iterator(),
        vi = values.iterator();
        while(ki.hasNext())
            entries.add(new MPair(ki.next(), vi.next()));
        return entries;
    }
    public static void main(String[] args) {
        SlowMap m = new SlowMap();
        Collections2.fill(m,
        Collections2.geography, 25);
    }
}

```

```
        System.out.println(m);
    }
} //:~
```

El método **put()** simplemente coloca las claves y los valores en las correspondientes **ArrayLists**. En el main(), un **SlowMap** es cargado y luego impreso para mostrar que este trabaja.

Esto muestra que no es duro producir un nuevo tipo de **Map**. Pero como el nombre lo sugiere, un **SlowMap** no es muy rápido, así es que probablemente no lo use si se tiene una alternativa disponible. El problema es en la búsqueda de la clave: no hay orden así es que una búsqueda simple es utilizada, lo que es la forma mas lenta de buscar algo.

El punto mas importante del hashing es la velocidad: el hashing permite que la búsqueda se suceda rápidamente. Dado que el cuello de botella es la velocidad de búsqueda de la clave, una de las soluciones del problema puede ser mantener las claves ordenadas y luego utilizar **Collections.binarySearch()** para realizar una búsqueda (un ejercicio en el final de este capítulo trabajará sobre este proceso).

Hashing va mas allá diciendo que todo lo que quiere hacer es almacenar la clave en algún lugar de tal forma que pueda encontrarla rápidamente. Como se ha visto en este capítulo, la estructura mas rápida es aquella en la que se almacena un grupo de elementos en un arreglo, así es que será utilizada para representar la información de la clave (debe notarse cuidadosamente que he dicho “información de la clave”, y no la propia clave). También se puede ver en este capítulo el hecho de que un arreglo, una vez ubicado, no se le puede cambiar el tamaño, así es que tenemos un problema: queremos ser capaces de almacenar cualquier número de valores en el **Map**, pero el número de claves es fijo por el tamaño del arreglo. ¿Como puede ser esto?

La respuesta es que el arreglo no almacena las claves. Del objeto clave, un número será derivado que indexara dentro del arreglo. Este número es el código hash, producido por el método **hashCode()** (en el lenguaje de la ciencia de las computadoras, esta es la función hash definida en Object y presumiblemente sobreescrita por su clase. Para solucionar el problema del arreglo de tamaño fijo, mas de una clave puede producir el mismo índice. Esto es, puede haber colisiones. Es por esto, que no importa que tan grande sea el arreglo porque cada objeto clave aterrizará en alguna parte del arreglo.

Así es que el proceso de buscar un valor comienza computando el código hash y utilizándolo para indexar dentro del arreglo. Si se puede garantizar que no habrá colisiones (lo que puede ser posible si se tiene un número fijo de valores) entonces se tiene la función de hashing perfecta, pero eso es un caso especial. En todos los otros casos, las colisiones son manejadas mediante encadenado externo: al arreglo no apunta directamente a un valor, en lugar de eso apunta a una lista de valores. Estos valores son explorados

en una forma lineal utilizando el método **equals()**. Claro, este aspecto de la búsqueda es mucho mas lento, pero si la función de hash es buena solo se tendrá unos pocos valores en cada ubicación, a lo sumo. Así es que en lugar de buscar a través de la lista entera, se puede rápidamente saltar a una ubicación donde se tiene que comparar unas pocas entradas para encontrar el valor. Esto es mucho mas rápido, lo que hace a el **HashMap** tan rápido.

Conociendo las básicas del hashing. Es posible implementar un simple **Map** utilizando hash:

```
//: c09:SimpleHashMap.java
// A demonstration hashed Map.
import java.util.*;
import com.bruceeckel.util.*;

public class SimpleHashMap extends AbstractMap {
    // Elige un número primo para el tamaño de la tabla
    // hash, para alcanzar una distribución uniforme:
    private final static int SZ = 997;
    private LinkedList[] bucket= new LinkedList[SZ];
    public Object put(Object key, Object value) {
        Object result = null;
        int index = key.hashCode() % SZ;
        if(index < 0) index = -index;
        if(bucket[index] == null)
            bucket[index] = new LinkedList();
        LinkedList pairs = bucket[index];
        MPair pair = new MPair(key, value);
        ListIterator it = pairs.listIterator();
        boolean found = false;
        while(it.hasNext()) {
            Object iPair = it.next();
            if(iPair.equals(pair)) {
                result = ((MPair)iPair).getValue();
                it.set(pair); // Reemplaza el viejo con el nuevo
                found = true;
                break;
            }
        }
        if(!found)
            bucket[index].add(pair);
        return result;
    }
    public Object get(Object key) {
        int index = key.hashCode() % SZ;
        if(index < 0) index = -index;
        if(bucket[index] == null) return null;
        LinkedList pairs = bucket[index];
        MPair match = new MPair(key, null);
        ListIterator it = pairs.listIterator();
        while(it.hasNext()) {
            Object iPair = it.next();
            if(iPair.equals(match))
                return ((MPair)iPair).getValue();
        }
    }
}
```

```

        return null;
    }
    public Set entrySet() {
        Set entries = new HashSet();
        for(int i = 0; i < bucket.length; i++) {
            if(bucket[i] == null) continue;
            Iterator it = bucket[i].iterator();
            while(it.hasNext())
                entries.add(it.next());
        }
        return entries;
    }
    public static void main(String[] args) {
        SimpleHashMap m = new SimpleHashMap();
        Collections2.fill(m,
                           Collections2.geography, 25);
        System.out.println(m);
    }
} //:~
}

```

Dado que las ubicaciones en una tabla hash son a menudo referidas como **buckets** (cubos), el arreglo que representa la tabla actual es llamada **bucket**. Para promover una distribución pareja, el número buckets es típicamente un número primo. Note que es un arreglo de **LinkedList**, lo que automáticamente proporciona soporte para las colisiones -cada nuevo ítem es simplemente agregado al final de la lista.

El valor de retorno de **put()** es **null** o, si la clave ya se encuentra en la lista, el viejo valor asociado con esa clave. El valor de retorno es **result**, que es inicializado a **null**, pero la clave es descubierta en la lista luego **result** es asignado a esa clave.

Para **put()** y **get()**, la primera cosa que sucede es que el **hashCode()** es llamada para esa clave, y el resultado es forzado a ser un número positivo. Entonces es forzado a encajar dentro del arreglo de **buckets** utilizando el operador módulo y el tamaño del arreglo. Si la ubicación es **null**, significa que no hay elementos en esa ubicación, así es que una nueva **LinkedList** es creada para almacenar el objeto. Sin embargo, el proceso normal es buscar a través de la lista para ver si hay duplicados, y si hay, el viejo valor es colocado como **result** y el nuevo valor reemplaza el viejo. La bandera **found** mantiene el rastro de donde un viejo par clave-valor fue encontrado y, si no, el nuevo par es agregado al final de la lista.

En **get()**, se puede ver código similar que el contenido en **put()**, pero más simple. El índice es calculado dentro del arreglo **bucket**, y si una **LinkedList** existe es explorada por una coincidencia.

**entrySet()** debe encontrar y atravesar todas las listas, agregándolas a el resultante **Set**. Una vez que este método ha sido creado, el **Map** puede ser probado llenándolo de valores y luego imprimiéndolos.

## Factores de rendimiento de **HashMap**

Para entender estos temas alguna terminología es necesaria:

**Capacidad:** El número de buckets en la tabla

**Capacidad inicial:** El número de buckets cuando la tabla es creada.

**HashMap** y **HashSet**: tienen constructores que permiten especificar la capacidad inicial

**Tamaño:** El número de entradas actualmente en la tabla.

**Factor de carga:** El tamaño sobre la capacidad. Un factor de carga de cero es una tabla vacía. 0.5 es la mitad de la totalidad de la tabla, etc. Una tabla cargada ligeramente tendrá menos colisiones y por lo tanto óptima para inserciones y búsquedas (pero será lenta en el proceso de atravesarla con un iterator). **HashMap** y **HashSet** tienen constructores que permiten especificar el factor de carga, que significa que cuando este factor es alcanzado el contenedor automáticamente aumentará la capacidad (el número de buckets) doblando la capacidad, y luego redistribuyendo los objetos existentes en el nuevo grupo de buckets (esto es llamado rehashing).

El factor de carga utilizado por **HashMap** es 0.75 (no realiza un rehashing hasta que la tabla esta 3/4 llena). Esto parece ser un buen trueque entre el tiempo y los costos de espacio. Un factor de carga mas alto disminuye el espacio requerido por la tabla pero incrementa el costo de búsqueda, lo que es importante porque la búsqueda es lo que se hace la mayor parte del tiempo (incluyendo **get()** y **put()**).

Si se sabe cuantas entradas se van a almacenar en un **HashMap**, créelo con una capacidad inicial apropiada para prevenir la sobrecarga del rehashing automático.

## Sobrecargando el **hashCode()**

Ahora que se entiende que está involucrado en la función de **HashMap**, los temas involucrados en la escritura de **hashCode()** tendrá mas sentido.

Antes que nada, no se tiene control de la creación o del valor actual que se esta utilizando para indexar dentro del arreglo de buckets. Esto depende de la capacidad de un objeto **HashMap** en particular, y esa capacidad cambia dependiendo de que tan lleno el contenedor esta, y cual es el factor de carga. El valor producido por su **hashCode()** será procesado para crear el índice de buckets (en **SimpleHashMap** el calculo es solo un módulo entre el tamaño del arreglo de buckets).

El factor mas importante en la creación de **hashCode()** es ese, sin importar cuando **hashCode()** es llamado, esto produce el mismo valor para un objeto particular cada vez que es llamado. Si se termina con un objeto que produce

un valor **hashCode()** cuando esta en **put()** en un **HashMap**, y otro durante un **get()**, no será capaz de recuperar los objetos. Así es que si su **hashCode()** depende de datos que cambian en el objeto el usuario se debe hacer consciente que cambiar los datos efectivamente producirá una clave diferente que generará un **hashCode()** diferente.

Además, probablemente no se querrá generar un **hashCode()** que esté basado en información de un único objeto -en particular, el valor de **this** crea un **hashCode()** malo porque no puede generar una clave idéntica a la utilizada para colocar con **put()** el par clave-valor original. Este fue el problema que se sucedió en **SpringDetector.java** porque la implementación por defecto de **hashCode()** utiliza la dirección del objeto. Así es que si se querrá utilizar la información en el objeto que identifica a el objeto en una forma significativa.

Un ejemplo se encuentra en la clase **String**. Las cadenas tienen la característica especial que si un programa tiene varios objetos **String** que contienen secuencias idénticas de caracteres, entonces esos objetos **String** se acotan en la misma memoria (el mecanismo de esto esta descrito en el Apéndice A). Así es que tiene sentido que el **hashCode()** producido por dos instancias separadas de `new String("hello")` sean idénticas. Se puede ver esto corriendo este programa:

```
//: c09:StringHashCode.java
public class StringHashCode {
    public static void main(String[] args) {
        System.out.println("Hello".hashCode());
        System.out.println("Hello".hashCode());
    }
} ///:~
```

Para que esto trabaje, el **hashCode()** para **String** debe ser basado en el contenido de **String**.

Así es que para que un **hashCode()** sea efectivo, debe ser rápido y debe ser significativo: esto es, debe generar un valor basado en el contenido del objeto. Recordemos que este valor no tiene que ser único -se debe apuntar a la velocidad en lugar de la unicidad- y entre **hashCode()** y **equals()** la identidad del objeto puede ser completamente resuelta.

Dado que se favorece que **hashCode()** sea procesado antes que el incide de buckets sea producido, el rango de valores no es importante; solo se necesita generar un entero.

Hay otro factor: un buen **hashCode()** tiene que resultar en una distribución de valores pareja. Si los valores tienden a agruparse, entonces el **HashMap** o el **HashSet** será mas pesado de cargar en algunas áreas y no será tan rápido como podría ser en una función hash distribuida de forma pareja.

He aquí un ejemplo que sigue estas líneas:

```

//: c09:CountedString.java
// Creating a good hashCode().
import java.util.*;
public class CountedString {
    private String s;
    private int id = 0;
    private static ArrayList created =
        new ArrayList();
    public CountedString(String str) {
        s = str;
        created.add(s);
        Iterator it = created.iterator();
        // Id is the total number of instances
        // of this string in use by CountedString:
        while(it.hasNext())
            if(it.next().equals(s))
                id++;
    }
    public String toString() {
        return "String: " + s + " id: " + id +
            " hashCode(): " + hashCode() + "\n";
    }
    public int hashCode() {
        return s.hashCode() * id;
    }
    public boolean equals(Object o) {
        return (o instanceof CountedString)
            && s.equals(((CountedString)o).s)
            && id == ((CountedString)o).id;
    }
    public static void main(String[] args) {
        HashMap m = new HashMap();
        CountedString[] cs = new CountedString[10];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString("hi");
            m.put(cs[i], new Integer(i));
        }
        System.out.println(m);
        for(int i = 0; i < cs.length; i++) {
            System.out.print("Looking up " + cs[i]);
            System.out.println(m.get(cs[i]));
        }
    }
}
} //:~

```

**CountedString** incluye una cadena y un id que representa el numero de objetos **CountedString** que contiene una cadena idéntica. El conteo se logra en el constructor por iteración a través del **ArrayList** estático donde todos los **Strings** son almacenados.

**hashCode()** y **equals()** produce resultados basados en ambos campos; si ellos son basados solamente en los **String** o en solamente el id encontraremos coincidencias duplicadas para distintos valores.

Deba notarse lo simple que **hashCode()** es: el **hashCode()** para un **String** es multiplicado por el **id**. El menor es generalmente mejor (y rápido) para **hashCode()**.

En **main()**, un montón de objetos **CountedString** son creados, utilizando el mismo **String** para mostrar, que los duplicados crean valores únicos a causa del contador **id**. El **HashMap** es desplegado así es que se puede ver como esto se almacena internamente (en órdenes que no se pueden discernir) y entonces cada clave es buscada individualmente para demostrar que el mecanismo de búsqueda esta trabajando adecuadamente.

## Almacenando referencias

La librería **java.lang.ref** contiene un grupo de clases que permiten una gran flexibilidad en la recolección de basura, lo que es especialmente útil cuando se tiene grandes objetos que pueden causar agotamiento de memoria. Hay tres clases heredadas de la clase abstracta **Reference**: **SoftReference**, **WeakReference**, y **PhantomReference**. Cada uno de estas proporcionan un nivel diferente de vía indirecta para el recolector de basura, si el objeto en cuestión *solo* se puede alcanzar a través de uno de estos objetos **Reference**.

Si un objeto es *accesible* esto significa que en alguna parte del programa el objeto puede encontrarse. Esto puede significar que se tiene una referencia común en la pila que llega correctamente al objeto, pero se puede querer también tener una referencia a un objeto que sea referencia al objeto en cuestión; pueden ser muchos enlaces intermedios. Si un objeto es accesible, el recolector de basura no lo liberará porque sigue en uso por el programa. Si un objeto no es accesible, no hay forma de que el programa lo use así es que es seguro recolectar la basura de ese objeto.

Se utilizan objetos **Reference** cuando se quiere continuar almacenando una referencia a ese objeto -se quiere ser capaz de alcanzar ese objeto- pero también se quiere permitir que el recolector de basura libere ese objeto. De esta manera, se tiene una forma de seguir utilizando el objeto, pero si la agotamiento de memoria es inminente se permite que el objeto sea liberado.

Se logra esto utilizando un objeto **Reference** como un intermediario entre el programador y la referencia común, y no debe haber referencias comunes a el objeto (una que no haya sido envuelta dentro de un objeto **Reference**). Si el recolector de basura descubre que un objeto es alcanzado por una referencia común, no liberará ese objeto.

En orden, **SoftReference**, **WeakReference**, y **PhantomReference**, cada uno es “mas débil” que el anterior, y corresponden a diferentes niveles de accesibilidad. Las referencias soft son implementaciones de ocultación en memoria susceptible. Las referencias weak son implementaciones de

mapeos canonicalizados -donde las instancias de los objetos pueden ser simultáneamente utilizados en varios lugares en un programa, para ahorrar almacenamiento- esto no impide que sus claves (o valores) sean reclamados. Las referencias phantom son para fijar la hora de acciones de limpieza pre-mortem en una forma mas flexible de la posible con el mecanismo de finalización de Java.

Mediante **SoftReferences** y **WeakReferences**, se tiene la elección acerca de colocarlas en una **ReferenceQueue** (el dispositivo utilizado para acciones de limpieza pre-mortem), pero una **PhantomReference** solo puede ser creada en una **ReferenceQueue**. Aquí hay una demostración simple:

```
//: c09:References.java
// Demuestra los objetos Reference
import java.lang.ref.*;
class VeryBig {
    static final int SZ = 10000;
    double[] d = new double[SZ];
    String ident;
    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    public void finalize() {
        System.out.println("Finalizing " + ident);
    }
}
public class References {
    static ReferenceQueue rq= new ReferenceQueue();
    public static void checkQueue() {
        Object inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " +
                (VeryBig)((Reference)inq).get());
    }
    public static void main(String[] args) {
        int size = 10;
        // 0, se elige el tamaño mediante la linea de comandos:
        if(args.length > 0)
            size = Integer.parseInt(args[0]);
        SoftReference[] sa =
            new SoftReference[size];
        for(int i = 0; i < sa.length; i++) {
            sa[i] = new SoftReference(
                new VeryBig("Soft " + i), rq);
            System.out.println("Just created: " +
                (VeryBig)sa[i].get());
            checkQueue();
        }
        WeakReference[] wa =
            new WeakReference[size];
        for(int i = 0; i < wa.length; i++) {
            wa[i] = new WeakReference(
                new VeryBig("Weak " + i), rq);
            System.out.println("Just created: " +
                (VeryBig)wa[i].get());
        }
    }
}
```

```

        checkQueue();
    }
    SoftReference s = new SoftReference(
        new VeryBig("Soft"));
    WeakReference w = new WeakReference(
        new VeryBig("Weak"));
    System.gc();
    PhantomReference[] pa =
        new PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
        pa[i] = new PhantomReference(
            new VeryBig("Phantom " + i), rq);
        System.out.println("Just created: " +
            (VeryBig)pa[i].get());
        checkQueue();
    }
}
} //:~

```

Cuando se ejecute este programa (se va a querer canalizar la salida a través de “mas” programas utilitarios así se puede ver la salida en páginas), se podrá ver que el objeto es recogido por el recolector de basura, aun si se sigue teniendo acceso a ellos a través de un objeto **Reference** (para obtener la referencia a el objeto actual, se utiliza **get()**). Se puede ver también que la **ReferenceQueue** siempre produce una **Reference** conteniendo un objeto **null**. Para hacer uso de esto, se puede heredar de una clase **Reference** particular lo que se esté interesado y agregar más métodos útiles a el nuevo tipo de **Reference**.

## El WeakHashMap

La librería de contenedores tiene un **Map** especial para almacenar referencias weak; el **WeakHashMap**. Esta clase está diseñada para hacer fácil la creación de mapeos canonicalizados. Es este tipo de mapeos, se está ahorrando almacenamiento haciendo solo una instancia del valor en particular. Cuando el programa necesita ese valor, busca el objeto existente en el mapeo y utiliza ese (en lugar de crear uno a los arañazos). El mapeo puede hacer a los valores parte de la inicialización, pero es mejor que los valores sean creados en demanda.

Dado que esto es una técnica de ahorro de almacenamiento, es muy conveniente que el **WeakHashMap** permita que el recolector de basura automáticamente limpie las claves y los valores. No se tiene que hacer nada especial para que las claves y los valores que se quieran sean colocados en el **WeakHashMap**: estos son automáticamente envueltos en **WeakReferences** por el mapeo. El gatillo que permite la limpieza es si la clave no está más en uso, como se demuestra aquí:

```

//: c09:CanonicalMapping.java
// Demonstrates WeakHashMap.

```

```

import java.util.*;
import java.lang.ref.*;
class Key {
    String ident;
    public Key(String id) { ident = id; }
    public String toString() { return ident; }
    public int hashCode() {
        return ident.hashCode();
    }
    public boolean equals(Object r) {
        return (r instanceof Key)
            && ident.equals(((Key)r).ident);
    }
    public void finalize() {
        System.out.println("Finalizing Key "+ ident);
    }
}
class Value {
    String ident;
    public Value(String id) { ident = id; }
    public String toString() { return ident; }
    public void finalize() {
        System.out.println("Finalizing Value "+ident);
    }
}
public class CanonicalMapping {
    public static void main(String[] args) {
        int size = 1000;
        // 0, se elige el tamaño mediante la línea de comandos:
        if(args.length > 0)
            size = Integer.parseInt(args[0]);
        Key[] keys = new Key[size];
        WeakHashMap whm = new WeakHashMap();
        for(int i = 0; i < size; i++) {
            Key k = new Key(Integer.toString(i));
            Value v = new Value(Integer.toString(i));
            if(i % 3 == 0)
                keys[i] = k; // Guardar como referencias "reales"
            whm.put(k, v);
        }
        System.gc();
    }
}

```

La clase **Key** debe tener un **hashCode()** y un **equals()** dado que son utilizados como claves en una estructura de datos de hash, como se describió previamente en este capítulo.

Cuando se ejecute el programa se podrá ver que el recolector de basura saltara cada tercer clave, porque una referencia común a esta clave ha sido colocada en el arreglo de claves y por lo tanto esos objetos no pueden ser recolectados.

# Revisión de iteradores

Podemos ahora demostrar el verdadero poder del **Iterator**: la habilidad de separar la operación de atravesar una secuencia desde la estructura de capas bajas de esa secuencia. En el siguiente ejemplo, la clase **PrintData** utiliza un **Iterator** para moverse a través de una secuencia y llamar el método **toString()** para cada objeto. Dos tipos diferentes de contenedores son creados -un **ArrayList** y un **HashMap**- y pueden ser llenados con, respectivamente objetos **Mouse** y **Hamster** (estas clases son definidas mas atrás en este capítulo). Dado que un **Iterator** oculta la estructura de las capas mas bajas del contenedor. **PrintData** no sabe o no le importa el tipo de contenedor donde se origina el **Iterator**.

```
//: c09:Iterators2.java
// Revisiting Iterators.
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**, lo que contiene la clave y el valor para cada entrada, así es que se puede ver ambos cuando son impresos.

Puede verse que **PrintData.print()** toma ventaja del hecho de que los objetos en estos contenedores son del la clase **Object** así es que la llamada a **toString()** hecha por **System.out.println()** es automática. Es mejor en el problemas, se debe asumir de que el **Iterator** esta atravesando un contenedor de algún tipo específico. Por ejemplo, se debe asumir que todo en el contenedor es una **Shape** con un método **draw()**. Entonces se realiza una conversión descendente de el **Object** que retorna **Iterator.next()** para producir una **Shape**.

# Eligiendo una implementación

Por ahora se debe entender que hay solo tres componentes contendores: **Map**, **List** y **Set**, y solo dos de tres implementaciones de cada interfase. Si se necesita utilizar la funcionalidad que entrega una **interface** particular.

¿Como se decide la implementación particular a utilizar?

Para entender la respuesta, se debe ser consciente que cada una de las diferentes implementaciones tienen sus propias características, fuerzas y debilidades. Por ejemplo, se puede ver en el diagrama que la “característica” de **Hashtable**, **Vector**, y **Stack** es que hay clases heredadas, así es que el viejo código no se romperá. Por otro lado, es mejor si no se utiliza estos en nuevo código (Java 2).

La distinción entre los otros contenedores a menudo provienen de por “quien está apoyados”; esto es, las estructuras de datos que físicamente implementan la **interface** deseada. Esto significa que, por ejemplo, **ArrayList** y **LinkedList** implementan la interfase **List** así es que el programa producirá el mismo resultado independientemente del que se utilice.

Sin embargo, **ArrayList** es secundado por un arreglo, mientras que **LinkedList** es implementada en la forma usual para una lista doblemente enlazada, con objetos individuales cada uno contenido los datos junto con las referencias a el anterior y al siguiente elemento de la lista. A causa de esto, si se quiere hacer muchas inserciones y eliminaciones den el medio de la lista, una **LinkedList** es la elección apropiada (**LinkedList** también tiene funcionalidades adicionales que son establecidas en una **AbstractSequentialList**). Si no fuera así, una **ArrayList** es típicamente mas rápida.

Otro ejemplo, un **Set** puede ser implementado como un **TreeSet** o como un **HashSet**. Un **TreeSet** esta apoyado por un **TreeMap** y esta diseñado para producir un grupo constantemente ordenado. Sin embargo, si se va a tener **Set** muy grandes, el rendimiento de agregar en un **TreeSet** serán lentos. Cuando se esta escribiendo un programa que necesita un **Set**, se debe elegir **HashSet** por defecto, y cambiar a **TreeSet** cuando sea mas importante tener un grupo constantemente ordenado.

## Elección entre listas

La forma mas convincente de ver las diferencias entre las implementaciones de **List** es con una prueba de rendimiento. El siguiente código establece una

clase base interna para utilizar como framework de prueba, entonces se crea un arreglo de clases internas anónimas, una para cada grupo diferente. Cada una de estas clases internas es llamada por el método **test()**. Esta estrategia permite fácilmente agregar y quitar nuevos tipos de pruebas.

```
//: c09>ListPerformance.java
// Demonstrates performance differences in Lists.
import java.util.*;
import com.bruceekel.util.*;
public class ListPerformance {
    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, int reps);
    }
    private static Tester[] tests = {
        new Tester("get", 300) {
            void test(List a, int reps) {
                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, int reps) {
                for(int i = 0; i < reps; i++) {
                    Iterator it = a.iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
        new Tester("insert", 5000) {
            void test(List a, int reps) {
                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, int reps) {
                ListIterator it = a.listIterator(3);
                while(it.hasNext()) {
                    it.next();
                    it.remove();
                }
            }
        }
    }
}
```

```

        },
    };
    public static void test(List a, int reps) {
        // A trick to print out the class name:
        System.out.println("Testing " +
            a.getClass().getName());
        for(int i = 0; i < tests.length; i++) {
            Collections2.fill(a,
                Collections2.countries.reset(),
                tests[i].size);
            System.out.print(tests[i].name);
            long t1 = System.currentTimeMillis();
            tests[i].test(a, reps);
            long t2 = System.currentTimeMillis();
            System.out.println(": " + (t2 - t1));
        }
    }
    public static void testArray(int reps) {
        System.out.println("Testing array as List");
        // Can only do first two tests on an array:
        for(int i = 0; i < 2; i++) {
            String[] sa = new String[tests[i].size];
            Arrays2.fill(sa,
                Collections2.countries.reset());
            List a = Arrays.asList(sa);
            System.out.print(tests[i].name);
            long t1 = System.currentTimeMillis();
            tests[i].test(a, reps);
            long t2 = System.currentTimeMillis();
            System.out.println(": " + (t2 - t1));
        }
    }
    public static void main(String[] args) {
        int reps = 50000;
        // Or, choose the number of repetitions
        // via the command line:
        if(args.length > 0)
            reps = Integer.parseInt(args[0]);
        System.out.println(reps + " repetitions");
        testArray(reps);
        test(new ArrayList(), reps);
        test(new LinkedList(), reps);
        test(new Vector(), reps);
    }
} //:~

```

La clase interna **Tester** es **abstract**, para proporcionar una clase base para la prueba específica. Contiene un **String** que será impreso cuando las pruebas comiencen, un parámetro **size** que será utilizado para probar la calidad de los elementos o repeticiones de las pruebas, un constructor para inicializar los campos, y un método abstracto **test()** que hace el trabajo. Todos los diferentes tipos de pruebas son recolectadas en un lugar, los arreglos **test**, que son inicializados con las diferentes clases anónimas que heredan de **Tester**. Para agregar o quitar tareas, simplemente agregue o quite una

definición de clase interna del arreglo, y todo lo demás se sucede automáticamente.

Para comparar el acceso de los arreglos a el acceso a los contenedores (inicialmente contra **ArrayList**), una prueba especial es creada para arreglos envolviéndolo como una **List** utilizando **Arrays.asList()**. Puede verse que solo las dos primeras pruebas pueden ser realizadas en este caso, dado que no se puede insertar o quitar elementos de un arreglo.

La **List** que es manejada para **test()** es primero llenada con elementos, luego a cada prueba en los arreglos se le mide el tiempo. El resultado puede variar de máquina a máquina, estos pretenden dar solo un orden de magnitud para realizar una comparación entre el rendimiento de los diferentes contenedores. He aquí un resumen de una corrida:

Tipo	Extraer	Iteración	Insertar	Eliminar
arreglo	1430	3850	na	na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Como se esperaba, los arreglos son mas rápidos que cualquier contenedor para acceso aleatorios, búsquedas e iteración. Se puede ver que los acceso aleatorios (**get()**) son económicos para **ArrayLists** y costosos para **LinkedLists** (Curiosamente, la iteración es *mas rápida* para una **LinkedList** que para una **ArrayList**, lo que va un poco en contra de la intuición). Por el otro lado, las inserciones y las eliminaciones del medio de una lista son dramáticamente mas económicas para una **LinkedList** que para una **ArrayList** -especialmente las eliminaciones. **Vector** generalmente no es tan rápido como una **ArrayList**, y puede ser evitado; no solo en la librería para soporte de código heredado (la única razón por lo que funciona en este programa es porque fue adaptado para ser una **List** en Java 2). La mejor estrategia es probablemente elegir una **ArrayList** por defecto, y cambiar a una **LinkedList** si se descubren problemas de rendimiento debido a muchas inserciones y eliminaciones del medio de la lista. Claro, si se está trabajando con un grupo de elementos de tamaño fijo, utilice un arreglo.

## Eligiendo entre Sets

Se puede elegir entre un **TreeSet** y un **HashSet**, dependiendo de el tamaño del **Set** (si se necesita producir una secuencia ordenada de un **Set**, se debe utilizar un **TreeSet**). el siguiente programa de prueba da una indicación de este cambio:

```
| //: c09:SetPerformance.java
```

```

import java.util.*;
import com.bruceekel.util.*;
public class SetPerformance {
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Set s, int size, int reps);
    }
    private static Tester[] tests = {
        new Tester("add") {
            void test(Set s, int size, int reps) {
                for(int i = 0; i < reps; i++) {
                    s.clear();
                    Collections2.fill(s,
                        Collections2.countries.reset(),size);
                }
            }
        },
        new Tester("contains") {
            void test(Set s, int size, int reps) {
                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, int reps) {
                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, int reps) {
        System.out.println("Testing " +
            s.getClass().getName() + " size " + size);
        Collections2.fill(s,
            Collections2.countries.reset(), size);
        for(int i = 0; i < tests.length; i++) {
            System.out.print(tests[i].name);
            long t1 = System.currentTimeMillis();
            tests[i].test(s, size, reps);
            long t2 = System.currentTimeMillis();
            System.out.println(": " +
                ((double)(t2 - t1)/(double)size));
        }
    }
    public static void main(String[] args) {
        int reps = 50000;
        // O elija el número de repeticiones repetitions
        // mediante la linea de comandos:
        if(args.length > 0)

```

```

        reps = Integer.parseInt(args[0]);
        // Pequeño:
        test(new TreeSet(), 10, reps);
        test(new HashSet(), 10, reps);
        // Mediano:
        test(new TreeSet(), 100, reps);
        test(new HashSet(), 100, reps);
        // Grande:
        test(new TreeSet(), 1000, reps);
        test(new HashSet(), 1000, reps);
    }
} //:~

```

La siguiente tabla muestra el resultado de una corrida (Claro, esto puede ser de acuerdo a la computadora y a la JVM que se esté utilizando; todos debería correr la prueba):

Tipo	Tamaño de la Prueba	Agregador	Contenedores	Iteración
TreeSet	10	1 3 8. 0	115.0	187 .0
	100	1 8 9. 2	151.1	20 6.5
	1000	1 5 0 .6	177.4	40. 04
HashSet	10	5 5	82.0	192 .0
	100	4 5	90.0	20 2.2
	1000	3 6	106.5	39. 39

El rendimiento de **HashSet** es generalmente superior a **TreeSet** para todas las operaciones (pero en particular agregando y buscando, las operaciones mas importantes). La única razón por la que **TreeSet** existe es porque

mantiene sus elementos en orden, así es que solo debe utilizarse cuando se necesite un grupo ordenado.

## Eligiendo entre Maps

Cuando se elige entre implementaciones de **Map**, el tamaño del **Map** es lo que mas fuertemente afecta el rendimiento, y el siguiente programa de prueba da una indicación de este intercambio:

```
//: c09:MapPerformance.java
// Demuestra los diferentes rendimientos en Maps.
import java.util.*;
import com.bruceeeckel.util.*;
public class MapPerformance {
    private abstract static class Tester {
        String name;
        Tester(String name) { this.name = name; }
        abstract void test(Map m, int size, int reps);
    }
    private static Tester[] tests = {
        new Tester("put") {
            void test(Map m, int size, int reps) {
                for(int i = 0; i < reps; i++) {
                    m.clear();
                    Collections2.fill(m,
                        Collections2.geography.reset(), size);
                }
            }
        },
        new Tester("get") {
            void test(Map m, int size, int reps) {
                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, int reps) {
                for(int i = 0; i < reps * 10; i++) {
                    Iterator it = m.entrySet().iterator();
                    while(it.hasNext())
                        it.next();
                }
            }
        },
    };
    public static void
    test(Map m, int size, int reps) {
        System.out.println("Testing " +
            m.getClass().getName() + " size " + size);
        Collections2.fill(m,
            Collections2.geography.reset(), size);
        for(int i = 0; i < tests.length; i++) {
```

```

        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(m, size, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
            ((double)(t2 - t1)/(double)size));
    }
}
public static void main(String[] args) {
    int reps = 50000;
    // O, elija el número de repeticiones
    // mediante la linea de comandos:
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);
    // Pequeño:
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);
    // Medio:
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);
    // Grande:
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
}
} //:~

```

Dado que el tamaño del mapa es el tema, se verá que las pruebas de cronometraje dividen el tiempo por el tamaño para normalizar cada medida. Aquí hay un grupo de resultados (correr el programa probablemente entregue valores diferentes).

Tipo	Tamaño de la prueba	Poner	Tomar	Iteración
TreeMap	10	143.0	110.0	186.0
	100	201.1	188.4	280.1
	1000	222.8	205.2	40.7
HashMap	10	66.0	83.0	197.7
	100	80.7	135.7	278.5
	1000	48.2	105.7	41.4
Hashtable	10	61.0	93.0	302.0
	100	90.6	143.3	329.0
	1000	54.1	110.95	47.3

Como se podía esperar, el rendimiento de **Hashtable** es mas o menos el equivalente a **HashMap** (se puede ver también que **HashMap** generalmente es un poquito mas rápido. **HashMap** pretende remplazar **Hashtable**).

¿**TreeMap** es mas lento generalmente que **HashMap**, así es que por que se utiliza? Así es que no se debería utilizar como un **Map**, sino como una forma de crear una lista ordenada. El comportamiento de un árbol es tal que siempre esta en orden y no tiene por que estar especialmente clasificado.

Una vez que se llena un **TreeMap**, se puede llamar a **keySet()** para obtener una vista del **Set** para estas claves, luego **toArray()** para producir un arreglo con estas claves. Se puede utilizar entonces el método estático

**Arrays.binarySearch()** (discutido mas adelante) para encontrar objetos rápidamente en el arreglo ordenado. Claro, que se debería hacer esto si, por alguna razón, el comportamiento de un **HashMap** es inaceptable, dado que **HashMap** esta diseñado para encontrar cosas rápidamente. También, se puede fácilmente crear un **HashMap** de un **TreeMap** con la creación de un solo objeto al final, cuando se esta utilizando un **Map** la primera elección suele ser un **HashMap**, y luego si se necesita un **Map** continuamente ordenado se necesitará un **TreeMap**.

## Ordenando y explorando Lists

Los utilitarios para realizar ordenamientos y búsquedas en **Lists** tienen los mismos nombres y firmas que aquellos para realizar búsquedas de arreglos de objetos, pero son métodos estáticos de **Collections** en lugar de **Arrays**. He aquí un ejemplo, modificado de **ArraySearching.java**:

```
///: c09>ListSortSearch.java
// Sorting and searching Lists with 'Collections.'
import com.bruceeckel.util.*;
import java.util.*;
public class ListSortSearch {
    public static void main(String[] args) {
        List list = new ArrayList();
        Collections2.fill(list,
        Collections2.capitals, 25);
        System.out.println(list + "\n");
        Collections.shuffle(list);
        System.out.println("After shuffling: "+list);
        Collections.sort(list);
        System.out.println(list + "\n");
        Object key = list.get(12);
        int index =
            Collections.binarySearch(list, key);
        System.out.println("Location of " + key +
            " is " + index + ", list.get(" +
```

```

        index + ") = " + list.get(index));
    AlphabeticComparator comp =
        new AlphabeticComparator();
    Collections.sort(list, comp);
    System.out.println(list + "\n");
    key = list.get(12);
    index =
        Collections.binarySearch(list, key, comp);
    System.out.println("Location of " + key +
        " is " + index + ", list.get(" +
        index + ") = " + list.get(index));
}
} //:~

```

El uso de estos métodos es idéntico a los de **Arrays**, pero se está utilizando una **List** en lugar de un arreglo. Exactamente como explorando y ordenando con arreglos, si se ordena utilizando un **Comparator** se debe realizar una **binarySearch()** utilizando el mismo **Comparator**.

Este programa también demuestra el método **shuffle()** en **Collections**, que desordena el orden de una **List**.

## Utilitarios

Hay algunas otras utilidades útiles en la clase **Collections**:

enumeration(Collection)	Produce una <b>Enumeration</b> del viejo estilo para el argumento.
max(Collection) min(Collection)	Produce el máximo o mínimo elemento en el argumento utilizando el método natural de comparación de los objetos en la <b>Collection</b> .
max(Collection, Comparator) min(Collection, Comparator)	Produce el máximo o el mínimo elemento en la <b>Collection</b> utilizando el <b>Comparator</b> .
reverse()	Invierte de lugar todos los elementos.
copy(List destino, List origen)	Copia elementos de origen a destino.

fill(List lista, Object o)	Reemplaza todos los elementos de una lista por o.
nCopies(int n, Object o)	Retorna una <b>List</b> inmutable de tamaño n cuyas referencias apuntan todas a o.

Debe notarse que **min()** y **max()** trabajan con una objetos **Collection**, no con **Lists**, así es que no necesita preocuparse por cuando la **Collection** debe ser ordenada o no (Como se ha mencionado antes, se *necesita* realizar un **short()** (ordenar) una **List** o un arreglo antes de realizar una **binarySearch()**).

## Haciendo una **Collection** o un **Map** inmodificable

A menudo es conveniente crear una versión de solo lectura de una **Collection** o un **Map**. La clase **Collection** permite hacer esto pasando el contenedor original dentro de un método que retorne una versión de solo lectura. Hay cuatro variaciones de este método, una para cada **Collection** (si no se quiere manejar una **collection** de un tipo mas específico), **List**, **Set**, y **Map**. Este ejemplo muestra la forma adecuada de crear versiones solo lectura de cada una:

```
//: c09:ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import com.bruceeckel.util.*;
public class ReadOnly {
    static Collections2.StringGenerator gen =
        Collections2.countries;
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Collections2.fill(c, gen, 25); // Insertar datos
        c = Collections.unmodifiableCollection(c);
        System.out.println(c); // Lectura OK
        c.add("one"); // No se puede cambiar
        List a = new ArrayList();
        Collections2.fill(a, gen.reset(), 25);
        a = Collections.unmodifiableList(a);
        ListIterator lit = a.listIterator();
        System.out.println(lit.next()); // Lectura OK
        lit.add("one"); // No se puede cambiar
        Set s = new HashSet();
        Collections2.fill(s, gen.reset(), 25);
        s = Collections.unmodifiableSet(s);
        System.out.println(s); // Reading OK
```

```

    //! s.add("one"); // No se puede cambiar
Map m = new HashMap();
Collections2.fill(m,
    Collections2.geography, 25);
m = Collections.unmodifiableMap(m);
System.out.println(m); // Lectura OK
//! m.put("Ralph", "Howdy!");
}
} //:~

```

En cada caso, se debe llenar el contenedor con datos significativos *antes* de hacerlo de solo lectura. Una vez que es cargada, la mejor estrategia es reemplazar la referencia existente con la referencia que fue producido por la llamada a el “inmodificable”. De esta forma, no se corre el riesgo de accidentalmente cambiar el contenido una vez que se ha hecho inmodificable. Por el otro lado, esta herramienta también permite mantener un contenedor inmodificable como privada dentro de una clase y retornar una referencia de solo lectura como contenedor de una llamada a método. Así es que se puede cambiar esta dentro de la clases, pero cualquier otro solo puede leerla.

Llamando a el método “inmodificable” para un tipo en particular no causa una verificación en tiempo de compilación, pero una vez que la transformación ha sucedido, cualquier llamada a métodos que modifican el contenido de un contenedor en particular producirán una **UnsupportedOperationException**.

## Sincronizando una **Collection** o un **Map**

La palabra clave **synchronized** es una parte importante del tema de *hilos múltiples*, un tema mas complicado que no será introducido hasta el capítulo 14. Aquí, veremos solamente que la clase **Collections** contiene una forma para sincronizar automáticamente un contenedor entero. La sintaxis es similar a los métodos “inmodificable”.

```

//: c09:Synchronization.java
// Using the Collections.synchronized methods.
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, inmediatamente se pasa el nuevo contenedor a través del método “synchronized”; de esta forma no hay posibilidad de accidentalmente exponerlo a una versión no sincronizada.

## Falla rápida

Los contenedores de Java tienen también un mecanismo para prevenir que mas de un proceso modifique el contenido de un contenedor. El problema sucede si se esta iterando a través de un contenedor y algún otro proceso interviene e inserta, borra o cambia un objeto en el contenedor. Tal vez ya se haya pasado ese objeto, tal vez este adelante, tal vez el tamaño del contenedor se contrae antes de que se llame a **size()** -hay muchos escenarios para el desastre. La librería de contenedores de Java incorpora un mecanismo de *falla-rápida* que mira cualquier otro cambio en el contenedor del cual no sea personalmente responsable el proceso. Detecta si alguien mas está modificando el contenedor, e inmediatamente produce un **ConcurrentModificationException**. Este es el aspecto “falla rápida” -no trata de detectar un problema mas tarde utilizando un algoritmo mas complejo.

Es fácil de ver el mecanismo de falla rápida en operación -todo lo que se tiene que hacer es crear un iterator y luego agregar algo a la colección al que el iterator esta apuntando, de la siguiente forma:

```
///: c09:FailFast.java
// Demonstrates the "fail fast" behavior.
import java.util.*;
public class FailFast {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Iterator it = c.iterator();
        c.add("An object");
        // Causes an exception:
        String s = (String)it.next();
    }
} //:/~
```

La excepción se sucede porque alguien esta colocando algo en el contenedor *luego* que el iterator es obtenido por el contenedor. La posibilidad que dos partes del programa puedan modificar el mismo contenedor produce un estado de incertidumbre, así es que la excepción notifica que se debe cambiar su código -en este caso, obtener el iterator *después* de que se han agregado todos los elementos al contenedor.

Debe notarse que no se puede beneficiar de este tipo de monitoreo cuando se esta accediendo a los elementos de una **List** utilizando **get()**.

# Operaciones no soportadas

Es posible convertir un arreglo en una **List** con el método **Array.asList()**:

```
//: c09:Unsupported.java
// Algunos métodos definidos en la interfase
// de Collection no funcionan!
import java.util.*;
public class Unsupported {
    private static String[] s = {
        "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten",
    };
    static List a = Arrays.asList(s);
    static List a2 = a.subList(3, 6);
    public static void main(String[] args) {
        System.out.println(a);
        System.out.println(a2);
        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]));
        // Atravesar a través hacia atrás:
        ListIterator lit = a.listIterator(a.size());
        while(lit.hasPrevious())
            System.out.print(lit.previous() + " ");
        System.out.println();
        // Grupo de elementos de diferente valor:
        for(int i = 0; i < a.size(); i++)
            a.set(i, "47");
        System.out.println(a);
        // Compila pero no corre:
        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
    }
} ///:~
```

Se descubrirá que una parte de la interfase de la **Collection** y la **List** son implementadas actualmente. El resto de los método causan la indeseada venida de algo llamado **UnsupportedOperationException**. Se aprenderá todo acerca de excepciones en el siguiente capítulo, pero la historia corta es que

la interfase **Collection** -al igual que algunas de las otras interfases en la librería de contenedores de Java- tienen métodos “opcionales”, que pueden o pueden no ser “soportados” en la clase concreta que implementa esa interfase. Llamar a un método no soportado produce una **UnsupportedOperationException** para indicar un error de programación.

“¿¡¿Que?!?” se puede decir, incrédulamente. “!El punto mas importante de las interfases o las clases base es que prometen esos métodos harán algo significativo! ¡Esto rompe esa promesa -ellos dicen que no solo llamando algunos métodos *no* se tendrá un comportamiento significante, detendrán el programa! ¡La seguridad de tipo simplemente es tirada por la ventana!”

No es tan malo. Con una **Collection**, **List**, o **Map**, el compilador a pesar de ello restringe el llamar solo los métodos en esa interfase, así es que no es como Smalltalk (en donde se puede llamar cualquier método para cualquier objeto, y tener respuesta solo cuando se ejecuta un programa donde su llamada hace algo). Además, muchos métodos que toman una **Collection** como argumentos solo leen de esa **Collection** -todos los métodos que “leen” de **Collection** *no* son opcionales.

Esta estrategia previene una explosión de interfases en el diseño. Otros diseños para las librerías de contenedores siempre parecen terminar con un confuso exceso de interfaces para describir cada una de las variantes del tema principal y son de esta forma difíciles de aprender. Esto no siempre es posible para tomar todos los casos especiales de interfaces, dado que alguien puede siempre inventar una nueva interfase. La estrategia de la “operación no soportada” alcanza una importante meta en la librería de contenedores de Java, el contenedor es simple de aprender y utilizar; las operaciones no soportadas son un caso especial que pueden ser aprendidas mas adelante. Para que esta estrategia funcione, sin embargo:

1. La **UnsupportedOperationException** debe ser un evento raro. Esto es, para la mayoría de las clases todas las operaciones deben funcionar, y solo en casos especiales debe una operación no ser soportada. Esto es verdad en las librerías de contenedores de Java, en las clases que se utilizan el 99 por ciento de las veces -**ArrayList**, **LinkedList**, **HashSet**, y **HashMap**, de la misma forma que otras implementaciones en concreto- soportan todas las operaciones. El diseño proporciona una “puerta trasera” si se quiere crear una nueva **Collection** sin proporcionar definiciones significativas para todos los métodos en la interfase de la **Collection**, y encajar igual en la librería existente.
2. Cuando una operación no es soportada, debería ser razonable la posibilidad de que una **UnsupportedOperationException** apareciera en tiempo de implementación, en lugar de que sucediera luego de enviado el producto al cliente. Después de todo, esta indica un error de programación: se está utilizando una implementación de forma

incorrecta. Este punto es menos cierto, y es donde la naturaleza experimental de este diseño comienza a jugar. Solo en horas extras se encontrara que tan bien trabaja esto.

En el ejemplo anterior, **Arrays.asList()** produce una **List** que es respaldada por un arreglo de tamaño fijo. Sin embargo tiene sentido que solo soporte operaciones que no cambien el tamaño del arreglo. Si, por el otro lado, una nueva interfase fue requerida para expresar este tipo diferente de comportamiento (llamada, tal vez, “**FixedSizeList**”), dejará una puerta abierta a la complejidad y pronto no se sabrá donde comenzar cuando se intente utilizar la librería.

La documentación para un método que toma una **Collection**, **List**, **Set**, o **Map** como argumento especificará que métodos opcionales deben ser implementados. Por ejemplo, ordenar requiere los métodos **set()** e **Iterator.set()**, pero no **add()** y **remove()**.

## Contenedores de Java 1.0/1.1

Desafortunadamente un montón de código fue escrito utilizando contendores de Java 1.0/1.1, y a aún nuevo código es a veces escrito utilizando estas clases. Así es que a pesar nunca se utilice los viejos contendores cuando se escriba nuevo código, se debe estar concientes de estos. Sin embargo, los viejos contendores son muy limitados, así es que no hay mucho para saber de ellos (Dado que ellos son pasado, me trataré de refrenar de darle demasiada importancia a las horribles decisiones de diseño).

### Vector y Enumeración

La única secuencia que se explicaba sola de Java 1.0/1.1 era el **Vector**, y de esta forma se utilizaba mucho. Sus defectos eran muchos para describir aquí (vea la primera edición de este libro, disponible en el CD-ROM y en [www.BruceEckel.com](http://www.BruceEckel.com) libremente). Básicamente, se puede pensar en esto como un **ArrayList** con lagos y complicados nombres de métodos. En la librería de contendores de Java 2, **Vector** fue adaptado de tal forma que se puede encontrar como una **Collection** y una **List**, así es que en el siguiente ejemplo el método **collections2.fill()** es utilizado exitosamente. Esto es un poco perverso, así como puede confundir algunas personas y hacerlas pensar que **Vector** se ha mejorado, cuando es incluido actualmente para poder soportar código anterior a Java 2.

Se decidió por elegirle un nuevo nombre a la versión de Java 1.0/1.1 del iterator, “enumeración”, en lugar de utilizar un termino que para todos ya era familiar. La interfase **Enumeration** es mas pequeña que la de **Iterator**, con solo dos métodos, utiliza nombres de métodos muy grandes: **boolean hasMoreElements()** produce un **true** si la enumeración contiene mas elementos, y **Object nextElement()** retorna el siguiente elemento de esta enumeración si hay mas (de otra forma lanza una excepción).

**Enumeration** es solo una interfase, no una implementación, y aun nuevas librerías a veces utilizan la vieja **Enumeration** -que es desafortunadamente pero generalmente inofensiva. Aún cuando se debería utilizar siempre **Iterator** cuando se pueda en el código, se debe estar preparado para librerías que quieren utilizar una **Enumeration**.

Además, se puede producir una **Enumeration** para cada **Collection** utilizando el método **Collections.enumeration()**, como se ha visto en este ejemplo:

```
///: c09:Enumerations.java
// Java 1.0/1.1 Vector y Enumeration.
import java.util.*;
import com.bruceeeckel.util.*;
class Enumerations {
    public static void main(String[] args) {
        Vector v = new Vector();
        Collections2.fill(
            v, Collections2.countries, 100);
        Enumeration e = v.elements();
        while(e.hasMoreElements())
            System.out.println(e.nextElement());
        // Produce una Enumeration de una Collection:
        e = Collections.enumeration(new ArrayList());
    }
} ///:~
```

El **Vector** de Java 1.0/1.1 tiene solo un método **addElement()**, pero **fill()** utiliza el método **add()** que incorporado en **Vector** cuando se convirtió en una **List**. Para producir una **Enumeration**, se debe llamar a **elements()**, luego se puede utilizar para realizar una iteración hacia adelante.

La última línea crea un **ArrayList** y utiliza **enumeration()** para adaptar una **Enumeration** de un **ArrayList Iterator**. De esta manera, si se tiene código viejo que necesita una **Enumeration**, se puede seguir utilizando los nuevos contenedores,

## Hashtable

Como se ha visto en las comparaciones de rendimiento en este capítulo, la **Hashtable** básica es muy similar a la **HashMap**, aún en los nombres de

métodos. No hay razón para utilizar **Hashtable** en lugar de **HashMap** en el nuevo código.

## Stack

El concepto de pila fue introducido temprano, con la **LinkedList**. Lo que es realmente extraño con el **Stack** de Java 1.0/1.1 es que en lugar de utilizar un **Vector** como un bloque para construirlo, **Stack** es *heredado* de **Vector**. Así es **Stack** que tiene todas las características y comportamientos de un **Vector** mas algunos comportamientos extras de **Stack**. Esto es difícil de saber si los diseñadores explicitamente decidieron que era una forma útil de hacer las cosas, o si simplemente es un diseño sencillo.

Aquí hay una demostración simple de **Stack** que empuja las líneas de un arreglo de **String**:

```
//: c09:Stacks.java
// Demonstracion de la clase Stack.
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);
        // Tratando un stack como un 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 en el arreglo **months** es insertada dentro de **Stack** con **push()**, y luego traído de la parte superior de la pila con un **pop()**. Para aclarar el asunto, las operaciones de **Vector** son también realizadas por en el objeto **Stack**. Esto es posible porque, por virtud de la herencia, un **Stack** es un **Vector**. De esta manera, todas las operaciones que pueden ser realizadas en un **Vector**, pueden también ser realizadas en un **Stack**, como **elementAt()**.

Como mencionamos antes, de debe utilizar una **LinkedList** cuando se quiere el comportamiento de una pila.

## BitSet

Un **Bitset** es utilizado si se quiere almacenar eficientemente un montón de información de llaves. Es eficiente solo desde el punto de vista del tamaño, si se esta buscando un acceso eficiente, es ligeramente mas lento que utilizar un arreglo de algún tipo nativo.

Además, el tamaño mínimo para un **BitSet** es el de un **long**: 64 bits. Esto implica que si se esta almacenando algo menor, como 8 bits, un **BitSet** será antieconómico; es mejor crear su propia clase, o simplemente un arreglo, para almacenar sus banderas si el tamaño es el tema.

Un contenedor normal se expande a medida que se van agregando elementos, y el **BitSet** hace esto de la misma forma. El siguiente ejemplo muestra como el **BitSet** trabaja:

```
// Demonstración de BitSet.
import java.util.*;
public class Bits {
    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);
    }
    public static void main(String[] args) {
        Random rand = new Random();
        // Toma el LSB del 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);
        // Prueba 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);
        b1023.set(1023);
        b1023.set(1024);
        System.out.println("set bit 1023: " + b1023);
    }
} //:~

```

El generador de números aleatorios es utilizado para crear números **byte**, **short**, y **int** aleatorios, y cada uno es transformado en su correspondiente patrón en bits en el **BitSet**. Esto trabaja bien porque un **BitSet** son 64 bits, así es que ninguno de estos produce un incremento en tamaño. Entonces un **BitSet** de 512 bits es creado. El constructor asigna espacio para el doble de bits. Sin embargo, se pueden configurar 1024 o mas.

## Resumen

Para repasar los contenedores proporcionados por la librería estándar de Java:

1. Un arreglo asocia índices numéricos a objetos. Este almacena objetos de un tipo conocido así es que no se tiene que realizar una conversión del resultado cuando se está buscando un objeto. Este puede ser multidimensional, y puede almacenar primitivas. Sin embargo, el tamaño no puede ser cambiado una vez que se ha creado.
2. Una **Collection** almacena elementos simples, mientras que un **Map** almacena pares asociados.
3. Como un arreglo, una **List** también asocia índices numéricos a objetos -se puede pensar en arreglos y **Lists** como contenedores ordenados. La **List** automáticamente cambia el tamaño sola cuando se agregan mas elementos. Pero una **List** solo puede almacenar referencias a objetos, así es que no almacena primitivas y se debe siempre realizar una conversión del resultado cuando se extraiga una referencia a un **Objeto** de un contenedor.
4. Use un **ArrayList** si esta haciendo una gran cantidad de accesos aleatorios, y una **LinkedList** si se esta haciendo una gran cantidad de inserciones y eliminaciones en el medio de la lista.

5. El comportamiento de las colas, deques, y pilas es proporcionado por la **LinkedList**.
6. Un **Map** es la forma de asociar no números, *objetos* con otros objetos. El diseño de un **HashMap** esta enfocado en el acceso rápido, donde un **TreeMap** tiene sus claves en orden, y de esta manera no es tan rápido como un **HashMap**.
7. Un **Set** solo acepta uno de cada tipo de objeto. **HashSets** proporciona las búsquedas mas rápidas, mientras que **TreeSets** mantiene los elementos en orden.
8. No hay necesidad de utilizar las clases heredadas **Vector**, **Hashtable** y **Stack** en el nuevo código.

Los contenedores son herramientas que se pueden utilizar día a día para hacer sus programas mas simples, mas poderosos y mas efectivos.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide*, disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Cree un arreglo de **double** y llénelo utilizando **fill()** utilizando **RandDoubleGenerator**. Imprima el resultado.
2. Cree una nueva clase **Roedor** con un **int numeroDeRoedor** que es inicializado en el constructor (similar a el ejemplo **Mouse** en este capítulo). Dele un método llamado **salte()** que imprima que numero de roedor es, y lo que esta saltando. Cree un **ArrayList** y agregue algunos objetos **Roedor** a la **List**. Ahora use el método **get()** para moverse a través de la **List** y llame **salte()** para cada **Gerbil**.
3. Modifique el ejercicio 2 de tal manera de que se use un **Iterator** para moverse a través de la **List** mientras se llama **salte()**.
4. Tome la clase **Roedor** en el ejercicio 2 y colóquela dentro de un **Map**, asocie el nombre del **Roedor** con un **String** (la clave) para cada **Roedor** (el valor) que se coloca en la tabla. Obtenga un **Iterator** para el **ketSet()** y úselo para moverse a través del **Map**, busque el **Roedor** para cada clave e imprime la clave e indíquele al **Roedor** que **salte()**.
5. Crea una **List** (intente con ambas, **ArrayList** y **LinkedList**) y llénela utilizando **collections2.countries**. Ordene la lista e imprimala, luego aplique **Collections.shuffle()** a la lista repetidamente, imprimala cada vez así puede ver como el método **shuffle()** desordena la lista de forma diferente cada vez.

6. Demuestre que no puede agregar nada mas que un **Mouse** a la **MouseListener**.
7. Modifique **MouseListener.java** de tal forma que se herede de **ArrayList** en lugar de utilizar composición. Demuestre el problema con esta estrategia.
8. Repare **CatsAndDogs.java** creando un contenedor **Cats** (utilizando una **ArrayList**) que no solo acepte y recupere objetos **Cat**.
9. Cree un contendor que encapsule un arreglo de **String**, y que solo agregue **String** y se obtengan **Strings**, de tal forma que no haya temas de conversiones mientras se utiliza. Si el arreglo interno no es suficientemente grande para cada agregado, su contendor debe automáticamente cambiar de tamaño. En **main()**, compare el rendimiento de su contenedor con un **ArrayList** almacenando **Strings**.
10. Repita el ejercicio 9 para un contendor de **int**, y compare el rendimiento con un **ArrayList** almacenando objetos **Integer**. En su comparación de rendimiento, incluya el proceso de incrementar cada objeto en el contenedor.
11. Utilice las utilidades en **com.bruceeckel.util**, cree un arreglo con cada tipo primitivo y con **String**, luego llene cada arreglo utilizando un generador apropiado, e imprima cada arreglo utilizando el método **print()** apropiado.
12. Cree un generador que produzca nombres de caracteres para su película favorita (se puede utilizar *Snow White* o **Star Wars** como un sistema de soporte), y regresar al principio cuando se salga de los nombres. Utilice las utilidades en **com.bruceeckel.util** para llenar un arreglo, un **ArrayList**, un **LinkedList** y los dos tipos de **Set**, luego imprima cada contenedor.
13. Cree una clase conteniendo dos objetos **String**, y hágala **Comparable** de tal forma que a la comparación solo le importe el primer **String**. Llene un arreglo y una **ArrayList** con objetos de su clase, utilice el generador **geography**. Demuestre que ordena palabra adecuadamente. Ahora cree un **Comparator** que solo le importe el segundo **String** y demuestre que ordena palabras adecuadamente; también realice una búsqueda binaria utilizando su **Comparator**.
14. Modifique el ejercicio 13 de tal forma que un ordenamiento alfabético sea utilizado.
15. Use **Array2.RandStringGenerator** para llenar un **treeSet** pero utilizando un ordenamiento alfabético. Imprima el **TreeSet** para verificar el orden.

16. Cree una **ArrayList** y una **LinkedList**, y llene cada una utilizando el generador **Collection2.capitals**. Imprima cada lista utilizando un **Iterator** común, luego inserte una lista dentro de la otra utilizando un **ListIterator**, insertando en cada una de las otras ubicaciones. Ahora realice la inserción comenzando desde el final del la primera lista y moviéndose hacia atrás.
17. Escriba un método que utilice un **Iterator** para ir a través de una **Collection** e imprima el **hashCode()** de cada objeto en el contenedor. Llene todos los diferentes tipos de **Collections** con objetos y aplique su método a cada contenedor.
18. Repare el problema en **InfiniteRecursion.java**.
19. Cree una clase, luego cree un arreglo de objetos inicializado de su clase. Llene una **List** de su arreglo. Cree un subgrupo de su **List** utilizando **subList()**, y luego elimine este subset de su **List** utilizando **removeAll()**.
20. Cambie el ejercicio 6 del capítulo 7 para utilizar un **ArrayList** para almacenar los **Rodents** y un **Iterator** para moverse a través de la secuencia de **Rodents**. Recuerde que una **ArrayList** almacena solo **Objects** así se debe realizar una conversión cuando se acceda a **Rodents** individuales.
21. Siga el ejemplo **Queue.java**, cree una clase **Deque** y pruébela.
22. Use un **TreeMap** en **Statistics.java**. Ahora agregue el código que prueba las diferencias de rendimiento entre **HashMap** y **TreeMap** en ese programa.
23. Produzca un **Map** y un **Set** conteniendo todos los países que comienzan con 'A'.
24. Utilice **Collections2.countries**, llene un **Set** varias veces con los mismos datos y verifique que al final el **Set** tiene solo una instancia de cada uno. Intente esto con ambos tipos de **Set**.
25. Comience con **Statistics.java**, cree un programa que ejecute la prueba repetidamente y busque si algún número tiende a aparecer mas que los otros en los resultados.
26. Escriba nuevamente **Statics.java** utilizando un **HashSet** de objetos **Counter** (tendrá que modificar **Counter** para que trabaje con **HashSet**). ¿Que estrategia parece mejor?
27. Modifique la clase en el ejercicio 13 de tal forma que trabaje con **HashSets** y tenga una clave en **HashMaps**.
28. Use **SlowMap.java** como inspiración, cree un **SlowSet**.

29. Aplique las pruebas en **Map1.java** para **SlowMap** para verificar que trabaja. Corrija cualquier cosa en **SlowMap** que no trabaje correctamente.
30. Implemente el resto de la interfase **Map** para **SlowMap**.
31. Modifique **MapPerformance.java** para incluir pruebas de **SlowMap**.
32. Modifique **SlowMap** de tal forma que en lugar de dos **ArrayLists**, almacene una solo **ArrayList** u objetos **MPair**. Verifique que la versión modificada trabaja correctamente. Utilice **MapPerformance.java**, pruebe la velocidad de su nuevo **Map**. Ahora cambie el método **put()** de tal forma que realice una **sort()** luego que cada par es entrado, y modifique **get()** para utilizar **Collections.binarySearch()** para buscar la clave. Compare el rendimiento de la nueva versión con la vieja.
33. Agregue un campo **char** a **CountedString** que también sea inicializado en el constructor, y modifique los métodos **hashCode()** y el **equals()** para incluir los valores de este **char**.
34. Modifique **SimpleHashMap** de tal forma que reporte colisiones, y pruébelo agregando el mismo grupo de datos dos veces para ver las colisiones.
35. Modifique **SimpleHashMap** de tal forma que reporte el número de “pruebas” necesarias cuando una colisión se sucede. ¿Esto, es cuantas llamadas a **next()** deben ser hechas en los **Iterators** que se mueven en los **LinkedLists** buscando coincidencias?
36. Implemente los métodos **clear()** y **remove()** para **SimpleHashMap**.
37. Implemente el resto de la interfase **Map** para **simpleHashMap**.
38. Agregue un método privado **rehash()** a **SimpleHashMap** que sea invocado cuando el factor de carga exceda los 0.75. Durante el rehashing, doble el número de cubos, luego busque el primer número primo mayor que ese para determinar el nuevo número de cubos.
39. Siga el ejemplo en **SimpleHashMap.java**, cree y pruebe un **SimpleHashSet**.
40. Modifique **SimpleHashMap** para utilizar **ArrayList** en lugar de **LinkedLists**. Modifique **MapPerformance.java** para comparar el rendimiento de las dos implementaciones.
41. Utilice la documentación HTML para la JDK (que puede bajar de [java.sun.com](http://java.sun.com)), busque la clase **HashMap**. Cree un **HashMap**, llénelo con elementos, y determine el factor de carga. Pruebe la velocidad de búsqueda con este mapa, luego intente incrementar la velocidad haciendo un nuevo **HashMap** con una capacidad inicial mas grande y

copiando el viejo mapa en el nuevo, ejecute la prueba de velocidad nuevamente en el nuevo mapa.

42. En el capítulo 8, encuentre el ejemplo **GreenhouseControls.java**, que consiste en tres ficheros. En **Controller.java**, la clase **EventSet** es solo un contenedor. Cambie el código para utilizar una **LinkedList** en lugar de un **EventSet**. Esto va a requerir mas que simplemente reemplazar **EventSet** con **LinkedList**; necesitará también utilizar un **Iterator** para ir a través del grupo de eventos.
43. (Desafío). Escriba su propia clase map hashed, hecha a la medida para una clave de tipo particular: **String** para este ejemplo. No herede de **Map**. En lugar de eso, duplique los métodos de tal forma que los métodos **put()** y **get()** especialmente tomen objetos **String**, no **Objetcts**, como claves. Todo lo que invoque claves no debe utilizar tipos genéricos, pero en lugar de eso deben trabajar con cadenas, para evitar el costo de la conversión ascendente y descendente. Su meta es hacer una implementación hecha a la medida lo mas rápida posible. Modifique **MapPerformance.java** para probar su implementación contra un **HashMap**.
44. (Desafío). Encuentre el código fuente para un **List** en las librerías de código fuente de Java que vienen con las distribuciones. Copie este código y haga una versión especial llamada **intList** que almacene solo enteros. Considere que tomará para hacer una versión especial de **List** para todos los tipos primitivos. Ahora considere que sucede si quiere hacer una clase de lista enlazada que trabaje con todos los tipos primitivos. Si los tipos parametrizados son siempre implementados en Java, ellos proporcionaran una forma de hacer este trabajo para usted automáticamente (de la misma forma que muchos otros beneficios).

# 10: Manejo de errores con excepciones

La filosofía básica en Java es “el código mal formado no correrá”.

El momento ideal para atrapar un error es en tiempo de compilación, antes de que incluso trate de ejecutar el programa. Sin embargo, no todos los errores pueden ser detectados en tiempo de compilación. El resto de los problemas debe ser manejado en tiempo de ejecución, a través de alguna formalidad que permita que lo que origine el error pase la información adecuada a un receptor que sepa como manejar la dificultad adecuadamente.

En C y en otros lenguajes precoses, pueden haber muchas de esas formalidades, y ellas son generalmente establecidas por convención y no como parte del lenguaje de programación. Típicamente, se retorna un valor especial o un grupo de banderas, y el receptor supuestamente observaba el valor de las banderas y determinaba que estaba mal. Sin embargo, los años han pasado, y se ha descubierto que los programadores que utilizan una librería tienden a pensar que son invencibles -para adentro, “Si, los errores le pueden suceder a otros, pero no en *mi* código”. Así es que, no sorprendentemente, ellos no verifican las condiciones de error (y a veces las condiciones de error son tan tontas de verificar<sup>1</sup>). Si se es suficientemente minucioso para verificar errores cada vez que se llama a un método, su código se vuelve una pesadilla ilegible. Dado que los programadores pueden persuadir los sistemas con esos lenguajes son resistentes a admitir la verdad: Esta estrategia de manejar errores es la mayor limitación para crear programas grandes, robustos y fáciles de mantener.

La solución es tomar la naturaleza casual fuera del manejo de error y forzar la formalidad. Esto es actualmente una larga historia, puesto que las implementaciones de *manejo de excepciones* regresan a los sistemas operativos de los 60s, e incluso a el “**on error goto**” de BASIC. Pero el

---

<sup>1</sup> El programador C puede buscar el valor de retorno de `printf()` para un ejemplo de esto.

manejo de errores de C++ fue basado en Ada, y Java es basado primariamente en C++ (a pesar de que se ve mas como Pascal orientado a objetos).

La palabra “excepción” esta utilizada en el sentido de “tomo excepción de esto”. En el punto en que el problema se sucede se puede no saber que hacer con el, pero se sabe que no se puede continuar alegremente; se debe parar y alguien, en algún lugar, debe entender que hacer. Pero no se tiene suficiente información en el contexto actual para solucionar el problema. Así es que se reparte el problema a un contexto mayor donde alguien está calificado para tomar la decisión apropiada (muy parecido a una cadena de comandos).

El otro beneficio significante de las excepciones que limpian el código de manejo de excepciones. En lugar de verificar un error en particular y tratar con el en muchos lugares en su programa, no se necesita verificar en el punto de la llamada al método (dado que la excepción garantiza que alguien la tiene que atrapar). Y, se necesita manejar el problema en solo un lugar, el llamado *manejador de excepciones*. Esto saca adelante el código, y separa el código que describe que es lo que quiere hacer del código que es ejecutado cuando las cosas van mal. En general, la lectura, escritura, y depuración del código se hace mucho mas limpia con excepciones que utilizando las viejas formas de manejo de errores.

Dado que el manejo de excepciones esta implementado por el compilador Java, hay solo unos tantos ejemplos que puedan ser escritos en este libro sin aprender acerca de manejo de excepciones. Este capítulo introduce al código que se necesita escribir para manejar adecuadamente excepciones, y la forma en que se puede generar excepciones propias si un método esta en problemas.

## Excepciones básicas

Una *condición excepcionales* un problema que evita la continuación del método o el campo de acción donde se encuentra. Esto es importante de distinguir de una condición excepcional de un problema normal, en el cual se tiene la información suficiente en el contexto actual para que alguien haga frente a la dificultad. Con una condición excepcional, no se puede continuar procesando porque no se tiene la información necesaria para tratar con el problema *en el contexto actual*. Todo lo que se puede hacer es saltar fuera del contexto y relegar el problema a un contexto superior. Esto es lo que sucede cuando lanza una excepción.

Un ejemplo simple es una división. Si se esta dividiendo y es posible hacerlo por cero, vale la pena verificar para asegurarse que no se va a realizar esa división. ¿Pero que significa que el denominador sea cero? Tal vez se sepa,

en el contexto del problema que se está tratando de solucionar en este método en particular, como tratar con un denominador cero. Pero si es un valor inesperado, no se puede tratar con él y se debe lanzar una excepción en lugar de continuar adelante por ese camino.

Cuando se lanza una excepción, muchas cosas suceden. Primero, el objeto excepción es creado de la misma forma en que cualquier objeto Java es creado: en el heap, con **new**. Luego el actual camino de ejecución (el que no se puede continuar) es parado y la referencia al objeto excepción es expulsado del contexto actual. En este punto el mecanismo de excepción se encarga y comienza a buscar un lugar apropiado para continuar ejecutando el programa. Este lugar apropiado es el *manejador de excepciones*, cuyo trabajo es recuperarse del problema así el programa puede cambiar de dirección o simplemente continuar.

Como un ejemplo simple de lanzado de excepción, considere una referencia a un objeto llamada **t**. Es posible que pueda pasar una referencia que no fue inicializada, así es que se puede querer verificar antes de tratar de llamar un método utilizando esa referencia a objetos. Se puede enviar información acerca del error dentro de un gran contexto creando un objeto que represente su información y “lanzarla” fuera de su contexto. Esto es llamado *lanzar una excepción*. Aquí vemos como se vería:

```
| if (t== null)
|     throw new NullPointerException();
```

Esto lanza la excepción, lo que permite -en el contexto actual- abdicar responsabilidades para pensar mas acerca de este tema. Esto es simplemente mágicamente manejado por alguien mas. Precisamente *donde* será mostrado en breve.

## Argumentos de excepciones

Como un objeto en Java, se crean siempre excepciones en el heap utilizando **new**, quien asigna espacio y llama al constructor. Hay dos constructores en todas las excepciones estándar: el primero es el constructor por defecto, y el segundo toma una cadena como argumento así es que se puede colocar información pertinente en la excepción:

```
| if(t == null)
|     throw new NullPointerException("t = null");
```

Esta cadena puede ser mas tarde extraída utilizando varios métodos, como se verá en breve.

La palabra clave **throw** causa que sucedan una gran cantidad de cosas mágicas. Típicamente, se utilizará primero **new** para crear un objeto que represente la condición de error. Se da la referencia resultante a **throw**. El objeto es, en efecto, “retornado” del método, aún si el tipo de objeto no es normalmente el que el método está diseñado a retornar. Una forma

simplista de pensar en el manejo de excepciones es como un mecanismo alternativo de retorno, a pesar de que se está en problemas si lleva la analogía muy lejos. Se puede también salir del ámbitos usuales lanzando una excepción. Pero un valor es retornado, y el método o alcance sale.

Cualquier similitud con un retorno común de un método termina aquí, dado que *donde* retorna es un lugar completamente diferente de donde se retorna en una llamada a método normal (Se termina en un manejador de excepciones apropiado a millas -muchos niveles abajo de la llamada- de donde la excepción fue lanzado).

Además, se puede lanzar una excepción de cualquier tipo de objeto **Throwable** que se quiera. Típicamente, se lanza una clase diferente de excepción para cada diferente tipo de error. La información acerca del error es representadas dentro del objeto de la excepción e implícitamente en el tipo de objeto de excepción elegido, así es que alguien en el gran contexto puede darse cuenta de que hacer con su excepción (A menudo, la única información es el tipo de objeto de excepción, y nada significante es almacenado con el objeto de la excepción).

## Capturando una excepción

Si un método lanza una excepción, se debe asumir que esa excepción es “capturada” y tratada. Una de las ventajas del manejo de excepciones es que permite concentrarse en los problemas que se está tratando de resolver en un lugar, y el trato con los errores de ese código en otro lugar.

Para ver como una excepción es capturada, se debe entender primero el concepto de una *región protegida* que es una sección de código que puede producir excepciones, y la cual está seguida del código que maneja esas excepciones.

### El bloque **try**

Si está dentro de un método y se lanza una excepción (u otro método que se llama dentro de este método lanza una excepción), ese método saldrá en el proceso de lanzado de excepción. Si no se quiere que se salga del método lanzando una excepción se puede configurar un bloque especial dentro del método para capturar la excepción. Esto es llamado el *bloque try* porque se “intenta” las distintas llamadas a métodos aquí. El bloque try es un ámbito común, precedido por la palabra clave **try**.

```
try {  
    // Código que puede generar excepciones  
}
```

Si se va a verificar por errores cuidadosamente en un lenguaje de programación que no soporta manejo de excepciones, se tiene que rodear cada llamada a un método con configuraciones y código de prueba de errores, aún si se llama al mismo método muchas veces. Con el manejo de excepciones, se coloca todo en un bloque try y se captura todas las excepciones en un solo lugar. Esto significa que su código es mucho más fácil de escribir y más fácil de leer porque la meta del código no es confundirse con las verificaciones de errores.

## Manejadores de error

Claro, la excepción lanzada debe terminar en algún lugar. Este “lugar” es el *manejador de excepciones* y hay uno para cada tipo de excepción que se quiera capturar. Los manejadores de excepción siguen inmediatamente al bloque try y son indicados por la palabra clave **catch**:

```
try {  
    // Code that might generate exceptions  
} catch(Type1 id1) {  
    // Handle exceptions of Type1  
} catch(Type2 id2) {  
    // Handle exceptions of Type2  
} catch(Type3 id3) {  
    // Handle exceptions of Type3  
}  
// etc...
```

Cada cláusula catch (manejador de excepción) es como un pequeño método que toma uno y solo un argumento de un tipo particular. El identificador (**id1**, **id2**, y así sucesivamente) puede ser utilizado dentro del manejador, exactamente igual a un argumento de un método. A veces nunca se utiliza este identificador porque el tipo de excepción le da la suficiente información para tratar con la excepción, pero el identificador debe seguir estando allí.

Los manejadores deben aparecer directamente luego del bloque try. Si una excepción es lanzada, el mecanismo de manejo de excepciones busca el primer manejador con un argumento que corresponda con el tipo de excepción. Luego entra en la cláusula catch, y la excepción es considerada manejada. La búsqueda por manejadores se detiene una vez que la cláusula es finalizada. Solo la cláusula catch que corresponde se ejecuta; no es como una instrucción **switch** en donde se necesita un **break** luego de cada **case** para evitar que el resto se ejecute.

Debe notarse que, dentro del bloque try, un gran número de llamadas a métodos puede generar la misma excepción, pero se necesita solo uno manejador.

## Finalización contra reanudación

Hay dos modelos básicos en la teoría de manejo de excepciones. En la *finalización* (que es lo que Java y C++ soportan), se asume que el error es tan crítico que no hay forma de retornar a donde la excepción ha ocurrido. Quienquiera que ha lanzado la excepción decide que no hay forma de salvar la situación, y no *quiere* regresar.

La alternativa es llamada *reanudación*. Esto significa que el manejador de excepciones está esperando hacer algo para rectificar la situación, y entonces el método culpable vuelve a intentarlo, suponiendo el éxito la segunda vez. Si se quiere reanudar, significa que se tiene la esperanza de que se pueda continuar la ejecución luego de que la excepción es manejada. En este caso, la excepción tiene el comportamiento más como una llamada a un método -que es lo que se tiene que configurar en situaciones en Java en las cuales se quiere reanudar (esto es, no se lanza una excepción; se llama a un método que arregla el problema). Alternativamente, se puede colocar el bloque `try` dentro de un bucle `while` que mantiene el bloque `try` reingresando hasta que el resultado es satisfactorio.

Históricamente, los programadores utilizan sistemas operativos que soportan reanudación en el manejo de excepciones parándolo utilizando código como el de terminación y saltándose la reanudación. Así es que a pesar de que la reanudación suena atractiva al principio, no es tan útil en la práctica. La principal razón es probablemente el acoplamiento que resulta: su manejador a menudo tiene cuidado de donde la excepción es lanzada y contiene código no genérico específico a la ubicación de lanzamiento. Esto hace el código difícil de escribir y de mantener, especialmente para sistemas grandes donde la excepción puede ser generada de muchos puntos.

# Creando excepciones propias

No se está atascado utilizando las excepciones existentes de Java. Esto es importante porque a menudo se necesita crear excepciones propias para mostrar un error especial que su librería es capaz de crear, pero que no se previeron cuando la jerarquía de excepciones de Java fue creada.

Para crear su propia clase de excepción, se está forzado a heredar de un tipo de excepción, preferiblemente una que esté cercana al significado de la nueva excepción (esto a menudo no es posible, sin embargo). La forma más trivial de crear un nuevo tipo de excepción es simplemente indicarle al

compilador que cree un constructor por defecto para usted, así es que se requiere al menos nada de código después de todo:

```
//: c10:SimpleExceptionDemo.java
// Heredando sus propias excepciones.
class SimpleException extends Exception {}
public class SimpleExceptionDemo {
    public void f() throws SimpleException {
        System.out.println(
            "Lanzando una excepción simple de f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        SimpleExceptionDemo sed =
            new SimpleExceptionDemo();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.err.println("Capturandola!");
        }
    }
} //:~
```

Cuando el compilador crea un constructor por defecto, automáticamente (y de forma transparente) llama el constructor de la clase base. Claro, en este caso no se tiene un constructor para **SimpleException(String)**, pero en la práctica no es muy utilizado. Como se vera, la cosa mas importante acerca de las excepciones es el nombre de clases, así es que la mayoría del tiempo una excepción como la mostrada arriba es satisfactoria.

Aquí, el resultado es impreso a la consola de *error estándar* mediante **System.err**. Esto es usualmente mejor lugar para enviar información acerca de errores que **System.out**, el cual puede ser redirigido. Si se envía la salida a **System.err** no puede ser redirigido junto con **System.out** así es que es mas probable que el usuario tenga aviso de el.

Creando una clase de excepción que solo tenga un constructor que tome un **String** es también bastante simple:

```
//: c10:FullConstructors.java
// Heredando sus propias excepción.
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}
public class FullConstructors {
    public static void f() throws MyException {
        System.out.println(
            "Lanzando MyException de f()");
        throw new MyException();
    }
    public static void g() throws MyException {
```

```

        System.out.println(
            "Lanzando MyException de g()");
        throw new MyException("Originada en g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace(System.err);
        }
        try {
            g();
        } catch(MyException e) {
            e.printStackTrace(System.err);
        }
    }
} //:~

```

El código agregado es pequeño -el agregado de dos constructores que definen la forma en que **MyException** es creada. En el segundo constructor, el constructor de la clase base con un argumento **String** es explícitamente invocado utilizando la palabra clave **super**.

La información de trazado de pila es enviada a **System.err** así es que es mas probable que sea notificado en el evento que en **System.out** redirigido.

La salida del programa es:

```

Throwing MyException from f()
MyException
    at FullConstructors.f(FullConstructors.java:16)
    at FullConstructors.main(FullConstructors.java:24)
Throwing MyException from g()
MyException: Originated in g()
    at FullConstructors.g(FullConstructors.java:20)
    at FullConstructors.main(FullConstructors.java:29)

```

Se puede ver la ausencia del mensaje de detalle en **MyException** lanzado desde **f()**.

El proceso de creación de excepciones propias puede ir mas allá aún. Se puede agregar constructores extra y miembros:

```

//: c10:ExtraFeatures.java
// Further embellishment of exception classes.
class MyException2 extends Exception {
    public MyException2() {}
    public MyException2(String msg) {
        super(msg);
    }
    public MyException2(String msg, int x) {
        super(msg);
        i = x;
    }
    public int val() { return i; }
    private int i;
}

```

```

public class ExtraFeatures {
    public static void f() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }
    public static void h() throws MyException2 {
        System.out.println(
            "Throwing MyException2 from h()");
        throw new MyException2(
            "Originated in h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace(System.err);
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace(System.err);
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace(System.err);
            System.err.println("e.val() = " + e.val());
        }
    }
} //:~

```

Todos los datos miembros **i** han sido agregados, junto con un método que lee el valor y un constructor adicional que lo configura. La salida es:

```

Throwing MyException2 from f()
MyException2
    at ExtraFeatures.f(ExtraFeatures.java:22)
    at ExtraFeatures.main(ExtraFeatures.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
    at ExtraFeatures.g(ExtraFeatures.java:26)
    at ExtraFeatures.main(ExtraFeatures.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
    at ExtraFeatures.h(ExtraFeatures.java:30)
    at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47

```

Dado que una excepción es solo otro tipo de objeto, se puede continuar este proceso de embellecimiento del poder de sus clases de excepciones. Tenga

en mente, sin embargo, que toda estas vestiduras pueden ser perdida en el programa cliente que utiliza los paquetes, dado que simplemente puede considerar la excepción a ser lanzada y nada mas (Esto es la forma en que la mayoría de las excepciones de Java son utilizadas).

## Especificación de excepciones

En Java, se requiere que se informe al cliente programador, que llama a el método, de las excepciones que pueden ser lanzadas desde el método. Esto es civilizado, dado que la persona que llama puede saber exactamente que código escribir para capturar todas las potenciales excepciones. Claro, si el código fuente esta disponible, el cliente programador puede buscar y encontrar instrucciones **throw**, pero a menudo las librerías no vienen con el código fuente. Para prevenir esto de ser un problema, Java proporciona la sintaxis (*y fuerza a utilizar esa sintaxis*) para permitir que políticamente indique al cliente programador que excepciones lanza este método, así el cliente programador puede manejarlas. Esto es la *especificación de excepciones*, y es parte de la declaración de métodos, que se hace presente luego de la lista de argumentos.

La especificación de excepciones utiliza una palabra clave adicional, **throws**, seguida de una lista de todos los tipos potenciales de excepción. Así es que su definición de método puede verse como esto.

```
| void f() throws TooBig, TooSmall, DivZero { //...
| Si se dice
| void f() { // ...
| significa que no hay excepciones que serán lanzadas desde el método
| (Excepto por las excepciones del tipo RuntimeException, que pueden
| razonablemente ser lanzadas en cualquier lado -esto será descrito mas
| tarde.)
```

No se puede mentir acerca de la especificación de excepciones -si el método causa excepciones y no se manejan, el compilador las detectara y e indicara que se debe manejar o indicar con una especificación de excepciones que pueden ser lanzadas por el método. Haciendo cumplir la especificación de excepciones del principio al fin, Java garantiza que la exactitud de excepciones puede ser asegurada *en tiempo de compilación*<sup>2</sup>.

---

<sup>2</sup> Esto es una mejora significante sobre el manejo de excepciones de C++, que no captura las violaciones de especificaciones de excepciones hasta el tiempo de ejecución, donde no es muy útil.

Hay un lugar donde se puede mentir: se puede reclamar el lanzar una excepción que realmente no se tiene. El compilador toma la información, y fuerza a l usuario de su método a tratarla como si realmente se lanzara la excepción. Esto tiene el beneficio de que se tiene un lugar para esa excepción, así es que se puede comenzar a lanzar la excepción mas tarde sin requerir cambios en el código existente. Esto es también importante para la creación de clases base abstractas e interfaces cuyas clases derivadas o implementaciones pueden necesitar lanzar excepciones.

## Capturando cualquier excepción

Es posible crear un manejador que capture cualquier tipo de excepción. Se realiza esto capturando el tipo base la excepción **Exception** (hay otros tipos de excepciones base, pero **Exception** es la base que es aplicable a virtualmente todas las actividades de programación):

```
catch(Exception e) {  
    System.err.println("Captura una excepción");  
}
```

Esto capturará cualquier excepción, así es que si se usa, se podrá poner al final de su lista de manejadores para evitar la atribución de cualquier manejador de excepciones que puedan de otra forma seguirlo.

A causa de que la clase **Exception** es la base para todas las clases de excepción que son importantes para el programador, no se tiene mucha información específica acerca de la excepción, pero se puede llamar los métodos que vienen de su tipo base **Throwable**:

**String getMessage()**

**String getLocalizedMessage()**

Obtiene el detalle del mensaje, o un mensaje ajustado para este escenario particular.

**String toString()**

Retorna una corta descripción del **Throwable**, incluyendo el mensaje de detalle si hay uno.

**void printStackTrace()**

**void printStackTrace(PrintStream)**

**void printStackTrace(PrintWriter)**

Imprime el **Throwable** y la traza de las llamadas de pila **Throwable**. La llamadas de pila muestra la secuencia de las llamadas a métodos que nos traen a el punto en que la excepción fue lanzada. La primer versión imprime a la salida de error estándar, la segunda y tercera imprime a un flujo de su elección (en el Capítulo 11, se entenderá por que hay dos tipos de flujo).

## Throwable fillInStackTrace()

Registra información dentro de este objeto **Throwable** acerca del estado actual de la trama de la pila. Útil cuando una aplicación vuelve a lanzar un error o excepción (mas acerca de esta historia).

Además, se tienen algunos otros métodos del objeto base del tipo **Throwable** (el tipo base de todo el mundo). El único que se hace cómodo para las excepciones es **getClass()**, que retorna un objeto representando la clase de este objeto. Se puede a su vez indagar a este objeto **Class** por su nombre con **getName()** o **toString()**. También se pueden hacer cosas mas sofisticadas con objetos **Class** que no son necesarias para el manejo de excepciones. Los objetos **Class** serán estudiados mas adelante en este libro.

He aquí un ejemplo que muestra el uso de estos métodos básicos de **Exception**:

```
//: c10:ExceptionMethods.java
// Demonstrating the Exception Methods.
public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Here's my Exception");
        } catch(Exception e) {
            System.err.println("Caught Exception");
            System.err.println(
                "e.getMessage(): " + e.getMessage());
            System.err.println(
                "e.getLocalizedMessage(): " +
                e.getLocalizedMessage());
            System.err.println("e.toString(): " + e);
            System.err.println("e.printStackTrace():");
            e.printStackTrace(System.err);
        }
    }
} ///:~
```

La salida para este programa es:

```
Caught Exception
e.getMessage(): Here's my Exception
e.getLocalizedMessage(): Here's my Exception
e.toString(): java.lang.Exception:
    Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
at ExceptionMethods.main(ExceptionMethods.java:7)
java.lang.Exception:
    Here's my Exception
at ExceptionMethods.main(ExceptionMethods.java:7)
```

Se puede ver que el método proporciona sucesivamente mas información - cada vez es efectivamente un super grupo del anterior.

# Lanzando nuevamente una excepción

A veces se quiere volver a lanzar una excepción que simplemente se capturó, particularmente cuando se utiliza **Excepción** para capturar cualquier excepción. Dado que ya se tiene la referencia a la excepción actual, simplemente se puede volver a lanzar esa referencia:

```
    catch(Exception e) {
        System.err.println("Una excepción fué lanzada ");
        throw e;
    }
```

Volver a lanzar una excepción causa que la excepción vaya a el manejador de excepciones en el siguiente nivel superior del contexto. Cualquier **catch** superior para el mismo bloque **try** es ignorada. Además, todo acerca del objeto de la excepción es preservado, así es que el manejador en el contexto mas alto que captura la excepción especificada puede extraer toda la información de ese objeto.

Si simplemente se vuelve a lanzar la excepción actual, la información que se imprime acerca de esa excepción en **printStackTrace()** pertenecerá a la excepción original, no a el lugar donde se volvió a lanzar. Si se quiere instalar nueva información de trazado de pila, se puede hacer llamando a **fillInStackTrace()**, el que retornara un objeto de excepción que es creado llenando la información actual de la pila dentro del objeto de excepción anterior. Aquí vemos como se vería:

```
//: c10\Rethrowing.java
// Demonstrating fillInStackTrace()
public class Rethrowing {
    public static void f() throws Exception {
        System.out.println(
            "originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Throwable {
        try {
            f();
        } catch(Exception e) {
            System.err.println(
                "Inside g(), e.printStackTrace()");
            e.printStackTrace(System.err);
            throw e; // 17
            // throw e.fillInStackTrace(); // 18
        }
    }
    public static void
main(String[] args) throws Throwable {
        try {
            g();
        }
```

```

        } catch(Exception e) {
            System.err.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace(System.err);
        }
    }
} //:~

```

Los números de línea importantes son marcados como comentarios. Con la línea 17 sin comentar (como se muestra), la salida es:

```

originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
    at Rethrowing.f(Rethrowing.java:8)
    at Rethrowing.g(Rethrowing.java:12)
    at Rethrowing.main(Rethrowing.java:24)

```

Así es que el trazado de pila de la excepción siempre recuerda su verdadero punto de origen, no importa cuantas veces es vuelta a lanzar.

Con la línea 17 comentada y la línea 18 sin comentar, **fillInStackTrace()** es utilizado en su lugar, y el resultado es:

```

originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
at Rethrowing.f(Rethrowing.java:8)
at Rethrowing.g(Rethrowing.java:12)
at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
at Rethrowing.g(Rethrowing.java:18)
at Rethrowing.main(Rethrowing.java:24)

```

Puesto que por **fillInStackTrace()**, la línea 18 se convierte en el punto de origen de la excepción.

La clase **Throwable** debe aparecer en la especificación de la excepción para **g()** y **main()** porque **fillInStackTrace()** produce una referencia a un objeto **Throwable**. Dado que **Throwable** es una clase base de **Exception**, es posible obtener un objeto que es un **Throwable** pero *no* una **Exception**, así es que el manejador para **Exception** en **main()** puede perderse. Para asegurarse todo en este orden, el compilador fuerza a una especificación de excepción para **Throwable**. Por ejemplo, la excepción en el siguiente programa *no* es capturada en **main()**:

```

//: c10:ThrowOut.java
public class ThrowOut {
    public static void
    main(String[] args) throws Throwable {

```

```

    try {
        throw new Throwable();
    } catch(Exception e) {
        System.err.println("Caught in main()");
    }
}
} //:~

```

Es posible también volver a lanzar una excepción diferente de la que se capturó. Si se hace esto, se tiene un efecto similar que cuando se utiliza **fillInStackTrace()** -la información acerca del sitio original de la excepción se pierde, y lo que queda es la información pertinente a el nuevo **throw**:

```

//: c10:RethrowNew.java
// Rethrow a different object
// from the one that was caught.
class OneException extends Exception {
    public OneException(String s) { super(s); }
}
class TwoException extends Exception {
    public TwoException(String s) { super(s); }
}
public class RethrowNew {
    public static void f() throws OneException {
        System.out.println(
            "originating the exception in f()");
        throw new OneException("thrown from f()");
    }
    public static void main(String[] args)
    throws TwoException {
        try {
            f();
        } catch(OneException e) {
            System.err.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace(System.err);
            throw new TwoException("from main()");
        }
    }
}
} //:~

```

La salida es:

```

originating the exception in f()
Caught in main, e.printStackTrace()
OneException: thrown from f()
at RethrowNew.f(RethrowNew.java:17)
at RethrowNew.main(RethrowNew.java:22)
Exception in thread "main" TwoException: from main()
at RethrowNew.main(RethrowNew.java:27)

```

La excepción final solo conoce lo que viene del **main()**, y no del **f()**.

No hay que preocuparse nunca acerca de limpiar la excepción previa, o cualquier excepción si vamos a eso. Están todos los objetos basados en el heap creados con **new**, así es que el recolector de basura automáticamente los limpia a todos ellos.

# Excepciones estándar de Java

La clase de Java **Throwable** describe todo lo que puede ser lanzado como una excepción. Hay dos tipos generales de objetos **Throwable** (“tipos de” = “heredados de”). **Error** representa errores en tiempo de compilación y errores de sistema de los cuales no hay que preocuparse por capturar (excepto en casos especiales). **Exception** es el tipo básico que puede ser lanzado desde cualquiera de los métodos de la librería de clases estándar de Java, desde los métodos propios y en accidentes en tiempo de ejecución. Así es que el tipo base de interés para el programador Java es **Exception**.

La mejor forma de tener una visión general de las excepciones es buscar la documentación Java que se puede bajar desde *java.sun.com*. Tiene mérito hacer esto una vez solo para obtener un sentido de las diferentes excepciones, pero rápidamente verá que no hay nada especial entre una excepción y la siguiente excepto el nombre. También, el número de excepciones en Java se sigue expandiendo; básicamente no tiene sentido imprimirlas en un libro. Cada nueva librería que se obtenga de un tercero probablemente tenga sus propias excepciones también. Lo importante de entender es el concepto y que se puede hacer con excepciones.

La idea básica es que el nombre de la excepción represente el problema que ha ocurrido, y el nombre de excepción pretende ser relativamente auto explicativa. Las excepciones no están definidas todas en **java.lang**; algunas son creadas para apoyar a otras librerías como **util**, **net**, y **io**, lo que se puede ver de sus nombres de clases completos o de donde son heredadas. Por ejemplo, todas las excepciones E/S son heredadas de **java.io.IOException**.

## El caso especial de **RuntimeException**

El primer ejemplo en este capítulo fue

```
| if(t == null)
|     throw new NullPointerException();
```

Puede ser un poco horripilante pensar que se debe verificar por **null** en cada referencia que es pasada a un método (dado que no se puede conocer si el que llama el método ha pasado una referencia válida). Afortunadamente, no se necesita -esto es parte de las pruebas en tiempo de ejecución que Java realiza por usted, y si cualquier llamada es hecha a una referencia **null**, Java

automáticamente lanzará una **NullPointerException**. Así es que el pedacito de código arriba es siempre superfluo.

Hay un conjunto completo de tipos de excepciones que están en esta categoría. Estas siempre se lanzan automáticamente por Java y no se necesita incluirlas en la especificación de excepción. De forma muy adecuada, están agrupadas juntas en una sola clase base llamada **RuntimeException**, que es un ejemplo perfecto de herencia: esta establece una familia de tipos que tienen algunas características y comportamientos en común. También, nunca se necesita escribir una especificación de excepción a no ser que un método pueda lanzar una **RuntimeException**, puesto que simplemente es asumido. Dado que indican errores de programación, virtualmente nunca se captura una **RuntimeException** -se trata automáticamente. Si se está forzado a verificar excepciones **RuntimeException** su código puede volverse desordenado. Aún cuando no se capturen típicamente excepciones **RuntimeException**, en paquetes propios se debe elegir lanzar algunas de las **RuntimeException**.

¿Qué sucede cuando no se capturan estas excepciones? Dado que el compilador no fuerza la especificación de estas, es bastante plausible que una excepción **RuntimeException** pueda filtrarse de todas formas fuera de su método **main()** sin ser capturada. Para ver qué sucede en este caso, intente con el siguiente ejemplo:

```
//: c10:NeverCaught.java
// Ignoring RuntimeExceptions.
public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
    }
} ///:~
```

Ya se puede ver que una excepción **RuntimeException** (o cualquiera heredada de esta) es un caso especial, dado que el compilador no requiere una especificación de excepción para estos tipos.

La salida es:

```
Exception in thread "main"
java.lang.RuntimeException: From f()
    at NeverCaught.f(NeverCaught.java:9)
    at NeverCaught.g(NeverCaught.java:12)
    at NeverCaught.main(NeverCaught.java:15)
```

Así es que la respuesta es: Si una excepción **RuntimeException** se obtiene de todas formas fuera del **main()** sin ser capturada, **printStackTrace()** es llamado para esa excepción así como se sale del programa.

Se tiene que tener en mente que solo se puede ignorar excepciones **RuntimeException** en su código, dado que todos los demás manejos son forzados cuidadosamente por el compilador. El razonamiento es que una excepción **RuntimeException** representa un error de programación:

1. Un error no se puede capturar (recibir una referencia nula manejada por un método por el cliente programador, por ejemplo).
2. Un error que, como programador, debe ser verificado en el código (como un **ArrayOutOfBoundsException** donde se debe prestar atención al tamaño del arreglo).

Se puede ver que tremendo beneficio es tener excepciones en este caso, dado que ayuda en el proceso de depuración.

Es interesante notar que no se puede clasificar el manejo de excepciones en Java como una herramienta con un solo propósito. Si, esta diseñada para manejar estos molestos errores en tiempo de ejecución que producirán una salida del control del código, pero esto es algo esencial para ciertos tipos de errores de programación que el compilador no puede detectar.

## Realizando una limpieza con finally

A menudo alguna pieza de código que se quiere ejecutar se lance o no una excepción dentro de un bloque **try**. Esto es usualmente se aplica a algunas operaciones aparte de las de recuperación de memoria (dado que se está siendo cuidadoso con el recolector de basura). Para alcanzar este efecto, se utiliza una cláusula<sup>3</sup> **finally** en el final de un manejador de excepciones. La imagen completa de una sección de manejo de excepciones es de esta forma:

```
try {
    // La región protegida: Actividades peligrosas
    // que pueden lanzar A, B, o C
} catch(A a1) {
    // Manejador para la situación A
} catch(B b1) {
    // Manejador para la situación B
} catch(C c1) {
    // Manejador para la situación C
} finally {
    // Actividades que deben suceder todas las veces
}
```

---

<sup>3</sup> El manejo de excepciones de C++ no tiene la cláusula **finally** porque confía en los destructores para lograr este tipo de limpieza.

Para demostrar que la cláusula **finally** siempre se ejecuta, se puede intentar con este programa:

```
//: c10:FinallyWorks.java
// The finally clause is always executed.
class ThreeException extends Exception {}
public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // Post-increment is zero first time:
                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; // out of "while"
            }
        }
    }
} ///:~
```

Este programa también sugiere como se puede tratar con el hecho de que las excepciones en Java (como las excepciones en C++) no permiten reanudar donde la excepción fue lanzada, como se planteó antes. Si se coloca el bloque **try** en un bucle, se puede establecer una condición que pueda ser encontrada antes de continuar con el programa. Se puede también agregar un contador estático o algún otro dispositivo para permitir que el bucle intente con diferentes estrategias antes de abandonar. De esta forma se puede crear un gran nivel de robustez dentro de los programas.

La salida es:

```
ThreeException
In finally clause
No exception
In finally clause
```

Sea o no lanzada una excepción, la cláusula **finally** es siempre ejecutada.

## ¿Para qué es **finally**?

En un lenguaje sin recolección de basura y sin llamadas automáticas a destructores<sup>4</sup>, **finally** es importante porque permite que el programador garantice la liberación de memoria sin importar que sucede en el bloque **try**.

---

<sup>4</sup> Un destructor es una función que siempre es llamada cuando un objeto no se utiliza más. Se debe saber exactamente donde y cuando el destructor es llamado. C++ tiene llamadas automáticas a destructores, pero las versiones Delphi de Pascal orientado a objetos 1 y 2 no (lo que cambia el significado y uso del concepto de destructor para ese lenguaje).

Pero Java tiene un recolector de basura, así es que la liberación de memoria nunca es realmente un problema. También, no tiene destructores a llamar. ¿Así es que cuando se necesita utilizar **finally** en Java?

**finally** es necesaria cuando se necesita ajustar alguna *otracosa* que volver la memoria a su estado original. Esto es algún tipo de limpieza como un fichero abierto o una conexión a redes, a veces se dibuja en la pantalla, o incluso se intercambia en el mundo exterior, como se modela en el siguiente ejemplo:

```
///: c10:OnOffSwitch.java
// ¿Por que utilizar finally?
class Switch {
    boolean state = false;
    boolean read() { return state; }
    void on() { state = true; }
    void off() { state = false; }
}
class OnOffException1 extends Exception {}
class OnOffException2 extends Exception {}
public class OnOffSwitch {
    static Switch sw = new Switch();
    static void f() throws
        OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            f();
            sw.off();
        } catch(OnOffException1 e) {
            System.err.println("OnOffException1");
            sw.off();
        } catch(OnOffException2 e) {
            System.err.println("OnOffException2");
            sw.off();
        }
    }
} ///:~
```

El objetivo aquí es asegurarse que la llave esté apagada cuando haya terminado **main()**, así es que **sw.off()** se coloca en el final del bloque **try** y al final de cada manejador de excepción. Pero es posible que una excepción pueda ser lanzada que no sea capturada aquí, así es que **sw.off()** sea echada de menos. Sin embargo, con **finally** se puede colocar código de limpieza en un bloque **try** en solo un lugar.

```
///: c10:WithFinally.java
// Finally Guarantees cleanup.
public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
```

```

        // Code that can throw exceptions...
        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 solamente un lugar, donde está garantizado que se ejecute sin importar que suceda.

Incluso en casos en donde la excepción no es capturada en el grupo actual de cláusulas **catch**, **finally** será ejecutado antes que el mecanismo de excepciones continúe su búsqueda por un manejador en el siguiente nivel mas alto:

```

//: c10:AlwaysFinally.java
// Finally is always executed.
class FourException extends Exception {}
public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println(
            "Entering first try block");
        try {
            System.out.println(
                "Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println(
                    "finally in 2nd try block");
            }
        } catch(FourException e) {
            System.err.println(
                "Caught FourException in 1st try block");
        } finally {
            System.err.println(
                "finally in 1st try block");
        }
    }
} //:~

```

La salida para este programa muestra que sucede:

```

Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block

```

La instrucción **finally** también será ejecutada en situaciones en donde las instrucciones **break** y **continue** están involucradas. Note que, junto con el

**break** etiquetado y **continue** etiquetado, **finally** elimina la necesidad de una instrucción **goto** en Java.

## Dificultad: la excepción perdida

En general, la implementación de excepciones de Java es bastante excepcional, pero desafortunadamente hay un desperfecto. A pesar de que las excepciones son una indicación de una crisis en un programa y nunca deben ser ignoradas, es posible para una excepción simplemente ser perdida. Esto sucede con una configuración particular utilizando una cláusula **finally**:

```
//: c10:LostMessage.java
// How an exception can be lost.
class VeryImportantException extends Exception {
    public String toString() {
        return "A very important exception!";
    }
}
class HoHumException extends Exception {
    public String toString() {
        return "A trivial exception";
    }
}
public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args)
        throws Exception {
        LostMessage lm = new LostMessage();
        try {
            lm.f();
        } finally {
            lm.dispose();
        }
    }
} ///:~
```

La salida es:

```
Exception in thread "main" A trivial exception
at LostMessage.dispose(LostMessage.java:21)
at LostMessage.main(LostMessage.java:29)
```

Se puede ver que no hay evidencia de **VeryImportantException**, la que es simplemente reemplazada por **HoHumException** en la cláusula **finally**. Esto es una falla seria, dado que significa que una excepción puede ser completamente perdida, y de una forma más sutil y difícil de detectar que el ejemplo anterior. En contraste, C++ trata la situación en donde una segunda

excepción es lanzada antes que la primera es manejado como un calamitoso error de programación. Tal vez una futura versión de Java reparará este problema (por el otro lado, normalmente se puede envolver cualquier método que lance una excepción, como un **dispose()**, dentro de una cláusula **try-catch**).

## Restricciones en la excepción

Cuando se sobrescribe un método, se puede lanzar solo las excepciones que han sido especificadas en la versión del método en la clase base. Esto es una restricción útil, dado que significa que el código que trabaja con la clase base automáticamente trabajará con cualquier objeto derivado de la clase base (un concepto fundamental de la POO, claro), incluyendo excepciones.

Este ejemplo demuestra el tipo de restricciones impuesta (en tiempo de compilación) para las excepciones:

```
//: c10:StormyInning.java
// Los métodos sobrecargados pueden lanzar
// solo excepciones especificadas en sus versiones
// de la clase base, o excepciones derivadas de
// la clase base de excepciones.
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}
abstract class Inning {
    Inning() throws BaseballException {}
    void event () throws BaseballException {
        // No tiene actualmente que lanzar nada
    }
    abstract void atBat() throws Strike, Foul;
    void walk() {} // No lanza nada
}
class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}
interface Storm {
    void event() throws RainedOut;
    void rainHard() throws RainedOut;
}
public class StormyInning extends Inning
    implements Storm {
    // Se pueden agregar nuevas excepciones para
    // constructores, pero se debe tratar
    // con las excepciones de los constructores
    // de la clase base:
    StormyInning() throws RainedOut,
    BaseballException {}}
```

```

StormyInning(String s) throws Foul,
BaseballException {}
// Metodos regulares deben asemejarse a la clase base:
// !!! void walk() throws PopFoul {} //Error de compilación
// La interfse NO PUEDE agregar excepciones a los métodos
// existentes de la clase base:
// !!! public void event() throws RainedOut {}
// Si el método no existe ya en la clase
// base, la excepción esta bien:
public void rainHard() throws RainedOut {}
// Se puede elegir no lanzar ninguna excepción,
// aún si la clase base lo hace:
public void event() {}
// Los métodos sobrescritos pueden lanzar
// excepciones heredadas:
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 error");
    }
    // Strike no lanza en la versión derivada.
    try {
        // Que sucede si se realiza una conversión ascendente?
        Inning i = new StormyInning();
        i.atBat();
        // Se deben capturar excepciones de la version de la
        // clase base del método:
    } 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 el constructor y el método **event()** expresan que lanzarán una excepción, pero nunca lo hacen. Esto es legal porque permite que se fuerce al usuario a atrapar cualquier excepción que pueda ser agregada en versiones sobrescritas de **event()**. La misma idea se mantiene para métodos abstractos, como se ve en **atBat()**.

La interfase de **Storn** es interesante porque contiene un método (**event()**) que esta definido en **Inning**, y un método que no lo esta. Ambos métodos

lanzan un nuevo tipo de excepción, **RainedOut**. Cuando **StormyInning** **extends Inning** e **implements Strom**, se podrá ver que el método **event()** en **Strom** *no puede* cambiar la interfase de **event()** en **Inning**. Nuevamente, esto tiene sentido porque de otra forma nunca se sabe si se está capturando lo correcto cuando se está trabajando con la clase base. Claro, si un método descrito en una **interfase** no está en la clase base, como lo es **rainHard()**, entonces no hay problema si lanza una excepción.

La restricción de las excepciones no se aplica a los constructores. Se puede ver en **StormyInning** que un constructor puede lanzar cualquier cosa que se quiera, sin importar de que lanza el constructor de la clase base. Sin embargo, dado que un constructor de la clase base debe siempre ser llamado de una forma u otra (aquí, el constructor por defecto es llamado automáticamente), el constructor de la clase derivada debe declarar cualquier excepción en el constructor de la clase base en su especificación de excepciones. Debe notarse que un constructor de una clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.

La razón por la cual **StormyInning.walk()** no compilara es que lanza una excepción, donde **Inning.walk()** no lo hace. Si esto fuera permitido, entonces se puede escribir código que llama a **Inning.walk()** y que no tenga que manejar ninguna excepción, pero cuando se substituye un objeto de la clase derivada de **Inning**, las excepciones pueden ser lanzadas así es que su código se puede quebrar. Forzando los métodos de la clase derivada para amoldarse a las especificaciones de los métodos de la clase base, la capacidad para sustituir los objetos es mantenida.

El ejemplo sobreescrito **event()** muestra que la versión de la clase derivada de un método puede elegir no lanzar ninguna excepción, aún si la versión de la clase base lo hace. Nuevamente, esto está bien dado que no quiebra ningún código que sea escrito -asumiendo que la versión de la clase base lanza excepciones. Una lógica similar se aplica a **atBat()**, que lanza **PopFoul**, una excepción que es derivada de **Foul** lanzada por la versión de la clase base de **atBat()**. Esta forma, si alguien escribe código que trabaje con **Inning** y llamadas a **atBat()**, estas deben capturar la excepción **Foul**. Dado que **PopFoul** es derivada de **Foul**, el manejador de excepciones capturará también **PopFoul**.

El último punto de interés se encuentra en el **main()**. Aquí se puede ver que si se está tratando con objetos **StormyInning** y no otros, el compilador fuerza a atrapar solo las excepciones que son específicas de esa clase, pero si se realiza una conversión ascendente a el tipo de la clase base entonces el compilador (correctamente) fuerza a capturar las excepciones para el tipo

base. Todas estas limitaciones producen código de manejo de excepciones mas robusto<sup>5</sup>.

Es útil darse cuenta que a pesar de que las especificaciones de excepciones son implementadas por el compilador durante la herencia, las especificaciones de excepciones no son parte del tipo en un método, lo que abarca solo en nombre del método y el tipo de argumento. Por lo tanto, no se puede sobrescribir métodos basados en especificaciones de excepciones. Además solo porque una especificación de excepciones existe en una versión de la clase base de un método no significa que debe existir en la versión del método para la clase derivada. Esto es muy diferente de las reglas de herencia, donde un método en la clase base debe también existir en la clase derivada. Poniéndolo de otra forma, la “interfase de especificación de excepciones” par un método en particular puede estrecharse durante la herencia y cuando se sobrescribe, pero esto puede ampliarse -esto es precisamente el opuesto a la regla de la interfase de clases durante la herencia.

## Constructores

Cuando se escribe un código con excepciones, es particularmente importante que se siempre se pregunte, “¿Si una excepción ocurre, será propiamente limpia?” La mayoría del tiempo se esta bastante seguro, pero en los constructores hay un problema. El constructor coloca el objeto dentro de un estado seguro de arranque, pero puede realizar algunas operaciones -como abrir un fichero- que no son limpiadas hasta que el usuario ha terminando con el objeto y llama un método de limpieza especial. Si se lanza una excepción dentro de un constructor, este comportamiento de limpieza puede no suceder correctamente. Esto significa que se debe ser especialmente diligente cuando se escriba su constructor.

Dado que solamente se ha aprendido acerca de **finally**, se puede pensar que esta es la solución correcta. Pero no es tan simple, dado que **finally** realiza la limpieza del código *todo el tiempo*, aún en situaciones en donde no se quiere ejecutar código de limpieza hasta que el método de limpieza se ejecute. De esta forma, si se realiza la limpieza en **finally** se debe configurar algún tipo de bandera cuando el constructor finalice correctamente y de esta forma no hacer nada en el bloque **finally** si la bandera esta configurada. Dado que no es particularmente elegante (se esta asociando el código de un lugar a otro),

---

<sup>5</sup> ISO C++ agrega limitaciones similares que requieren excepciones en métodos derivados para hacer lo mismo, o derivados de, las excepciones lanzadas por el método de la clase base. Esto es un caso en donde C++ actualmente es capaz de verificar especificaciones de excepciones en tiempo de compilación.

es mejor que se intente evitar este tipo de limpieza en **finally** a no ser que se este forzado.

En el siguiente ejemplo, una clase llamada **InputFile** es creada que abre un fichero y permite leer una línea (convertida a **String**) a la vez. Se utilizan las clases **FileReader** y **BufferedReader** de la librería estándar de Java que serán tratadas en el Capítulo 11, pero que son suficientemente simples para que probablemente no se tenga problema entendiendo su uso básico:

```
///: c10:Cleanup.java
// Prestar attention a las excepciones
// en constructores.
import java.io.*;
class InputFile {
    private BufferedReader in;
    InputFile(String fname) throws Exception {
        try {
            in =
                new BufferedReader(
                    new FileReader(fname));
            // Otro código que puede lanzar excepciones
        } catch(FileNotFoundException e) {
            System.err.println(
                "Could not open " + fname);
            // No fué abierto, así es que no se cierra
            throw e;
        } catch(Exception e) {
            // Todas las otras excepciones deben cerrarse
            try {
                in.close();
            } catch(IOException e2) {
                System.err.println(
                    "in.close() unsuccessful");
            }
            throw e; // Rethrow
        } finally {
            // No cierre aquí!!!
        }
    }
    String getLine() {
        String s;
        try {
            s = in.readLine();
        } catch(IOException e) {
            System.err.println(
                "readLine() unsuccessful");
            s = "failed";
        }
        return s;
    }
    void cleanup() {
        try {
            in.close();
        } catch(IOException e2) {
```

```

        System.err.println(
            "in.close() unsuccessful");
    }
}
public class Cleanup {
    public static void main(String[] args) {
        try {
            InputFile in =
                new InputFile("Cleanup.java");
            String s;
            int i = 1;
            while((s = in.getLine()) != null)
                System.out.println(""+ i++ + ": " + s);
            in.cleanup();
        } catch(Exception e) {
            System.err.println(
                "Caught in main, e.printStackTrace()");
            e.printStackTrace(System.err);
        }
    }
} //:~

```

El constructor para **InputFile** toma un argumento **String**, que es el nombre del fichero que se quiere abrir. Dentro del bloque **try**, se crea un **FileReader** utilizando el nombre de fichero. Un **FileReader** no es particularmente útil hasta que se convierta y se use para crear un **BufferedReader** con el cual se puede establecer una comunicación -debe notarse que uno de los beneficios de **InputFile** es que combina estas dos acciones.

Si el constructor de **FileReader** no es exitoso, lanza una **FileNotFoundException**, que debe ser capturada separadamente porque es el único caso en donde no se quiere cerrar el fichero porque *fue* abierto en el momento en que se entraba a las cláusulas **catch** (Claro, esto es difícil si más de un método puede lanzar un **FileNotFoundException**. En ese caso, debe querer dividir las cosas en muchos bloques **try**). El método **close()** puede lanzar una excepción o sea que es probado y atrapado aún cuando esta dentro del bloque de otra cláusula **catch** -esto es solo otro par de llaves para el compilador de Java. Luego de realizar las operaciones locales la excepción es lanzada nuevamente, lo que es apropiado porque este constructor ha fallado, y no se quiere que la llamada al método asuma que el objeto fue creado adecuadamente y era válido.

En este ejemplo, que no usa la técnica antes mencionada de las banderas, la cláusula **finally** *no* es definitivamente el lugar para llamar a **close()** y cerrar el fichero, dado que lo cerrará cada vez que el constructor sea completado. Dado que queremos que el fichero sea abierto durante el tiempo de vida del objeto **InputFile** esto no sería apropiado.

El método **getLine()** retorna un **String** conteniendo la siguiente línea del fichero. Este llama a **readLine()**, que puede lanzar una excepción, pero esa

excepción es capturada así es que **getLine()** no lanza ninguna excepción. Uno de los temas de diseño con excepciones es cuando manejar una excepción completamente en este nivel, para manejarla parcialmente y pasar la misma excepción (o una diferente), o cuando pasarlá simplemente. Pasándola, cuando es apropiado, puede ciertamente simplificar el código. El método **getLine()** se convierte en:

```
| String getLine() throws IOException {  
|     return in.readLine();  
| }
```

Pero claro, el método que ha llamado ahora es responsable por el manejo de cualquier **IOException** que pueda originarse.

El método **cleanup()** debe ser llamado por el usuario cuando se finalice utilizando el objeto **Inputfile**. Esto liberará los recursos del sistema (como el manejador de fichero) que es utilizado por **BufferedReader** y/o por los objetos **FileReader**<sup>6</sup>. No se quiere hacer esto hasta que se ha finalizado con el objeto **Inputfile**, en el punto que se lo quiera dejar ir. De puede pensar que colocar alguna funcionalidad dentro de un método **finalize()** será llamado (aún si se *puede* asegurar que será llamado, no se sabe *cuando*). Esto es uno de las desventajas de Java: toda la limpieza -otras además de la memoria- no se suceden automáticamente, así es que se debe informar al cliente programador que son responsables, y posiblemente garantizar que la limpieza sucede utilizando **finalize()**.

En **Cleanup.java** e **Inputfile** es creada para abrir el mismo fichero fuente que crea el programa, el fichero es leído una línea a la vez, y son agregados números de línea. Todas las excepciones son capturadas genéricamente en **main()**, a pesar de que se puede elegir por una gran granularidad.

Uno de los beneficios de este ejemplo es mostrar como las excepciones son introducidas en este punto en el libro -no se puede crear entradas y salidas básicas sin utilizar excepciones. Las excepciones son integrales para programar en Java, especialmente porque el compilador las fuerza, lo que se puede lograr sin mucho conocimiento de como trabajar con ellas.

## Excepciones que coinciden

Cuando una excepción es lanzada, el sistema de manejo de excepciones mira los manejadores mas “cercanos” en el orden en que son escritos. Cuando encuentra una coincidencia, la excepción es considerada manejada, y ninguna otra búsqueda se realiza.

---

<sup>6</sup> En C++, un destructor puede manejar esto.

Que coincide una excepción no requiere una perfecta coincidencia entre la excepción y su manejador. Una clase derivada coincidirá un manejador para la clase base, como se muestra en este ejemplo:

```
//: c10:Human.java
// Catching exception hierarchies.
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}
public class Human {
    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");
        }
    }
} //:~
```

La excepción **Sneeze** será capturada por la primer cláusula **catch** que coincide -la que sea la primera, claro esta. Sin embargo, si se quita la primer cláusula de captura, dejando solo:

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.err.println("Caught Annoyance");
}
```

El código seguirá trabajando porque se esta capturando la clase base de **Sneeze**. Poniéndolo de otra forma, **catch(Annoyance e)** capturará una **Annoyance** o *cualquier clase derivada de esta*. Esto es útil porque si se decide agregar a un método mas excepciones derivadas, entonces el código del cliente programador no necesitará cambios mientras el cliente capture las excepciones de la clase base.

Si se trata de “disfrazar” las excepciones de la clase derivada poniendo la clase base el la clase base primero, como esto:

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
    System.err.println("Caught Annoyance");
} catch(Sneeze s) {
    System.err.println("Caught Sneeze");
}
```

El compilador dará un mensaje de error, dado que ve que la cláusula de captura de **Sneeze** nunca será alcanzada.

## Líneas guía para la excepción

Use excepciones para:

1. Solucionar el problema y llamar el método que produce la excepción nuevamente.
2. Arreglar las cosas y continuar sin intentar de nuevo con el método.
3. Calcular algún resultado alternativo en lugar de lo que el método se supone tiene que producir.
4. Hacer lo que se pueda en el contexto actual y lanzar la *misma* excepción a un contexto mas alto.
5. Hacer lo que se pueda en el contexto actual y lanzar una excepción *diferente* en un contexto mas alto.
6. Terminar el programa.
7. Simplificar (Si el esquema de la excepción hace cosas mas complicadas, entonces es doloroso y es molesto de utilizar).
8. Hacer su librería y programa mas seguro (Es una inversión a corto plazo para depurar, y una inversión a largo plazo (para robustez de aplicación)).

## Resumen

Mejorar la recuperación del error es una de las formas mas poderosas que se tiene para incrementar la robustez del código. La recuperación del error es una preocupación fundamental para cualquier programa que se escriba, pero es especialmente importante en Java, donde una de las metas principales es crear componentes de programa para que otros utilicen. *Para crear un sistema robusto, cada componente debe ser robusto*

Las metas para el manejo de excepciones en Java son simplificar la creación de programas largos y confiables utilizando la menor cantidad de código que sea posible, y con mas confianza de que su aplicación no tiene un error sin manejar.

Las excepciones no son horriblemente difíciles de aprender, y son uno se esas características que proporcionan beneficios inmediatos y significantes para su proyecto. Afortunadamente, Java implementan todos los aspectos de las excepciones así es que es garantido que ellas sean utilizadas consistentemente por los diseñadores de librerías y los clientes programadores.

# Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Cree una clase con un **main()** que lance un objeto de la clase **Exception** dentro de un bloque **try**. Dele el constructor para la **Exception** un argumento **String**. Capture la excepción dentro de una cláusula **catch** e imprima el argumento **String**. Agregue una cláusula **finally** e imprima el mensaje para probar que está ahí.
2. Cree su propia clase excepción utilizando la palabra clave **extends**. Escriba un constructor para esta clase que tome un argumento **String** y lo almacene dentro del objeto con una referencia **String**. Escriba un método que imprima el **String** almacenado. Cree una cláusula **try-catch** para ejercitarse con su nueva excepción.
3. Escriba una clase con un método que lance una excepción del tipo creado en el Ejercicio 2. Trate de compilarla sin una especificación de excepción para ver que dice el compilador. Agregue la especificación de excepción apropiada. Ponga a prueba su clase y su excepción dentro de una cláusula **try-catch**.
4. Defina una referencia a un objeto e inicialícela a **null**. Trate de llamar a un método con 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 nuevo tipo que defina. En **f()**, llame a **g()**, capture la excepción y, en la cláusula **catch**, lance una excepción diferente (de un segundo tipo que defina). Prueba su código en el **main()**.
6. Cree tres nuevos tipos de excepciones. Escriba una clase con un método que lance las tres. En el **main()**, llame el método pero solo use una sola cláusula **catch** que capturará los tres tipos de excepciones.
7. Escriba un código para generar y capturar un **ArrayIndexOutOfBoundsException**.
8. Cree su propio comportamiento análogo a una reanudación utilizando un bucle **while** que repita hasta que una excepción no sea lanzada.
9. Cree una jerarquía de excepciones de tres niveles. Ahora cree una clase base **A** con un método que lance una excepción en la base de su jerarquía. Herede **B** de **A** y sobre escriba el método así este lanza una excepción en el nivel dos de su jerarquía. Repita heredando la clase **C** de la **B**. En el **main()**, cree una **C** y realícela una conversión ascendente hacia **A**, luego llame el método.

10. Demuestre que un constructor de una clase derivada no puede capturar excepciones lanzadas por el constructor de su clase base.
11. Muestre que **OnOffSwitch.java** puede fallar lanzando una **RuntimeException** dentro del bloque **try**.
12. Muestre que **WithFinally.java** puede fallar lanzando una **RuntimeException** dentro del bloque **try**.
13. Modifique el ejercicio 6 agregando una cláusula **finally**. Verifique que su cláusula **finally** es ejecutada, aún si un **NullPointerException** es lanzada.
14. Cree un ejemplo donde se use una bandera de control si un código de limpieza es llamado, como se describe en el segundo párrafo luego del encabezado “Constructores”.
15. Modifique **StormyInning.java** agregando una excepción **UmpireArgument**, y los métodos que lanzan esta excepción. Pruebe la jerarquía modificada.
16. Quite la primer cláusula catch en **Human.java** y verifique que el código sigue compilando y corre correctamente.
17. Agregue un segundo nivel de excepción perdida a **LostMessage.java** de tal forma que **HoHumException** es remplazada por una tercer excepción.
18. En el Capítulo 5, encuentre los dos programas llamados **Assert.java** y modifique estos para lanzar sus propios tipos de excepción en lugar de imprimir a **System.err**. Esta excepción debe ser una clase interna que extienda **RuntimeException**.
19. Agregue un grupo apropiado de excepciones a **co8:GreenhouseControls.java**.

# 11: El sistema de E/S de Java

Crear un buen sistema de entrada/salida(E/S) es una de las tareas mas difíciles para el diseñador de lenguajes.

Esto queda en evidencia por la cantidad de diferentes estrategias. El reto parece ser cubrir todas las eventualidades. No solo tienen diferentes fuentes y sinks de E/S con los que se quiere comunicar (ficheros, la consola, conexiones de redes), sino que se necesita hablar con ellos en una amplia variedad de formas (secuencial, acceso aleatorio, con buffers, binario, caracteres, por líneas, palabras, etc.).

Los diseñadores de la librería de Java atacaron este problema creando muchas clases. De hecho, hay tantas clases para el sistema de E/S que puede ser intimidante al principio (irónicamente, los actuales diseños de la E/S de Java previene una explosión de clases). También hubo un cambio significante en la librería de E/S luego de Java 1.0, cuando la librería originalmente orientada a **byte** fue complementada con la orientada a **char**, clases basadas en Unicode. Como resultado hay una justa cantidad de clases para aprender antes de que se entienda lo suficiente la imagen de la E/S de Java para que se pueda utilizar adecuadamente. Además, es muy importante entender la historia evolutiva de la librería de E/S, aun si su primera reacción es “¡No me molesten con historia, solo muéstrenme como se utiliza!”. El problema es que sin la perspectiva histórica se comienzan a confundir con algunas de las clases y cuando utilizarlas o no.

Este capítulo dará una introducción a la variedad de clases de E/S en la librería estándar de Java y como utilizarlas.

## La clase **File**

Antes de introducirnos a las clases que actualmente leen y escriben datos en flujos, veremos una utilidad proporcionada con la librería para asistir en el manejo de temas de directorios de ficheros.

La clase **File** tiene un nombre engañoso -se puede pensar que se refiere a un fichero, pero no lo es. Puede representar tanto el *nombre* de un fichero particular como los *nombres* de un grupo de ficheros en un directorio. Si es un grupo de ficheros en un directorio, se puede preguntar por el grupo con

el método **list()**, y este retorna un arreglo de **String**. Le da sentido a retornar un arreglo en lugar de uno de las flexibles clases contenedoras porque el número de elementos es fijo, y si se quiere un listado de directorio diferente solo se crea un objeto **File** diferente. De hecho, “FilePath” podría haber sido un mejor nombre para la clase. Esta sección muestra un ejemplo del uso de esta clase, incluyendo la interfase asociada **FilenameFilter**.

## Una clase que crea lista de directorios

Supóngase que se quiere ver un listado de un directorio. El objeto **File** puede ser listado de dos formas. Si se llama a **list()** sin argumentos, se obtiene la lista completa que el objeto **File** contiene. Sin embargo, si se quiere una lista clasificada -por ejemplo, si se quiere todos aquellos ficheros con una extensión de **.java**- entonces se puede utilizar un “filtro de directorio”, que es una clase que indica como seleccionar los objetos de **File** para desplegar.

He aquí el código para el ejemplo. Note que el resultado ha sido fácilmente ordenado (alfabéticamente) utilizando el método **java.util.Arrays.sort()** y el **AlphabeticComparator** definido en el capítulo 9:

```
//: 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) {
        // información desnuda de la ruta:
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
} ///:~
```

La clase **DirFilter** “implementa” la interfase **FilenameFilter**. Esta es útil para ver que tan simple la interfase **FilenameFilter** es:

```
public interface FilenameFilter {  
    boolean accept(File dir, String name);  
}
```

Esta dice todo lo que el tipo de objeto hace es proporcionar una llamada al método **accept()**. La única razón detrás de la creación de esta clase es proporcionar el método **accept()** para el método **list()** de tal forma que **list()** pueda devolverle la llamada a **accept()** para determinar que nombres de fichero deben ser incluidos en la lista. De esta forma, esta técnica es a menudo referida como una *callback* a veces como *functor* (esto es, **DirFilter** es un functor porque su único trabajo es almacenar un método) o el *Command Pattern*. Porque **list()** toma un objeto **FilenameFilter** como argumento, esto significa que se puede pasar un objeto de cualquier clase que implemente **FilenameFilter** para elegir (aún en tiempo de ejecución) como el método **list()** debe comportarse. El propósito de la devolución de la llamada es proporcionar flexibilidad en el comportamiento del código.

**DirFilter** muestra que solo porque una interfase contiene solo un grupo de métodos, se debe estar restringido a escribir solo estos métodos (se debería por lo menos proporcionar definiciones para todos los métodos es una interfase, sin embargo). En este caso, el constructor de **DirFilter** es también creado.

El método **accept()** debe aceptar un objeto **Fichero** representando el directorio donde se encuentra un fichero en particular, y un **String** conteniendo el nombre de este fichero. Se puede elegir utilizar o ignorar estos argumentos, pero probablemente al menos se utilizará el nombre de fichero. Se debe recordar que el método **list()** llama a **accept()** por cada uno de los nombres de ficheros encontradas en el objeto directorio para ver cual debe ser incluido -esto es indicado por un resultado **boolean** retornado por **accept()**. Para asegurarse que el elemento con el que se está trabajando solo es el nombre del fichero y no contiene información de la ruta, todo lo que se tiene que hacer es tomar el objeto **String** y crear un objeto **File** fuera de él, luego llamar a **getName()**, que quita toda la información de ruta (de una forma independiente de la plataforma). Luego **accept()** utiliza el método **indexOf()** de la clase **String** para ver si la cadena de búsqueda **afn** aparece en algún lado en el nombre del fichero. Si **afn** se encuentra dentro de la cadena, el valor retornado es el índice inicial de **afn**, pero si no se encuentra el valor de retorno es -1. Se debe tener presente que esto es una simple búsqueda de cadenas y no tiene ni una gota de expresiones de correspondencia con comodines -como lo es “fo?.b?r”- que es mucho más difícil de implementar.

El método **list()** retorna un arreglo. Se puede indagar este arreglo por su largo y luego moverse a través de él seleccionando los elementos. Esta

habilidad de pasar un arreglo dentro y fuera de un método es una tremenda mejora sobre el comportamiento de C y C++.

## Clases internas anónimas

Este ejemplo es ideal para rescribir utilizando una clase anónima interna (descrita en el Capítulo 8). Como una primera aproximación, se crea un método **filter()** que retorna una referencia a **FilenameFilter**:

```
//: c11:DirList2.java
// Uso de las clases anónimas internas.
import java.io.*;
import java.util.*;
import com.bruceekel.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) {
                // Desnuda la información de ruta:
                String f = new File(n).getName();
                return f.indexOf(fn) != -1;
            }
        }; // Fin de la clase anónima interna
    }
    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]);
    }
} ///:~
```

Debe notarse que el argumento para **filter()** debe ser **final**. Esto es requerido por la clase interna anónima así es que puede utilizar un objeto fuera de su alcance.

Este diseño es una mejor porque la clase **FilenameFilter** esta ahora estrechamente unida a **DirList2**. Sin embargo, se puede ir mas allá con esta estrategia y definir la clase anónima interna como un argumento para **list()**, en tal caso es aún mas pequeña:

```
//: c11:DirList3.java
// Creando la clase anónima interna "en el lugar."
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 para **main()** es ahora **final**, dado que la clase anónima interna utiliza **args[0]** directamente.

Para mostrar como una clase anónima interna permite la creación de clases rápidas y sucias para solucionar problemas. Dado que todo en Java gira alrededor de las clases, esto puede ser una técnica de codificación útil. Uno de los beneficios es que mantiene el código que soluciona un problema en particular aislado junto en un sitio. Por otro lado, no es siempre tan fácil de leer, así es que debe utilizarse juiciosamente.

## Probando y creando directorios

La clase **File** es mas que simplemente una representación para un fichero o directorio existente. También se puede utilizar un objeto **File** para crear un nuevo directorio o una ruta completa si no existe. Se puede también ver las características de los ficheros (tamaño, la fecha de la última modificación, lectura o escritura), ver cuando un objeto **File** representa un fichero o un directorio, y borrar el fichero. Este programa muestra alguno de los otros métodos disponibles con la clase **file** (si se quiere ver el grupo completo vea en la documentación HTML de *java.sun.com*):

```

//: 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 **fileData()** se puede ver varios métodos de investigación de ficheros utilizados para desplegar información acerca de la ruta de un fichero o directorio.

El primer método que es ejercitado por **main()** es **renameTo()**, que permite cambiar el nombre (o mover) un fichero a una ruta totalmente nueva representada por el argumento, que es otro objeto **File**. Esto también trabaja con directorios de cualquier largo.

Si se experimenta con el programa mas arriba, encontrará que se puede crear una ruta a directorio de cualquier complejidad porque **mkdirs()** hará todo el trabajo.

## Entrada y salida

Las librerías de E/S a menudo usan la abstracción de un *flujo*, que representa cualquier fuente de datos o drenaje como un objeto capaz de producir o recibir pedazos de datos. El flujo oculta los detalles de que sucede con los datos dentro del dispositivo actual de E/S.

La librería de clases de Java para la E/S esta dividida en entrada y salida, como se puede ver en la jerarquía de clases de Java en línea con un navegador Web. Por herencia, todo lo derivado de las clases **InputStream** o **Reader** tienen métodos básicos llamados **read()** para leer un solo byte o arreglos de bytes. De la misma forma, todas las cosas derivadas de las clases **OutputStream** o **Writer** tienen métodos básicos llamados **write()** para escribir un solo byte o un arreglo de bytes. Sin embargo, generalmente no se pueden utilizar estos método; estos existen para que otras clases puedan utilizarlos -estas otras clases proporcionan una interfase mas útil. De esta forma, raramente se crea un objeto flujo utilizando una sola clase, en lugar de eso acomodaremos múltiples objetos juntos para proporcionar la funcionalidad deseada. El hecho de que se cree mas de un objeto para crear un solo flujo resultante es la razón primaria para que la librería de flujo de Java sea confusa.

Es útil categorizar las clases por su funcionalidad. En Java 1.0, los diseñadores de la librería comenzaron por decidir que todas las clases que tengan algo que hacer con la entrada serían heredados de **InputStream** y todas las clases que son asociadas con la salida serían heredadas de **OutputStream**.

# Tipos de InputStream

El trabajo de **InputStream** es representar clases que produzcan entrada de diferentes fuentes. Estas fuentes pueden ser:

1. Un arreglo de bytes.
2. Un objeto **String**.
3. Un fichero.
4. Una “tubo” o “pipe”, que trabaja como un tubo físico: se colocan cosas en un lado y aparecen por el otro.
5. Una secuencia de otros flujos, así se pueden juntar en un solo flujo.
6. Otras fuentes, como una conexión a la Internet (Esto será discutido en capítulo mas adelante).

Cada uno de estos tiene una subclase asociada de **InputStream**. Además, el **FilterInputStream** también es un tipo de **InputStream**, para proporcionar una clase base para clases “decoradoras” que enganchan atributos o interfaces útiles a los flujos de entrada. Esto se discute mas adelante.

**Tabla 11-1. Tipos de InputStream**

Class	Function	Argumentos del constructor
		Como utilizarlos
<b>ByteArray-InputStream</b>	Permite que un buffer en memoria sea utilizado como un <b>InputStream</b>	El buffer de donde se extraen los datos  Como una fuente de datos. Conectada a un objeto <b>FilterInputStream</b> para proporcionar una interfase útil
<b>StringBuffer-InputStream</b>	Convierte un <b>String</b> en un <b>InputStream</b>	Un <b>String</b> . La implementación de capas mas bajas actualmente utiliza un <b>StringBuffer</b> .  Como una fuente de datos. Conectada a un objeto <b>FilterInputStream</b> para proporcionar una interfase útil.
<b>File-InputStream</b>	Para leer información de un fichero	Un <b>String</b> representando el nombre del fichero, o un objeto <b>File</b> o un objeto <b>FileDescriptor</b> .

		Como una fuente de datos. Conectada a un objeto <b>FilterInputStream</b> para proporcionar una interfase útil.
<b>Piped-InputStream</b>	Produce que los datos sean escritos en la <b>PipedOutputStream</b> asociada. Implementa el concepto de canalización.	<b>PipedOutputStream</b> Como una fuente de datos en multihilado. Conectado a un objeto <b>FilterInputStream</b> para proporcionar una interfase útil.
<b>Sequence-InputStream</b>	Convierte dos o mas objetos <b>InputStream</b> en un solo <b>InputStream</b> .	Dos objetos <b>InputStream</b> o una <b>Enumeration</b> de contenedores de objetos <b>InputStream</b> .  Como una fuente de datos. Conectada a un objeto <b>FilterOutputStream</b> para proporcionar una interfase útil.
<b>Filter-InputStream</b>	Clase abstracta que es una interfase para decoradores que proporcionan funcionalidades útiles a otras clases <b>InputStream</b> . Vea la tabal 11-3.	Vea la tabal 11-3.  Vea la tabal 11-3.

## Tipos de OutputStream

Esta categoría incluye las clases que deciden donde irá la salida: un arreglo de bytes (no un **String**, sin embargo; presumiblemente se puede crear uno utilizando el arreglo de bytes), un fichero, o una “tubería”.

Además, el **FilterOutputStream** proporciona una clase base para clases “decoradoras” que enganchan atributos o interfaces útiles a los flujos de salida. eso será discutido mas tarde.

**Tabla 11-2. Tipos de OutputStream**

Clase	Función	Argumentos del constructor
		<b>Como utilizarla</b>
<b>ByteArray-OutputStream</b>	Crea un buffer en memoria. Todos los datos que se envían a él	Tamaño inicial del buffer opcional.

	datos que se envían a el flujo es colocado en este buffer.	Para designar el destino de los datos. Debe conectarse a un objeto <b>FilterOutputStream</b> que provea una interfase útil.
<b>File-OutputStream</b>	Para enviar información a un fichero	Una cadena representando en nombre del fichero, o un objeto <b>File</b> o un objeto <b>FileDescriptor</b> .
		Para designar el destino de los datos. Debe conectarse a un objeto <b>filterOutputStream</b> para proporcionar una interfase útil.
<b>Piped-OutputStream</b>	Cualquier información que se escriba aquí automáticamente termina como entrada para la <b>PipedInputStream</b> asociada. Implementando el concepto de “canalización”	<b>PipedInputStream</b> Para designar el destino de los datos para multihilado. Debe conectarse a un objeto <b>FilterOutputStream</b> para proporcionar una interfase útil.
<b>Filter-OutputStream</b>	Clase abstracta que es una interfase para decoradores que proporcionan funcionalidad útil para las otras clases <b>OutputStream</b> . Vea la tabla 11-4.	Vea la tabla 11-4. Vea la tabla 11-4.

## Agregando atributos y interfaces útiles

El uso de objetos en capas para agregar responsabilidades de forma dinámica y transparente a objetos individuales es referido como patrones

*Decoradores* (Patrones<sup>1</sup> son el tema de *Pensando en patrones con Java* el que se puede bajar en [www.BruceEckel.com](http://www.BruceEckel.com)). Los patrones decoradores especifican que todos los objetos que envuelven los objetos iniciales tienen la misma interfase. esto hace el uso básico de los decoradores transparente - se envía el mismo mensaje a un objeto esté decorado o no. Esta es la razón para la existencia de clases “filtro” en la librería de E/S de Java: las clases abstractas “filtros” son la clase base para todos los decoradores (Un decorador debe tener la misma interfase que el objeto que esta decorando, pero el decorador puede también extender la interfase, lo que ocurre en muchas de las clases “filtros”).

Los decoradores son a menudo utilizado donde realizar simples subclases resulta en una gran cantidad de subclases para satisfacer cada combinación posible que se necesite -de esta manera muchas subclases se vuelven imprácticas. La librería de E/S de Java requiere muchas combinaciones diferentes de características, que es por que el patrón decorador es utilizado. Esto es un inconveniente para los patrones decoradores, sin embargo. Los decoradores dan mucho mas flexibilidad cuando se esta escribiendo un programa (dado que se puede fácilmente mezclar y hacer corresponder atributos), pero agregan complejidad a el código. La razón para la que la librería de E/S de Java es complicada de utilizar es que se debe crear muchas clases -el tipo “corazón” de E/S mas todos los decoradores- para obtener el único objeto de E/S que se quiere.

Las clases que proporcionan la interfase decoradora para controlar un **InputStream** o un **OutputStream** particular son las **FilterInputStream** y **FilterOutputStream** - que no tienen nombres muy intuitivos.

**FilterInputStream** y **FilterOutputStream** son clases abstractas que son derivadas de las clases base de la librería de E/S, **InputStream** y **OutputStream**, que son el requerimiento clave del decorador (así es que proporcionan la interfase común a todos los objetos que son decorados).

## Leyendo de un **InputStream** con **FilterUIInputStream**

Las clases **FilterInputStream** logran dos cosas significativamente diferentes. **DataInputStream** permite leer diferentes tipos de datos primitivos de la misma forma que objetos **String** (Todos los métodos comienzan con “read”, como **readByte()**, **readFloat()**, etc.). Esto, junto con su compañera **DataOutputStream**, permite mover datos primitivos de un lugar a otra mediante un flujo. Estos “lugares” son determinados por las clases en la Tabla 11-1.

---

<sup>1</sup> *Design Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995.

Las clases restantes modifican la forma en que **InputStream** se comporta internamente: si utiliza un buffer o no, si lleva un control de las líneas que lee (permitiendo que se pregunte por el número de líneas o configurar el número de línea), y cuando se puede rechazar un solo carácter. Las dos últimas clases se ven mucho como soporte para construir un compilador (esto es, son agregadas para apoyar la construcción del compilador de Java), así es que probablemente no las use en programación general.

Probablemente se necesite un buffer en su entrada al menos todo el tiempo, independientemente del dispositivo de E/S al que se esté conectando, así es que tiene mas sentido para la librería de E/S hacer una excepción (o simplemente una llamada a un método) para una entrada que no utilice un buffer en lugar de una que si lo utilice.

Tabla 11-3. Tipos de FilterInputStream

Clase	Función	Argumentos del constructor
		Como utilizarla
<b>Data-InputStream</b>	Utilizada coordinada con <b>DataOutputStream</b> , de tal forma que se puedan leer primitivas ( <b>int</b> , <b>char</b> , <b>long</b> , etc.) de un flujo en una forma portátil.	<b>InputStream</b> Contiene una interfase entera que permite leer tipos primitivos
<b>Buffered-InputStream</b>	Se debe utilizar para prevenir una lectura física cada vez que se quiera mas datos. Se está diciendo “Utilice un buffer” .	<b>InputStream</b> , con un tamaño de buffer opcional. Esta no proporciona una interfase <i>de por si</i> , solo una exigencia de que un buffer sea utilizado. Enganchando un objeto interfase.
<b>LineNumber-InputStream</b>	Mantiene la pista de los números de línea en la entrada de flujo; se puede llamar a <b>getLineNumber()</b> y <b>setLineNumber(int)</b> .	<b>InputStream</b> Esto solo agrega numeración de líneas, así es que probablemente se enganche un objeto interfase.

<b>Puchback-InputStream</b>	Tiene un buffer de rechazo de un solo carácter de tal forma que se puede rechazar el último carácter leído.	<b>InputStream</b>
		Generalmente utilizado en la búsqueda por un compilador y probablemente incluido porque el compilador de Java lo necesita. Probablemente nunca se utilice.

## Escribiendo un OutputStream con FilterOutputStream

El complemento de **DataInputStream** es **DataOutputStream**, que organiza los datos de cada uno de los tipos primitivos y objetos **String** en un flujo de tal forma que cualquier **DataInputStream**, en cualquier máquina, pueda leerlo. Todos los métodos comienzan con “write”, como **writeByte()**, **writeFloat()**, etc.

El intento original de **PrintStream** fue imprimir todos los tipos de datos primitivos y objetos **String** en un formato que se pudiera ver. Esto es diferente de **DataOutputStream**, cuya meta es colocar elementos de datos en un flujo de una forma en que **DataInputStream** puede reconstruirlo sin de forma portátil.

Los dos métodos importantes en **PrintStream** son **print()** y **println()**, que son sobrecargados para imprimir todos los distintos tipos. La diferencia entre **print()** y **println()** es que el último agrega una nueva línea cuando termina.

**PrintStream** puede ser problemático porque atrapa todas las **IOExceptions** (Se debe explicitamente verificar el estado de error con **checkError()**, que retorna **true** si un error ha sucedido). También, **PrintStream** no esta correctamente internacionalizado y no maneja los saltos de línea de una forma independiente de la plataforma (estos problemas son solucionados con **PrintWriter**).

**BufferedOutputStream** es un modificador e indica el flujo a utilizar con buffer así es que no se tiene una escritura física cada vez que se escribe en el flujo. Probablemente siempre se quiera utilizar esto con ficheros, y posiblemente con la E/S de consola.

**Tabla 11-4. Tipos de FilterOutputStream**

Clase	Función	Argumentos del constructor
		Como utilizarlo
<b>Data-OutputStream</b>	Utilizado en coordinación con <b>DataInputStream</b> así es que se puede escribir primitivas (int, char, long, etc.) a un flujo en una forma portátil.	<b>OutputStream</b> Contiene una interfase completa para permitir escribir tipos primitivos.
<b>PrintStream</b>	Para producir una salida con formato. Donde <b>DataOutputStream</b> maneja el <i>almacenamiento de datos</i> , <b>PrintStream</b> maneja el <i>despliegue</i> .	<b>OutputStream</b> , con un <b>boolean</b> opcional indicando que el buffer es limpiado con cada nueva línea.  Suele ser la envoltura “final” para su objeto <b>OutputStream</b> . Normalmente se utiliza mucho.
<b>Buffered-OutputStream</b>	Use este para prevenir escritura física cada vez que se envía una pieza de datos. Se está diciendo “Utilice un buffer”. Se puede llamar a <b>flush()</b> para limpiar el buffer.	<b>OutputStream</b> , con un tamaño de buffer opcional.  Esto no proporciona una interfase de por si, solo un requerimiento de que un buffer es utilizado. Enganche un objeto interfase.

## Readers & Writers

Java 1.1 realiza algunas modificaciones significantes a la librería de flujo de E/S fundamental (Java 2, sin embargo, no tiene modificaciones fundamentales). Cuando se ven las clases **Reader** y **Writer** el primer pensamiento (que fue el que tuve yo) puede ser que estas dieran a entender que se debe reemplazar las clases **InputStream** y **OutputStream**. Pero este no es el caso. A pesar de que algunos aspectos de la librería de flujo original son deprecadas (si se utilizan estas se recibirá una advertencia del compilador), las clases **InputStream** y **OutputStream** siguen proporcionando funcionalidad valiosa en la forma de E/S orientada a **byte**, donde las clases **Reader** y **Writer** obedecen a el estándar Unicode, en la E/S basadas en caracteres. Además:

1. Java 1.1 agrega nuevas clases en la jerarquía de **InputStream** y **OutputStream**, así es que es obvio aquellas clases que no fueron reemplazadas.
2. Hay momentos donde se debe utilizar clases de la jerarquía de “byte” *en combinación* con clases en la jerarquía de “caracteres”. Para lograr esto hay clases “puente”: **InputStreamReader** convierte un **InputStream** en un **Reader** y **OutputStreamWriter** convierte un **OutputStream** en un **Writer**.

La razón mas importante para las jerarquías de **Reader** y **Writer** es para internacionalización. La vieja jerarquía de flujo de E/S soporta solo flujos de 8 bytes y no maneja caracteres Unicode de 16 bits bien. Dado que Unicode es utilizado para internacionalización (y los **char** nativos de Java son Unicode de 16 bits), las jerarquías del **Reader** y **Writer** donde se agregan para soportar en todas las operaciones de E/S. Además, las nuevas librerías son diseñadas para operaciones mas rápidas que las viejas.

Como es la práctica en este libro, intentaremos proporcionar una visión general de las clases, pero se asume que se utilizará documentación para determinar todos los detalles, como la exhaustiva lista de métodos.

## Fuentes y sinks de datos

Al menos todas las clases de flujos de E/S de Java tienen su clase **Reader** y **Writer** para proporcionar manipulación Unicode nativa. Sin embargo, hay algunos lugares donde los **InputStreams** y **OutputStreams** orientados a **byte** son la solución correcta; en particular, las librerías **java.util.zip** son orientadas a **byte** en lugar de orientarse a **char**. Así es que la estrategia mas sensible para tomar es *tratar de utilizar Reader y Writer donde quiera que se pueda*, y si se descubre situaciones donde se tiene que utilizar librerías orientadas a **byte** porque su código no compila.

Aquí hay una tabla que muestra la correspondencia entre las fuentes y sinks de información (esto es, donde los datos físicamente llegan o se van) en dos jerarquías.

Fuentes y Sinks: Java 1.0 clase	Clase Java correspondiente
<b>InputStream</b>	Convertidor <b>Reader</b> : <b>InputStreamReader</b>
<b>OutputStream</b>	Convertidor <b>Writer</b> : <b>InputStreamReader</b>
<b>FileInputStream</b>	<b>FileReader</b>
<b>FileOutputStream</b>	<b>FileWriter</b>

<b>StringBufferInputStream</b>	<b>StringReader</b>
(sin clase correspondiente)	<b>StringWriter</b>
<b>ByteArrayInputStream</b>	<b>CharArrayReader</b>
<b>ByteArrayOutputStream</b>	<b>CharArrayWriter</b>
<b>PipedInputStream</b>	<b>PipedReader</b>
<b>PipedOutputStream</b>	<b>PipedWriter</b>

En general, se encontrará que las interfaces para las dos diferentes jerarquías son similares pero no idénticos.

## Modificando el comportamiento del flujo

Para **InputStreams** y **OutputStreams**, los flujos fueron adaptados para necesidades particulares utilizando subclases “decoradoras” de **FilterInputStream** y **FilterOutputStream**. Las clases **Reader** y **Writer** continúan con el uso de esta idea -pero no exactamente.

En la siguiente tabla, la correspondencia es una aproximación grosera en la tabla previa. La diferencia es a causa de la organización de las clases: donde **BufferedOutputStream** es una subclase de **FilterOutputStream**, **BufferedWriter** *no* es una subclase de **FilterWriter** (la cual, a pesar de que si bien es abstracta, no tiene subclases y de esta forma parece haber sido colocada como un placeholder o simple así es que no nos deberíamos preocupar por donde está). Sin embargo, las interfaces a las clases corresponden bastante mejor.

Filtros: Clases de Java 1.0	Clases de Java 1.1 correspondientes
<b>FilterInputStream</b>	<b>FilterReader</b>
<b>FilterOutputStream</b>	<b>FilterWriter</b> (clase abstracta sin subclases)
<b>BufferedInputStream</b>	<b>BufferedReader</b> (también como <b>readLine()</b> )
<b>BufferedOutputStream</b>	<b>BufferedWriter</b>
<b>DataInputStream</b>	Utilice <b>DataInputStream</b> (Excepto cuando se necesite utilizar <b>readLine()</b> , cuando se debe utilizar un <b>BufferedReader</b> )
<b>PrintStream</b>	<b>PrintWriter</b>
<b>LineNumberInputStream</b>	<b>LineNumberReader</b>

<b>StremTokenizer</b>	<b>StreamTokenizer</b> (utilice un constructor que tome un <b>Reader</b> en su lugar)
<b>PuchBackInputStream</b>	<b>PushBackReader</b>

Hay una dirección que es muy clara: Donde sea que se quiera utilizar **readLine()**, no se debe hacer con **DataInputStream** nunca mas (es informado por un mensaje de depreciación en tiempo de compilación), en lugar se de este se deba utilizar un **BufferedReader**. Fuera de eso, **DataInptStrem** sigue siendo un miembro “preferido” de la librería de E/S.

Para hacer la transición de utilizar un **PrintWriter** de forma simple, hay constructores que toman cualquier objeto **OutputStream**, de la misma forma que con objetos **Writer**. Sin embargo, **PrintWriterno** tiene mas soporte para formato que el que tiene **PrintStream**; las interfases son virtualmente las mismas.

El constructor de **PrintWriter** también tiene una opción para realizar una limpieza automática, lo que sucede luego de cada **println()** si la bandera del constructor esta configurada.

## Clases que no cambiaron

Algunas clases que no cambiaron entre Java 1.0 y Java 1.1:

Clases de Java 1.0 sin clases correspondientes en Java 1.1
<b>DataOutputStream</b>
<b>File</b>
<b>RandomAccessFile</b>
<b>SequenceInputStream</b>

**DataOutputStream**, en particular, es utilizada sin cambios, así es que para almacenar y recuperar datos en un formato transportable se debe utilizar las jerarquías **InputStream** y **OutputStream**.

## Fuera de sitio por si mismo: RandomAccessFile

**RandomAccessFile** es utilizada para ficheros que contienen registros de tamaño conocido en los que se puede mover de un registro a otro utilizando **seek()**, y luego leer o cambiar los registros. Los registros no tienen que tener

el mismo tamaño; simplemente ser capaz de determinar que tan grande son y donde están colocados en el fichero.

Al principio es un poquito duro creer que **RandomAccessFile** no es parte de la jerarquía de **InputStream** o **OutputStream**. Sin embargo, no tiene otra asociación con estas jerarquías que la que sucede de implementar las interfaces de **DataInput** y **DataOutput** (las que son también implementadas por **DataInputStream** y **DataOutputStream**). No utilizan tampoco ninguna de las clases existentes **InputStream** u **OutputStream** -es una clase completamente separada, escrita de la nada, con todo sus propios métodos (en su mayor parte nativos). La razón para esto puede ser que **RandomAccessFile** tiene un comportamiento esencialmente diferente que otros tipos de E/S, dado que nos podemos mover adelante y atrás sin un ficheros. En cualquier evento, hace todo sin ayuda, como un descendiente directo de **Object**.

Esencialmente, un **RandomAccessFile** trabaja como un **DataInputStream** colocado junto con un **DataOutputStream**, junto con los métodos de **getFilePointer()** para encontrar donde está el registro, **seek()** para moverse a un nuevo punto dentro del fichero, y **length()** para determinar el tamaño máximo del fichero. Además, los constructores requieren un segundo argumento (exactamente igual a **fopen()** en C) que indican si es simplemente una lectura aleatoria ("r") o es una lectura y escritura("rw"). No hay soporte para escribir ficheros solamente, lo que puede sugerir que **RandomAccessFile** puede trabajar bien si hubiera sido heredada de **DataInputStream**.

Los métodos de búsqueda están disponibles solo en **RandomAccessFile**, que trabaja solo con ficheros, **BufferedInputStream** permite mediante **mark()** marcar una posición (cuyo valor es guardado en una sola variable interna) y se puede realizar un **reset()** de esa posición, pero esto es limitado y no muy útil.

## Usos típicos de flujos de E/S

A pesar de que se pueden combinar las clases de flujos de E/S en muchas formas diferentes, probablemente solo se utilicen unas pocas combinaciones. El siguiente ejemplo puede ser utilizado como referencia básica; muestra la creación y uso de una configuración típica de E/S. Note que cada combinación comienza con un número comentado y un título que corresponde con el encabezada para la explicación adecuada que sigue en el texto.

```

//: c11:IOStreamDemo.java
// Typical I/O stream configurations.
import java.io.*;
public class IOStreamDemo {
    // Lanza excepciones a la consola:
    public static void main(String[] args)
        throws IOException {
        // 1. Leyendo la entrada por líneas:
        BufferedReader in =
            new BufferedReader(
                new FileReader("IOStreamDemo.java"));
        String s, s2 = new String();
        while((s = in.readLine())!= null)
            s2 += s + "\n";
        in.close();
        // 1b. Leyendo la entradas estandar:
        BufferedReader stdin =
            new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print("Enter a line:");
        System.out.println(stdin.readLine());
        // 2. Entrada desde memoria
        StringReader in2 = new StringReader(s2);
        int c;
        while((c = in2.read()) != -1)
            System.out.print((char)c);
        // 3. Entrada de memoria con formato
        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. Salida de fichero
        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. Almacenando y recuperando datos
        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));
// Se debe utilizar DataInputStream para datos:
System.out.println(in5.readDouble());
// Ahora se puede utilizar el readLine() 'apropiado':
System.out.println(in5br.readLine());
// Pero la linea se vuelve graciosa.
// La única creada con writeBytes esta OK:
System.out.println(in5br.readLine());
} catch(EOFException e) {
    System.err.println("End of stream");
}
// 6. Lectura/escritura de ficheros acceso aleatorio
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();
}
}

```

Aquí están las descripciones para las secciones numeradas del programa:

## Flujos de entrada

Desde la parte 1 a la 4 se demuestra la creación y uso de los flujos de entrada. La parte 4 también muestra el uso simple de un flujo de salida.

## 1. Fichero de entrada con buffer

Para abrir un fichero para la entrada de caracteres, se utiliza **FileInputStream** con un objeto **String** o **File** como nombre de fichero. Por un tema de velocidad, de va a querer que el fichero tenga un buffer así es se entrega la referencia resultante del constructor para **BufferedReader**. Dado que **BufferedReader** también proporciona el método **readLine()**, este es el objeto final y la interfase de donde se leerá. Cuando se alcance el final del fichero, **readLine()** retorna **null** así es que es utilizado para quebrar el bucle **while**.

El **String s2** utilizado para acumular el contenido completo del fichero (incluyendo saltos de línea que pueden ser agregados ya que **readLine()** los desnuda). **s2** es utilizado entonces en las porciones mas tardías de este programa. Finalmente, **close()** es llamado para cerrar el fichero. Técnicamente, **close()** es llamado cuando se ejecuta **finalize()**, y esto supuestamente sucede (se recolecte o no la basura) cuando se sale del programa. Sin embargo, esto puede ser inconsistentemente implementado, así es que la única estrategia segura es explícitamente llamar **close()** para los ficheros.

La sección 1b muestra como se puede envolver a **System.in** para leer la entrada de consola. **System.in** es un **DataInputStream** y un **BufferedReader** que necesita un argumento **Reader**, así es que **InputStreamReader** realiza la translación.

## 2. Entrada de memoria

Esta sección toma un **String s2** que ahora contiene la totalidad del contenido del fichero y lo utiliza para crear un **StringReader**. Entonces **read()** es utilizado para leer cada carácter de a una vez y enviarlos fuera a la consola. Debe verse que **read()** retorna el siguiente byte como un **int** y debe convertirse a **char** para imprimirse correctamente.

## 3. Entrada de memoria con formato

Para leer datos con “formato”, se debe utilizar **DataInputStream**, la cual es una clase de E/S orientada a bytes (en lugar de ser orientada a **char**). De esta manera de deben utilizar todas las clases **InputStream** en lugar de las clases **Reader**. Claro, se puede leer todo (como un fichero) como bytes utilizando la clase **InputStream**, pero aquí es utilizado un **String**. Para convertir un **String** a un arreglo de bytes, que es lo apropiado para un **ByteArrayInputStream**, el objeto **String** tiene un método **getBytes()** que hace el trabajo. En este punto, se tiene un **InputStream** para manejar para **DataInputStream**.

Si se lee los caracteres de un **DataInputStream** un byte a la vez utilizando **readByte()**, cualquier valor de byte es un resultado legítimo así es que el

valor de retorno no puede ser utilizado para detectar el final de la entrada. En lugar de eso, se puede utilizar el método **available()** para encontrar cuantos caracteres mas están disponibles. He aquí un ejemplo que muestra como leer un fichero de a un byte por vez:

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

Debe verse que **available()** trabaja diferente dependiendo del tipo de medio del cual se esta leyendo; es literariamente “el número de bytes que pueden ser leídos *sin bloqueo*”. Con un fichero esto significa la totalidad del fichero, pero con un tipo de flujo diferente esto puede no ser verdad, así es que debe utilizarse atentamente.

Se puede también detectar el final de la entrada en casos como este capturando una excepción. Sin embargo, el uso de excepciones para controlar flujo es considerado una mala utilización de esta característica.

## 4. Salida de fichero

El ejemplo también muestra como escribir datos en un fichero. Primero, un **FileWriter** es creado para conectar a el fichero. Virtualmente siempre se quiere agregar un buffer a la salida envolviéndola en un **BufferedWriter** (de puedes intentar quitando esta envoltura para ver el impacto en el rendimiento -colocar un buffer tiende a incrementar dramáticamente el rendimiento de las operaciones de E/S). Entonces para las formateadas son colocadas dentro de un **PrintWriter**. El fichero de datos creado de esta forma se puede leer como un fichero de texto común.

Así como las líneas son leídas a el fichero, los números de línea son agregados. Debe notarse **LineNumberInputStream** *no* es utilizado, dado que es una clase boba y no se necesita. Como es mostrado aquí, es trivial seguir el rastro de sus propios números de línea.

Cuando la entrada de flujo está exhausta, **readLine()** retorna **null**. Se podrá ver un explícito **close()** para **out1**, porque si no se llama a **close()** para todos

los ficheros de salida, se descubrirá que los buffers no se limpian así es que estarán incompletos.

## Flujos de salida

Los dos tipos primarios de flujos de salida son separados por la forma en que escriben los datos: uno los escribe para ser consumidos por humanos, y el otro lo escribe para ser readquiridos por un **DataInputStream**. El **RandomAccessFile** permanece solo, a pesar de que su formato de datos es compatible con el **DataInputStream** y el **DataOutputStream**.

### 5. Almacenando y recuperando datos

Así es que el formato de datos de **PrintWriter** se puede leer por humanos. Sin embargo, para que la salida de datos pueda ser recuperada por otro flujo, se utiliza un **DataOutputStream** para escribir los datos y un **DataInputStream** para recuperar los datos. Claro, estos flujos pueden ser cualquiera, pero un fichero es utilizado aquí, colocando un buffer para lectura y escritura. **DataOutputStream** y **DataInputStream** son orientados a bytes y de esta forma se requiere **InputStreams** y **OutputStreams**.

Si se utiliza **DataOutputStream** para escribir los datos, Java garantiza que se puedan recuperar los datos con exactitud utilizando **DataInputStream** sin interesar la plataforma. Esto tiene una importancia increíble, como todos sabes se pierde mucho tiempo preocupándose acerca de los temas de datos específicos de la plataforma. Este problema desaparece si se tiene Java en ambas plataformas<sup>2</sup>.

Debe notarse que la cadena de caracteres es escrita utilizando **writeChar()** y **writeBytes()**. Cuando se ejecuta el programa, se descubrirá que las salidas de **writeChars()** son de caracteres Unicode de 16 bits. Cuando se lea la línea utilizando **readLine()**, se vera que hay un espacio entra cada carácter, a causa de el byte extra insertado por Unicode. Dado que no hay métodos “readChars” complementarios en **DataInputStream**, se estarán adhiriendo estos caracteres una a la vez con **readChar()**. Así es que para ASCII, es mas fácil escribir caracteres como bytes seguidos de una nueva línea; entonces se utiliza **readLine()** para volver a leer los bytes como una línea ASCII común.

El método **writeDouble()** almacena el número **double** en el flujo y el **readDouble()** complementario lo recupera (hay métodos similares para la lectura y escritura de los otros tipos). Pero para que cualquiera de los métodos de lectura trabajen correctamente, se debe conocer la correcta

---

<sup>2</sup> XML es otra forma de solucionar el problema de mover datos a través de diferentes plaraformas, y no depende de si se tiene Java o no es todas las plataformas. Sin embargo, Existen herramientas de Java que soportan XML,

colocación de los ítems de datos en el flujo, dado que puede ser igualmente posible leer los **double** almacenados como una secuencia simple de bytes, o como **char**, etc. Así es que se debe tener un formato fijo para los datos o información extra debe ser almacenada en el fichero que se analiza para determinar donde están localizados los datos.

## 6. Leyendo y escribiendo ficheros de acceso aleatorio

Como previamente se ha notado, **RandomAccessFile** esta casi totalmente aislado del resto de la jerarquía de E/S, salvo por el hecho de que implementa las interfaces de **DataInput** y **DataOutput**. Así es que no se puede combinar con ninguno de los aspectos de las subclases **InputStream** y **OutputStream**. Aún cuando tenga sentido tratar un **ByteArrayInputStream** como un elemento de acceso aleatorio, se puede utilizar **RandomAccessFile** solo para abrir un fichero. Se debe asumir que **RandomAccessFile** tiene un apropiado buffer dado que no se puede agregar.

La única opción que se tiene es en el segundo argumento del constructor: se puede abrir un **RandomAccessFile** para leer (“**r**”) o leer y escribir (“**rw**”).

Utilizando un **RandomAccessFile** es como utilizar de forma combinada **DataInputStream** y **DataOutputStream** (porque se implementan interfaces equivalentes). Además, se puede ver que **seek()** es utilizado para moverse en el fichero y cambiar uno de los valores.

## ¿Un error de programación?

Si se observa en la sección 6, se podrá ver que los datos son escritos *antes* que el texto. Esto es porque un problema ha sido introducido en Java 1.1 (y persiste en Java 2) que seguramente parece un error de programación para mi, pero lo he reportado y las personas que corrigen los errores en JavaSoft dicen que esta es la forma en que supuestamente funciona (sin embargo, el problema *no* sucede en Java 1.0, lo que me hace sospechar). El problema es mostrado en el siguiente código:

```
//: c11:IOProblem.java
// Java 1.1 and higher I/O Problem.
import java.io.*;
public class IOProblem {
    // Lanza excepciones a la consola:
    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));
        // Los doubles escritos ANTES de la linea de texto
        // se leen correctamnte:
        System.out.println(in.readDouble());
        // Se leen las lineas de texto:
        System.out.println(inbr.readLine());
        System.out.println(inbr.readLine());
        // Intentan leer los doubles luego de la linea
        // una excepción de fin de fichero:
        System.out.println(in.readDouble());
    }
}
} //:~
```

Parece que todo lo que se escribe después de una llamada a **writeBytes()** no es recuperable. La respuesta es aparentemente la misma que la respuesta a la vieja broma de vodevil: “¡Doctor, duele cuando hago esto!” “¡No lo haga!”.

## Flujos dentro de pipes

**PipedInputStream**, **PipedOutputStream**, **PipedReader** y **PipedWriter** han sido mencionados solo brevemente en este capítulo. Esto no sugiere que no sean útiles, pero su valor no es aparentemente útil hasta que se comience a entender multihilado, dado que los flujos dentro de pipes son utilizados para comunicarse entre hilos. Esto está cubierto en un ejemplo en el capítulo 14.

## E/S estándar

El término *E/S estándarse* refiere al concepto Unix (que es reproducido en alguna forma en Windows y muchos otros sistemas operativos) de un solo flujo de información que ese utilizado por un programa. Todas las entradas de un programa pueden venir de la *entrada estándar* y toda la salida puede ir a la *salida estándar* y todos los mensajes de error pueden ser enviados a la salida de *error estándar*. El valor de la E/S es que los programas pueden ser fácilmente encadenados juntos y una salida estándar de un programa puede convertirse en entrada estándar de otro programa. Esto es una poderosa herramienta.

## Leyendo de la entrada estándar

Siguiendo el modelo estándar de E/S, Java tiene **System.in**, **System.out**, y **System.err**. En todo este libro se verá como escribir a la salida estándar utilizando **System.out**, que ya esta envuelta como un objeto **PrintStream**. **System.err** es como un **PrintStream**, pero **System.in** es un **InputStream**. **err** en bruto, sin envoltura. Esto significa que cuando se utilice **System.out** y **System.err** también. **System.in** debe ser envuelto antes de que se pueda leer de ella.

Típicamente, se querrá leer de la entrada una línea a la vez utilizando **readLine()**, así es que querrá envolver **System.in** en un **BufferedReader**. Para hacer esto, se debe convertir **System.in** en un **Reader** utilizando **InputStreamReader**. Aquí hay un ejemplo que realiza un eco de cada línea que se escribe:

```
//: c11:Echo.java
// Como leer de una entrada estándar.
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);
        // Una línea vacía termina el programa
    }
} ///:~
```

La razón para la especificación de excepción es que **readLine()** puede lanzar una excepción **IOException**. Debe notarse que **System.in** debe usualmente ser utilizada con un buffer, como la mayoría de los flujos.

## Cambiando System.out en un PrintWriter

**System.out** es un **PrintStream**, que es un **OutputStream**. **PrintWriter** tiene un constructor que toma un **OutputStream** como argumento. De esta manera, si se quiere se puede convertir **System.out** en un **PrintWriter** utilizando 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 utilizar la versión de **PrintWriter** que tiene dos argumentos en su constructor y configurar el segundo argumento a **true** para habilitar el flujo automático, de otra forma puede no verse la salida.

### Redirigiendo la E/S estándar

La clase **System** de Java permite redirigir los flujos de entrada estándar, salida, y error utilizando simples llamadas a métodos estáticos:

**setIn(InputStream)**

**setOut(PrintStream)**

**setErr(PrintStream)**

Redirigir la salida es especialmente útil si de pronto comienza a crear una gran cantidad de salida en su salida y el desplazamiento es mas rápido de lo que se puede leer<sup>3</sup>. Redirigir la entrada es de valor para un programa en línea de comandos en donde se quiere probar una secuencia de entrada de usuario repetidamente. He aquí un ejemplo simple que muestra el uso de estos métodos:

```

//: c11:Redirecting.java
// Demuestra la redirección de E/S estándar.
import java.io.*;
class Redirecting {
    // Lanza una excepción a la consola:
    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);
    }
}

```

---

<sup>3</sup> El Capítulo 13 muestra una solución mas conveniente para esto: un programa GUI con un área de texto desplazable.

```

        out.close(); // Remember this!
    }
} ///:~

```

El programa engancha la entrada estándar a un fichero, y redirige la salida estándar y el error estándar a otro fichero.

La redirección de E/S manipula flujos de bytes, no flujos de caracteres, de esta manera **InputStreams** y **OutputStreams** son utilizados en lugar de **Readers** y **Writers**.

## Compresión

La librería de E/S de Java contiene clases para soportar la lectura y la escritura de flujos en un formato comprimido. Estos son envueltos alrededor de las clases de E/S para proporcionar funcionalidad de compresión.

Estas clases no son derivadas de las **Reader** y **Writer**, pero en lugar de eso son parte de las jerarquías **InputStream** y **OutputStream**. Esto es porque la librería de compresión trabaja con bytes, no con caracteres. Sin embargo, a veces se puede forzar a mezclar los dos tipos de flujos (Recuerde que se puede utilizar **InputStreamReader** y **OutputStreamWriter** para proporcionar una fácil conversión entre un tipo y otro).

Clase de compresión	Función
<b>CheckedInputStream</b>	<b>GetChecksum()</b> produce la suma de chequeo para cualquier <b>InputStream</b> (no solo para descompresión).
<b>CheckedOutputStream</b>	<b>GetChecksum()</b> produce la suma de chequeo para cualquier <b>OutputStream</b> (no solo compresión).
<b>DeflaterOutputStream</b>	Clase base para clases de compresión
<b>ZipOutputStream</b>	Una <b>DeflaterOutputStream</b> que comprime datos dentro de un fichero de formato Zip.
<b>GZIPOutputStream</b>	Una <b>DeflaterOutputStream</b> que comprime datos dentro de un fichero con formato GZIP.
<b>InflaterInputStream</b>	Clase base para clases de descompresión.
<b>ZipInputStream</b>	Una <b>InflaterInputStream</b> que descomprime datos que son

	almacenados en el formato Zip.
<b>GZIPInputStream</b>	Una <b>InflaterInputStream</b> que descomprime datos que son almacenados en un fichero GZIP.

A pesar de que hay muchos algoritmos de compresión, Zip y GZIP son posiblemente los mas comúnmente utilizados. De esta forma se puede fácilmente manipular sus datos comprimidos con las muchas herramientas disponibles para lectura y escritura de estos formatos.

## Compresión simple con GZIP

La interfase GZIP es simple y de esta forma es probablemente la mas apropiada cuando se tiene un solo flujo de datos que se quiera comprimir (en lugar de un contenedor con piezas de datos distintas). Aquí hay un ejemplo que comprime un solo fichero:

```
//: c11:GZIPcompress.java
// Uso de compresión GZIP para comprimir un fichero
// Cuyo nombre se pasa en la linea de comandos.
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 las clases de compresión es directo -simplemente se envuelve el flujo de salida en un **GZIPOutputStream** o en un **ZipOutputStream** y su flujo de entrada en un **GZIPInputStream** o en un **ZipInputStream**. Todo lo demás es escritura común de E/S. Esto es un ejemplo de mezclar los flujos orientados a caracteres con los flujos orientados a bytes: **in** utilice las clases **Reader**, mientras que el constructor de **GZIPOutputStream**, puede aceptar solo un objeto **OutputStream**, no un objeto **Writer**. Cuando un fichero es abierto, el **GZIPInputStream** es convertido a un **Reader**.

## Almacenamiento de varios ficheros con Zip

La librería que soporta el formato Zip es mucho más extensa. Con esta se puede fácilmente almacenar varios ficheros, y hay incluso una clase separada para hacer el proceso de lectura de un fichero Zip fácil. La librería utiliza el fichero estándar Zip así es que funciona sin parches con todas las herramientas que se pueden bajar actualmente de la Internet. El siguiente ejemplo tiene la misma forma que el ejemplo anterior, pero este maneja tantos argumentos de la línea de comandos como se quiera. Además, muestra el uso de las clases de **Checksum** para calcular y verificar la suma de chequeo del fichero. Hay dos tipos de **Checksum**: **Adler32** (que es más rápida) y **CRC32** (que es más lenta pero mucho más precisa).

```
//: c11:ZipCompress.java
// Uso de compresión Zip para comprimir cualquier
// numero de ficheros dados en la linea de comandos.
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");
        // getComment() no correspondiente, sin embargo.
        for(int i = 0; i < args.length; i++) {
            System.out.println(
                "Writing file " + args[i]);
            BufferedReader in =
                new BufferedReader(
                    new FileReader(args[i]));
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
        }
        out.close();
    }
}
```

```

        out.putNextEntry(new ZipEntry(args[i]));
        int c;
        while((c = in.read()) != -1)
            out.write(c);
        in.close();
    }
out.close();
// Validación de suma de chequeo solo despues de que el
// fichero fué cerrado!
System.out.println("Checksum: " +
    csum.getChecksum().getValue());
// Ahora extrae los ficheros:
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();
// Una forma alternativa de abrir y leer
// fichros zip:
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);
    // ... y extraer los datos como antes
}
}
}
} //:~
```

Para cada fichero que se agrega a el archivo, se debe llamar a **putNextEntry()** y pasarla a el objeto **ZipEntry**. El objeto **ZipEntry** contiene una gran interfase que permite obtener y configurar todos los datos disponibles en esa entrada particular en su fichero Zip: nombre, tamaños de compresión y descompresión, fecha, suma de chequeo CRC, campos de datos extra, comentarios, método de compresión y si es una entrada de directorio. Sin embargo, a pesar de que el formato Zip tiene una forma de configurar una palabra clave, esto no es soportado en la librería Zip de Java. Y a pesar de que **CheckedInputStream** y **CheckedOutputStream** soportan las sumas de chequeo **Adler32** y **CRC32**, la clase **ZipEntry** soporta solo la

interfase para CRC. Esto es una restricción de capas bajas del formato Zip, pero lo limitará de utilizar el **Adler32** mas rápido.

Para extraer ficheros, **ZipInputStream** tiene un método **getNextEntry()** que retorna la siguiente entrada **ZipEntry** si hay una. Como una alternativa mas resumida se puede leer el fichero utilizando un objeto **ZipFile**, que tiene un método **entries()** para retornar una Enumeración a las entradas **ZipEntries**.

Para leer la suma de chequeo se debe tener una forma de acceder al objeto **Checksum** asociado. Aquí, una referencia a los objetos **CheckedOutputStream** y **CheckedInputStream** es guardada, pero se puede almacenar solamente una referencia a el objeto **Checksum**.

Un desconcertante método en los flujos Zip es **setComment()**. Como se ha mostrado antes, se puede configurar un comentario cuando se esta escribiendo un fichero, pero no hay forma de recuperar el comentario en el **ZipInputStream**. Los comentarios están soportados totalmente entrada por entrada solo mediante **ZipEntry**.

Claro, no se esta limitado a ficheros donde utilizando **GZIP** o **Zip** -se puede comprimir cualquier cosa, incluyendo datos a ser enviados a través de una conexión de red.

## Archivos de Java (JARs)

El formato Zip es también utilizado en el fichero de formato JAR (Java ARchive), que es la forma de recoger un grupo de ficheros en un solo fichero comprimido, exactamente igual que un Zip. Sin embargo, como todo lo demás en Java, los ficheros Jar son de plataforma cruzada así es que no se necesita preocuparse acerca de temas de plataforma. Se puede incluir audio e imágenes de la misma forma que ficheros class.

Los ficheros JAR son particularmente útiles cuando se trata con la Internet. Antes de los ficheros JAR, su navegador debía hacer muchas consultas a un servidor Web para bajar todos los ficheros para crear un applet. Además, cada uno de estos ficheros no estaba comprimido. Combinando todos los ficheros para un applet en particular en un solo fichero JAR, solo una consulta a el servidor es necesario y la transferencia es mas rápida por la compresión. Cada entrada en el fichero JAR puede ser firmada digitalmente para seguridad (véase la documentación por detalles).

Un fichero JAR consiste en un solo fichero que contiene un grupo de ficheros comprimidos con Zip con un “manifiesto” que los describe (Se puede crear su propio fichero manifiesto; de otra forma el programa **jar** lo hará por usted). Se puede encontrar mas acerca de los manifiestos JAR en la documentación HTML de JDK.

La utilidad **jar** que viene con JDK de Sun automáticamente comprime los ficheros de su elección. Se invoca en la línea de comandos:

```
| jar [opciones] destino [manifiesto] ficheros_de_entrada
```

Las opciones son simplemente una colección de letras (ningún guión u otro indicador es necesario). Los usuarios de UNIX/Linux notarán la similitud con las opciones **tar**. Estas son:

<b>e</b>	Crear una fichero nuevo o vacío
<b>t</b>	Lista de tabla de contenidos
<b>x</b>	Extraer todos los fichero
<b>x file</b>	Extraer el fichero indicado
<b>f</b>	Dice: "te daré el nombre del fichero." Si no se usa esto, jar asume que la entrada vendrá de la entrada estándar, o, si se esta creando un fichero, su salida será a la salida estándar.
<b>m</b>	Dice que el primer argumento será el nombre del fichero de manifiesto de usuario.
<b>v</b>	Genera una salida elocuente que describe que esta haciendo jar,
<b>o</b>	Solo almacena los ficheros; no comprime los ficheros (utilizado para crear un fichero JAR que se pueda colocar en su ruta de clases).
<b>M</b>	No crea automáticamente un fichero de manifiesto.

Si un subdirectorio es incluido en los ficheros a colocarse en un fichero JAR, ese directorio es automáticamente agregado, incluyendo todos sus subdirectorios, etc. La información de ruta es también preservada.

He aquí una de las típicas formas de invocara **jar**:

```
| jar cf miFichero.jar *.class
```

Esto crea un fichero JAR llamado **miFichero.jar** que contiene todos los ficheros clases en el directorio actual, junto con un manifiesto automáticamente generado.

```
| jar cmf miFicheroJar.jar miFicheroDeManifiesto.mf *.class
```

Al igual que el ejemplo anterior, pero agregando un fichero de manifiesto creado por el usuario llamado **miFicheroDeManifiesto.mf**.

```
| jar tf miFicheroJar.jar
```

Produce una tabla de contenido de los ficheros en **miFicheroJar.jar**.

```
| jar tvf miFicheroJar.jar
```

Agrega la bandera "elocuente" para dar información mas detallada acerca de los ficheros en **miFicheroJar.jar**.

```
| jav cvf miAplicacion.jar audio classes image
```

Asumiendo que **audio**, **classes**, e **image** son subdirectorios, esto combina todos los subdirectorios en un fichero **miAplicacion.jar**. La bandera de “elocuente” es también incluida para dar información extra cuando el programa **jar** esta trabajando.

Si se crea un fichero JAR utilizando la opción **o**, ese fichero puede ser colocado en su CLASSPATH:

```
| CLASSPATH="lib1.jar;lib2.jar;"
```

Entonces Java puede buscar en **lib1.jar** y **lib2.jar** por ficheros de clases.

La herramienta **jar** no es tan útil como lo es la utilidad **zip**. Por ejemplo, no se puede agregar o actualizar ficheros en un fichero JAR existente; se puede crear ficheros JAR solo de la nada. Tampoco se pueden mover ficheros dentro de un fichero JAR, borrándolos luego de que sean movidos. Sin embargo, un fichero JAR creado en una plataforma se podrá leer de forma transparente por la herramienta **jar** de otra plataforma (un problema que a veces infecta las utilidades **zip**).

Como se verá en el Capítulo 13, los ficheros JAR son también utilizados para empaquetar JavaBeans.

## Serialización de objetos

La *serialización de objetos* de Java nos permite tomar cualquier objeto que implemente la interfase **Serializable** y convertirlo en una secuencia de bytes que pueda mas tarde ser totalmente restaurada para regenerar el objeto original. Esto es verdad inclusive a través de la red, lo que significa que el mecanismo de serialización compensa automáticamente las diferencias de los sistemas operativos. Esto es, se puede crear un objeto en una máquina Windows, serializarlo, y enviarlo a través de la red a una máquina Unix donde será correctamente reconstruido. No nos tenemos que preocupar por la representación de los datos en las diferentes máquinas, por el orden de los bytes o cualquier otro detalle.

Por si mismo, la serialización de objetos es interesante porque permite implementar *persistencia ligera*. Recuerde que esta persistencia significa que el tiempo de vida de un objeto no esta determinada por si un programa es ejecutado o no -el objeto vive entre *invocaciones* del programa. Tomando un objeto serializable y escribiéndolo en el disco, luego restaurándolo cuando el programa es invocado nuevamente, se es capaz de producir el efecto de persistencia. La razón por la cual es llamado “persistencia ligera” es que no se puede simplemente definir un objeto utilizando algún tipo de palabra clave “persistente” y dejar que el sistema se haga cargo de los detalles (a pesar de que esto puede suceder en el futuro). En lugar de eso, se debe explícitamente serializar y deserializar los objetos en su programa.

La serialización de objetos fue agregada a el lenguaje para soportar dos características mas grandes. El *método de invocación remota* de Java o (RMI remote method invocation) que permite que los objetos que viven en otras máquinas se comporten como si vivieran en nuestra máquina. Cuando se envían mensajes a objetos remotos, la serialización de objetos es necesaria para transportar los argumentos y retornar los valores. RMI se discute en el Capítulo 15.

La serialización de objetos es necesaria también par JavaBean, descritos en el Capítulo 13. Cuando un Bean es utilizado, su información de estado es generalmente configurada en tiempo de diseño. Esta información de estado debe ser almacenada y luego recuperada cuando el programa es iniciado; la serialización de objetos realiza esta tarea.

Serializar un objeto es bastante simple, mientras el objeto implemente la interfase **Serializable** (esta interfase es solo una bandera y no tiene métodos). Cuando la serialización fue agregada al lenguaje, muchas clases de la librerías estándar fueron cambiadas para hacerlas serializables, incluyendo todas las envolturas de los tipos primitivos, todos los contenedores de clases, y muchos otros. Incluso lo objetos **Class** pueden ser serializados (Véase el capítulo 12 por las implicaciones de esto).

Para serializar un objeto, se puede crear una suerte de objeto **OutputStream** y luego envolverlo dentro de un objeto **ObjectOutputStream**. En este punto se necesita solo llamar a **writeObject()** y su objeto es serializado y enviado a la **OutputStream**. Para invertir el proceso, se envuelve un **InputStream** dentro de un **ObjectInputStream** y se llama a **readObject()**. Lo que recupera lo que es, como es usual, una referencia a un **Object** con una conversión ascendente, por lo que se deberá realizar una conversión descendente para ver las cosas correctamente.

Un aspecto particularmente brillante de la serialización de objetos es que no solo guarda una imagen del objeto sino que también sigue las referencias contenidas en su objeto y guarda *aquellos* objetos, y sigue las referencias en cada uno de aquellos objetos, etc. Esto es a veces referido como los “objetos atrapados en la telaraña” a los que un solo objeto pide estar conectado, y esto incluye arreglos de referencias a objetos de la misma forma que objetos miembro. Si se tiene que mantener su propio esquema de serialización, mantener el código que sigue todos estos enlaces puede aturdir un poco la mente. Sin embargo, la serialización de objetos de Java parece lograrlo perfectamente, sin duda el uso de un algoritmo optimizado recorre la telaraña de objetos. El siguiente ejemplo prueba el mecanismo de serialización haciendo un “gusano” de objetos enlazados, cada uno de los cuales tiene un enlace a el siguiente segmento en el gusano de la misma forma que un arreglo de referencias a objetos de una clase diferente, **Data**:

```
//: c11:Worm.java
// Demuestra la serializacion de objetos.
```

```

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 {
    // Genera un valor entero aleatorio:
    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;
    // Valor de i == número de segmentos
    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;
    }
    // Lanza excepciones a la consola:
    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(); // Tambien limpia la salida
        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 arreglo de objetos **Data** dentro de **Worm** es inicializado con números aleatorios (De esta forma no se puede sospechar que el compilador mantiene algún tipo de meta-information). Cada segmento de **Worm** es etiquetado con un **char** que es automáticamente generado en el proceso de recursividad generando la lista de **Worms** enlazados. Cuando se crea un **Worm**, se le indica al constructor que tan largo se quiere que sea. Para hacer que la referencia **next** llame al constructor de **Worm** con un largo de uno menos, etc. La referencia **next** final es dejada como **null**, indicando el final de **Worm**.

El punto de todo esto es crear algo razonablemente complejo que no pueda ser fácilmente serializado. El acto de serializar, sin embargo, es bastante simple. Una vez que el **ObjectOutputStream** es creado de algún otro flujo, **writeObject()** serializa el objeto. Debe notarse la llamada a **writeObject()** para un **String**, de la misma forma. Se puede escribir también todos los tipos de datos primitivos utilizando los mismos métodos que **DataOutputStream** (ellos comparten la misma interfase).

Hay dos secciones separadas de código que se ven similares. La primera escribe y lee un fichero y la segunda, para variar, escribe y lee un **ByteArray**. Se puede leer y escribir un objeto utilizando serialización de cualquier **DataInputStream** o **DataOutputStream** incluyendo, como se verá en el capítulo 15, una red de trabajo. La salida para una corrida 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 realmente contiene todos los enlaces que estaban en el objeto original.

Debe notarse que ningún constructor, incluso el constructor por defecto, es llamado en el proceso de deserialización de un objeto **Serializable**. El objeto entero es restaurado recuperando datos de la **InputStream**.

La serialización de un objeto es orientada a byte, y de esta forma utiliza las jerarquías de **InputStream** y **OutputStream**.

## Encontrando la clase

Nos preguntaremos que es necesario para que un objeto sea recuperado desde su estado serializado. Por ejemplo, supongamos que se serializa un objeto y se envía como un fichero a través de una red a otra máquina. ¿Puede un programa en la otra máquina reconstruir el objeto utilizando solo el contenido del fichero?

La mejor forma de contestar esta pregunta es (como es usual) realizando un experimento. El siguiente fichero viene en el subdirectorio para este capítulo:

```
//: c11:Alien.java
// A serializable class.
import java.io.*;
public class Alien implements Serializable {
} ///:~
```

El fichero que crea y serializa un objeto **Alien** viene en el mismo directorio:

```
//: c11:FreezeAlien.java
// Crea un fichero con una salida serializada.
import java.io.*;
public class FreezeAlien {
    // Lanza las excepciones a la consola:
    public static void main(String[] args)
        throws IOException {
        ObjectOutputStream out =
            new ObjectOutputStream(
                new FileOutputStream("X.file"));
        Alien zorcon = new Alien();
        out.writeObject(zorcon);
    }
} ///:~
```

Antes que capturar y manejar excepciones, este programa toma la rápida y sucia estrategia de pasar las excepciones fuera del **main()**, así es que serán reportadas en la línea de comandos.

Una vez que el programa es compilado y ejecutado, copia el **X.file** resultante en un subdirectorios llamada **xfiles**, donde el siguiente código acude:

```
//: 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 fichero y lee en el objeto **mystery** exitosamente. Sin embargo, tan pronto como se intente encontrar algo acerca del objeto -que requiere del objeto **Class** para **Alien**- La máquina virtual de Java (JVM) no podrá encontrar **Alien.class** (a no ser que suceda que este en la ruta de clases, lo que no debería ser en este ejemplo). Se obtendrá una **ClassNotFoundException** (¡Una vez mas, toda la evidencia de la vida del extraterrestre se desvanece antes de una prueba de su existencia pueda ser verificada!).

Si se espera hacer mucho luego de recuperar un objeto que ha sido serializado, debe asegurarse de que la JVM pueda encontrar el fichero.**.class** asociado o en la ruta local o en alguna parte de la Internet.

## Controlando la serialización

Como se ha visto, el mecanismo de serialización por defecto es trivial de utilizar. ¿Pero que si se tienen necesidades especiales? Tal vez se tienen temas especiales de seguridad y no se quiere serializar partes de su objeto, o simplemente no tiene sentido para un subobjeto el ser serializado si esa parte necesita ser creada nuevamente cuando el objeto es recuperado.

Se puede controlar el proceso de serialización implementando la interfase **Externalizable** en lugar de la interfase **Serializable**. La interfase **Externalizable** extiende la interfase **Serializable** y agrega dos métodos, **writeExternal()** y **readExternal()**, que son automáticamente llamados para su objeto durante la serialización y deserialización así es que se pueden realizar sus operaciones especiales.

El siguiente ejemplo muestra una implementación simple de los métodos de la interfase **Externalizable**. Debe notarse que **Blip1** y **Blip2** son casi idénticas excepto por una util diferencia (vea si puede descubrirla mirando el código):

```

//: c11:Blips.java
// Simple uso de Externalizable & una trampa.
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 {
    // Lanza las excepciones a la consola:
    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();
        // Ahora lo recuperamos:
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("Blips.out"));
        System.out.println("Recovering b1:");
        b1 = (Blip1)in.readObject();
        // OOPS! Lanza una excepción:
        // System.out.println("Recovering b2:");
        // b2 = (Blip2)in.readObject();
    }
} //:~

```

La salida para este programa es:

```

Constructing objects:
Blip1 Constructor

```

```

Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal

```

La razón para que el objeto **Blip2** no sea recuperado es lo que se está tratando de hacer causa una excepción. ¿Se puede ver la diferencia entre **Blip1** y **Blip2**? El constructor para **Blip1** es público, mientras que el constructor para **Blip2** no lo es, y esto causa una excepción durante la recuperación. Trate de hacer el constructor de **Blip2** público y quite el comentario `//!` para ver los resultados correctos.

Cuando **b1** es recuperado, el constructor por defecto de **Blip1** es llamado. Esto es diferente de recuperar un objeto **Serializable**, en donde el objeto es construido enteramente de sus bits almacenados, no con una llamada a un constructor. Con un objeto **Externalizable**, todo el comportamiento normal de construcción se sucede (incluyendo la inicialización en el punto de la definición de campos), y *entonces* **readExternal()** es llamado. Se necesita tener cuidado con esto -en particular, del hecho de que todas las construcciones por defecto siempre tienen lugar- para producir el correcto comportamiento de sus objetos **Externalizable**.

He aquí un ejemplo que muestra como se debe almacenar y recuperar un objeto **Externalizable**:

```

//: c11:Blip3.java
// Reconstruyendo un objeto externalizable.
import java.io.*;
import java.util.*;
class Blip3 implements Externalizable {
    int i;
    String s; // Sin inicialización
    public Blip3() {
        System.out.println("Blip3 Constructor");
        // s, i no son inicializados
    }
    public Blip3(String x, int a) {
        System.out.println("Blip3(String x, int a)");
        s = x;
        i = a;
        // s & i inicializados solo en constructores
        // que no son por defecto.
    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Blip3.writeExternal");
        // Se deb hacer esto:
        out.writeObject(s);
        out.writeInt(i);
    }
}
```

```

}
public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // Se debe hacer esto:
    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();
    // Ahora lo recuperamos:
    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** son inicializados solo en el segundo constructor, pero no en el constructor por defecto. Esto significa que si no se inicializa **s** e **i** en **readExternal()**, habrá un **null** (dado que el almacenaje de los objetos son llevados a cero en el primer paso de la creación de un objeto). Si se quita el comentario de las dos líneas de código que están seguidas “Se debe hacer esto” y se ejecuta el programa, se verá que cuando el objeto es recuperado, **s** es **null** e **i** es cero.

Si se ha heredado de un objeto **Externalizable**, normalmente se llama a las versiones de la clase base de **writeExternal()** y **readExternal()** para proporcionar el almacenaje y la recuperación apropiada de los componentes de la clase base.

Así es que hacer que las cosas trabaje correctamente no solamente se puede escribir los datos importante de un objeto durante el método **writeExternal()** (no hay comportamiento por defecto que escriba algo en los objetos miembro para un objeto **Externalizable**) sino que también se puede recuperar los datos en el método **readExternal()**. Esto puede ser un poco confuso al principio porque el comportamiento del constructor por defecto para un objeto **Externalizable** puede hacer que parezca como un tipo de almacenaje y recuperación que tome lugar automáticamente. Cosa que no es.

## La palabra clave **transient**

Cuando se esta controlando la serialización, puede haber un subobjeto particular que no se quiere que el mecanismo de serialización automáticamente guarde o recupere. Esto es común en el caso de que ese subobjeto represente información sensible que no se quiere serializar, como una palabra clave. Aún si esta información es privada en el objeto, una vez que es serializada es posible para alguien acceder leyendo el fichero o interceptando la transmisión de la red.

Una forma de prevenir que las partes sensibles de su objetos sea serializada es implementar su clases como **Externalizable** como se ha mostrado previamente. Luego nada es automáticamente serializado y se puede explícitamente serializar solo las partes necesarias dentro de **writeExternal()**.

Si se esta trabajando con un objeto **Serializable**, sin embargo, toda la serialización sucede automáticamente. Para controlar esto, se puede apagar la serialización campo por campo utilizando la palabra clave **transient**, que dice "No se moleste en guardar o serializar esto -yo me ocuparé"

Por ejemplo, considere un objeto **Login** que contiene información acerca de una sesión de acceso particular. Supongamos que, una vez que se verifica el acceso, se debe querer almacenar el dato, pero sin la palabra clave de acceso. La forma mas simple de hacer esto es implementando **Serializable** y hacer que el campo **password** como **transient**. Así es como se vería:

```
//: c11:Logon.java
// Demuestra la palabra clave "transient".
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();
// Retardo:
int seconds = 5;
long t = System.currentTimeMillis()
    + seconds * 1000;
while(System.currentTimeMillis() < t)
;
// Ahora lo recuperamos:
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 **date** y **username** son comunes (no **transient**), y de esta forma son automáticamente serializados. Sin embargo, **password** es **transient** y de esta forma no es almacenado en el disco; tampoco el mecanismo de serialización no hace intento por 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 un objeto es recuperado, el campo **password** es **null**. Debe notarse que **toString()** debe verificar por un valor **null** de **password** porque si se trata de volver a armar un objeto **String** utilizando el operador sobrecargado '+', y el operador encuentra una referencia **null**, se obtendrá un **NullPointerException** (las versiones mas nuevas de Java pueden contener código para evitar este problema).

Se puede ver también que el campo **date** es almacenado y recuperado del disco y no generado nuevamente.

Dado que los objetos **Externalizable** no almacenan ningunos de sus campos por defecto, la palabra clave **transient** es utilizada solo con objetos **Serializable**.

## Una alternativa para **Externalizable**

Si no se es entusiasta al implementar la interfase **Externalizable**, hay otra estrategia. Se puede implementar la interfase **Serializable** y *agregar* (debe notarse que digo “agregar” y no “sobrecargar” o “implementar”) métodos llamados **writeObject()** y **readObject()** que automáticamente serán llamados cuando el objeto sea serializado y deserializado, respectivamente. Esto es, si se proporcionan estos dos métodos serán utilizados en lugar de la serialización por defecto.

Estos métodos deben tener exactamente estas firmas:

```
private void  
    writeObject(ObjectOutputStream stream)  
        throws IOException;  
private void  
    readObject(ObjectInputStream stream)  
        throws IOException, ClassNotFoundException
```

Desde un punto de vista del diseño, las cosas se hacen extrañas aquí. Antes que nada, se puede pensar que dado que estos métodos no son parte de la clase base de la interfase **Serializable**, deben ser definidos en sus propias interfaces. Pero la noticia es que son definidos como **private**, lo que significa que son llamados solo por otros miembros de esta clase. Sin embargo, no se llaman actualmente de otros miembros de esta clase, en lugar de eso los métodos **writeObject()** y **readObject()** de los objetos **ObjectOutputStream** y **ObjectInputStream** llaman a nuestros métodos **writeObject()** y **readObject()** (Debe notarse mi tremenda abnegación a no lanzarme en una larga diatriba acerca de utilizar los mismos nombres de métodos aquí. En una palabra: confuso). Podríamos preguntarnos como los objetos **ObjectOutputStream** y **ObjectInputStream** tienen acceso a los métodos **private** de su clase. Solo podemos asumir que esto es parte de la magia de la serialización.

En cualquier evento, cualquier cosa definida en una **interface** es automáticamente **public** así es que si **writeObject()** y **readObject()** debiesen ser privados, entonces estos no pueden ser parte de una interfase. Dado que se debe seguir las firmas exactamente, el efecto es el mismo que si se estuviera implementando una **interface**.

Parecerá que cuando se llama a **ObjectOutputStream.writeObject()**, el objeto **Serializable** que se pasa es interrogado (utilizando reflexión, no hay duda) para ver si implementa su propio **writeObject()**. Si es así, el proceso de serialización normal es saltado y **writeObject()** es llamado. La misma suerte de situación existe para **readObject()**.

Hay otro giro. Dentro de su **writeObject()**, se puede elegir realizar la acción por defecto la acción de **writeObject()** llamando a **defaultWriteObject()**. De la misma forma, dentro de **readObject()** se puede llamar a **defaultReadObject()**. He aquí un simple ejemplo que demuestra como se

puede controlar el almacenamiento y la recuperación de un objeto **Serializable**:

```
//: c11:SerialCtl.java
// Controlando la serialization agregando su propios
// métodos writeObject() y readObject().
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);
        // Ahora a recuperarlo:
        ObjectInputStream in =
            new ObjectInputStream(
                new ByteArrayInputStream(
                    buf.toByteArray()));
        SerialCtl sc2 = (SerialCtl)in.readObject();
        System.out.println("After:\n" + sc2);
    }
} ///:~
```

En este ejemplo, un campo **String** es común y otro es **transient**, para probar que un campo que no es **transient** es guardado y recuperado explícitamente. Los campos son inicializados dentro del constructor en lugar de hacerlo en el punto de definición para probar que no son inicializados por algún mecanismo automático durante la deserialización.

Si se va a utilizar el mecanismo por defecto para escribir las partes que no son **transient** de su objeto, se debe llamar a **defaultWriteObject()** como la primer operación en **writeObject()** y **defaultReadObject()** como la primera operación en **readObject()**. Estas son extrañas llamadas a métodos.

Parecerán, por ejemplo, que se esta llamando **defaultWriteObject()** para un **ObjectOutputStream** sin pasarle argumentos, y a pesar de eso de alguna forma da vueltas y conoce la referencia a el objeto y como escribir todas las partes que no son **transient**. Fantasmagórico.

el guardar y la recuperación de los objetos **transient** utilizan código mas familiar. Y aún, pensando acerca de que sucede aquí. En el **main()**, un objeto **SerialCtl** es creado, y luego es serializado a un **ObjectOutputStream** (Debe notarse en este caso que un buffer es utilizado en lugar de un fichero -esto es todo lo mismo para **ObjectOutputStream**). La serialización sucede en la línea:

```
| o.writeObject(sc);
```

El método **writeObject()** debe examinar **sc** para ver si tiene su propio método **writeObject()** (No mirando en la interfase -no hay ninguna- o en el tipo de clase, si cazando el método utilizando reflexión). Si lo hace, lo utiliza. Una estrategia similar se cumple para **readObject()**. A pesar de que esta fue la única forma práctica en que se puede resolver el problema, es ciertamente extraña.

## Asignando una versión

Es posible que se quiera cambiar la versión de una clase serializable (los objetos de la clase original pueden ser almacenados en una base de datos, por ejemplo). Se da soporte a esto pero probablemente se haga en casos especiales, y esto requiere una profundidad extra de entendimiento que no intentaremos alcanzar aquí. El documento HTML de la JDK que se puede bajar de [java.sun.com](http://java.sun.com) cubre este tema bastante a fondo..

Se notará también en la documentación HTML de JDK muchos comentarios que comienzan con:

***Cuidado:** Los objetos serializables de esta clase pueden no ser compatibles con futuras versiones Swing. El soporte actual para serialización es apropiado para almacenamientos cortos o RMI entre aplicaciones.*

Esto es porque el mecanismo de versiones es muy simple para trabajar confiablemente en todas las situaciones, especialmente con JavaBeans. No trabajarán correctamente en una corrección del diseño, y por esto es la advertencia.

# Utilizando persistencia

Es bastante atractivo utilizar la tecnología de serialización para almacenar algunos de los estados de su programa así es que se puede fácilmente arrancar el programa el estado actual mas tarde. Pero antes de que se puede hacer esto, algunas preguntas deben ser contestadas. ¿Que sucede si se serializan dos objetos que tienen ambos una referencia a un tercer objeto? Cuando se recuperan estos objetos de su estado serializado, se obtienen solo una ocurrencia del tercer objeto? ¿Que si se serializan los dos objetos en fichero separados y se deserializan en diferentes partes 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 ol =
            new ObjectOutputStream(buf1);
        ol.writeObject(animals);
        ol.writeObject(animals); // Write a 2nd set
        // Escribir a un flujo diferente:
        ByteArrayOutputStream buf2 =
            new ByteArrayOutputStream();
        ObjectOutputStream o2 =
            new ObjectOutputStream(buf2);
        o2.writeObject(animals);
```

```

    // Ahora recuperarlo:
    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 cosa que es interesante aquí es que es posible utilizar una serialización de un objeto hacia y desde un arreglo de bytes como una forma de hacer una “copia profunda” de cualquier objeto que sea **Serializable** (Una copia profunda significa que se duplica la telaraña entera de objetos, en lugar de solo el objeto básico y sus referencias). La copia es cubierta en profundidad en el Apéndice A.

Los objetos **Animal** contienen campos del tipo **House**. En el **main()**, un **ArrayList** de estos **Animals** es creado y son serializados dos veces a un flujo y luego nuevamente en un flujo separado. Cuando estos son deserializados e impresos, se ven los siguientes resultados para una corrida (los objetos estarán en diferentes posiciones de memoria en cada corrida):

```

animals: [Bosco the dog[Animal@lcc76c], House@lcc769
, Ralph the hamster[Animal@lcc76d], House@lcc769
, Fronk the cat[Animal@lcc76e], House@lcc769
]
animals1: [Bosco the dog[Animal@lcca0c], House@lcca16
, Ralph the hamster[Animal@lcca17], House@lcca16
, Fronk the cat[Animal@lcca1b], House@lcca16
]
animals2: [Bosco the dog[Animal@lcca0c], House@lcca16
, Ralph the hamster[Animal@lcca17], House@lcca16
, Fronk the cat[Animal@lcca1b], House@lcca16
]
animals3: [Bosco the dog[Animal@lcca52], House@lcca5c
, Ralph the hamster[Animal@lcca5d], House@lcca5c
, Fronk the cat[Animal@lcca61], House@lcca5c
]

```

Por supuesto que se esperaba que los objetos deserializados tuvieran diferentes direcciones que sus originales. Pero se debe notar que en **animals1** y en **animals2** la misma dirección aparece, incluyendo las

referencias a el objeto **House** que ambos comparten. Por el otro lado, cuando **animals3** es recuperado en el sistema no hay forma de conocer que los objetos en este otro flujo son alias de los objetos en el primer flujo, así es que se crea una completamente diferente telaraña de objetos.

Mientras se esta serializando todo en un solo flujo, se será capas de recuperar la misma telaraña de objetos que se escribió, sin duplicaciones de objetos. Claro, se puede cambiar el estado de sus objetos entre el momento que se escribió el primero y el último, pero es su responsabilidad -los objetos serán escritos en cualquier estado en que estén (y con cualquier conexión que tengan con otros objetos) en el tiempo que se serialicen.

Lo mas seguro para hacer si se quiere guardar el estado de un sistema es serializarlo como una operación “atómica”. Si se serializan algunas cosas, se puede hacer otro trabajo, y serializar algo mas, etc., entonces no se estará almacenando el sistema de forma segura. En lugar de esto, coloque todos los objetos que comprenden el estado de su sistema en un solo contenedor y simplemente escriba el contendor afuera en una sola operación. Luego se puede restaurar con una sola llamada a un método.

El siguiente ejemplo es un sistema de diseño asistido por computador (CAD) que demuestra esta estrategia. Además, incursiona en el tema de los campos **statics** -si se ve la documentación se podrá ver que **Class** es **Serializable**, así es que será fácil almacenar campos estáticos simplemente serializando el objeto **Class**. Esto parece ser una estrategia sensata, de cualquier forma.

```
//: 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();
            // Agregar referencias a los objetos de la clase:
            shapeTypes.add(Circle.class);
            shapeTypes.add(Square.class);
            shapeTypes.add(Line.class);
            // Hacer algunas formas:
            for(int i = 0; i < 10; i++)
                shapes.add(Shape.randomFactory());
            // Configura todos los colores a GREEN:
            for(int i = 0; i < 10; i++)
                ((Shape)shapes.get(i))
                    .setColor(Shape.GREEN);
            // Guarda el estado del vector:
            ObjectOutputStream out =
                new ObjectOutputStream(
                    new FileOutputStream("CADState.out"));
            out.writeObject(shapeTypes);
            Line.serializeStaticState(out);
            out.writeObject(shapes);
        } else { // Hay un argumento de línea de comando
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream(args[0]));
            // Lee en el mismo orden en que fueron escritos:
            shapeTypes = (ArrayList)in.readObject();
            Line.deserializeStaticState(in);
            shapes = (ArrayList)in.readObject();
        }
        // Despliega las formas:
        System.out.println(shapes);
    }
}
} //:~

```

La clase **Shape** implementa **Serializable**, así es que cualquier cosa que sea heredada de **Shape** es automáticamente **Serializable** igualmente. Cada **Shape** contiene datos, y cada una de las que son derivadas de la clase **Shape** contiene un campo **static** que determina el color de todos estos tipos de **Shapes** (Colocando un campo **static** en la clase base resultara en solo un campo, dado que los campos **static** no son duplicados en la clase derivada). Los métodos en la clase base pueden ser sobrecargados para ajustar el color para varios tipos (los métodos **static** no son enlazados dinámicamente, así es que estos son métodos normales). El método **randomFactory()** crea un **Shape** diferente cada vez que se llama, utilizando valores aleatorios para los datos **Shape**.

**Circle** y **Square** son extensiones simples de **Shape**; la única diferencia es que **Circle** inicializa **color** en el punto de la definición y **Square** lo inicializa en el constructor. Dejaremos la discusión de **Line** para mas adelante.

En el **main()**, un **ArrayList** es utilizado para almacenar los objetos **Class** y otro para almacenar las curvas. Si no se quiere proporcionar un argumento en la línea de comandos el **ArrayList shapeTypes** es creado y los objetos **Class** son agregados, y luego el **ArrayList shapes** es creado y los objetos **Shape** son agregados. Después, todos los colores **static color** son configurados a **GREEN**, y todo es serializado en el fichero **CADState.out**.

Si se proporciona un argumento de línea de comandos (presumiblemente **CADState.out**), ese fichero es abierto y utilizado para restaurar el estado del progr5ama. En ambas situaciones, el **ArrayList** de **Shapes** es impreso. El resultado para una corrida es:

```
>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** son todos almacenados y recuperados correctamente, pero hay algo mal con esta recuperación de información **static**. Es toda “3” cuando va, pero no regresa de esta forma. Los **Circles** tienen un valor de 1 (**RED**, que es la definición), y los **Squares** tienen un valor de 0 (recuerde, son inicializados en el constructor). ¡Esto es como si los **statics** no fueran serializados después de todo! Esto es correcto - a pesar de que la clase **Class** es **Serializable**, no hace lo que se esperaba. Así es que si se quiere serializar **statics**, lo debe hacer uno mismo.

Esto es lo que los métodos estáticos **serializeStaticState()** y **deserializeStaticState()** de **Line** hacen por nosotros. Se puede ver que hay

llamadas explícitas como parte del proceso de almacenamiento y del proceso de recuperación (Note que el orden de escritura del fichero serializado y de lectura se debe mantener). Así es que para hacer que **CADState.java** corra correctamente se debe:

1. Agregar un **serializeStaticState()** y un **deserializeStaticState()** para las curvas.
2. Quite el **ArrayList shapeTypes** y todo el código relacionado con el.
3. Agregue llamadas a los nuevos métodos para serializar y deserializar en las formas.

Otro tema en el que se debe pensar es en la seguridad, dado que la serialización solo guarda datos **privados**. Si se tiene un tema de seguridad, aquellos campos deben ser marcados como **transient**. Pero entonces se tiene que diseñar una forma segura de almacenar esta información así es que cuando se restaure se puede reiniciar estas variables **privadas**.

## Entrada convertida a símbolos

*Convertir a símbolo* o *tokenizing* es el proceso de dividir una secuencia de caracteres en una secuencia de “símbolos”, que son bits de texto delimitados por cualquier cosa que se elija. Por ejemplo, sus símbolos pueden ser palabras, y entonces estas pueden ser delimitadas por espacios en blancos y puntuaciones. Hay dos clases proporcionadas en la librería estándar de Java que pueden ser utilizadas para tokenization:

## Convertidores de símbolos de flujo

A pesar de que **StremTokenizer** no es una derivada de **InputStream** u **OutputStream**, trabaja solo con objetos **InputStream**, así es que legítimamente se convierte en una parte de la librería de E/S.

Considere el programa para contar el número de ocurrencias de palabras en un fichero de texto:

```
//: c11:WordCount.java
// Cuenta palabras en un fichero, entregando los
// resultados en orden.
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;
    // Un TreeMap mantiene las claves ordenadas:
    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();
    }
} //:~

```

Presentar las palabras de forma ordenada es fácil de hacer almacenando los datos en un **TreeMap**, que automáticamente organiza las claves en orden (véase el capítulo 9). Cuando se obtiene un grupo de claves utilizando **keySet()**, estas estarán también en orden.

Para abrir el fichero, un **FileReader** es utilizado, y para convertir el fichero en palabras un **StreamTokenizer** es creado desde un **FileReader** envuelto en un **BufferedReader**. En **StreamTokenizer**, hay una lista por defecto de separadores, y se puede agregar mas con un grupo de métodos. Aquí, **ordinaryChar()** es utilizado para decir “Este carácter no tiene un significado que me interese”, así es que el analizador no lo incluye como parte de alguna de las palabras que crea. Por ejemplo, diciendo **st.ordinaryChar('.)** significa que el punto no será incluida como parte de las palabras que sean analizadas. Se pueden encontrar mas información en la documentación de JDK HTML en *java.sun.com*

En **countWords()**, los símbolos son empujados una por ves del flujo, y la información de **ttype** es utilizada para determinar que hacer con cada símbolo, dado que un símbolo puede ser un final de línea, un número, una cadena o un solo carácter.

Una vez que un símbolo es encontrado, el **count** de **TreeMap** es consultado para ver si contiene el símbolo como clave. Si lo tiene, el objeto **Counter** correspondiente es incrementado para indicar que otra instancia de esta palabra fue encontrada. Si no, una nuevo **Counter** es creado -dado que el constructor de **Counter** inicializa el valor a uno, esto actúa también como contador de la palabra.

**WordCount** no es un tipo de **TreeMap**, así es que no fue heredado. Este realiza un tipo específico de funcionalidad, así es que aunque los métodos **keys()** y **values()** deben ser expuestos nuevamente, esto no significa que la herencia deba ser utilizada puesto que una cantidad de métodos **TreeMap** son inapropiados aquí. Además, otros métodos como **getCounter()**, que tienen el **Counter** para un **String** particular, y **sortedKeys()**, que producen un **Iterator**, terminan el cambio en la forma de la interfase **WordCount**.

En el **main()** se puede ver el uso de un **WordCount** para abrir y contar las palabras en un fichero -esto solo toma dos líneas de código. Entonces un iterator para una lista ordenadas de claves (palabras) es extraído, y esto es utilizando para sacar cada clave y **Count** asociado. La llamada a **cleanup()** es necesaria para asegurar que el fichero es cerrado.

## Cadenas convertidas a símbolos

A pesar de que no es parte de la librería de E/S, el  **StringTokenizer** tiene funcionalidad suficiente para el  **StreamTokenizer** que será descrito aquí.

El  **StringTokenizer** retorna los símbolos dentro de una cadena uno a la vez. Estos símbolos son caracteres consecutivos delimitados por tabulaciones, espacios y saltos de líneas. De esta forma, los símbolos de la cadena "¿Donde esta mi gato?" son "¿Donde", "esta", "mi" y "gato?". Como el  **StreamTokenizer**, se puede indicar a el  **StringTokenizer** que divida la entrada en la forma que se quiera, pero con  **StringTokenizer** se puede hacer esto pasando un segundo argumento a el constructor, que es un  **String** de delimitadores que se desean utilizar. En general, si necesita mas sofisticación, utilice un  **StreamTokenizer**.

Se pregunta a un objeto  **StringTokenizer** por el siguiente símbolo en la cadena utilizando el método **nextToken()**, el que retorna el símbolo y una cadena vacía indicando que no hay mas símbolos.

Como ejemplo, el siguiente programa realiza un análisis limitado de una sentencia, buscando secuencias de claves para indicar cuando la felicidad o la tristeza está implicada.

```
//: c11>AnalyzeSentence.java
// Busca secuencias particulares en una sentencia.
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();
            // Busca hasta que encuentra uno de los dos
            // símbolos de inicio:
            if(!token.equals("I") &&
               !token.equals("Are"))
                continue; // Inicio del bucle while
            if(token.equals("I")) {
                String tk2 = next();
                if(!tk2.equals("am")) // Debe estar antes del 'I'
                    break; // Sale del bucle while
                else {
                    String tk3 = next();
                    if(tk3.equals("sad")) {
                        sad = true;
                        break; // Sale del bucle while
                    }
                    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; // Debe estar despues del 'Are'
                String tk3 = next();
                if(tk3.equals("sad"))
                    sad = true;
                break; // Sale del bucle while
            }
            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);
    }
} //:~

```

Para cada cadena que es analizada, se entra a un bucle **while** y los símbolos son extraídos de la cadena. Debe observarse la primer instrucción **if**, que continúa (regresa al inicio del bucle y comienza nuevamente) si el símbolo no es ni “I”, ni “Are”. Esto significa que se tomarán símbolos hasta que se encuentra “I” o “Are”. Se puede pensar el utilizar el == en lugar de el método **equals()**, pero eso no trabajará correctamente, dado que == compara valores de referencia mientras que **equals()** compara contenidos.

La lógica del resto del método **analyze()** es que el patrón buscado es “I am sad”, “I am not happy”, o “Are you sad?”. Sin la instrucción **break**, el código de esto sería aún mas desordenado de lo que es. Se debe ser consciente de que un analizador típico (esto es un ejemplo primitivo de uno) normalmente tiene una tabla de estos símbolos y código que se mueve a través de los estados en la tabla a medida que nuevos símbolos son leídos.

Se puede pensar del  **StringTokenizer** solo como un atajo para un simple y específico tipo de  **StreamTokenizer**. Sin embargo, si se tiene un **String** que se quiere convertir a símbolos y  **StringTokenizer** es muy limitado, todo lo que tiene que hacer es colocarlo en un flujo con  **StringBufferInputStream** y luego utilizarlo para crear un  **StreamTokenizer** mucho mas poderoso.

## Verificando el estilo de las mayúsculas y minúsculas

En esta sección observaremos un ejemplo mas completo del uso de la E/S de Java, que también utiliza conversión a símbolos. Este proyecto es útil de forma directa porque realiza una verificación de estilo para asegurarnos que las mayúsculas y minúsculas están de acuerdo con el estilo de Java encontrado en [java.sun.com/docs/codeconv/index.html](http://java.sun.com/docs/codeconv/index.html) El programa abre cada **.java** en el directorio actual y extrae todos los nombres de clases e identificadores, luego los muestra si alguno no es apropiado con el estilo de Java.

Para que el programa funcione correctamente, se debe primero crear un depósito para nombres de clase para almacenar todos los nombres de clase en la librería estándar de Java. Se hace esto moviéndose dentro de los subdirectorios del código fuente para la librería estándar de Java y se ejecuta **ClassScanner** en cada subdirectorio. Se proporciona como argumento el nombre del fichero depósito (utilizando la misma ruta y

nombre cada vez) y la opción -a en la línea de comandos para indicar que los nombres de clase deben agregarse a el depósito.

Para utilizar este programa para verificar su código, hay que pasarle el nombre y la ruta del depósito a utilizar. El programa verificará todas las clases e identificadores en el directorio actual y de indicará cual de ellos no siguen el estilo de capitalización típico de Java.

Se debe ser conciente de que el programa no es perfecto; hay algunos momentos en donde indica lo que piensa que es un problema pero observando el código se puede ver que no se necesita cambiar nada. Esto es un poco molesto, pero sigue siendo mucho mas fácil tratar de encontrar todos esos casos mirando fijamente el código.

```
//: c11:ClassScanner.java
// Analiza todos los ficheros en un directorio buscando
// clase e identificadores, para verificar las
// mayúsculas y minúsculas.
// Se asume una compilación correcta del código.
// No hace todo correctamente, pero es una
// ayuda útil.
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)));
    // No parece funcionar:
    // 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")) {
                // Obtiene el nombre de la clase:
                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 // Es un identificador o palabra clave
                identMap.add(fname, in.sval);
        }
    }
}
void discardLine() throws IOException {
    while(in.nextToken() != StreamTokenizer.TT_EOF
        && in.ttype !=

```

```

        StreamTokenizer.TT_EOL)
        ; // Elimina símbolos al final de línea
    }
// Removedor de comentarios de StreamTokenizer que parecen
// estar quebrados. Esto los extrae:
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)) {
                // Ignora identificadores de largo 3 o
                // mas largos que esten en mayúsculas
                // (probablemente valores estáticos finales):
                if(id.length() >= 3 &&
                   id.equals(
                           id.toUpperCase())))
                    continue;
                // Mira para ver si el primer caracter
                // es mayúsculas:
                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 {
                // Intenta abrir un fichero existente
                // de propiedades:
                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();
}
// Escribe los nombres de clases en el depósito:
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 **MultiStringMap** es una herramienta que permite asociar un grupo de cadenas con cada entrada de una clave. Este utiliza **HashMap** (esta vez con herencia) con la clave como la única cadena que esta asociado a un valor en el **ArrayList**. El método **add()** simplemente busca si ya hay una clave en el **HashMap**, y si no está la coloca. El método **getArrayList()** produce un **ArrayList** para una clave en particular, y **printValues()**, que es principalmente útil para depurar, imprime todos los valores **ArrayList** a **ArrayList**.

Para que la vida se mantenga simple, los nombres de las clases de la librería estándar de Java son todos puestos dentro de el objeto **Properties** (desde la librería estándar de Java).

Se debe recordar que el objeto **Properties** es un **HashMap** que almacena solo objetos **String** para las claves y los valores. Sin embargo, se puede ser guardado en disco y recuperado del disco en una llamada a método, así es que es ideal para el almacén de nombres. Actualmente, necesitamos solo

una lista de nombres, y un **HashMap** no puede aceptar **null** para ninguna de sus entradas de claves o valores. Así es que el mismo objeto será utilizando para las claves y para el valor.

Para las clases y los identificadores que son descubiertos para los ficheros en un directorio en particular, dos **MultiStringMaps** son utilizados: **ClassMap** e **identMap**. Además, cuando el programa comienza carga el depósito de los nombres de clases en el objeto **Properties** llamado **classes**, y cuando un nuevo nombre de clase es encontrado en el directorio local es agregado a **classes** al igual que a **classMap**. De esta forma, **classMap** puede ser utilizado para moverse dentro de todas las clases dentro del directorio local, y **classes** puede ser utilizado para ver si el símbolo es un nombre de clase (lo que indica el comienzo de una definición de un objeto o método, así es que toma los siguientes símbolos -hasta un punto y coma- y luego los coloca dentro de **identMap**).

El constructor por defecto para **ClassScanner** crea una lista de nombres, utilizando la implementación **JavaFilter** o **FilenameFilter**, mostrado en el final del fichero. Luego llama a **scanListing()** para cada nombre de fichero.

Dentro de **scanListing()** el código fuente es abierto y enviado dentro de **StreamTokenizer**. En la documentación, pasando **true** a **slashStarComments()** y **slashSlashComments()** se supone desase de estos comentarios, pero esto parece ser un poco defectuoso, y no funciona totalmente. En lugar de eso, aquellas líneas que están comentadas y sus componentes son extraídas con otro método. Para hacer esto, el "/" debe ser capturado como un carácter común en lugar de autorizar que el **StreamTokenizer** lo absorba como parte de un comentario, y el método **ordinaryChar()** le indica a **StreamTokenizer** que haga esto. Esto es cierto también para puntos ("."), dado que queremos tomar la llamada al método y apartarla dentro de identificadores individuales. Sin embargo, el infragüion, que comúnmente es tratado con **StreamTokenizer** como un carácter individual, debe ser dejado como parte de identificadores dado que aparecen en valores **static final** como **TT\_EOF**, etc., utilizado en este mismo programa. El método **wordChars()** toma un rango de caracteres que se quiera agregar a aquellos que sobran dentro de un símbolo que es pasado como una palabra. Finalmente, cuando se analiza un comentario de una línea o se descarta una línea necesitamos conocer cuando un final de línea se sucede, así es que llamando **eolIsSignificant(true)** el EOL será mostrado en lugar de ser absorbido por el **StreamTokenizer**.

El resto de la lectura de **scanListing()** lee y reacciona a símbolos hasta el final del fichero, dado a entender cuando **nextToken()** retorna el valor **static final StreamTokenizer.TT\_EOF**.

Si el símbolo es un "/" es potencialmente un comentario, así es que **eatComments()** es llamado para tratar con el. La otra situación en la que

estamos interesados es si es una palabra, de las cuales hay algunos casos especiales.

Si la palabra es **class** o **interface** entonces el siguiente símbolo representa una clase o un nombre de interfase, y esta es colocada dentro de **classes** y en **classMap**. Si la palabra es **import** o **package**, entonces no queremos el resto de la línea. Cualquier otra cosa debe ser un identificador (en los cuales estamos interesados) o una palabra clave (en los cuales no estamos interesados, pero tienen que ser todos en minúsculas de todas formas así es que no vamos a echar a perder las cosas por ponerlas adentro). Estos son agregados a **identMap**.

El método **discardLine()** es una simple herramienta que busca el final de la línea. Debe notarse que en cualquier momento que se obtenga un nuevo símbolo, se debe verificar el final de el fichero.

El método **eatComments()** es llamado cada vez que una barra es encontrada en el bucle principal de análisis. Sin embargo, eso no necesariamente indica que un comentario se ha encontrado, así es que el siguiente símbolo debe ser extraído para ver si es otra barra (caso en el cual la línea es descartada) o un asterisco. ¡Pero si no es ninguno de estos, esto significa que el símbolo que se ha dejado atrás se necesita el bucle principal de análisis!

Afortunadamente, el método **pushBack()** permite “retroceder” el símbolo actual en el flujo de entrada así es que cuando el bucle principal de análisis llama a **nextToken()** podrá obtener el primero que simplemente fue retrocedido.

Por un tema de conveniencia, el método **classNames()** produce un arreglo de todos los nombres en el contenedor **classes**. Este método no es utilizado en el programa pero es útil para depuración.

Los siguientes dos métodos son los únicos en los cuales las pruebas actuales tienen lugar. En **checkclassNames()**, los nombres de clases son extraídos de el **classMap** (que, recordemos, solo contiene los nombres es este directorio, organizados por nombre de fichero, así es que el nombre de fichero puede ser impreso junto con el nombre de clase errante). Esto se logra empujando cada **ArrayList** asociado y moviéndonos a través de el, buscando para ver si el primer carácter es minúscula. Si lo es, el mensaje de error apropiado es impreso.

En **checkIdentNames()**, una estrategia similar es tomada: cada nombre de identificador es extraído de **identMap**. Si el nombre no esta en la lista **classes**, se asume que es un identificador o una palabra clave. Un caso especial es verificado: si el largo del identificador es tres o mas y todos los caracteres son mayúsculas, este identificador es ignorado porque es probable que sea un valor **static final** como lo es **TT\_EOF**. Claro, esto no es un algoritmo perfecto, pero asume que eventualmente note todos los identificadores en mayúsculas que estén fuera de lugar.

En lugar de reportar cada identificador que comienza con una mayúscula, este método mantiene la pista de cuales ya han sido reportados en una **ArrayList** llamando **reportSet()**. Esto trata el **ArrayList** como un “grupo” que nos indica cuando un elemento esta en el grupo. Este elemento es producido concatenando el nombre del fichero y el identificador. Si el elemento no esta en el grupo, es agregado y luego el reporte es hecho.

El resto del listado comprende el **main()**, que ocupa solo de manejar la línea de comandos y resuelve cuando se esta creando un almacén de nombres de clases de la librería estándar de Java o verificando si es válido el código que se ha escrito. En ambos casos crea un objeto **ClassScanner**.

Se este creando un almacén o utilizando uno, se debe tratar de abrir el almacén existente. Para crear un objeto **File** y verificar su existencia, se puede decidir por abrir el fichero y **load()** la lista de **classes Properties** dentro de **ClassScanner** (Las clases del almacén agregadas, en lugar de sobrescribir, las clases encontradas por el constructor **ClassScanner**). si se proporciona solamente un argumento en la línea de comandos significa que se quiere probar de los nombres de clases y de identificadores, pero si se proporcionan dos argumentos (el segundo comenzado con “-a”) se esta creando un nombre de un almacén de clases. En este caso, un fichero de salida es abierto y el método **Properties.save()** es utilizado para escribir la lista dentro de un fichero, junto con una cadena que proporcione la información del manejador de fichero.

## Resumen

La librería de E/S de Java satisface los requerimientos básicos: se puede realizar la lectura y escritura con la consola, un fichero, un bloque de memoria, o inclusive a través de la Internet (Como se verá en el capítulo 15). Con herencia, se pueden crear nuevos tipos de objetos de entrada y salida. Y se puede incluso agregar extensibilidad simple a los tipos de objetos en un flujo que acepta por redefinición el método **toString()** que es automáticamente llamado cuando se pasa un objeto a un método que esta esperando un **String** (“conversión de tipo automática” restringida de Java).

Hay preguntas sin contestar por la documentación y el diseño de la librería de flujo de E/S. Por ejemplo, si sería bueno si se podría decir que se quiere lanzar una excepción si se sobrescribe un fichero cuando se abre para salida -algunos sistemas de programación permiten especificar si se quiere abrir un fichero para salida, pero solo si no existe. En Java, se hace presente el uso del objeto **File** para determinar cuando un fichero existe, porque si se abre como un **FileOutputStream** o **FileWriter** este será siempre sobreescrito.

La librería de E/S trae sentimientos mezclados; hace mucho del trabajo y es portátil. Pero no se entiende todavía el patrón decorador, el diseño no

es intuitivo, así es que hay un trabajo extra en aprender y enseñarlo. Es también incompleto; no hay soporte para el tipo de salida con formato que al menos todos los otros paquetes de E/S soportan.

Sin embargo, una vez que se entienda el patrón decorador y se comience a utilizar la librería en situaciones que requiere su flexibilidad, se comenzara a beneficiarse de su diseño, hasta el punto en que el costo extra de líneas de código no importan mucho.

Si no se encontró lo que se estaba buscando en este capítulo (el cual es solo una introducción, y no trata de ser comprensivo), se puede encontrar cubierto en profundidad en *Java I/O*, por Elliotte Rusty Harold (O'Reilly, 1999)

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Abra un fichero de texto de tal forma en que se pueda leer el fichero una línea por vez. Lea cada línea como un **String** y coloque el objeto **String** en un **LinkedList**. Imprima todas las líneas en el **LinkedList** en orden reverso.
2. Modifique el Ejercicio 1 de tal forma en que el nombre del fichero que se lea sea proporcionado como un argumento en la línea de comandos.
3. Modifique el Ejercicio 2 para abrir también un fichero de texto de tal forma que se pueda escribir texto en él. Escriba las líneas en el **ArrayList**, junto con números de línea (no intente utilizar las clases "LineNumber"), fuera del fichero.
4. Modifique el Ejercicio 2 para forzar a que todas las líneas de **ArrayList** sean convertidas a mayúsculas y envíe los resultados a **System.out**.
5. Modifique el Ejercicio 2 para tomar argumentos adicionales en la línea de comandos de palabras para encontrar en el fichero. Imprima cualquier línea en donde haya una ocurrencia de estas palabras.
6. Modifique **DirList.java** de tal forma que el **FilenameFilter** realmente abra cada fichero y acepte el fichero basado en cualquiera de los argumentos rastreados en la línea de comando existan en ese fichero.
7. Cree una clase llamada **SortedDirList** con un constructor que tome la información de la ruta del fichero y cree una lista de directorios ordenada de los ficheros en esa ruta. Cree dos métodos sobrecargados **list()** que producirán la lista entera o un subgrupo de la lista basado en

un argumento. Agregue un método **size()** que tome un nombre de fichero y genere el tamaño para ese fichero.

8. Modifique **WordCount.java** de tal forma que el resultado este ordenado alfabéticamente, utilizando la herramienta del Capítulo 9.
9. Modifique **WordCount.java** para que utilice una clase que contenga un **String** y un valor contador que almacene cada palabra diferente, y un **Set** de estos objetos para mantener la lista de palabras.
10. Modifique **IOStreamDemo.java** de tal forma que utilice **LineNumberInputStream** para mantener una pista del contador de línea. Vea que es mucho mas fácil simplemente mantener la pista en forma de programa.
11. Comience con la sección 4 de **IOStreamDemo.java**, escriba un programa que compare el rendimiento de escribir en un fichero cuando se usa un buffer de E/S y cuando no se utiliza.
12. Modifique la sección 5 de **IOStreamDemo.java** para eliminar los espacios en la línea producida por la primer llamada a **in5br.readLine()**. Haga esto utilizando un bucle **while** y **readChar()**.
13. Repare el programa **CADState.java** como se describe en el texto.
14. En **Blips.java**, copie el fichero y cámbiele el nombre a **BlipCheck.java** y cámbiele el nombre también a la clase **Blip2** a **BlipCheck** (haciéndolo **public** y quitando el alcance público de la clase **Blips** en el proceso). Quite las marcas `//!` en el fichero y ejecute el programa incluyendo las líneas ofensivas. Luego, comente el constructor por defecto para **BlipCheck**. Ejecútelo y explique por que trabaja. Note que luego de compilar, se debe ejecutar el programa con “**java Blips**” porque el método **main()** sigue en la clase **Blips**.
15. En **Blip3.java**, comente las dos líneas luego de las frases “Se debe hacer esto”: y ejecute el programa. Explique el resultado y porque este difiere de cuando las dos líneas estaban en el programa.
16. (Inmediato) En el Capítulo 8, encuentre el ejemplo **GreenhouseControls.java**, que consiste en tres ficheros. En **GreenhouseControls.java**, la clase interna **Restart()** tiene un grupo de eventos muy codificados. Cambie el programa de tal forma que lea los eventos y sus tiempos relativos desde un fichero de texto. (Desafío: Use patrones de diseños *factory method* para crear los eventos -vea *Thinking in Patterns with Java* el que se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com))

# 12: Identificación de tipo en tiempo de ejecución

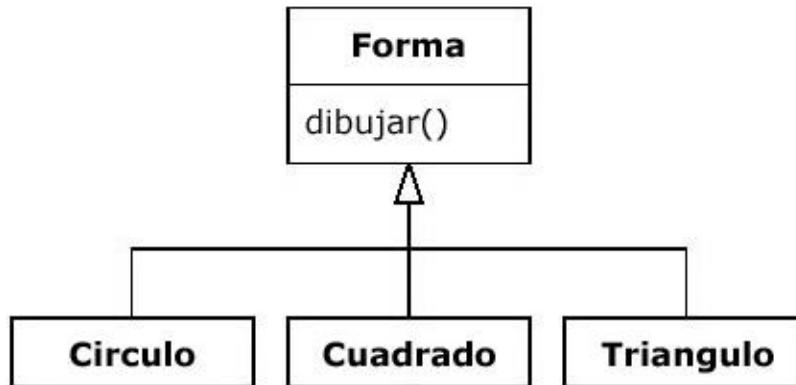
La idea de la identificación de tipo en tiempo de ejecución (RTTI run-time Type Identification) parece bastante simple al principio: esta permite encontrar el tipo exacto de un objeto cuando se tiene una referencia a el tipo base.

Sin embargo, la *necesidad* de RTTI descubre una abundancia de interesantes (y a menudo desconcertantes) temas de diseño OO, y surgen preguntas fundamentales de como se debe estructurar los programas.

Este capítulo analiza todas las formas en que Java permite descubrir información acerca de los objetos y clases en tiempo de ejecución. Esto toma dos formas: la “tradicional” RTTI, que asume que se tiene todos los tipos disponibles en tiempo de compilación y en tiempo de ejecución, y el mecanismo de “reflexión”, que permite descubrir la información de clases exclusivamente en tiempo de ejecución. La RTTI “tradicional” será cubierta primero, seguida por una discusión acerca de reflexión.

## La necesidad de una RTTI

Considerando el ahora familiar ejemplo de la jerarquía de clases que utiliza polimorfismo. El tipo genérico base es **Forma**, y el tipo derivado específico es **Círculo**, **Cuadrado** y **Triangulo**:



Este es un diagrama de una jerarquía de clases típica, con la clase base en la parte superior y las clases derivadas creciendo hacia abajo. El objetivo normal en la programación orientada a objetos es que la totalidad del código manipule referencias a el tipo base (**Forma**, en este caso), así si se decide extender el programa agregando nuevas clases (**Rombo**, derivada de **Forma**, por ejemplo), la totalidad del código no es afectado. En este ejemplo, el método de enlazado dinámico en la interfase de **Forma** es **dibujar()**, así es que lo que se intenta por parte del cliente programador es llamar a **dibujar()** a través de una referencia genérica a **Forma**. **dibujar()** es sobrescrita en todas las clases derivadas, y dado que es un método dinámico, el comportamiento correcto sucederá aún si es llamada a través de la referencia genérica **Forma**. Esto es polimorfismo.

De esta forma, generalmente se crea un objeto específico (**Circulo**, **Cuadrado**, o **Triangulo**), se realiza una conversión ascendente a **Forma** (se olvida el tipo específico del objeto), y se usa una referencia anónima **Forma** en el resto del programa.

Como un breve repaso de polimorfismo y conversiones ascendentes, se puede codificar el siguiente programa:

```

//: c12:Shapes.java
import java.util.*;
class Forma {
    void dibujar() {
        System.out.println(this + ".draw()");
    }
}
class Circulo extends Forma {
    public String toString() { return "Circulo"; }
}
class Cuadrado extends Forma {
    public String toString() { return "Cuadrado"; }
}
class Triangulo extends Forma {
    public String toString() { return "Triangulo"; }
}

```

```

public class Formas {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circulo());
        s.add(new Cuadrado());
        s.add(new Triangulo());
        Iterator e = s.iterator();
        while(e.hasNext())
            ((Forma)e.next()).draw();
    }
} //:~

```

La clase base contiene un método **dibujar()** que indirectamente utiliza **toString()** para imprimir un identificador para la clase pasando **this** a **System.out.println()**. Si esta función presencia un objeto, automáticamente llama el método **toString()** para producir una representación **String**.

Cada una de las clases derivadas sobrescriben el método **toString()** (de **Object**) así es que **draw()** termina imprimiendo algo diferente en cada caso. En el **main()**, tipos específicos de **Forma** son creados y luego agregados a un **ArrayList**. Este es el punto en donde la conversión ascendente sucede porque el **ArrayList** almacena solo **Objects**. Dado que todo en Java (con la excepción de las primitivas) es un **Object**, un **ArrayList** puede también almacenar objetos **Forma**. Pero durante una conversión ascendente a **Object**, se pierde alguna información específica, incluyendo el echo de que los objetos son **Formas**. Para el **ArrayList** son solo **Objects**.

En el punto en que se trae un elemento fuera del **ArrayList** con **next()**, las cosas se ponen un poco ajetreadas. Dado que **ArrayList** guarda solo **Objects**, **next()** naturalmente produce una referencia a **Object**. Pero sabemos realmente que es una referencia a **Forma**, y queremos enviar mensajes de **Forma** a ese objeto. Así es que es necesario convertir a **Forma** utilizando el tradicional conversión “(**Forma**)”. Esta es la forma mas básica de RTTI, dado que en Java todas las conversiones son verificadas en tiempo de ejecución para ver si son correctas. Esto es exactamente lo que RTTI significa: en tiempo de ejecución, el tipo de objeto es identificado.

En este caso, la conversión RTTI es solo parcial: el **Object** es convertido a **Forma**, y no del todo, hasta **Circulo**, **Cuadrado**, o **Triangulo**. Esto es porque lo único que *sabemos* en este punto es que el **ArrayList** está lleno de **Formas**.

En tiempo de compilación, esto es forzado solo por las reglas que uno se impone a si mismo, pero en tiempo de ejecución la conversión asegura esto.

Ahora el polimorfismo se encarga y el método exacto que es llamado para **Forma** es determinado por si la referencia es un **Circulo**, un **Cuadrado**, o un **Triángulo**. En general, esto es como suele ser; se quiere que la totalidad del código sepa lo mínimo posible acerca de los tipos *específicos* de objetos, y para tratar solamente con la representación general de una familia de objetos (en este caso, **Forma**). Como resultado, el código será mucho mas

fácil de escribir, leer, y mantener, y sus diseños serán mas fáciles de implementar, entender, y cambiar. Así es que polimorfismo es la meta general en la programación orientada a objetos.

¿Pero que si se tiene un problema especial de programación que es fácil de solucionar si se sabe el tipo exacto de una referencia genérica? Por ejemplo, supongamos que se quiere permitir a usuarios iluminar todas las formas de un tipo particular haciéndolas mas púrpuras. De esta forma, se pueden encontrar todos los triángulos en la pantalla iluminándolos. Esto es lo que RTTI logra: se puede preguntar a la referencia **Forma** el tipo exacto al cual se está refiriendo.

## El objeto **Class**

Para entender como RTTI trabaja en Java, se debe primero saber que tipo de información es representada en tiempo de ejecución. Esto se logra a través de un tipo especial de objeto llamado el *objeto Class*, que contiene información acerca de la clase (Esto es algo llamado una *meta-clase*). De echo, el objeto **Class** es utilizado para crear todos los objetos “comunes” de las clases.

Hay un objeto **Class** para cada clase que es parte de un programa. Esto es, cada vez que se escribe y se compila una nueva clase, un objeto **Class** simple es también creado (y almacenado, suficientemente apropiado, en un fichero **.class** nombrado idénticamente). En tiempo de ejecución, cuando se quiere crear un objeto de esta clase, la Maquina Virtual de Java (JVM) que está ejecutando su programa primero prueba para ver si el objeto **Class** para este tipo es cargado. Si no está, la JVM lo carga encontrando el fichero **.class** con ese nombre. De esta forma, un programa Java, no es completamente cargado antes de que comience, lo que es diferente de muchos lenguajes tradicionales.

Una vez que el objeto **Class** para ese tipo esta en memoria, es utilizado para crear todos los objetos de ese tipo.

Si esto parece sombrío o si no cree realmente en esto, aquí hay un programa demostración que lo prueba:

```
//: c12:SweetShop.java
// Examination of the way the class loader works.
class Candy {
    static {
        System.out.println("Loading Candy");
    }
}
class Gum {
    static {
        System.out.println("Loading Gum");
    }
}
```

```

    }
    class Cookie {
        static {
            System.out.println("Loading Cookie");
        }
    }
    public class SweetShop {
        public static void main(String[] args) {
            System.out.println("inside main");
            new Candy();
            System.out.println("After creating Candy");
            try {
                Class.forName("Gum");
            } catch(ClassNotFoundException e) {
                e.printStackTrace(System.err);
            }
            System.out.println(
                "After Class.forName(\"Gum\")");
            new Cookie();
            System.out.println("After creating Cookie");
        }
    }
}

```

Cada una de las clases **Candy**, **Gum**, y **Cookie** tienen una cláusula **static** que es ejecutada cuando la clase es cargada por primera vez. La información será impresa para indicar cuando la carga ocurre para esa clase. En el **main()**, la creación de los objetos son separadas en instrucciones para imprimir para ayudar a detectar el tiempo de carga.

Una línea particularmente interesante es:

```
| Class.forName("Gum");
```

Este método es un miembro **static** de **Class** (a el cual todos los objetos **Class** pertenecen). Un objeto **Class** es como cualquier otro objeto y se puede obtener y manipular la referencia a el (Esto es lo que el cargador hace). Una de las formas para obtener una referencia a el objeto **Class** es **forName()**, que toma un **String** contenido el nombre textual (¡Se debe observar la ortografía y la mayúsculas y minúsculas!) de la clase particular a la cual se quiere hacer referencia. Esta 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

```

Se puede ver que cada objeto **Class** es solo cargada solo cuando es necesario, y la inicialización estática es realizada en la carga de la clase.

## Literales de clase

Java proporciona una segunda forma de producir la referencia a el objeto **Class**, utilizando un *literal de clase*. En el programa anterior esto se vería como:

```
| Gum.class;
```

lo que no solo es simple, también es seguro dado que es verificado en tiempo de compilación. Dado que elimina la llamada a el método, es también mas eficiente.

Los literales de clases trabajan con clases regulares de la misma forma que las interfaces, arreglos y tipos primitivos. Además, hay un campo estándar llamado **TYPE** que existe para cada una de las clases que envuelven primitivas. El campo **TYPE** produce una referencia a el objeto **Class** para el tipo primitivo asociado, como estas:

... es equivalente a ...	
<b>boolean.class</b>	<b>Boolean.TYPE</b>
<b>char.class</b>	<b>Character.TYPE</b>
<b>byte.class</b>	<b>Byte.TYPE</b>
<b>short.class</b>	<b>Short.TYPE</b>
<b>int.class</b>	<b>Integer.TYPE</b>
<b>long.class</b>	<b>Long.TYPE</b>
<b>float.class</b>	<b>Float.TYPE</b>
<b>double.class</b>	<b>Double.TYPE</b>
<b>void.class</b>	<b>Void.TYPE</b>

Mi preferencia es utilizar las versiones “.class” si se puede, dado que son mas consistentes con las clases regulares.

## Probar antes de convertir

Hasta el momento, se ha visto las formas RTTI incluyendo:

1. La conversión clásica; e.g. “(**Shape**)” que utiliza RTTI para asegurarse que la conversión es correcta y lanza una **ClassCastException** y se ha realizado una mala conversión.
2. El objeto **Class** que representa el tipo del objeto. El objeto **Class** puede ser consultado por información útil en tiempo de ejecución.

En C++, la conversión clásica “(**Forma**)”, *no* realiza una RTTI. Simplemente indica que el compilador debe tratar el objeto como al nuevo tipo. En Java, que realiza un chequeo de tipo, esta conversión es llamada a menudo una “conversión descendiente segura de tipo” (type safe downcast). La razón para el término “conversión descendiente” (downcast) es el ordenamiento histórico del diagrama de jerarquía de clases. Si se realiza una conversión de un **Círculo** en una **Forma** es una conversión ascendente, entonces la conversión de una **Forma** en un **Círculo** es una conversión descendente. Sin embargo, se sabe que un **Círculo** es también una **Forma**, y el compilador libremente permite una asignación mediante una conversión ascendente, pero *no* sabe que una **Forma** es necesariamente un **Círculo**, así es que el compilador no permite realizar una asignación con una conversión descendiente sin utilizar una conversión explícita.

Hay una tercera forma de RTTI en Java. Es la palabra clave **instanceof** que indica si un objeto es una instancia particular de un tipo. Este retorna un **boolean** así es que utilice en la forma de una pregunta, como esta:

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

La instrucción **if** anterior verifica si el objeto **x** pertenece a la clase **Dog** *antes* de realizar la conversión de **x** a **Dog**. Es importante el uso de **instanceof** *antes* de realizar la conversión descendiente cuando no se tiene otra información que indique el tipo de objeto; de otra forma se terminará con una **ClassCastException**.

Normalmente, se debe ser un cazador para un tipo (triángulos para poner en púrpura, por ejemplo), pero se puede fácilmente etiquetar *todos* los objetos utilizando **instanceof**. Supóngase que se tiene una familia de clases **Pet**:

```
//: c12:Pet.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; } ///:~
```

Su clase **Counter** es utilizada para mantener la pista del número de cualquier tipo particular de **Pet**. Se puede pensar en esto como un **Integer** que puede ser modificado.

Utilizando **instanceof**, todas las mascotas pueden ser contadas:

```
//: c12:PetCount.java
// Using instanceof.
import java.util.*;
public class PetCount {
    static String[] typenames = {
        "Pet", "Dog", "Pug", "Cat",
```

```

        "Rodent", "Gerbil", "Hamster",
    };
// Exceptions thrown out to console:
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("Cannot instantiate");
        throw e;
    } catch(IllegalAccessException e) {
        System.err.println("Cannot access");
        throw e;
    } catch(ClassNotFoundException e) {
        System.err.println("Cannot find class");
        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] + " quantity: " +

```

```

        ((Counter)h.get(typenames[i])).i;
    }
} //:~

```

Hay aquí mas bien una estrecha restricción de **instanceof**: se puede comparar con nombrar solamente el tipo, y no con un objeto **Class**. En el ejemplo anterior se puede sentir que es tedioso escribir todas estas expresiones, y es cierto. Pero no hay forma de automatizar inteligentemente **instanceof** para crear un **ArrayList** de objetos **Class** y compararlos aquellos (siga sincronizado-mostraremos una alternativa). Esto no es mayormente una restricción como se puede pensar, dado que eventualmente se entenderá que el diseño es defectivo si se termina escribiendo un montón de expresiones **instanceof**.

Claro que este ejemplo es fraguado -probablemente se pondrá un miembro de datos **static** en cada tipo y se incrementará este en el constructor para mantener la pista de los contadores. Se puede hacer algo como esto *sí* se tiene el control sobre el código fuente para la clase y cambiarla. Dado que este no es siempre el caso, RTTI se puede volver conveniente.

## Usando literales de clases

Es interesante ver como el ejemplo **PetCount.java** puede ser reescrito utilizando literales de clases. El resultado es mas limpio de muchas formas:

```

//: c12:PetCount2.java
// Using class literals.
import java.util.*;
public class PetCount2 {
    public static void main(String[] args)
        throws Exception {
        ArrayList pets = new ArrayList();
        Class[] petTypes = {
            // Class literals:
            Pet.class,
            Dog.class,
            Pug.class,
            Cat.class,
            Rodent.class,
            Gerbil.class,
            Hamster.class,
        };
        try {
            for(int i = 0; i < 15; i++) {
                // Offset by one to eliminate Pet.class:
                int rnd = 1 + (int)(Math.random() * (petTypes.length - 1));
                pets.add(petTypes[rnd].newInstance());
            }
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        }
    }
}

```

```

    } catch(IllegalAccessException e) {
        System.err.println("Cannot access");
        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("class Pet")).i++;
        if(o instanceof Dog)
            ((Counter)h.get("class Dog")).i++;
        if(o instanceof Pug)
            ((Counter)h.get("class Pug")).i++;
        if(o instanceof Cat)
            ((Counter)h.get("class Cat")).i++;
        if(o instanceof Rodent)
            ((Counter)h.get("class Rodent")).i++;
        if(o instanceof Gerbil)
            ((Counter)h.get("class Gerbil")).i++;
        if(o instanceof Hamster)
            ((Counter)h.get("class 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) +
            " quantity: " + cnt.i);
    }
}
}
} //:~
}

```

Aquí, los **typenames** fueron quitados para propiciar la obtención de las cadenas con el nombre del tipo del objeto **Class**. Debe notarse que el sistema puede distinguir entre clases e interfaces.

Se puede ver también que la creación de **petTypes** no necesita ser rodeada con un bloque **try** dado que es evaluada en tiempo de compilación y de esta forma no lanza ninguna excepción, a diferencia de **Class.forName()**.

Cuando los objetos **Pet** son creados dinámicamente, se puede ver que el número aleatorio está limitado así es que está entre uno y **petTypes.length** y no incluye el cero. Esto es porque el cero se refiere a **Pet.class**, y presumiblemente un objeto **Pet** genérico no tiene interés. Sin embargo, dado que **Pet.class** es parte de **petTypes** el resultado es que todas las mascotas son contadas.

## Un **instanceof** dinámico

El método **isInstance** de **Class** proporciona una forma de llamar dinámicamente el operador **instanceof**. De esta forma, todos esas tediosas instrucciones **instanceof** pueden ser quitados en el ejemplo **PetCount**:

```
//: c12:PetCount3.java
// Using 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++) {
                // Offset by one to eliminate Pet.class:
                int rnd = 1 + (int)(Math.random() * (petTypes.length - 1));
                pets.add(
                    petTypes[rnd].newInstance());
            }
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            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);
            // Using isInstance to eliminate individual
            // instanceof expressions:
            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) +
                " quantity: " + cnt.i);
        }
    }
} //:~

```

Se puede ver que el método **isInstance()** ha eliminado la necesidad de las expresiones **instanceof**. Además, esto significa que se puede agregar nuevos tipos de mascotas cambiando simplemente el arreglo **petTypes**; el resto del programa no necesita modificación (como cuando se utilizaban las expresiones **instanceof**).

## **instanceof** contra equivalencia de **Class**

Cuando se consulta por el tipo de información, hay una diferencia importante entre la forma en que **instanceof** (esto es, **instanceof** o **isInstance()**, que producen resultados equivalentes) y la comparación directa entre objetos **Class**. Aquí hay un ejemplo que demuestra la diferencia:

```

//: c12:FamilyVsExactType.java
// The difference between instanceof and class
class Base {}
class Derived extends Base {}
public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println("Testing x of type " +
            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 la prueba del tipo con su argumento utilizando ambas formas de **instanceof**. Esta toma entonces la referencia a **Class** y utiliza == y **equals()** para probar la igualdad de los objetos **Class**. He aquí 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** y **isInstance()** producen exactamente el mismo resultado, como lo hacen **equals()** y ==. Pero estas pruebas por si solas obtienen diferentes conclusiones. Manteniendo el concepto de tipo, **instanceof** dice “¿eres tu de esta clase o una clase derivada de esta clase?” Por el otro lado, si se compara los objetos **Class** actuales utilizando ==, no hay preocupación por la herencia- es el tipo exacto o no lo es.

## Sintaxis de RTTI

Java realiza su RTTI utilizando el objeto **Class**, aún si se está haciendo algo como una conversión. La clase **Class** también tiene otras formas para utilizar RTTI.

Primero, se debe tener una referencia a el objeto **Class** apropiado. Una forma de hacer esto, es mostrada en el ejemplo anterior, utilizando una cadena y el método **Class.forName()**. Esto es conveniente porque no se necesita un objeto de ese tipo para obtener la referencia **Class**. Sin embargo, si ya se tiene un objeto del tipo en que se está interesado, se puede obtener la referencia a **Class** llamando un método que sea parte de la clase raíz **Object: getClass()**. Este retorna la referencia a **Class** representando el actual tipo del objeto. **Class** tiene muchos métodos interesantes, demostrado en el siguiente ejemplo:

```

//: c12:ToyTest.java
// Testing class Class.
interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
    // Comment out the following default
    // constructor to see
    // 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("Can't find 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 {
            // Requires default constructor:
            o = cy.newInstance(); // (*1*)
        } catch(InstantiationException e) {
            System.err.println("Cannot instantiate");
            throw e;
        } catch(IllegalAccessException e) {
            System.err.println("Cannot access");
            throw e;
        }
        printInfo(o.getClass());
    }
    static void printInfo(Class cc) {
        System.out.println(
            "Class name: " + cc.getName() +
            " is interface? [" +
            cc.isInterface() + "]");
    }
} ///:~

```

Se puede ver que **class FancyToy** es bastante complicada, dado que hereda de **Toy** e implementa las interfaces de **HasBatteries**, **Waterproof**, y

**ShootsThings**. En el **main()**, una referencia **Class** es creada e inicializada a la **Class FancyToy** utilizando **forName()** dentro de un bloque **try** apropiado.

El método de la clase **Class.getInterfaces** retorna un arreglo de objetos **Class** representando las interfases que son contenidas en el objeto **Class** que interesa.

Si se tiene un objeto **Class** se puede también preguntar por este directamente a la clase base utilizando **getSuperclass()**. Esto, claro, retorna una referencia **Class** que se puede consultar mas adelante. Esto significa que, en tiempo de ejecución, se puede descubrir una jerarquía de clases de objetos completa.

El método **newInstance()** de **Class** puede, en un inicio, parecer simplemente como otra forma de utilizar **clone()** para clonar un objeto. Sin embargo, se puede crear un objeto nuevo con **newInstance()** *sin* un objeto existente, como se ha visto aquí, dado que no hay ningún objeto **Toy** -solo **ey**, que es una referencia a un objeto **Class** de **y**. Esto es una forma de implementar un “constructor virtual”, que permite que se diga “No se exactamente que tipo eres, pero créate a ti mismo de una forma apropiada”. En el ejemplo mas arriba, **ey** es simplemente una referencia a **Class** sin información adicional de tipo conocida en tiempo de compilación. Y cuando se crea una nueva instancia, se regresa una **Object reference**. Pero esta referencia apunta a un objeto **Toy**. Claro, antes de que se pueda enviar cualquier mensaje mas que aquellos aceptados por **Object**, se tiene que investigar un poco y realizar alguna conversión. Además, la clase que es creada con **newInstance()** debe tener un constructor por defecto. En la siguiente sección, se verá como se crea dinámicamente objetos de clases utilizando cualquier constructor, con la API de *reflexión* de Java.

El último método en el listado es **printInfo()**, que toma una referencia **Class** y toma su nombre con **getName()**, y encuentra si tiene una interfase mediante **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 forma, con el objeto **Class** se puede encontrar simplemente todo lo que se quiere saber de un objeto.

# Reflexión: información de clase en tiempo de ejecución

Si no se conoce el tipo exacto de un objeto, RTTI lo dirá. Sin embargo, hay una limitación: el tipo debe ser un tipo conocido en tiempo de compilación para poder ser capaz de detectarlo utilizando RTTI y hacer algo útil con la información. Poniéndolo de otra forma, el compilador debe saber de todas las clases con las que se está trabajando con RTTI.

Esto no parece mucho una limitación al comienzo, pero supongamos que se está dando una referencia a un objeto que no está en el espacio del programa. De hecho, la clase del objeto incluso no está disponible para el programa en tiempo de compilación. Por ejemplo, supongamos que se tiene un grupo de bytes de un fichero de disco o de una conexión de red y se dijo que estos bytes representan una clase. ¿Dado que el compilador no puede saber acerca de la clase que compilo el código, como es posible utilizar la clase?

En un ambiente tradicional de programación parece como un escenario ilógico. Pero mientras más nos movemos en un mundo de programación más amplio hay casos importantes en donde esto sucede. El primero es la programación basada en componentes, en donde se crean proyectos utilizando *Aplicaciones Rápidas de Desarrollo (RAD- Rapid Application Development)* en una herramienta creadora de aplicaciones. Esto es una estrategia visual para crear un programa (en donde se ve en la pantalla como un “formulario”) se mueven íconos que representan componentes en la forma. Estos componentes son luego configurados ajustando algunos de sus valores en tiempo de programación. Esta configuración en tiempo de diseño requiere que se pueda tener una instancia de cualquier componente, esto da a conocer parte de si mismo, y permite que sus valores sean leídos y configurados. Además, los componentes que manejan eventos GUI deben exponer información acerca de métodos apropiados así es que el ambiente RAD puede asistir al programador sobrecargando estos métodos manejadores de eventos. La reflexión proporciona los mecanismos para detectar los métodos disponibles y producir los nombres de métodos. Java proporciona una estructura para programación basada en componentes a través de JavaBeans (Descrito en el Capítulo 13).

Otra motivación convincente para descubrir la información de clase en tiempo de ejecución es proporcionar la habilidad de crear y ejecutar objetos en plataformas remotas a través de la red. Esto es llamado *Método Remoto*

*de Invocación (RMI - Remote Method Invocation)* y esto permite a un programa Java tener objetos distribuidos a través de muchas máquinas. Esta distribución puede suceder a causa de una cantidad de razones: por ejemplo, tal vez se este haciendo una tarea de computación muy intensa y se quiere dividir y colocar estas partes en máquinas que están desocupadas para que la velocidad de las cosas aumente. En algunas situaciones se debe querer colocar el código que maneja tipos particulares de tareas (por ejemplo, "Reglas de negocios" en una arquitectura cliente/servidor multiter) en una máquina particular, así es que la máquina se convierte en un depósito común describiendo aquellas acciones y puede ser fácilmente cambiado para afectar a cualquiera en el sistema (¡Esto es un desarrollo interesante, dado que la máquina existe solamente para hacer cambios en software fácilmente!). A lo largo de estas líneas, la computación distribuida también soporta hardware especializado que puede ser bueno en una tarea particular-inversiones de matrices, por ejemplo-pero inapropiada o muy cara para programación de propósito general.

La clase **Class** (descrita previamente en este capítulo) soporta el concepto de reflexión, y aquí hay una librería adicional, **java.lang.reflect**, con clases **Field**, **Method**, y **Constructor** (cada una de los cuales implementa el **Member interface**). Los objetos de estos tipos son creados por la JVM en tiempo de ejecución para representar los correspondientes miembros en la clase desconocida. Se puede entonces utilizar los **Constructors** para crear nuevos objetos, los métodos **get()** y **set()** para leer y modificar los campos asociados con los objetos **Field**, y el método **invoke()** para llamar a el método asociado con el objeto **Method**. Además, se puede llamar los métodos por conveniencia **getFields()**, **getMethods()**, **getConstructors()**, etc., para retornar arreglos de estos objetos representando los campos métodos, y constructores (Se puede encontrar mas buscando la clase **Class** en su documentación en línea). De esta forma, la información de clase para objetos anónimos puede ser completamente determinada en tiempo de ejecución, y no se necesita conocer nada en tiempo de compilación.

Es importante darse cuenta que no hay nada mágico acerca de la reflexión. Cuando se está utilizando reflexión para interactuar con un objeto de un tipo desconocido, la JVM simplemente busca en el objeto y ve que pertenece a una clase particular (exactamente igual que RTTI) pero entonces, antes de que se pueda hacer algo más, el objeto **Class** debe ser leído. De esta forma el fichero **.class** para ese tipo en particular debe seguir estando disponible para la JVM, en la máquina local o a través de la red. Así es que la verdadera diferencia entre RTTI y reflexión es que con RTTI, el compilador abre y examina el fichero **.class** en tiempo de compilación. Puesto de otra forma, se puede llamar todos los métodos de un objeto de la forma "normal". Con reflexión, el fichero **.class** no está disponible en tiempo de compilación; este es abierto y examinado por el ambiente en tiempo de ejecución.

# Un extractor de métodos de clase

Raramente se necesitará utilizar herramientas de reflexión directamente; están en el lenguaje para soportar otras características de Java, como la serialización de objetos (Capítulo 11), JavaBeans (Capítulo 13), y RMI (Capítulo 15). Sin embargo, hay momentos en que es bastante útil ser capaz de extraer información acerca de una clase. Una herramienta extremadamente útil es un extractor de métodos de clase. Como mencionamos antes, buscando en el código fuente de definición de clase o en documentación en línea se muestra solo los métodos que son definidos o sobrescritos *dentro de la definición de clase*. Pero puede haber docenas mas disponibles que vienen de la clase base. El localizar estos es tedioso y consumen tiempo<sup>1</sup>. Afortunadamente, la reflexión proporciona una forma de escribir una herramienta simple que automáticamente muestre la interfase entera. Aquí esta la forma en que trabaja:

```
//: c12>ShowMethods.java
// Utilizando reflexión para mostrar todos los métodos de
// una clase, inclusive los métodos que estan definidos en
// la clase base.
import java.lang.reflect.*;
public class ShowMethods {
    static final String usage =
        "use: \n" +
        "ShowMethods qualified.class.name\n" +
        "Para motrar todos los métodos en la clase o: \n" +
        "ShowMethods qualified.class.name palabra\n" +
        "PAra buscar métodos que involucren '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++)
```

---

<sup>1</sup> Especialmente en el pasado. Sin embargo, Sun ha mejorado enormemente su documentación HTML, para que fácilmente se vean los métodos de la clase base.

```

        if(ctor[i].toString()
           .indexOf(args[1])!= -1)
          System.out.println(ctor[i]);
    }
} catch(ClassNotFoundException e) {
  System.err.println("No such class: " + e);
}
}
} //:~

```

Los métodos de **Class getMethods()** y **getConstructors()** retornan un arreglo de **Method** y **Constructor**, respectivamente. Cada una de estas clases tienen métodos adicionales para analizar los nombres, argumentos, y valores de retorno de los métodos que representan. Pero se puede también simplemente utilizar **toString()**, como se ha hecho aquí, para producir un **String** con la firma de método entera. El resto del código es simplemente para extraer la información de la línea de comandos, determinando si una firma en particular coincide con una cadena (utilizando **indexOf()**), e imprimiendo los resultados.

Esto muestra reflexión en acción, dado que el resultado producido por **Class.forName()** no puede ser conocida en tiempo de compilación, y sin embargo toda la información de firma de los métodos es extraída en tiempo de ejecución. Si se investiga la documentación en línea sobre reflexión, se verá que hay suficiente soporte para actualmente configurar y hacer una llamada a método en un objeto que es totalmente desconocido en tiempo de compilación (hay ejemplos de esto mas adelante en este libro). Nuevamente, esto es algo que nunca se necesitará hacer por si mismo -el soporte esta ahí para RMI y de esta manera un ambiente de programación puede manipular JavaBeans- pero es interesante.

Un experimento interesante es ejecutar

```
| java ShowMethods ShowMethods
```

Esto produce un listado que incluye un constructor **public** por defecto, aún cuando se puede ver en el código que ningún constructor fue definido. El constructor que se ve es el que es automáticamente sintetizado por el compilador. Si luego hace **ShowMethods** en una clase que no es pública (esto es, amigable), el constructor por defecto sintetizado no se muestra mas en la salida. El constructor por defecto sintetizado automáticamente tiene el mismo acceso que la clase.

La salida para **ShowMethods** sigue siendo un poco tediosa. Por ejemplo, he aquí una parte de la salida producida invocando **java ShowMethods 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)  
Sería mas bonita si los calificadores como **java.lang** puedan ser eliminados.  
La clase **StreamTokenizer** introducida en el capítulo anterior puede ayudar a  
crear una herramienta para solucionar 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(' '); // Keep the spaces
    }
    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: // single character in ttype
                        s = String.valueOf((char)st.ttype);
                }
            }
        } catch(IOException e) {
            System.err.println("Error fetching 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, esto utiliza **StreamTokenizer** y la manipulación de **String** para hacer el trabajo.

La nueva versión del programa utiliza la clase anterior para limpiar la salida:

```
//: c12>ShowMethodsClean.java
// ShowMethods sin los calificadores
// para hacer los resultados mas fáciles de leer.
import java.lang.reflect.*;
import com.bruceeckel.util.*;
public class ShowMethodsClean {
    static final String usage =
        "use: \n" +
        "ShowMethodsClean qualified.class.name\n" +
        "Para mostrar todos los métodos en la clase o: \n" +
        "ShowMethodsClean qualif.class.name palabra\n" +
        "Para buscar métodos que involucren '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();
            // Convertir a un arreglo de cadenas 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 such class: " + e);
        }
    }
} ///:~
```

La clase **ShowMethodsClean** es bastante similar a la anterior **ShowMethods**, a no ser porque toma los arreglos de **Method** y los **Constructor** y los

convierte en un solo arreglo de **String**. Cada uno de estos objetos **String** es pasado a través de **StripQualifiers.Strip()** para quitar todas las calificaciones de métodos.

Esta herramienta puede ser un salvador en tiempo real cuando se este programando, cuando no pueda recordar si la clase tiene un método en particular y no se quiere buscar en la jerarquía de clases en la documentación en línea, o si no se sabe si la clase puede hacer algo, por ejemplo, objetos **Color**.

El Capítulo 13 tiene la versión GUI de este programa (optimizado para extraer información de componentes Swing), así es que se puede dejar corriendo cuando se esta escribiendo código, para permitir miradas rápidas.

## Resumen

RTTI permite descubrir la información de tipo de una referencia de clase base anónima. De esta forma, se presta para ser mal utilizado por los aficionados dado que puede tener sentido antes de hacer las llamadas a métodos polimórficos. Para muchas personas que vienen con bases procesales, es difícil no organizar sus programas en un grupo de instrucciones **Switch**. Se puede lograr esto con RTTI y de esta forma perder el importante valor del polimorfismo en el desarrollo de código y mantenimiento. El empeño de Java es que se utilicen llamadas a métodos polimórficos a través de su código, y se utiliza RTTI solo cuando se debe.

Sin embargo, utilizando llamadas a métodos polimórficos como es la idea requiere que se tenga control sobre la definición de la clase base dado que en algún punto en la extensión de su programa de debe descubrir que la clase base no incluye el método que se necesita. Si la clase base viene de una librería o es de otra forma controlada por alguien mas, una solución al problema es RTTI: se puede heredar un nuevo tipo y agregar el método extra. En alguna parte del código se puede detectar el tipo en particular y llamar ese método especial. Esto no destruye el polimorfismo y la extensibilidad del programa porque el agregar un nuevo tipo no requiere que se busque en instrucciones **switch** en el programa. Sin embargo, cuando se agrega nuevo código en el cuerpo principal que requiere nuevas características, se debe utilizar RTTI para detectar el tipo en particular.

El colocar una característica en la clase base puede significar que, para el beneficio de una clase particular, todas las otras clases derivadas de esta base requieran alguna eliminación sin sentido de un método. Eso hace que la interfase sea menos limpia y molesta a aquellos que tienen que sobrecargar métodos abstractos cuando derivan de la clase base. Por ejemplo, considere una jerarquía de clases que represente instrumentos musicales. Supongamos que se quiere limpiar las válvulas de saliva de todos

los instrumentos apropiados en la orquesta. Una opción es colocar un método **limpiarValvulasDeSaliva()** en la clase base **Instrumento**, pero esto es confuso dado que implica que los instrumentos de **Percusión** y **Electrónica** también tienen válvulas de saliva. RTTI proporciona una forma mas razonable de solucionar en este caso porque se puede colocar el método en una clase específica (**Viento** en este caso), donde es apropiado. Sin embargo, una solución mas apropiada es colocar un método **prepareInstrumento()** en la clase base, pero puede que no se vea esto cuando se esta inicialmente solucionando el problema y puede de forma equivocada asumir que se debe utilizar RTTI.

Finalmente, RTTI puede a veces solucionar problemas eficientemente. Si su código utiliza polimorfismo, pero uno de sus objetos reacciona a este código de propósito general en una forma horriblemente ineficiente, se puede seleccionar ese tipo utilizando RTTI y escribir código específico para ese caso para mejorar la eficiencia. Hay que ser precavido, sin embargo, de programar eficiencia muy temprano. Es una trampa muy seductora. Es mejor tener el programa trabajando *primero*, luego se decide si esta ejecutándose suficientemente rápido y solo entonces se debe atacar los temas de eficiencia -con un perfilador.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Agregue **Rombo** a **Shapes.java**. Cree un **Rombo**, realice una conversión ascendente a **Forma**, luego realice una conversión descendente a **Rombo** nuevamente. Trate de realizar una conversión descendente a un **Circulo** y vea que sucede.
2. Modifique el ejercicio 1 de tal forma que utilice **instanceof** para verificar el tipo antes de realizar la conversión descendiente.
3. Modifique **Shapes.java** de tal forma que puede “iluminar” (configure una bandera) todas las formas de un tipo en particular. El método **toString()** para cada **Forma** derivada puede indicar que forma es “iluminada”.
4. Modifique **SwetShop.java** de tal forma que cada tipo de creación de objeto sea controlada por un argumento en la línea de comandos. Esto es, si su línea de comandos es “**java SwetShop Candy**”, entonces solo el objeto **Candy** es creado. Note como se puede controlar que objeto **Class** es cargado mediante un argumento en la línea de comandos.

5. Agregue un nuevo tipo de Mascota a **PetCount3.java**. Verifique que es creada y contada correctamente en el **main()**.
6. Escriba un método que tome un objeto y de forma recursiva imprima todas las clases en esa jerarquía de objetos.
7. Modifique el Ejercicio 6 de tal forma que utilice **Class.getDeclaredFields()** para desplegar información acerca de los campos en una clase.
8. En **ToyTest.java**, quite el comentario al constructor por defecto y explique que sucede.
9. Cree un nuevo tipo de contenedor que utilice un arreglo privado **ArrayList** para almacenar los objetos. Capture el tipo del primer objeto que se coloque en el, y permita entonces que el usuario inserte objetos de este tipo solamente de ahí en adelante.
11. Escriba un programa para determinar cuando un arreglo de **char** es un tipo primitivo o un verdadero objeto.
12. Implemente **limpieValvulaDeSaliva()** como se describe en el resumen.
13. Implemente el método **rotate(Shape)** como se describe en este capítulo, tal que verifique para ver si esta rotando un **Circulo** (y, si lo es, no realice la operación).
14. Modifique el Ejercicio 6 de tal forma que utilice reflexión en lugar de RTTI.
15. Modifique el Ejercicio 7 de tal forma que utilice reflexión en lugar de RTTI.
16. Busque la interfase para **java.lang.Class** en la documentación de *java.sun.com*. Escriba un programa que tome el nombre de la clase como un argumento en la línea de comandos, luego utilice los métodos de **Class** para volcar toda la información disponible para esta clase. Verifique su programa con una clase de la librería estándar y con una clase que usted cree.

# 13: Creando Ventanas & Applets

Una línea guía fundamental en el diseño es “hacer las cosas simples fáciles, y las cosas difíciles posibles”<sup>1</sup>.

La meta original del diseño de la librería de interfase gráfica de usuario (GUI - Graphical User Interface) en Java 1.0 fue permitir al programador crear una GUI que se vea bien en todas las plataformas. Esta meta no fue alcanzada. En lugar de eso, la *Abstract Window Toolkit*(AWT) de Java 1.0 produce una GUI que se ve igualmente mediocre en todos los sistemas.

Además, es restrictiva: solo se pueden utilizar cuatro fuentes y no se puede acceder a ninguna de los elementos GUI mas sofisticados que existen en el sistema operativo. El modelo de programación de la AWT de java 1.0 es también es complicada y no esta orientada a objetos. Un estudiante en uno de mis seminarios (que estaba en Sun durante la creación de Java) explico por que: el AWT original ha sido conceptualizado, diseñado, e implementado en un mes. Ciertamente una maravilla de productividad, y también una lección a objetar porque el diseño es importante.

La situación mejora con la AWT de Java 1.1 modelada a eventos, que toma una estrategia mucho mas clara, orientada a objetos, junto con el agregado de JavaBeans, un modelo de componente de programación que es orientado a la fácil creación de ambientes de programación visuales. Java 2 termina la transformación fuera de la vieja AWT de Java 1.0 esencialmente reemplazando todo con la *Java Fundation Classes*(JFC), la porción de la GUI que es llamada “Swing”. Esta es un rico conjunto de JavaBeans fáciles de utilizar y fáciles de entender) que pueden ser arrastrados y tirados (de la misma forma que programando con la mano) para crear una GUI que se puede (finalmente) satisfacer. La regla “revisión 3” de la industria de software (un producto no es bueno hasta la revisión 3) parece mantenerse verdadera con los lenguajes de programación.

---

<sup>1</sup> Una variación de esto es llamado “el principio del mínimo asombro”, que esencialmente dice: “no sorprenda al usuario”.

Este capítulo no cubre nada mas que lo moderno, la librería Swing de Java 2, y razonablemente asume que Swing es el destino final de la librería GUI para Java. Si por alguna razón se necesita utilizar la original “vieja” AWT (porque esta modificando viejo código o se tienen limitaciones en el navegador), se puede encontrar la introducción en la primera edición de este libro, que se puede bajar desde [www.BruceEckel.com](http://www.BruceEckel.com)(también incluida en CD ROM que viene con este libro).

Inmediatamente en este capítulo, se podrán ver como las cosas son diferentes cuando se quiere crear un applet contra una aplicación regular utilizando Swing, y como crear programas que son ambos, applets y aplicaciones así es que pueden ser ejecutadas dentro de un navegador o en la línea de comandos. Al menos todos los ejemplos de GUI en este libro son ejecutables como applets o aplicaciones.

Por favor, hay que ser cuidadoso con que esto no es un glosario para cada uno de los componentes Swing, o todos los métodos para las clases descritas. Lo que se verá aquí pretende ser simple. La librería Swing es enorme, y el objetivo de este capítulo es tener un comienzo con lo esencial y confortable con los conceptos. Si necesita mas, entonces Swing puede probablemente lo que quiere si esta gustoso para realizar la investigación.

Asumo aquí que se ha bajado e instalado la (libre) documentación de librería de Java en formato HTML de [java.sun.com](http://java.sun.com)y se ha navegado en las clases **javax.swing** en esa documentación para ver los detalles completos y los métodos de la librería Swing. Dado la simplicidad del diseño de Swing, esto a menudo puede ser suficiente información para solucionar los problemas. Hay muchos (mas bien muchos) libros dedicados solamente a Swing y se debería ir en su búsqueda si necesita mas profundidad, o si se quiere modificar el comportamiento por defecto de Swing.

A medida que se aprenda acerca de Swing se descubrirá:

1. Swing es un modelo mucho mejor de programación que el que probablemente haya visto en otros lenguajes y ambientes de desarrollo. JavaBeans (que serán introducidos al final de este capítulo) que es el marco de trabajo para esta librería.
2. “Creadores de GUI” (ambientes visuales de programación) son un aspecto *de rigueur* de un ambiente de desarrollo Java. JavaBeans y Swing permiten al creador GUI escribir código colocando componentes en hojas utilizando herramientas gráficas. Esto no solo acelera el desarrollo GUI, también permite una mayor experimentación y de esta forma la habilidad de tratar mas diseños y presumiblemente terminar con uno mejor.
3. La simplicidad y la naturaleza bien diseñada de Swing da a entender que aún si se usa un creador GUI en lugar de codificar a mano, el

código resultante seguirá siendo comprensible -esto soluciona un problema grande con los creadores GUI del pasado, que fácilmente podían generar fácilmente código imposible de leer.

Swing contiene todos los componentes que se espera ver en una UI moderna, todos desde botones que contienen imágenes hasta árboles y tablas. Es una librería grande, pero esta diseñada para tener una apropiada complejidad para la tarea a mano -si algo es simple, no se tiene que escribir mucho código pero a medida que se trata de hacer las cosas mas complejas, el código se vuelve proporcionalmente mas complejo. Esto significa que se tiene un punto de entrada fácil, pero se tendrá el poder si se necesita.

Mucho de lo que gustará acerca de Swing puede ser llamado “ortogonalidad de uso”. Esto es, una vez que se tiene una idea general acerca de la librería se puede aplicar en cualquier parte. Primariamente por las convenciones de nombres, mucho del tiempo que he escrito estos ejemplos he podido adivinar en los nombres de métodos y obtener lo correcto de primera, sin mirar nada mas. Esto es ciertamente es el sello de un buen diseño de librería. Además, se puede generalmente colocar componentes dentro de otros componentes y las cosas funcionarán correctamente.

Por un tema de velocidad, todos los componentes son “de peso liviano”, y Swing esta escrita totalmente en Java por un tema de portabilidad.

La navegación de teclado en automática -se puede ejecutar una aplicación Swing sin utilizar el ratón, y no requiere ninguna programación extra. El soporte para desplazamientos se logra sin ningún esfuerzo -simplemente se envuelve los componentes en un **JScrollPane** cuando se agrega a la hoja. Las características al igual que las agregados en herramientas típicamente requieren una sola línea de código para utilizarse.

Swing soporta también una característica mas bien radical llamada “pluggable look and feel”, que significa que la apariencia de la UI puede ser dinámicamente cambiada para satisfacer las expectativas de los usuarios que trabajan bajo diferentes plataformas y sistemas operativos. Es posible aún (a pesar de que es difícil) diseñar un look and feel propio.

## El applet básico

Una de las metas de diseño de Java es crear *applets*, que son pequeños programas que se ejecutan dentro de un navegador Web. Dado que estos deben ser seguros, los applets son limitados en lo que pueden lograr. Sin embargo, los applets son poderosas herramientas que soportan programación del lado del cliente, un tema mayor para la Web.

## Restricciones del applet

Programando dentro de un applet es tan restrictivo que a menudo es referido como estar “dentro del corral”, dado que siempre hay algo -esto es, el sistema de seguridad en tiempo de ejecución de Java- cuidándolo.

Sin embargo, se puede saltar sobre el corralito y escribir aplicaciones regulares en lugar de estos applets, en cuyo caso se puede acceder a las otras características de su OS. A lo largo de todo este libro hemos escrito aplicaciones comunes, pero estas han sido *aplicaciones de consola* sin ningún componente gráfico. Swing puede ser utilizado para crear interfaces GUI para aplicaciones regulares.

Generalmente se puede contestar la pregunta de que es capaz de hacer un applet observando que se *supone* que hace: extender la funcionalidad de una página web en un navegador. Dado que, como navegador en la Red, nunca se sabe cuando una página forma parte de un lugar amigable o no, se quiere que cualquier código que se ejecuta sea seguro. Así es que las grandes restricciones que se notarán son probablemente:

1. *Un applet no puede tocar el disco local* Esto significa escritura o lectura, dado que no quiere que un applet lea y transmita información privada a través de la Internet sin su permiso. La escritura previene, claro, dado que puede ser una invitación abierta para un virus. Java ofrece *firmas digitales* para applets. Muchas restricciones en los applets son sosegadas cuando se elige permitir *applets confiables* (aquejlos firmados por una fuente confiable) para tener acceso a su máquina.
2. *Los applets no pueden tardar mucho en desplegarse* dado que se debe bajar la totalidad de las cosas cada vez, incluyendo un servidor separado para cada clase diferente. Su navegador puede almacenar el applet en memoria caché, pero no hay garantías. Por esto, se debe siempre empaquetar el applet en un fichero JAR (Java ARchive) que combina todos los componentes del applet (incluyendo otros ficheros .class así como imágenes y sonidos) juntos en un solo fichero comprimido que puede bajarse en una sola transacción del servidor. “Las firmas digitales” estás disponibles para cada entrada individual en el fichero JAR.

## Ventajas de los applets

Si puede vivir dentro de las restricciones, los applets tienen ventajas definitivas especialmente cuando se construyen aplicaciones cliente/servidor u otro tipo de aplicaciones para redes:

1. *No hay temas de instalación* Un applet tiene una verdadera independencia de la plataforma (incluyendo la habilidad de fácilmente ejecutar ficheros de audio, etc) así es que no se necesita hacer ningún tipo de cambio en el código para las diferentes plataformas ni tampoco nadie tiene que hacer ninguna treta en la instalación. De hecho, la instalación es automática cada vez que el usuario carga la página Web que contiene applets, así es que la actualización se sucede silenciosamente y automáticamente. En un sistema tradicional cliente/servidor, la creación e instalación de una nueva versión del software del cliente es siempre una pesadilla.
2. *No hay que preocuparse acerca de algún código de mala calidad cause daño a alguien en el sistema* por la seguridad creada dentro del corazón del lenguaje Java y de la estructura del applet. Esto, junto con los puntos anteriores, hace que Java tan popular para las llamadas aplicaciones intranet cliente/servidor que viven solo dentro de una compañía o en una arena de operación restringida donde el ambiente de usuario (Navegador Web y agregados) puedan ser especificados y/o controlados.

Dado que los applets son automáticamente integrados con HTML, se tiene un sistema de documentación incluido independiente de la plataforma para soportar el applet. Es un interesante giro, puesto que estamos acostumbrados a tener la parte de documentación del programa en lugar de al revés.

#### Marcos de trabajo de aplicación

Las librerías son a menudo agrupadas de acuerdo con sus funcionalidades. Algunas librerías, por ejemplo, son utilizadas tal como están, sacado del estante. Las clases de la librería estándar de Java **String** y **ArrayList** son ejemplos de estas. Otras librerías están diseñadas específicamente como bloques de armado para crear otras clases. Una categoría inconfundible de estas librerías es el *marco de trabajo de aplicación (application framework)* cuya meta es ayudarlo a crear aplicaciones proporcionando una clase o grupo de clases que producen el comportamiento básico que se necesita en cada aplicación de un tipo particular. Entonces, para adaptar el comportamiento a necesidades particulares, se hereda de la clase aplicación y se sobrescriben los métodos de interés. El mecanismo de control por defecto de los marcos de trabajo de aplicación llamarán a los métodos sobrescritos en el momento apropiado. Un marco de trabajo de aplicación es un buen ejemplo de “separación de cosas que cambian de las cosas que se quedan igual”, dado que intenta localizar todas las partes exclusivas de un programa de los métodos sobrescritos<sup>2</sup>.

---

<sup>2</sup> Este es un ejemplo de patrones de diseño llamado *método de plantilla*.

Los applets son construidos utilizando un marco de trabajo de aplicación. Se hereda de la clase **JApplet** y se sobrescriben los métodos apropiados. Hay algunos pocos métodos que controlan la creación y ejecución de un applet en una página Web:

Método	Operación
<b>init()</b>	Automáticamente llamado para realizar la inicialización del inicio del applet, incluyendo el diseño de los componentes. Siempre se sobrescribirá este método.
<b>start()</b>	Llamado cada vez que el applet se mueve dentro de la vista del navegador Web para permitir que el applet comience su operación normal (especialmente aquellos que tienen que apagarse con <b>stop()</b> ). También llamada luego de <b>init()</b> .
<b>stop()</b>	Llamado cada vez que el applet se mueve fuera de la vista del navegador Web para permitir que el applet apague operaciones de alto consumo. También llamado inmediatamente antes de <b>destroy()</b> .
<b>destroy()</b>	Llamado cuando el applet esta siendo descargado de la página para realizar la liberación final de recursos cuando el applet no se va a utilizar mas.

Con esta información se está listo para crear un simple applet:

```
//: c13:Applet1.java
// Un applet muy simple.
import javax.swing.*;
import java.awt.*;
public class Applet1 extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
} ///:~
```

Debe notarse que los applets no requieren tener un **main()**. Esto es todo armado dentro del marco de trabajo de aplicación; se coloca todo el código de arranque en **init()**.

La única actividad en este programa es colocar una etiqueta de texto en el applet, mediante la clase **JLabel** (la vieja AWT destinó en nombre de **Label** de la misma forma que otros nombres de componentes, así es que a menudo se verá una “J” conduciendo utilizada con los componentes Swing). El constructor para esta clase toma un **String** y lo utiliza para crear la etiqueta. En el programa mas arriba esta etiqueta es colocada en la hoja.

El método **init()** es responsable por colocar todos los componentes en la hoja utilizando el método **add()**. Se puede pensar que se debe ser capas de simplemente llamar a **add()** por si mismo, y de hecho esta es la forma en que es utilizado en la vieja AWT. Sin embargo, Swing necesita que se agreguen todos los componentes a el “panel contenedor” de una hoja, y de esta forma de debe llamar a **getContentPane()** como parte del proceso de agregar que realiza **add()**.

## Ejecutando applets dentro de un navegador Web

Para ejecutar este programa se debe colocar dentro de una página Web y ver esta página dentro de un navegador Web con Java habilitado. Para colocar un applet dentro de una página Web se coloca una etiqueta especial dentro de el código HTML para esa página<sup>3</sup> para indicarle como cargar y ejecutar el applet.

Este proceso solía ser muy simple, cuando Java por si mismo era simple y todos usaban el mismo coche ganador e incorporaban el mismo soporte dentro de sus navegadores Web. Luego, se puede querer ser capaz de salir con una muy simple pequeña cantidad de HTML dentro de su página Web, como esta:

```
<applet code=Applet1 width=100 height=50>  
</applet>
```

Entonces a lo largo de la vendida de la guerra de navegadores y lenguaje, y nosotros (los programadores y los usuarios igualmente) perdemos. Luego de algún tiempo, JavaSoft se dio cuenta de que no podían esperar que los navegadores soportaran el sabor correcto de Java, y la única solución fue proporcionar algún tipo de mejoramiento de sus ejecuciones que puedan conformar el mecanismo de extensión de los navegadores. Utilizando el mecanismo de extensión (el que un proveedor de navegadores no puede deshabilitar -en un intento de ganar ventaja competitiva- sin romper todas las extensiones de terceros), JavaSoft garantiza que Java no puede negar la entrada del navegador por un proveedor antagónico.

Con Internet Explorer, el mecanismo de extensión es el control ActiveX, y con Netscape es el plug-in. En el código HTML, de debe proporcionar

---

<sup>3</sup> Esto es asumido que el lector esta familiarizado con lo básico de HTML. No es muy difícil entender, y hay muchos libros y recursos.

etiquetas para soportar ambos. He aquí como la simple página HTML resultante se ve para **Applet1**:<sup>4</sup>

```
//:! 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/ji
nstall-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-javaapplet;
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/p
lugin-install.html">
        <NOEMBED>
    </COMMENT>
    No Java 2 support for APPLET! !
    </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Alguna de estas líneas que son muy largas y tienen que ser plegadas para entrar en la página. El código fuente del libro (en el CD ROM, y se puede bajar desde [www.BruceEckel.com](http://www.BruceEckel.com)) trabajará sin tener que preocuparse por corregir las líneas plegadas.

El valor de **code** da el nombre del fichero **.class** donde el applet reside. El **width** y **height** especifica el tamaño inicial para el applet (en puntos, como antes). Hay otros ítems que se pueden colocar dentro de la etiqueta del applet: un lugar para encontrar las otras **.class** en la Internet (**codebase**), la información de alineación (**align**), un identificador especial que hace posible a los applets comunicarse con otros (**name**), y los parámetros del applet para proporcionar información que el applet puede recuperar. Los parámetros son de la forma:

| <param name="identifier" value = "information">  
| y aquí pueden ser tantos como se quieran.

El paquete de código fuente para este libro proporciona una página HTML para cada uno de los applets de este libro, y de esta manera muchos

---

<sup>4</sup> Esta página -en particular, la parte 'clsid'- parece trabajar bien con JDK1.2.2 y JDK 1.3 rc-1. Sin embargo, se puede encontrar que se tiene que cambiar la etiqueta en algún momento en el futuro. Los detalles pueden ser encontrados en [java.sun.com](http://java.sun.com).

ejemplos en la etiqueta del applet. Se puede encontrar una completa y actual descripción de los detalles de colocar applet en las páginas Web en [java.sun.com](http://java.sun.com)

## Utilizando el Appletviewer

JDK de Sun (se puede bajar libremente de [java.sun.com](http://java.sun.com)) contiene una herramienta llamada el *Appletviewer* que seleccionan las etiquetas <**applet**> del fichero HTML y ejecuta los applets sin despegar el texto HTML que las rodea. Dado que el Appletviewer ignora todo excepto las etiquetas APPLET, se puede colocar estas etiquetas en el fuente Java como comentarios:

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

De esta forma se puede ejecutar “**appletviewer MyApplet.java**” y no se necesita crear pequeñas ficheros HTML para ejecutar pruebas. Por ejemplo, se puede agregar las etiquetas HTML comentadas a **Applet1.java**:

```
//: c13:Applet1b.java
// Insertando la etiqueta para el applet para el 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 el applet con el comando

```
appletviewer Applet1b.java
```

En este libro, esta forma será utilizada para probar fácilmente los applets. En poco tiempo, se puede ver otra estrategia de codificación que permitirá ejecutar applet de la línea de comandos sin el Appletviewer.

## Probando applets

Se puede realizar una prueba simple sin ninguna conexión a redes ejecutando el navegador Web y abriendo el fichero HTML contenido la etiqueta applet. Mientras el fichero HTML es cargado, el navegador descubrirá la etiqueta applet y irá a capturar el fichero **.class** especificada por el valor de **code**. Claro, busca en la CLASSPATH para encontrar donde cazar, y si el fichero **.class** no está en la CLASSPATH entonces dará un mensaje de error en la línea de estado del navegador como resultado de que no puedo encontrar el fichero **.class**.

Cuando se quiera poner a prueba esto en un sitio Web las cosas son un poco más complicadas. Antes que nada, se debe tener un sitio Web, que para

muchas personas significan un tercero que provea de servicios de Internet (ISP) en una ubicación remota. Dado que el applet es simplemente un fichero o un grupo de ficheros, los ISP no tienen que proporcionar ningún soporte especial para Java. Se debe también tener una forma de mover los ficheros HTML y los ficheros **.class** de su sitio a el directorio correcto en la máquina ISP. Esto es realizado típicamente con un programa FTP (File Transfer Protocol), de los cuales hay muchos tipos diferentes disponibles gratis o como shareware. ¿Así es que parece que todo lo que se necesita hacer es mover los ficheros a la máquina del ISP con FTP, luego conectar a el sitio y a el fichero HTML utilizando su navegador; si el applet aparece y trabaja, entonces todo esta revisado, correcto?

Aquí es donde se puede ser engañado. Si el navegador en la máquina cliente no puede encontrar el fichero **.class** en el servidor, esta buscará a través de la CLASSPATH en su máquina *local*. De esta forma, el applet puede no ser cargado propiamente del servidor, pero para nosotros se ve bien durante el proceso de prueba porque el navegador lo encuentra en nuestra máquina. Cuando alguien mas se conecta, sin embargo, el navegador no puede encontrarlo. Así es que cuando se esta probando, hay que asegurarse de borrar los ficheros **.class** significantes (o el fichero **.jar**) en su máquina local para verificar que existan en la localización apropiada en el servidor.

Uno de los lugares mas insidiosos donde esto me sucede a mi es cuando inocentemente coloco un applet dentro de un paquete. Luego de subir el fichero HTML y el applet, descartamos que la ruta del servidor a el applet fue confusa a causa del nombre del paquete. Sin embargo, mi navegador lo ha encontrado en la CLASSPATH local. Así es que el único que puede cargar adecuadamente el applet. Toma algún tiempo descubrir que la instrucción **package** fue la culpable. En general, se debe querer dejar la instrucción **package** fuera de un applet.

## Ejecutando applets de la línea de comandos

Hay momentos cuando se quiere crear un programa con ventanas que haga algo mas que ubicarse en la pagina Web. Tal vez también quiera que haga alguna de las cosas que una aplicación “regular” puede hacer pero seguir jactándose de la portabilidad proporcionada por Java. En los capítulos previos en este libro hemos hecho aplicaciones de línea de comandos, pero en algunos ambiente operativos (como Macintosh, por ejemplo) no hay línea de comandos. Así es que por algunas razones de puede querer crear un programa sin ventanas, que no sea un applet utilizando Java. Esto es ciertamente un deseo razonable.

La librería Swing permite hacer una aplicación que preserve la apariencia del sistema operativo que está en capas mas bajas. Si se quiere crear aplicaciones con ventanas, tiene sentido hacerlo<sup>5</sup> solo si se puede utilizar la última versión de Java y herramientas asociadas así se puede distribuir aplicaciones que no confundan a sus usuarios. Si por alguna razón se está forzado a utilizar una versión vieja de Java, hay que pensar mucho antes de comprometerse a crear una aplicación significante con ventanas.

A menudo se quiere ser capaz de crear una clase que pueda ser invocada como una ventana o como un applet. Esto es especialmente conveniente cuando se están probando los applets, dado que típicamente es mucho más rápido y fácil ejecutar la aplicación resultante de la línea de comandos que ejecutando un navegador Web o el Appletviewer.

Para crear un applet que pueda ser ejecutado de la línea de comandos de la consola, se puede simplemente agregar el **main()** a su applet que crea una instancia del applet dentro de un **JFrame**<sup>6</sup>. Como ejemplo simple, veamos **Applet1b.java** modificado para trabajar como aplicación y como applet;

```
//: c13:Applet1c.java
// Una aplicación y un applet.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.swing.*;
public class Applet1c extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }
    // Un main() para la aplicación:
    public static void main(String[] args) {
        JApplet applet = new Applet1c();
        JFrame frame = new JFrame("Applet1c");
        // Para cerrar la aplicación:
        Console.setupClosing(frame);
        frame.getContentPane().add(applet);
        frame.setSize(100,50);
        applet.init();
        applet.start();
        frame.setVisible(true);
    }
} ///:~
```

**main()** es el único elemento agregado al applet, y el resto del applet está sin tocar. El applet es creado y se agrega un **JFrame** así es que puede ser desplegado.

---

<sup>5</sup> En mi opinión. Y luego de que se ha aprendido acerca de Swing, no se quiere gastar el tiempo en las cosas anteriores.

<sup>6</sup> Como se dijo anteriormente, “Frame” ya fue utilizado por la AWT, así es que Swing utiliza JFrame.

La línea:

```
| Console.setupClosing(frame);
```

Produce que la ventana sea adecuadamente cerrada. **Console** viene de **com.bruceeckel.swing** y será explicado un poco mas tarde.

Como se puede ver en el **main()**, el applet es explícitamente inicializado y ejecutado dado que en este caso no esta el navegador disponible para hacerlo por usted. Claro, esto no proporciona el comportamiento completo del navegador, que también llama a **stop()** y a **destroy()**, pero para la mayoría de las situaciones es aceptable. Si es un problema, se puede forzar las llamadas<sup>7</sup>.

Note la última línea:

```
| frame.setVisible(true);
```

Sin esto, no se podrá ver nada en la pantalla.

## Un marco de trabajo de visualización

A pesar de que el código que convierte programas en applets y aplicaciones produce resultados de valor, si es utilizado en todos lados se convierte en molesto y en papel usado. En lugar de eso, el siguiente marco de trabajo de visualización será utilizado para los ejemplos Swing en el resto de este libro:

```
//: com:bruceeckel:swing:Console.java
// Herramienta para ejecutar demostraciones Swing
// desde la consola, applets y JFrames.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;
public class Console {
    // Crea una cadena de titulo para el nombre de clase:
    public static String title(Object o) {
        String t = o.getClass().toString();
        // Quita la palabra "class":
        if(t.indexOf("class") != -1)
            t = t.substring(6);
        return t;
    }
    public static void setupClosing(JFrame frame) {
        // La solución de JDK 1.2 es una
        // clase anónima interna:
        frame.addWindowListener(new WindowAdapter() {
```

---

<sup>7</sup> Esto tiendrá sentido luego de que se haya leído mas adelante en este capítulo. Primero, haga la referencia **JApplet** un miembro **static** de la clase (en lugar de una variable local del **main()**), y luego se llama a **applet.stop()** y a **applet.destroy()** dentro de **WindowAdapter.windowClosing()** luego que se llame a **System.exit()**.

```

        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    // La solucion mejorada en 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 es una herramienta que puede querer utilizarse para si mismo, así es que es colocada en la librería **com.bruceeckel.swing**. La clase **Console** consiste enteramente en métodos **static**. El primero es utilizado para extraer el nombre de la clase (usando RTTI) de cualquier objeto y quitando la palabra “class”, que es típicamente anexada por **getClass()**. Esto utiliza el método **String indexOf()** para determinar si la palabra “class” esta ahí, y **substring()** para producir la nueva cadena sin “class” o los espacios. Este nombre es utilizado para etiquetar la ventana que será desplegada por el método **run()**.

**setupClosing()** es utilizada para ocultar el código que causa que **JFrame** salga del programa cuando **JFrame** es cerrada. El comportamiento por defecto es no hacer nada, así es que si no llama a **setupClosing()** o se escribe el código equivalente para su **JFrame**, la aplicación no cerrará. La razón para este código es ocultarlo en lugar de colocarlo directamente en los métodos **run()** subsecuentes en parte porque permite la utilización del método por si mismo cuando lo que se quiere hacer es mas complicado de lo que **run()** proporciona. Sin embargo, también aísla el factor cambio: Java 2

tiene dos formas de lograr que ciertos tipos de ventanas se cierren. En JDK 1.2, la solución es crear una clase **WindowsAdapter** nueva e implementar **windowsClosing()**, como se ha visto mas arriba (el significado de esto será explicado totalmente en este capítulo). Sin embargo, durante la creación de JDK 1.3 los diseñadores de la librería observaron que normalmente no se necesita cerrar la ventana cuando quiera que se este creando algo que no sea un applet, y de esta forma se agregan las **setDefaultCloseOperation()** para **JFrame** y **JDialog**. Desde este punto de vista de escritura de código, el nuevo método es mucho mas agradable de utilizar, pero este libro fue escrito cuando no estaba JDK 1.3 implementado en Linux y en otras plataformas, así es que por interés de portabilidad en plataformas cruzadas el cambio fue aislado adentro de **setupClosing()**.

Para el método **run()** es sobrescrito para trabajar con **JApplets**, **JPanels**, y **JFrames**. Debe notarse que solo si es un **JApplet** tiene llamadas a **init()** y **start()**.

Ahora cualquier applet puede ser ejecutado de la consola creando un **main()** que contenga una línea como esta:

```
| Console.run(new MyClass(), 500, 300);  
en donde los dos últimos argumentos son el ancho y el largo cuando se  
despliegue en pantalla. He aquí el Applet1d.java modificado para utilizar  
Console:
```

```
///: c13:Applet1d.java  
// Console ejecuta applets de la línea de comandos.  
// <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 mientras se proporciona la enorme flexibilidad ejecutando los ejemplos.

## Utilizando el Explorador de Windows

Si está utilizando Windows, se puede simplificar el proceso de ejecutar un programa Java en la línea de comandos configurando el Explorador de Windows -el navegador en Windows, *no* el Internet Explorer- así es que se

puede hacer simplemente un doble clic en un fichero **.class** para ejecutarlo. Hay varios pasos en el proceso.

Primero, baje e instale el lenguaje Perl de [www.Perl.org](http://www.Perl.org) Encontrará las instrucciones y documentación del lenguaje en este sitio.

Luego, cree el siguiente script sin la primera y última línea (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';
_____
:endofperl
///:~
```

Ahora, abra el Explorador de Windows, seleccione “Ver”, “Opciones de carpeta”, luego haga un clic en la lengüeta “Tipos de ficheros”. Presione el botón de “Nuevo tipo”. En la “Descripción de tipo” entre “Fichero de tipo Java”. En la “Extensión asociada” entre “class”. En “Acciones” presione el botón “Nuevo”. Para “Acción” entre “Abrir”, y para “Aplicación utilizada para realizar la acción” entre una línea como esta:

```
| "C:\aaa\Perl\RunJava.bat" "%L"
```

Se debe adaptar la ruta antes de “RunJava.bat” para adaptarse a la localización donde se coloco el fichero .bat.

Una vez que se haya realizado esta instalación, de puede ejecutar cualquier programa Java haciendo simplemente un doble clic en el fichero **.class** contenido en el **main()**.

## Creando un botón

Crear un botón es bastante simple: solo se llama a el constructor de la clase **JButton** con la etiqueta que se quiere en el botón. Se verá mas adelante que se pueden colocar adornos, como colocar imágenes gráficas en los botones.

Usualmente se querrá crear un campo para el botón dentro de la clase así es que se puede hacer referencia a el mas tarde.

El **JButton** es un componente -su propia pequeña ventana- que automáticamente se pintará como parte de una actualización. Esto significa que si no se pinta explícitamente el botón o cualquier otro tipo de control;

simplemente se coloca en la hoja y se deje que automáticamente se encargue de pintarse ellos mismos. Así es que para colocar un botón en una hoja, se tiene que hacer dentro del **init()**:

```
//: c13:Button1.java
// Colocando botones dentro de un applet.
// <applet code=Button1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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);
    }
} ///:~
```

Algo nuevo ha sido agregado aquí: antes que cualquier elemento haya sido agregado en el cuadro de contenido, se le ha asignado un nuevo “manejador de trazado”, del tipo **flowLayout**. El manejador de trazado es la forma en que el cuadro implícitamente decide donde se colocará en control en la hoja. El comportamiento normal de un applet es utilizar el **BorderLayout**, pero esto no trabajará aquí porque (como se aprenderá mas tarde en este capítulo cuando el control de trazado de una hoja sea examinado con mas detalle) por defecto se cubre cada entidad de control con cada nueva que es agregada. Sin embargo, **FlowLayout** produce que los controles fluyan equitativamente en la hoja, de izquierda a derecha y de arriba a abajo.

## Capturando un evento

Se notará que si se compila y se ejecuta el applet de mas arriba, nada sucederá cuando se presione los botones. Esto es donde se debe intervenir y escribir algún código para determinar que sucederá. Lo básico de la programación de eventos conducidos, que comprende un montón de que una GUI es, es atar los eventos para codificar lo que se responde a esos eventos.

La forma en que se logra esto es realizada en Swing es separando la interfase (los componentes gráficos) y la implementación (el código que se quiere ejecutar cuando cada evento sucede en un componente). Cada componente

Swing puede reportar todos los eventos que pueden sucederle, y este puede reportar cada tipo de evento individualmente. Así es que si no se está interesado en, por ejemplo, cuando el ratón es movido encima del botón, no se registra el interés en ese evento. Esta es una forma muy directa y elegante de manejar programación de conducción de eventos, y una vez que se entiendan los conceptos básicos se puede utilizar fácilmente los componentes Swing que no se han visto antes -de hecho, este modelo se extiende a todo lo que pueda ser clasificado como un JavaBean (lo que se aprenderá más tarde en este capítulo).

Al inicio, nos enfocaremos solamente en el evento principal de interés para los componentes que serán utilizados. En este caso de un **JButton**, este “evento de interés” es que el botón es presionado. Para registrar el interés en cuando el botón es presionado, se puede llamar a el método **addActionListener()** de **JButton**. Este método espera un argumento que es un objeto que implementa la interfase **ActionListener**, que contiene un solo método llamado **actionPerformed()**. Así es que se tiene que enganchar el código a un **JButton** que es implementado a la interfase **ActionListener** en una clase, y registrar un objeto de esa clase con el **JButton** mediante **addActionListener()**. El método será llamado cuando el botón sea presionado (esto es normalmente referido como una *callback*).

¿Pero cual debería ser el resultado de presionar ese botón? Nos hubiera gustado ver algo que cambiara en la pantalla, así es que un nuevo componente Swing será introducido: el **JTextField**. Esto es un lugar donde el texto puede ser escrito, o en este caso modificado por el programa. A pesar de que hay una cantidad de formas de crear un **JTextField**, la mas simple es solo indicarle al constructor de que tamaño se quiere que sea este campo. Una vez que el **JTextField** es colocado en la hoja, se puede modificar su contenido utilizando el método **setText()** (hay muchos otros métodos en **JTextField**, pero se deberían buscar en la documentación HTML para la JDK en [java.sun.com](http://java.sun.com)). Aquí está como se vería:

```
//: 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.bruceekel.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 **JTextField** y colocarlo en el lienzo toma los mismos pasos que para **JButtons**, o para cualquier componente Swing. La diferencia entre el programa anterior es en la creación de las anteriormente dichas clases **ActionListener** **BL**. El argumento para **actionPerformed()** es un tipo de **ActionEvent**, que contiene toda la información acerca del evento y como llega a la hoja. En este caso, queremos describir el botón que fue presionado: **getSource()** produce el objeto donde el evento fue originado, y asumo que en un **JButton**.**getText()** retorna el texto que está en el botón, y este es colocado en el **JTextField** para probar que el código es actualmente llamado cuando el botón es presionado.

En el **init()**, **addActionListener()** es utilizado para registrar el objeto **BL** con los botones.

Este es a menudo mas conveniente para codificar el **ActionListener** es una clase anónima interna, especialmente dado que se tiende a utilizar solo una instancia de cada clase listener. **Button2.java** puede ser modificado para utilizar una clase anónima interna 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);
}
} //:~

```

La estrategia de utilizar una clase interna será preferida (cuando sea posible) en los ejemplos de este libro.

### Áreas de texto

Una **JTextArea** es como un **JTextField** excepto que puede tener muchas líneas y mas funcionalidad. Un método particularmente útil es **append()**; con este se puede fácilmente verter la salida dentro de un **JTextArea**, de esta forma crear un programa Swing con una mejora (dado que se puede desplazar para atrás) sobre lo que se ha logrado utilizando programas de línea de comandos que imprimen en la salida estándar. Como ejemplo, el siguiente programa llena una **JTextArea** con la salida del generador geográfico del Capítulo 9:

```

//: c13:TextArea.java
// Utilizando el control JTextArea.
// <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("Aregar datos"),
        c = new JButton("Borrar datos");
    JTextArea t = new JTextArea(20, 40);
    Map m = new HashMap();
    public void init() {
        // Use todos los datos:
        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 **init()**, el **Map** es llenado con todos los países y sus capitales. Debe notarse que para ambos botones el **ActionListener** es creado y agregado sin definir una variable intermedia, dado que nunca se necesitará referirse a ese oyente nuevamente durante el programa. El botón “Aregar datos” genera y agrega todos los datos, mientras que el botón “Borrar datos” utiliza **setText()** para quitar todo el texto del **JTextArea**.

Así como el **JTextArea** es agregado al applet, este es envuelto en un **JScrollPane**, para controlar el desplazamiento cuando mucho texto sea colocado en la pantalla. Esto es todo lo que se tiene que hacer para producir capacidades completas de desplazamiento. Considerando que se ha probado a resolver como hacer lo equivalente en otros ambiente de programación GUI, estoy muy impresionado con la simplicidad y el buen diseño de los componentes como **JScrollPane**.

## Controlando el trazado

La forma en que se colocan los componentes en una hoja de Java es probablemente diferente de otro sistema GUI que hayamos utilizado. Primero, es todo código; no hay “recursos” que controlan la posición de los componentes. Segundo, la forma en que los componentes son colocados en la hoja es controlado no por posicionamiento absoluto sino que es realizado por un “manejador de trazado” que decide como los componentes se sitúan basado en el orden en que se agregan. El tamaño, forma y colocación de los componentes será notablemente diferente de un manejador de trazado a

otro. Además, los manejadores de trazado adaptan a las dimensiones de su applet o ventana de aplicación, así es que si el tamaño de la ventana es cambiado, el tamaño, forma y colocación de los componentes pueden cambiar en respuesta.

**JApplet**, **JFrame**, **JWindows** y **JDialog** pueden todos producir un contenedor (**Container**) mediante **getContentPane()** (tomar cuadro de contenido) que puede contener y mostrar componentes (**Component**). En cada contenedor, hay un método llamado **setLayout()** que permite elegir distintos manejadores de trazado. Otras clases, como **JPanel**, contienen y despliegan componentes directamente y de esta forma se puede también configurar el manejador de trazado directamente, sin utilizar el cuadro de contenido (content pane).

En esta sección exploraremos los distintos manejadores de trazado colocando botones en ellos (dado que es lo mas simple para hacer). No queremos ninguna captura de eventos de botón dado que estos ejemplos solo intentan mostrar como los botones son colocados.

## BorderLayout

El applet utiliza un esquema de trazado por defecto: el **BorderLayout** (un montón de ejemplos previos han cambiado el manejador de trazado a **FlowLayout**). Sin otra instrucción, este toma lo que sea que se agregue y lo coloca en el centro, estirando el objeto hasta los bordes.

Sin embargo, hay mas para el **BorderLayout**. Este manejador de trazado tiene el concepto de cuatro regiones en los bordes y un área central. Cuando se agrega algo a el panel que esta utilizando un **BorderLayout** se puede utilizar el método sobrecargado **add()** que toma valores constantes como el primer argumento. Este valor puede ser cualquier de los siguientes:

**BorderLayout. SOUTH** (abajo)

**BorderLayout. EAST** (derecha)

**BorderLayout. NORTH** (arriba)

**BorderLayout. WEST** (izquierda)

**BorderLayout.CENTER** (rellena el medio, arriba de los otros componentes o hasta los bordes)

Si no se especifica un área para colocar el objeto, por defecto es el centro.

Aquí hay un ejemplo simple. El trazado por defecto es utilizado, dado que por defecto en **JApplet** es **BorderLayout**:

```
//: c13:BorderLayout1.java
// Demuestra BorderLayout.
// <applet code=BorderLayout1
```

```

// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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 todos las ubicaciones excepto **CENTER**, el elemento que se esta agregando queda comprimido para llenar la menor cantidad de espacio a lo largo de una dimensión mientras que es estirado para el máximo largo en la otra dimensión. **CENTER**, sin embargo, se extiende en ambas direcciones para ocupar el medio.

## FlowLayout

Esto simplemente hacer “fluir” los componentes en la hoja, de la izquierda a la derecha hasta que el espacio superior esta lleno, luego se mueve abajo una columna y continúa fluyendo.

Aquí hay un ejemplo que configura el manejador de trazado a **FlowLayout** y luego coloca botones en la hoja. Se notara que con **FlowLayout** los componentes toman su tamaño “natural”. Un **JButton**, por ejemplo, será del tamaño de su cadena.

```

//: c13:FlowLayout1.java
// Demuestra FlowLayout.
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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);
    }
} //:~

```

Todos los componentes son compactados a su tamaño más pequeño en un **FlowLayout**, así es que puede tener un pequeño comportamiento sorprendente. Por ejemplo, dado que un **JLabel** será del tamaño de su cadena, el intentar justificar a la derecha su texto no produce nada cuando se utiliza **FlowLayout**.

## GridLayout

Un **GridLayout** permite crear una tabla de componentes, y a medida que se van agregando son colocados de izquierda a derecha y de arriba a abajo en la grilla. En el constructor se especifica el número de filas y columnas que se necesitan y estos son colocados en iguales proporciones.

```

//: c13:GridLayout1.java
// Demonstrates GridLayout.
// <applet code=GridLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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 posiciones pero solo 20 botones. La última posición es dejada vacía porque no se realiza un “balance” con una **GridLayout**.

## GridBagLayout

La **GridBagLayout** provee un tremendo control en decidir como las regiones de su ventana serán situadas y se modificarán cuando sea cambiado el tamaño de la ventana. Sin embargo, es también el manejador de trazado mas complicado, y bastante difícil de entender. Pretende ser primariamente para generación automática de código de un creador de GUI (los buenos creadores de GUI utilizan **GridBagLayout** en lugar de ubicaciones absolutas). Si el diseño es tan complicado que se siente la necesidad de utilizar **GridBagLayout**, entonces se debería utilizar una herramienta

creadora de GUI para generar el diseño. Si siente que debe conocer los intrincados detalles, deberemos referirnos a *Core Java 2* by Horstmann & Cornell (Prentice-Hall, 1999), o un libro dedicado a Swing, como punto de partida.

## Posicionamiento absoluto

Es posible configurar la posición absoluta de un componente gráfico de esta forma:

1. Configure un manejador de trazado **null** para su contenedor: **setLayout(null)**.
2. Llame a **setBounds()** o a **reshape()** (dependiendo de la versión del lenguaje) para cada componente, pasando un rectángulo limitado en coordenadas de pixel. Se puede hacer esto en el constructor, o en el **paint()**, dependiendo de lo que se quiera alcanzar.

Algunos creadores de GUI utilizan esta estrategia extensivamente, pero no es usualmente la mejor forma de generar código. Creadores de GUI más útiles utilizarán **GridBagLayout** en lugar de esto.

## BoxLayout

Dado que las personas tienen muchos problemas entendiendo como trabajando con **GridBagLayout**, Swing también incluye el **BoxLayout**, con el que se obtienen muchos beneficios de **GridBagLayout** sin la complejidad, así es que a menudo puede utilizarse esta cuando se necesita hacer trazados codificados a mano (nuevamente, si el diseño se vuelve muy complejo, se debería utilizar un creador de GUI que genere los **GridBagLayouts** por nosotros). **BoxLayout** permite controlar la ubicación de los componentes verticalmente y horizontalmente, y controlar el espacio entre los componentes utilizando algo llamado “puntales y goma”. Primero, veamos como utilizar el **BoxLayout** directamente, en la misma forma que los otros manejadores de trazados han sido demostrados:

```
//: c13:BoxLayout1.java
// BoxLayouts Verticales y horizontales.
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeeckel.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, jpvt);
    cp.add(BorderLayout.SOUTH, jph);
}
public static void main(String[] args) {
    Console.run(new BoxLayout1(), 450, 200);
}
} //:~

```

El constructor para **BoxLayout** es un poco diferente que otros manejadores de trazado - se proporciona el contenedor que será controlado por el **BoxLayout** como primer argumento, y la dirección de trazado como el segundo argumento.

Para simplificar el asunto, hay un contenedor especial llamado **Box** que utiliza **BoxLayout** como manejador nativo. El siguiente ejemplo sitúa componentes horizontales y verticales utilizando **Box**, que tiene dos métodos estáticos para crear cajas con alineación vertical y horizontal:

```

//: c13:Box1.java
// BoxLayouts verticales y horizontales.
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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 esta como un segundo argumento cuando se agregan componentes al marco de contenido.

Los puntales agregan espacio entre los componentes, medidos en pixels. Para utilizar un puntal, simplemente se agregan puntales entre los componentes que se quieren separar en el momento en que se agregan.

```
//: c13:Box2.java
// Agregando puntales.
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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 componentes en una cantidad conocida, pero la goma es lo opuesto: esta separa componentes todo lo que sea posible. De esta manera es mas una “elasticidad” que “goma” (y el diseño en que esto esta basado fue llamado “elásticos y puntales” así es que la elección del término es un poco misteriosa).

```
//: c13:Box3.java
// Usando goma.
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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 trabaja en una dirección, pero un área rígida fija el espacio entre los componentes en ambas direcciones:

```

//: c13:Box4.java
// Las áreas rígidas son como pares de puntales.
// <applet code=Box4
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceekel.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);
    }
} //:~

```

Se debe ser consciente que las áreas rígidas son un poco controversiales. Dado que ellas utilizan valores absolutos, algunas personas sienten que ellas causan mas problemas que lo que valen.

## ¿La mejor estrategia?

Swing es poderoso: puede tener mucho terminado con unas pocas líneas de código. El ejemplo mostrado en este libro es razonablemente simple, y con propósito de aprendizaje tiene sentido escribirlas a mano. Se puede actualmente lograr bastante combinando trazados simples. En algún punto,

sin embargo, deja de tener sentido codificar a mano hojas GUI -comienza a complicarse demasiado y no es bueno desperdiciar el tiempo de programación. Los diseñadores de Java y Swing orientaron el lenguaje y las librerías para soportar herramientas de creación de GUI, que han sido creadas para el propósito expreso de hacer la experiencia de programación fácil. Siempre y cuando se entienda que se está haciendo con trazados y como tratar con los eventos (descritos mas adelante), no es particularmente importante que se conozcan actualmente los detalles de como colocar componentes a mano -permítanos que la herramienta adecuada haga esto por nosotros (Java es, después de todo, diseñado para incrementar la productividad del programador).

## El modelo de eventos de Swing

En el modelo de eventos de Swing un componente puede ser iniciar (“disparar”) un evento. Cada tipo de evento es representado por una clase distinta. Cuando un evento es disparado, es recibido por uno o más “listener” (el que escucha), que actúan conectado con ese evento. De esta forma, la fuente de un evento y el lugar donde el evento es manejado puede ser separado. Puesto que típicamente se utilizan los componentes Swing como son, se necesita escribir código que es llamado cuando los componentes reciben un evento, esto es un excelente ejemplo de separación de interfase e implementación.

Cada evento escuchado es un objeto de una clase que implementa un tipo en particular de interfase listener. Así es que como programador, todo lo que tiene que hacer es crear un objeto listener y registrarlo con el componente que dispara el evento. El registro es realizado llamando a un método **addXXXListener()** en el componente que dispara el evento, en donde “**XXX**” representa el tipo de evento que se escucha. Se puede fácilmente saber que tipos de eventos pueden ser manejados observando los nombres de los método “addListener”, y si se trata de escuchar los eventos equivocados se notara el error en tiempo de compilación. Se vera mas tarde en el capítulo que JavaBeans también utiliza los nombre de los métodos “addListener” para determinar que eventos un Bean puede manejar.

Toda la lógica de eventos, entonces, irá dentro de una clase listener. Cuando se cree una clase listener, la única restricción es que debe implementar la interfase apropiada. Se puede crear una clase listener global, pero esta es una situación en donde las clases internas tienden a ser útiles, no solo porque proporcionan una agrupación lógica de sus clases listener dentro de la UI o de las clases lógicas de negocios que están sirviendo, pero dado que

(como se verá mas tarde) el echo de que una clase interna mantiene una referencia a su objeto padre proporciona una forma bonita de llamar a través de las clases y de los límites del subsistema.

Todos los ejemplos hasta ahora en este capítulo han sido utilizando el modelo de eventos de Swing, pero en el resto de esta sección completaremos los detalles de este modelo.

## Tipos de eventos y listeners

Todos los componentes Swing incluyen los métodos **addXXListener()** y **removeXXListener()** así es que los listeners adecuados pueden ser agregados y quitados de cada componentes. Se notará que el “**XXX**” en cada caso también representan el argumento para el método, por ejemplo: **addMyListener(MyListener m)**. La siguiente tabal incluye los eventos básicos asociados, listeners y métodos, junto con los componentes básicos que soportan esos eventos en particular proporcionando los métodos **addXXListener()** y **removeXXListener()**. Se debería tener en mente que el modelo de eventos esta diseñado para ser extensible, así es que se pueden encontrar otros tipos de eventos y listeners que no están cubiertos en esta tabla.

Evento, interfase listener y métodos add y remove	Componentes que soportan este evento
<b>ActionEvent</b> <b>ActionListener</b> <b>addActionListener()</b> <b>removeActionListener()</b>	<b>JButton</b> , <b>JList</b> , <b>JTextField</b> , <b>JMenuItem</b> y sus derivados incluyendo <b>JCheckBoxMenuItem</b> , <b>JMenu</b> , y <b>JPopupMenu</b> .
<b>AdjustmentEvent</b> <b>AdjustmentListener</b> <b>addAdjustmentListener()</b> <b>removeAdjustmentListener()</b>	<b>JScrollbar</b> y todo lo que se cree que implemente la interfase <b>Adjustable</b> .
<b>ComponentEvent</b> <b>ComponetListener</b> <b>addComponentListener()</b> <b>removeComponentListener()</b>	* <b>Component</b> y sus derivadas, incluyendo <b>JButton</b> , <b>JCanvas</b> , <b>JCheckBox</b> , <b>JComboBox</b> , <b>Containter</b> , <b>JPanel</b> , <b>JApplet</b> , <b>JScrollPane</b> , <b>Windows</b> , <b>JDialog</b> , <b>JFileDialog</b> , <b>JFrame</b> , <b>JLabel</b> , <b>JList</b> , <b>JScrollbar</b> , <b>JTextArea</b> , y <b>JTextField</b> .
<b>ContainerEvent</b>	<b>Container</b> y sus derivadas, incluyendo <b>JPanel</b> , <b>JApplet</b>

<b>ContainerListener</b> <b>addContainerListener</b> <b>addContainerListener()</b> <b>removeContainerListener()</b>	incluyendo <b>JPane</b> , <b>JApplet</b> , <b>JScrollPane</b> , <b>Window</b> , <b>JDialog</b> , <b>JFileDialog</b> , y <b>JFrame</b> .
<b>FocusEvent</b> <b>FocusListener</b> <b>addFocusListener()</b> <b>removeFocusListener()</b>	<b>Component</b> y derivadas*
<b>KeyEvent</b> <b>KeyListener</b> <b>addKeyListener()</b> <b>removeKeyListener()</b>	<b>Component</b> y derivadas*
<b>MouseEvent</b> (para clic y movimiento) <b>MouseListener</b> <b>addMouseListener()</b> <b>removeMouseListener()</b>	<b>Component</b> y derivadas*
<b>MouseEvent</b> <sup>8</sup> (para clic y movimiento)	<b>Component</b> y derivadas*.
<b>WindowEvent</b> <b>WindowListener</b> <b>addWindowListener</b> <b>addWindowListener()</b> <b>removeWindowListener()</b>	<b>Window</b> y sus derivadas, incluyendo <b>JDialog</b> , <b>JFileDialog</b> , y <b>JFrame</b> .
<b>ItemEvent</b> <b>ItemListener</b> <b>addItemListener()</b> <b>removeItemListener()</b>	<b>JCheckBox</b> , <b>JCheckBoxMenuItem</b> , <b>JComboBox</b> , <b>JList</b> , y todo lo que implemente la interfase <b>ItemSelectable</b> .
<b>TextEvent</b>	Cualquier cosa derivada de <b>JTextComponet</b> , incluyendo

<sup>8</sup> No hay **MouseMotionEvent** aún si pareciera que debería haber. Clic y movimiento esta combinado en **MouseEvent**, así es que la segunda aparición de **MouseEvent** en la tabla no es un error.

<b>TextListener</b>	<b>JTextArea y JTextField.</b>
<b>addTextListener()</b>	
<b>removeTextListener()</b>	

Se puede ver que cada tipo de componente soporta solo ciertos tipos de eventos. Descartamos que haya alguna dificultad para buscar todos los eventos soportados por cada componente. Una estrategia mas simple es modificar el programa **ShowMethodsClean.java** del Capítulo 12 de tal forma que muestre todos los eventos soportados por cada componente Swing que se entre.

El Capítulo 12 nos introduce *reflexión* y utiliza esta característica para buscar métodos para una clase particular .la lista de métodos entera o un parte de ellas cuyos nombres coincidas con una palabra clave que se proporcione. La magia de esto es que puede automáticamente mostrar todos los métodos de una clase sin forzarnos a caminar por toda la jerarquía de herencias examinando la clase base en cada nivel. De esta forma, proporciona una herramienta de valor para ahorrar tiempo de programación: dado que los nombres de la mayoría de los métodos de Java son hechos elocuentes y descriptivos, se puede buscar el nombre de método que contiene una palabra de interés en particular. Cuando se encuentra lo que se piensa que se esta buscando, se verifica la documentación en línea.

Sin embargo, como en el Capítulo 12 no se había visto Swing, la herramienta en este capítulo fue desarrollada como una aplicación de línea de comandos. Aquí hay una versión GUI mas útil, especializada para encontrar los métodos “addListener” in componentes Swing:

```
//: c13>ShowAddListeners.java
// Despliega los métodos "addXXXListener" de cualquier
// clase Swing.
// <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();
// Convierte a un arreglo de cadenas:
n = new String[m.length];
for(int i = 0; i < m.length; i++)
    n[i] = m[i].toString();
reDisplay();
}
void reDisplay() {
    // Crea un grupo de resultados:
    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.setLayout(BorderLayout.NORTH, top);
    cp.add(new JScrollPane(results));
}
public static void main(String[] args) {
    Console.run(new ShowAddListeners(), 500, 400);
}
} ///:~

```

La clase **StripQualifiers** definida en el capítulo 12 es reutilizada aquí para importar la librería **com.bruceeckel.util**.

La GUI contiene un **JTextField** **name** en donde se puede entrar el nombre de la clase Swing que se quiere buscar. El resultado es desplegado en una **JTextArea**.

Se notará que no hay botones u otro componente con el cual se quiera indicar que se quiere comenzar la búsqueda. esto es porque el **JTextField** es monitoreado por un **ActionListener**. Cuando quiera que se haga un cambio y se presione ENTER, la lista es inmediatamente actualizada. Si el texto no esta vacío, es utilizado dentro de **Class.forName()** para intentar buscar la clase. Si el nombre es incorrecto, **Class.forName()** fallará, lo que significa que lanzará una excepción. Esta es atrapada y el **JTextArea** es configurado a “No match”. Pero si su tipo es un nombre correcto (las mayúsculas y minúsculas cuentan), **Class.forName()** es exitosa y **getMethods()** retornará un arreglo de objetos **Method**. Cada uno de los objetos en el arreglo es transformado en un **String** mediante **toString()** (esto produce la firma de método completa) y se agrega a **n**, un arreglo **String**. El arreglo **n** es un miembro de la clase **ShowAddListeners** y es utilizado para actualizar la visualización cuando sea que **reDisplay()** sea llamado.

**reDisplay()** crea un arreglo de **String** llamado **rs** (por “result set”). El grupo de resultado es copiado condicionalmente de **String** en **n** que contiene “add” y “Listener”. **indexOf()** y **substring()**, son entonces utilizados para quitar los calificadores como **public**, **static**, etc. Finalmente, **StripQualifiers.strip()** quita los calificadores extra de nombres.

Este programa es una manera conveniente de investigar las capacidades de un componente Swing. Una vez que se conoce que eventos de un componente particular soporta, no se necesita buscar nada mas para reaccionar a ese evento. Simplemente:

1. Se toma el nombre de la clase de evento y se quita la palabra “**Event**”. Se agrega la palabra “**Listener**” a lo que queda. Esta es la interfase listener que se debe implementar es su clase interna.
2. Implementar la interfase mas arriba y escribir los método para los eventos que se quieren capturar. Por ejemplo, se pueden estar buscando movimientos de ratón, así es que se escribe código para el método **mouseMoved()** de la interfase **MouseMotionListener** (se deben implementar los otros métodos, claro, pero a menudo hay un atajo para estos que se verá mas adelante).
3. Se crea un objeto de la clase listener del paso 2. Se registra con el componente con el método producido agregando adelante “**add**” a el nombre del listener. Por ejemplo, **addMouseMotionListener()**.

Aquí vemos algunas de las interfaces listener:

Interfase listner con adaptador	Métodos en la interfase
<b>ActionListener</b>	<b>actionPerformed(ActionEvent)</b>
<b>AdjustmentListener</b>	<b>adjustmentValueChanged(AdjustmentEvent)</b>

<b>ComponentListener</b>	<b>componentHidden(ComponentEvent)</b>
<b>ComponentAdapter</b>	<b>componentShown(ComponentEvent)</b> <b>componentMoved(ComponentEvent)</b> <b>componentResized(ComponentEvent)</b>
<b>ContainerListener</b>	<b>componentAdded(ContainerEvent)</b>
<b>ContainerAdapter</b>	<b>componentRemoved(ContainerEvent)</b>
<b>FocusListener</b>	<b>focusGained(Focus Event)</b>
<b>FocusAdapter</b>	<b>focusLost(FocusEvent)</b>
<b>KeyListener</b>	<b>keyPressed(KeyEvent)</b>
<b>KeyAdapter</b>	<b>keyReleased(KeyEvent)</b> <b>keyTyped(KeyEvent)</b>
<b>MouseListener</b>	<b>mouseClicked(MouseEvent)</b>
<b>MouseAdapter</b>	<b>mouseEntered(MouseEvent)</b> <b>mouseExited(MouseEvent)</b> <b>mousePressed(MouseEvent)</b> <b>mouseReleades(MouseEvent)</b>
<b>MouseMotionListener</b>	<b>mouseDragged(MouseEvent)</b> <b>mouseMoved(MouseEvent)</b>
<b>WindowListener</b>	<b>windowOpened(WindowEvent)</b>
<b>WindowAdapter</b>	<b>windowClosing(WindowEvent)</b> <b>windowActivated(WindowEvent)</b> <b>windowDeactivated(WindowEvent)</b> <b>windowIconified(WindowEvent)</b> <b>windowsDeiconified(WindowEvent)</b>
<b>ItemListener</b>	<b>itemStateChanged(ItemEvent)</b>

Esto no es una lista exhaustiva, en parte porque el modelo de eventos permite crear sus propios tipos de eventos y listeners asociados. De esta forma, regularmente se verán librerías que han inventado sus propios eventos, y el conocimiento ganado en este capítulo permitirá que se imagina como utilizar estos eventos.

## Utilizando adaptadores listeners para simplificar

En la tabla anterior, se puede ver que algunas interfaces listeners tienen solo un método. Estas son triviales de implementar dado que solo se debe implementar cuando se quiere escribir ese método en particular. Sin embargo, la interfase listener que tiene muchos métodos puede ser menos placentera de utilizar. Por ejemplo, a veces se siempre debe hacer cuando se crea una aplicación es proporcionar un **WindowListener** a el **JFrame** de tal manera que cuando se obtiene el evento **windowClosing()** se puede llamar a **System.exit()** para salir de la aplicación. Pero dado que **WindowListener** es una interfase, se deben implementar todos los otros métodos aún si no hacen nada. Esto puede ser molesto.

Para solucionar el problema, algunas (pero no todas) las interfaces listeners que tienen mas de un método son proporcionadas con **adaptadores**, los nombres de se pueden ver en la tabla mas arriba. Cada adaptador proporciona métodos por defecto vacíos para cada uno de los métodos de la interfase. Entonces todo lo que se necesita hacer es heredar del adaptador y sobrecargar solo los métodos que necesita cambiar. Por ejemplo, el típico **WindowListener** que utilizará se ve como esto (recuerde que esto ha sido envuelto dentro de la clase **Console** en **com.bruceeckel.swing**):

```
class MyWindowListener extends WindowAdapter {  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
}
```

El único punto de los adaptadores es hacer la creación de las clases listeners fácil.

Hay una desventaja de los adaptadores, sin embargo, en la forma de trampa. Supongamos que se escribe un **WindowAdapter** como el de arriba:

```
class MyWindowListener extends WindowAdapter {  
    public void WindowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
}
```

Esto no trabaja, pero nos volverá locos tratando de darnos cuenta por que, dado que todo compilara y se ejecutaría bien -excepto porque cerrar la ventana no saldrá del programa. ¿Puede ver el problema? Está en el nombre del método: **WindowClosing()** en lugar de **windowClosing()**. Un simple desliz en una mayúscula resulta en el agregado de un método completamente nuevo. Sin embargo, eso no es el método que es llamado cuando la ventana se cierra, así es que no se obtienen los resultados deseados. A pesar del inconveniente, una interfase garantizara que los métodos son implementados adecuadamente.

## Rastrear eventos múltiples

Para probarnos a nosotros mismos que los eventos efectivamente fueron disparados, y como un experimento interesante, tiene valor crear un applet que trace comportamiento extra en un **JButton** (otros que solamente si es presionado o no). Este ejemplo también muestra como heredar un objeto de botón propio dado que es lo que es utilizado como blanco de todos los eventos de interés. Para hacerlo de esta manera, se puede simplemente heredar de **JButton**<sup>9</sup>.

La clase **MyButton** es una clase interna de **TrackEvent**, así es que **MyButton** puede alcanzar dentro de la ventana padre y manipular sus campos de texto, lo que es necesario ser capaz de escribir la información de estado dentro de los campos del parent. Claro que esta es una solución limitada, dado que **myButton** puede ser utilizado solo en conjunción con **TrackEvent**. Este tipo de código es llamado a veces “altamente acoplado”:

```
///: c13:TrackEvent.java
// Muestra los eventos que suceden.
// <applet code=TrackEvent
// width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceekel.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());
            }
        }
    }
}
```

---

<sup>9</sup> En Java 1.0/1.1 *no* se puede heredar exitosamente de un objeto botón. Eso fue solo una de los numerosos desperfectos de diseño fundamental.

```

};

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(f1);
    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 **MyButton**, el color del botón es configurado llamando a **setBackGround()**. Los listeners son todos instalados con una llamada simple a un método.

La clase **TrackEvent** contiene un **HashMap** para almacenar las cadenas que representan el tipo de evento y **JTextFields** donde se mantiene la información de los eventos. Claro, esto puede haber sido creado estáticamente en lugar de colocado dentro de un **HashMap**, pero pienso que se estará de acuerdo que es mas fácil de utilizar y de cambiar. En particular, si se necesita agregar o quitar un nuevo tipo de evento en **TrackEvent**, simplemente de agrega o quita una cadena en el arreglo **event** -todo lo demás se sucede automáticamente.

Cuando **report()** es llamado da el nombre del evento y de la cadena parámetro del evento. Esto utiliza el **HashMap h** en la clase externa para buscar el **JTextField** actual asociado con ese nombre de evento, y luego toma la cadena del parámetro en ese campo.

Este ejemplo es divertido para jugar con el dado que se puede ver realmente que seta sucediendo con los eventos en su programa.

## Un catálogo de componentes

Ahora que se entiende los manejadores de trazados y el modelo de eventos, se esta listo para ver como los componentes Swing pueden ser utilizados. Esta sección es un paseo no exhaustivo por los componentes Swing y características que probablemente se utilizarán la mayoría del tiempo. cada ejemplo intenta ser razonablemente pequeño así se puede levantar el código y utilizar en sus propios programas.

Se puede fácilmente ver que cada uno de estos ejemplos se ve como ejecutándolos mientras se observan las páginas HTML del código que se puede bajar de este capítulo.

Hay que tener en mente:

1. La documentación HTML de [java.sun.com](http://java.sun.com) contiene todas las clases Swing y sus métodos (solo unos pocos son mostrados aquí).
2. A causa de las convenciones de nombres utilizadas por los eventos Swing, es bastante fácil adivinar como escribir e instalar un manejador para un tipo particular de evento. Use el programa para buscar **ShowAddListeners.java** que esta en este capítulo para ayudar en la investigación de un componente en particular.
3. Cuando las cosas comienzan a ponerse complicadas se debería hacer de un constructor de GUI.

## Botones

Swing incluye una gran cantidad de diferentes tipos de botones. Todos los botones, casillas de verificación, botones de radio, e incluso ítems de menú son heredados de **AbstractButton** (que puesto que los ítems de menú son incluidos, probablemente sería mejor nombre “AbstractChooser” o algo igualmente general). Se verá brevemente el uso de los ítems de menú, pero el siguiente ejemplo muestra los varios tipos de botones disponibles:

```
//: c13:Buttons.java
// Varios botones Swing.
// <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.bruceekel.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);
    }
} //:~

```

Este comienza con el **BasicArrowButton** de **javax.swing.plaf.basic**, luego continua con los varios tipos específicos de botones. Cuando se ejecute el ejemplo, se verá que el botón commutador almacena su última posición, adentro o afuera. Solamente las casillas de verificación y los botones de radio se comportan idénticos a cada uno de los otros, solo pulsándolos encendido o apagado (son heredados de **JToggleButton**)

## Grupos de botones

Si se quiere que los botones de radio se comporten de una forma “o exclusivo”, se deben agregar a un “grupo de botones”. Pero, como el ejemplo mas adelante demuestra, cualquier **AbstractButton** puede ser agregado a un **ButtonGroup**.

Para evitar repetir un montón de código, este ejemplo utiliza reflexión para generar los grupos de los diferentes tipos de botones. Esto es visto en **makeBPanel()**, que crea un grupo de botones y un **JPanel**. El segundo argumento de **makeBPanel()** es un arreglo de **String**. Para cada **String**, un botón de clase representado por el primer argumento es agregado al **JPanel**:

```

//: c13:ButtonGroups.java
// Uso de reflexión para crear grupos
// de diferentes tipos de 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 {
                // Obtiene el constructor dinámico de método
                // que toma un argumento String:
                Constructor ctor = bClass.getConstructor(
                    new Class[] { String.class });
                // Crea un nuevo objeto:
                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 para el borde es tomado del nombre de la clase, eliminando toda la información de ruta. El **AbstractButton** es inicializado a un **JButton** que tiene la etiqueta “Failed” así es 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 arreglo de argumentos de los tipos en el arreglo **Class** pasado a **getConstructor()**. Entonces todo lo que se hace es llamar a **newInstance()**, pasando un arreglo de **Object** conteniendo los argumentos actuales -en este caso, solo el **String** del arreglo **ids**.

Esto agrega un poco de complejidad a lo que es un proceso simple. Para obtener comportamiento “o exclusivo” con botones, se crea un grupo de botones y se agrega cada botón para lo que se quiera ese comportamiento para el grupo. Cuando de ejecute el programa, se verá que todos los botones excepto **JButton** exhiben este comportamiento “o exclusivo”.

## Icons

Se puede utilizar un **Icon** dentro de una **JLabel** o cualquier cosa que se herede de **AbstractButton** (incluyendo **JButton**, **JCheckBox**, **JRadioButton**,

y los diferentes tipos de **JMenuItem**). Usar **Icons** con **JLabels** es bastante directo (como se verá en los ejemplos mas adelante). El siguiente ejemplo explora todas las formas adicionales en las que se pueden utilizar **Icons** con botones y sus descendientes.

Se puede utilizar cualquier fichero **gif** que se quiera, pero los únicos utilizados en este ejemplo son parte de el código de distribución de este libro, disponible en [www.BruceEckel.com](http://www.BruceEckel.com). Para abrir un fichero y traer la imagen, simplemente se crea una **ImageIcon** y este maneja el nombre de fichero. Desde ese momento, se puede utilizar el **Icon** resultante en el programa.

Debe notarse que la información de ruta es totalmente codificada en este ejemplo; se necesitará cambiar la ruta para que corresponda con los ficheros de imagen.

```
//: c13:Faces.java
// Comportamiento de Icon en 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 {
    // La siguiente información de ruta es necesaria
    // para ejecutar mediante un applet directamente del disco:
    static String path =
        "C:/aaa-TIJ2-distribution/code/c13/";
    static Icon[] faces = {
        new ImageIcon(path + "face0.gif"),
        new ImageIcon(path + "facel.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);
            }
        });
        add(jb);
        add(jb2);
    }
}
```

```

        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);
}
} //:~
}

```

Un **Icon** puede ser utilizado en muchos constructores, pero se puede utilizar también **setIcon()** para agregar o cambiar un **Icon**. este ejemplo también muestra como un **JButton** (o cualquier **AbstractButton**) puede configurar los diferentes tipos de íconos que aparecen cuando cosas le suceden a ese botón: cuando es presionado, deshabilitado, o se “hace rodar” (el rato se mueve por encima sin presionarlo). Se verá que esto le da al botón un bonito aspecto animado.

## Tool tips

El ejemplo anterior agregó un “tool tip” (información de herramienta) al botón. Al menos todas las clases que se utilizarán para crear interfaces de usuario son derivadas de **JComponent**, que contiene un método llamado **setToolTipText(String)**. Así es que, para virtualmente todo lo que se coloque en la hoja, todo lo que se necesita hacer es decir (para un objeto **jc** de cualquier clase derivada de **JComponent**):

| jc.setToolTipText("My tip");  
y cuando el ratón permanezca encima del **JComponet** por un período de tiempo predeterminado, una pequeña cajita conteniendo su texto aparecerá arriba 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 **JTextFiel t3** es incluido como un lugar para reportar cuando el listener las acciones para el **JTextField t1** es disparado. Se verá que el listener de acciones ara un **JTextField** es disparado solo cando se presiona la tecla “enter”.

El **JTextField t1** tiene varios listeners enganchados a el. El listener **T1** es un **DocumentListener** que responde a cualquier cambio en el “documento” (el contenido de **JTextField**, en este caso). Automáticamente copia todo el texto de **t1** a **t2**. Además, el documento de **t1** es configurado para una clase derivada de **PlainDocument**, llamado **UpperCaseDocument**, que fuerza a todos los caracteres a ser mayúsculas. Automáticamente detecta la tecla de retroceso y realiza la eliminación, ajustando el caret y manejar todo como se espera.

## Border

El **JComponent** contiene un método llamado **setBorder()**, que permite colocar varios bordes interesantes en cualquier componentes visibles. El siguiente ejemplo muestra varios diferentes bordes que estás disponibles, usando un método llamado **showBorder()** que crea un **JPanel** y coloca el borde en cada caso (borrando toas las rutas de información), luego coloca el nombre en una **JLabel** en el medio del panel:

```
//: c13:Borders.java
// Diferentes bordes Swing.
// <applet code=Borders
// width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceekel.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);
    }
} ///:~
```

Se pueden crear también sus propios bordes y colocarlos dentro de los botones, etiquetas, etc. -todo derivado de **JComponent**.

## JScrollPanes

La mayoría del tiempo solo dejaremos que un **JScrollPane** haga el trabajo, pero también puede controlar que barras de desplazamiento son permitidas- verticales, horizontales, ambas o ninguna:

```
//: c13:JScrollPanes.java
// Controlando las barras de desplazamiento en un 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.bruceekel.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());
        // Crear Borders para los componentes:
        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);
        // Inicializar listeners y agregar componentes:
        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);
    }
} //:~

```

Utilizando diferentes argumentos en el constructor del **JScrollPane** se controla las barras de desplazamiento que están disponibles. Este ejemplo también embellece las cosas usando un poco de bordes.

## Un mini editor

El control **JTextPane** proporciona una forma estupenda de soporte para edición, sin mucho esfuerzo. El siguiente ejemplo hace muy simple el uso de esto, ignorando el grueso de la funcionalidad de la clase:

```
| //: c13:TextPane.java
```

```

// El control JTextPane es un pequeño edito.
// <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 solo agrega texto generado de forma aleatoria. El objetivo de **JTextPane** es permitir que texto sea editado en el lugar, así es que se verá que no hay método **append()**. En este caso (admitiéndolo, un uso pobre de las capacidades de **JTextPane**), el texto debe ser capturado, modificado, y colocado atrás dentro del cuadro utilizando **setText()**.

Como ha sido mencionado antes, el comportamiento de trazado por defecto de un applet es utilizar el **BorderLayout**. Si se agrega algo al cuadro sin especificar ningún detalle, solo llena el centro del panel hasta los bordes. Sin embargo, si se especifica una de las regiones alrededor (NORTH, SOUTH, EAST, o WEST) como se ha hecho aquí, el componente entrará solo en esa región -en este caso, el botón será ubicado en la parte baja de la pantalla.

Note que las características insertadas en **JTextPane**, como el plegado automático de las líneas. Hay montones de otras características que se pueden buscar utilizando la documentación de JDK.

## Casillas de verificación

Una casilla de verificación proporciona una forma de hacer una selección única, si o no; esta consiste en una pequeña caja y una etiqueta. La caja

típicamente contiene una pequeña “x” (o alguna otra indicación de que está configurada) o está vacía, dependiendo de que está seleccionado.

Normalmente se creará una **JCheckBox** utilizando un constructor que tome la etiqueta como argumento. Se puede tomar y configurar el estado, y también tomar y configurar la etiqueta si se quiere leer o cambiar esta luego que la **JCheckBox** ha sido creada.

Cuando quiera que una **JCheckBox** sea creada o limpiada, un evento ocurre, que se puede capturar de la misma forma que se hace con un botón, utilizando un **ActionListener**. El siguiente ejemplo utiliza un **JTextArea** para enumerar todas las casillas de verificación que han sido seleccionadas:

```
//: c13:CheckBoxes.java
// Utilizando JCheckBoxes.
// <applet code=CheckBoxes width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceekel.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 **trace()** envía el nombre de la **JCheckBox** seleccionada y su estado actual a la **JTextArea** utilizando **append()**, así es que se puede ver una lista acumulativa de casillas de verificación que son seleccionadas y cual es su estado.

## Botones de radio

El concepto de botones de radio en programación GUI viene de las radios de autos con botones mecánicos: cuando se presiona uno, todos los demás botones que fueron presionados saltan. De esta manera, permite forzar una sola elección entre muchas.

Todo lo que se necesita hacer es configurar un grupo asociado de **JRadioButtons** sean agregados a un **ButtonGroup** (se puede tener cualquier numero de **ButtonGroups** en una hoja). Uno de los botones puede opcionalmente tener su estado inicial configurado a **true** (utilizando el segundo argumento en el constructor). Si se intenta configurar mas de un botón de radio a **true** entonces solo el último será **true**.

He aquí un ejemplo simple de el uso de botones de radio. Note que se capturan eventos de botones de radio como todos los demás:

```

//: c13:RadioButtons.java
// Utilizando JRadioButtons.
// <applet code=RadioButtons
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceekel.swing.*;
public class RadioButtons extends JApplet {
    JTextField t = new JTextField(15);
    ButtonGroup g = new ButtonGroup();
    JRadioButton
        rbl = 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() {
        rbl.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, un campo de texto es utilizado. Este campo es configurado para que no pueda ser editado porque es utilizado solo para mostrar datos, no para recogerlo. Como alternativa de esto se puede utilizar una **JLabel**.

## Combo boxes (listas desplegables)

Como un grupo de botones de radio, una lista desplegable es una forma de forzar a el usuario a seleccionar solo un elemento de un grupo de posibilidades. Sin embargo, es una forma mas compacta de lograr esto, y es mas fácil cambiar los elementos de la lista sin sorprender al usuario (Se pueden cambiar los botones de radio dinámicamente, pero eso tiende a ser irritante a la vista).

La caja **JComboBox** de Java no es como la de Windows, que deja que se seleccione de una lista o tipo de su propia selección. Con una caja **JComboBox** se elige uno y solo un elemento de la lista. En el siguiente ejemplo, la caja **JComboBox** comienza con un cierto número de entradas y luego nuevas entradas son agregadas cuando un botón es presionado.

```

//: c13:ComboBoxes.java
// Utilizando listas desplegables.
// <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", "Sommescent",
        "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);
}
} //:~

```

El **JTextField** despliega el “índice seleccionado”, que es la secuencia de números del elemento seleccionado actualmente, de la misma forma que la etiqueta del botón de radio.

## Cajas de listas

Las cajas de listas son significativamente diferentes de las cajas **JComboBox**, y no solo en apariencia. Una caja **JComboBox** se despliega cuando se activa, una **JList** ocupa algún número fijo de líneas en la pantalla todo el tiempo y no cambia. Si se quiere ver los ítems en una lista, simplemente se llama a **getSelectedValues()**, que produce un arreglo de **String** con los ítems que han sido seleccionados.

Una **JList** permite selección múltiple: si se hace clic en mas de un ítem (manteniendo presionada la tecla de control mientras se realiza los clic para agregar) el ítem original sigue margado y se puede seleccionar tantos como se quiera. Si se selecciona un ítem, y luego se realiza un clic manteniendo presionada la tecla “shift” en otro ítem, todos los ítems que se extienden entre esos dos serán seleccionados. Para quitar un ítem del grupo se puede realizar un clic manteniendo presionada la tecla de control.

```

//: 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.bruceckel.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 {
                // Desabilitado, dado que no hay mas
                // sabores para agregar a la lista
                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());
        // Crear bordes para los componentes:
        Border brd = BorderFactory.createMatteBorder(
            1, 1, 2, 2, Color.black);
        lst.setBorder(brd);
        t.setBorder(brd);
        // Agrega los primeros cuatro ítems a la lista
        for(int i = 0; i < 4; i++)
            lItems.addElement(flavors[count++]);
        // Agrega ítems a el Panel de contenido para mostrar
        cp.add(t);
        cp.add(lst);
        cp.add(b);
        // Registra los listeners de eventos
        lst.addListSelectionListener(ll);
    }
}

```

```

        b.addActionListener(bl);
    }
    public static void main(String[] args) {
        Console.run(new List(), 250, 375);
    }
} //:~

```

Cuando se presiona el botón se agregan ítems en la parte *superior*de la lista (porque el segundo argumento de **addItem()** es cero).

Se puede ver que los bordes han sido agregados a las listas.

Si solo se quiere colocar una arreglo de **Strings** en una **JList**, hay soluciones mucho mas simples: se pasa el arreglo al constructor de la **JList**, y ella crea la lista automáticamente. La única razón para utilizar el “modelo lista” en el ejemplo mas arriba es porque la lista puede ser manipulada durante la ejecución del programa.

**JLists** no proporcionan automáticamente soporte automático para desplazamiento. Claro, todo lo que se necesita hacer es envolver la lista en un **JScrollPane** y todos los detalles son manejados por usted.

## Paneles con lengüetas

El **JTabbedPane** permite crear un “cuadro de diálogo tabulado”, que son ficheros de carpetas con lengüetas que fluyen en un borde, y todo lo que se tiene que hacer es presionar una lengüeta para traer un cuadro de diálogo diferente.

```

//: c13:TabbedPane1.java
// Demuestra los paneles tabulados.
// <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 TabbedPanel1(), 350, 200);
}
} //:~

```

En Java el uso de algún tipo de mecanismo de “panel con lengüetas” es bastante importante porque en programación de applets el uso de diálogos desplegables es poco estimulante porque se agrega una pequeña etiqueta de advertencia para cualquier diálogo que aparezca fuera del applet.

Cuando se ejecute el programa se puede ver que **JTabbedPane** automáticamente amplia las lengüetas si hay muchas de ellas para adaptarlas en una sola fila. Se puede ver esto cambiando el tamaño de la ventana cuando se ejecute el programa desde la línea de comandos de la consola.

## Cuadros de mensajes

Los ambientes con ventanas comúnmente tienen un grupo de cuadros de mensajes estándar que permiten rápidamente colocar información para el usuario o capturar información del usuario. En Swing, estas cajas de mensajes son contenidas en **JOptionPane**. Se tienen muchas posibilidades diferentes (algunas bastante sofisticadas), pero las que probablemente se utilizarán más comúnmente son probablemente el dialogo de mensaje y el dialogo de confirmación, invocados utilizando el **static JOptionPane.showMessageDialog()** y **JOptionPane.showConfirmDialog()**. El siguiente ejemplo muestra un grupo de cajas de mensajes disponibles con **JOptionPane**:

```

//: c13:MessageBoxes.java
// Demuestra JOptionPane.
// <applet code=MessageBoxes
// width=200 height=150> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceekel.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 ser capaz de escribir un solo **ActionListener**, hemos utilizado una estrategia con algún riesgo de verificar las cadenas de las etiquetas en los botones. El problema con esto es que es fácil colocar una etiqueta con un pequeño error, típicamente en las mayúsculas o minúsculas, y este error puede ser difícil de divisar.

Debe notarse que **showOptionDialog()** y **showInputDialog()** proporciona objetos de retorno que contienen el valor entrado por el usuario.

## Menús

Cada componente capaz de almacenar un menú, incluyendo **JApplet**, **JFrame**, **JDialog** y sus descendientes, tiene un método **setJMenuBar()** que acepta un **JMenuBar** (solo se puede tener un solo **JMenuBar** en un componente en particular). Se agrega **JMenus** a el **JMenuBar**, y **JMenuItem**s a el **JMenus**. Cada **JMenuItem** puede tener un **ActionListener** enganchado a el, para ser disparado cuando ese ítem de menú es seleccionado.

De forma diferente a un sistema que utiliza recursos, con Java y Swing se debe manejar el armado de todos los menús en el código fuente. Aquí hay un menú simple de ejemplo:

```
//: c13:SimpleMenus.java
// <applet code=SimpleMenus
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceckel.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 ítems de menús a lo largo de tres **JMenus**. Cada **JMenuItem** debe tener un **ActionListener** enganchado a el; aquí, el mismo **ActionListener** es utilizado en todos lados pero normalmente se necesitará uno para cada **JMenuItem**.

**JMenuItem** hereda **AbstractButton**, así es que se tienen algunos comportamientos parecidos a los de un botón. Por si mismo, proporcionan un ítem que puede ser colocado en un menú desplegable. Hay también tres tipos heredados de **JMenuItem**: **JMenu** para almacenar otros **JMenuItems** (así es que se puede tener menús en cascada), **JCheckBoxMenuItem**, que produce una marca para indicar cuando el menú es seleccionado, y **JRadioButtonMenuItem**, que contiene un botón de radio.

Como un ejemplo mas sofisticado, aquí están los sabores de helados de crema nuevamente, utilizados para crear menús. Este ejemplo muestra también los menús en cascada, y las teclas mnemotécnicas, **JCheckBoxMenuItem**s, y la forma es que se pueden cambiar dinámicamente menús:

```

//: 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.bruceekel.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 se colocan los ítems de menú en arreglos y luego se colocan en cada arreglo llamando **add()** para cada **JMenuItem**. Esto hace que agregar o quitar un ítem de menú sea algo menos tedioso.

Este programa no crea un menú, crea dos **JMenuBar**s para demostrar que las barras de menú puede ser intercambiadas activamente mientras el programa se ejecuta. Se puede ver como un **JMenuBar** es creado con **JMenus**, y cada **JMenu** es creado por **JMenuItems**, **JCheckBoxMenuItems**, o aún otros **JMenus** (que producen submenús). CUando un **JMenuBar** es ensamblado puede ser instalado en el programa actual con el método **setJMenuBar()**. Debe notarse que cuando un botón es presionado, verifica por el menú actualmente instalado llamando a **getJMenuBar()**, luego se coloca la otra barra de menú en su lugar.

Cuando se verifica para “Open”, note que la ortografía y las mayúsculas y minúsculas son críticas, y Java no entrega ningún tipo de error se no hay coincidencia con “Open”. Este tipo de comparación de cadena es una fuente de errores de programación.

El cuidad de la verificación o no de los ítems de menú es automática. El manejo del código del **JCheckBoxMenuItem**s muestra dos formas diferentes de determinar que fue verificado: coincidencias de cadenas (las que son mencionadas mas arriba, no es una estrategia segura a pesar de que se puede ver que será utilizada) y coincidencias con el objeto de evento objetivo. Como es mostrado, el método **getState()** puede ser utilizado para dar a conocer el estado. Se puede también cambiar el estado de un **JCheckBoxMenuItem** con **setState()**.

Los eventos para menús son un poco inconsistentes y pueden caer en confusión: **JMenuItem**s utilizan **ActionListeners**, pero **JCheckboxMenuItem**s utiliza **ItemListeners**. Los objetos **JMenu** puede también soportar **ActionListeners**, pero esto no es usualmente útil. EN general, se enganchan listeners para cada **JMenuItem**, **JCheckBoxMenuItem**, o **JRadioButtonMenuItem**, pero el ejemplo muestra **ItemListeners** y **ActionListeners** enganchados a los diferentes componentes de menú.

Swing soporta mnemotécnicos, o “teclas rápidas”, así es que se puede seleccionar cualquier cosa derivada de un **AbstractButton** (botón, ítem de menú, etc.) utilizando el teclado en lugar del ratón. Esto es bastante simple: para **JMenuItem** se puede utilizar el constructor sobrecargado que toma como segundo argumento el identificador de la tecla. Sin embargo, la mayoría de los **AbstractButtons** no tienen constructores como esteasíes que la forma mas general de resolver el problema es utilizar el método **setMnemonic()**. El ejemplo anterior agrega mnemotécnicos a los botones y a alguno de los ítems de menú; los indicadores de las teclas rápidas aparecen automáticamente en los componentes.

Se puede también ver el uso de **setActionCommand()**. Esto parece un poco extraño dado que en cada caso el “comando de acción” es exactamente el mismo que la etiqueta en el componente de menú. ¿Por que no solo se utiliza la etiqueta en lugar de una cadena alternativa? El problema es la internacionalización. Si se traduce este programa en otro lenguaje, se debe cambiar solo la etiqueta en el menú y se necesita cambiar solo la etiqueta en el menú, y no cambiar el código (el cual no dudará en introducir nuevos errores). Así es que para hacer que el código fácilmente verifique la cadena de texto asociada con un componente del menú, el “comando de acción” puede ser inmutable mientras la etiqueta de menú puede cambiar. Todo el código trabaja con el “comando de acción”, así es que no es afectado con los cambios en la etiqueta de menú. Debe notarse esto en este programa, no todos los componentes de menú son examinados par sus comandos de acción, así es que aquellos que no lo son no tienen su comando de acción configurados.

La totalidad del trabajo se sucede en los listeners. **BL** realiza el intercambio de **JMenuBar**. En **ML**, la estrategia de “resuelva quien llama” es tomado tomando el fuente del **ActionEvent** y convirtiéndolo en un **JMenuItem**, luego tomando la cadena de comando de acción para pasarl a través de una instrucción **if** en cascada.

El listener **FL** es simple aún cuando maneja todos los diferentes sabores en el menú de sabores. Esta estrategia es útil cuando se tienen simplicidad suficiente en la lógica, pero en general, se querrá tomar la estrategia utilizada con **FooL**, **BarL**, y **BazL**, en donde son enganchados a un solo componente de menú así es que ninguna lógica es necesaria y se sabe exactamente quien llamó al listener. Aún con la abundancia de clases generada de esta forma, el código dentro tiende a ser menor y el proceso es mas a prueba de tontos.

Se puede ver que este código de menú rápidamente se vuelve rápidamente cansador y entreverado. Este es otro caso donde el uso de un constructor de GUI es la solución apropiada. Una buena herramienta manejará también el mantenimiento de los menús.

## Menús pop-up

La forma mas directa de implementar un **JPopupMenu** es crear una clase interna que extienda **MouseAdapter**, luego se agrega un objeto de esa clase interna a cada componente que se quiera producir un comportamiento pop-up:

```
//: 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.bruceekel.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);
    }
} //:~

```

El mismo **ActionListener** es agregado a cada **JMenuItem**, así este trae el texto de la etiqueta de menú y lo inserta en el **JTextField**.

## Trazado

En un buen marco de trabajo GUI, el trazado debe ser razonablemente fácil -y así es, en la librería Swing. El problema con cualquier ejemplo de trazado es que los cálculos que determinan donde las cosas se ponen típicamente un poco mas complicadas que las llamadas a las rutinas de trazado, y esos cálculos son a menudo mezclados juntos con las llamadas a trazados así es que pueden parecer que la interfase es mas complicada de lo que actualmente es.

Para facilitar las cosas, consideremos el problema de representar datos en pantalla -aquí, los datos son proporcionados por el método incluido **Math.sin()** que es una función matemática de seno. Para hacer las cosas un poco mas interesantes, y para demostrar mas adelante que tan fácil es utilizar componentes Swing, un deslizador es colocado en la parte inferior de la hoja para controlar dinámicamente el número de ciclos de la curva de seno que serán desplegados. Además, si se cambia el tamaño la ventana, se verá que la curva de seno vuelve a encajar solo en el nuevo tamaño de ventana.

A pesar de que todo **JComponent** puede ser pintando y de esta forma utilizado como lienzo, si solo queremos una superficie de trazado se heredará típicamente de un **JPanel**. El único método que necesitamos sobrescribir es **paintComponent()**, que es llamada cuando se quiera que ese componente deba ser repintado (normalmente no nos necesitaremos preocuparnos acerca de esto, la decisión es manejada por Swing). Cuando este es llamado, Swing pasa un objeto **Graphics** a este método, y se puede entonces utilizar este objeto para dibujar o pintar en la superficie.

En el siguiente ejemplo, todo el talento concerniente a pintar esta en la clase **SineDraw**; la clase **SineWave** simplemente configura el programa y el control deslizante. Dentro de **SineDraw**, el método **setCycles()** proporciona un gancho para permitir que otro objeto -el control deslizante, en este caso- 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.bruceeeckel.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 y arreglos son utilizados en el cálculo de los puntos de la onda del seno: **cycles** indican en número completo de períodos deseados de la onda, **points** contienen el número total de puntos que serán graficados, **sines** contiene los valores de la función seno, y **pts** contienen las coordenadas y de los puntos que serán dibujados en el **JPanel**. El método **setCycles()** crea los arreglos de acuerdo a el número de puntos que se necesitan y llena el arreglo de **sines** con números. Llamando a **repaint()**, **setCycles()** hace que se llame a **paintComponent()** así es que el resto de los cálculos y dibujados tendrán lugar.

La primera cosa que se debe hacer es cuando se sobrecarga **paintComponent()** es llamar a la versión de la clase base del método. Entonces se está libre de hacer lo que se quiera; normalmente, esto significa utilizar los métodos **Graphics** que se pueden encontrar en la documentación para **java.awt.Graphics** (en la documentación HTML de *java.sun*) para dibujar y pintar puntos en el **JPanel**. Aquí, se puede ver que al menos todo el código esta involucrado en realizar los cálculos; los únicos dos métodos llamados que actualmente manipulan la pantalla son **setColor()** y **drawLine()**. Probablemente se pueda tener una experiencia similar cuando se crea un programa propio que despliegue datos gráficos -se puede gastar la mayoría del tiempo imaginándose que es lo que se quiere dibujar, pero el proceso de trazado actual será bastante simple.

Cuando creé este programa, la mayor parte del tiempo fue utilizado en obtener la onda seno a desplegar. Una vez que hice esto, pensé que sería bonito ser capaz de cambiar dinámicamente el número de ciclos. Mis experiencias de programación cuando he tratado de hacer este tipo de cosas en otros lenguajes me ha hecho estar muy poco dispuesto a hacer esto, pero resultó la parte mas fácil del proyecto. He creado un **JSlider** (Los argumentos son los valores que toma cuando esta mas a la derecha, los valores que se encuentran mas a la izquierda y el valor de inicio, respectivamente, pero hay otros constructores igualmente) y lo he colocado dentro del **JApplet**. Luego he buscado en la documentación HTML y advierta que solo fue agregado solo el listener **addChangeListener**, que es disparado cuando el deslizador es cambiado lo suficiente como para producir un valor diferente. El único método para esto fue el obviamente llamado **stateChanged()**, el cual proporciona un objeto **ChangeEvent** así es que podría retrocederse a la fuente del cambio y encontrar el valor nuevo. Llamando a **setCycles()** del objeto **sines**, el nuevo valor fue incorporado y el **JPanel** se dibuja nuevamente.

En general, se encontrara que muchos de sus problemas con Swing pueden ser solucionados siguiendo un proceso similar, y se encontrará que por lo general es bastante simple, aún si nunca ha utilizado un componente en particular antes.

Si su problema es mas complejo, hay otras alternativas mas sofisticadas para dibujar, incluyendo componentes JavaBeans de terceros y la API 2D de Java. Esta soluciones están mas allá del alcance de este libro, y se deberían de buscar si su código de dibujo se torna muy oneroso.

## Cajas de diálogo

Una caja de diálogo es una ventana que se despliega fuera de otra ventana. Se supone que es para tratar con algún tema específico sin desordenar la ventana original con estos detalles. Las cajas de diálogo son muy utilizadas en ambientes de programación con ventanas, pero menos utilizadas en applets.

Para crear una caja de diálogo, se hereda de **JDialog**, que es simplemente otro tipo de **Window**, como una **JFrame**. Una **JDialog**, tiene un manejador de trazado (que por defecto es **BorderLayout**) y se agregan listener de eventos para tratar los eventos. Una diferencia significante cuando **windowClosing()** es llamado es que no se quiere cerrar la aplicación. En lugar de eso, se liberan los recursos utilizados por la ventana de dialogo llamando a **dispose()**. He aquí un ejemplo muy simple:

```
//: 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 **JDialog** es creado, el método **show()** debe ser llamado para desplegar y activarla. Para que la caja de diálogo se cierre se debe llamar a **dispose()**.

Se podrá ver que todo lo que se despliega fuera de un applet, incluyendo las cajas de diálogo, son “no confiables”. Esto es, se muestra una advertencia en la ventana que ha sido desplegada. Esto es porque, en teoría, es posible engañar a el usuario a pensar que esta tratando con una aplicación nativa común y conseguir que escriba el número de su tarjeta de crédito, que luego se envíe a través de la Web. Un applet esta siempre enganchado a una página Web y es visible dentro de su navegador Web, mientras que una caja de diálogo esta desprendida -así es que en teoría, es posible. Como resultado no es muy común ver un applet que utilice una caja de diálogo.

El siguiente ejemplo es mas complejo; la caja de diálogo esta constituida por una grilla (utilizando **GridLayout**) de un tipo especial de botón que esta definido aquí como la clase **ToeButton**. Este botón dibuja un marco alrededor de si mismo y, dependiendo de su estado, una “x”, una “o” o nada en el medio. Comienza en blanco, y luego dependiendo de quien mueve, cambia a una “x” o a un “o”. Sin embargo, también puede dar un giro y en adelante ser una “x” o una “o” cuando se haga clic en el botón (Esto hace el concepto de ta-te-ti solo un poco mas molesto de lo que realmente es).

Además, la caja de diálogo puede ser configurada para cualquier número de filas y columnas cambiando los números en 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.bruceekel.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 miembros estáticos solo pueden estar en el nivel mas externo de la clase, las clases internas no pueden tener datos estático o clases internas estáticas.

El método **paintComponent()** dibuja el cuadrado alrededor del panel, y la “x” o la “o”. Esto está lleno de cálculos molestos, pero es directo.

Un clic del ratón es capturado por el **MouseListener**, que primero verifica para ver si el panel tiene algo escrito en el. Si no, la ventana padre es consultada para encontrar de quien es el turno y esto es utilizado para establecer el estado del **ToeButton**. Mediante el mecanismo de la clase interna, el **ToeButton** accede luego a la ventana padre y cambia el turno. Si el botón ya esta mostrando una “x” o una “o” entonces es reprobado. Se puede ver en estos cálculos la conveniencia del if-else ternario descrito en el Capítulo 3. Luego que es estado cambia, el **ToeButton** es pintado nuevamente.

El constructor para el **ToeDialog** es bastante simple: agrega dentro de una **GridLayout** tantos botones como se hayan solicitado, luego cambia el tamaño a 50 pixel por lado para cada botón.

**TicTacToe** configura la totalidad de la aplicación creando el **JTextFields** (con la entrada de las filas y columnas en la grilla de botones) y el botón “go” con su **ActionListener**. Cuando el botón es presionado, los datos en el **JTextFields** debe ser traídos, y, dado que están en forma de cadena, se convierten en enteros utilizando el método **static Integer.parseInt()**.

## Diálogos de ficheros

Algunos sistemas operativos tienen algunas cajas de diálogo especiales incluidas para manejar la selección de cosas como fuentes, colores, impresoras y semejantes. Virtualmente todos los sistemas operativos soportan la apertura y almacenado de ficheros, sin embargo, y de esta forma el **JFileChooser** de Java encapsula esto para su fácil uso.

La siguiente aplicación ejercita dos formas de diálogos **JFileChooser**, uno para abrir y uno para guardar. Mucho del código puede a esta altura ser familiar, y todas las actividades interesantes suceden en los listeners de acciones para los dos botones:

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

```

Debe notarse que muchas variaciones pueden aplicarse a **JFileChooser**, incluyendo filtros para limitar los nombres de fichero que se permitirán.

Para un diálogo de “apertura de fichero”, se llama **showOpenDialog()**, y para un diálogo de “guardar fichero”, se llama a **showSaveDialog()**. Estos comandos no retornan hasta que la caja de diálogo es cerrada. El objeto **JFileChooser** sigue existiendo, así es que se puede leer datos de él. El método **getSelectedFile()** y **getCurrentDirectory()** son dos formas en que se puede interrogar los resultados de la operación. Si estos retornan **null** significa que el usuario ha cancelado el diálogo.

## HTML en componentes Swing

Cualquier componente que pueda tomar texto puede también tomar texto HTML, que será formateado de acuerdo con las reglas HTML. Esto significa que se puede fácilmente agregar texto adornado en un componente Swing. 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.bruceeeckel.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);
    }
} ///:~
```

Se debe comenzar el texto con “<html>”, y luego se puede utilizar las etiquetas normales de HTML. Debe notarse que no se está forzado a incluir una etiqueta normal de cierre.

El **ActionListener** agrega una nueva **JLabel** a el formulario, que también contiene texto HTML. Sin embargo, esta etiqueta no es agregada durante **init()** así es que de debe llamar a el método **validate()** del contenedor para forzar un nuevo trazado de los componentes (y de esta forma desplegar la nueva etiqueta).

Se puede también utilizar texto HTML para **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** y **JCheckBox**.

## Deslizadores y barras de progreso

Un deslizador (que ya ha sido utilizado en el ejemplo de la onda sinusoidal) permite a el usuario entrar datos moviendo un punto atrás y adelante, lo que es intuitivo en algunas situaciones (control de volumen, por ejemplo). Una barra de progreso despliega datos en una forma relativa de “lleno” a “vacío” así es que el usuario obtiene una perspectiva. Mi ejemplo favorito para esto es simplemente enganchar el deslizador a la barra de progreso así es que cuando se mueve el deslizador la barra de progreso cambia consecuentemente:

```
//: 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.bruceekel.swing.*;
public class Progress extends JApplet {
    JProgressBar pb = new JProgressBar();
    JSlider sb =
        new JSlider(JSeparator.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 enganchar los dos componentes juntos es compartir su modelo, en la línea:

```
| pb.setModel(sb.getModel());
```

Claro, se puede también controlar los dos utilizando un listener, pero esto es mas director para situaciones simples.

La **JProgressBar** es bastante directa, pero el **JSlider** tiene un montón de opciones, como la orientación y las marcas mayores y menores. Note que tan directo es agregar un borde de título.

# Árboles

Usar un **JTree** es tan simple como decir:

```
| add(new JTree(  
| new Object[] {"this", "that", "other"}));
```

Esto despliega un árbol primitivo. La API para árboles es enorme, sin embargo -ciertamente uno de los mas largos de Swing. Parece que se puede hacer todo con árboles, pero las tareas mas sofisticadas pueden requerir bastante un poco de investigación y experimentación.

Afortunadamente, hay un terreno neutral proporcionado en la librería: los componentes árbol “por defecto”, que generalmente hacen lo que se necesita. Así es que la mayoría del tiempo se puede utilizar estos componentes, y solo en casos especiales necesitará ahondar y entender árboles mas profundamente.

El siguiente ejemplo utiliza componentes árbol “por defecto” para desplegar un árbol en un applet. Cuando se presiona un botón, un nuevo subárbol es agregado bajo el nodo actualmente seleccionado (si no hay nodo seleccionado, el nodo raíz es utilizado):

```
///: 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.bruceekel.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 primer clase, **Branch**, es una herramienta que toma un arreglo de **String** y crea un **DefaultMutableTreeNode** con el primer **String** como raíz y el resto de los **Strings** en el arreglo como hojas. Luego **node()** puede ser llamado para producir la raíz de esta “rama”.

La clase **Trees** contienen un arreglo de dos dimensiones de cadenas de donde **Branch** puede ramificar y un **static int i** para contar a través de este arreglo. El objeto **DefaultMutableTreeNode** almacena los nodos, pero la representación física en pantalla es controlada por el **JTree** y su modelo

asociado, el **DefaultTreeModel**. Debe notarse que cuando el **JTree** es agregado a el applet, este es envuelto en un **JScrollPane** - esto es todo lo que toma para proporcionar desplazamiento automático.

El **JTree** es controlado a través de su *modelo*. Cuando se hace un cambio a el modelo, el modelo genera un evento que causa que el **JTree** realice las actualizaciones necesarias para la representación visible del árbol. En **init()**, el modelo es capturado llamando a **getModel()**. Cuando el botón es presionado, una nueva "rama" es creada. Entonces el componente seleccionado es encontrado (o la raíz es utilizada si nada ha sido seleccionado) y el método **insertNodeInto()** del modelo hace todo el trabajo de cambiar el árbol y causar que sea actualizado.

Un ejemplo como el anterior puede dar lo necesario en un árbol. Sin embargo, los árboles tienen el poder de hacer a penas todo lo que se puede imaginar -en todas partes se ve la palabra "por defecto" en el ejemplo anterior, se puede sustituir su propia clase para obtener un comportamiento diferente. Pero hay que tener cuidado: al menos todas estas clases tienen una interfase muy grande, así es que se puede perder un montón de tiempo luchando para entender la complejidad de los árboles. A pesar de esto, es un buen diseño y las alternativas son usualmente mucho peor.

## Tablas

Como los árboles, las tablas en Swing son basta y poderosas. Son inicialmente un intento de ser la popular interfase "grilla" para bases de datos vía conectividad a bases de datos de Java (Java Database Connectivity, JDBC, discutida en el capítulo 15) y de esta forma tienen una tremenda flexibilidad, que se paga con su complejidad. Hay suficiente aquí para obtener lo básico de una hoja de cálculo hecha y derecha y puede probablemente justificar un libro entero. Sin embargo, es también posible crear una relativamente simple **JTable** si se entiende las básicas.

La **JTable** controla como los datos son desplegados, pero el **TableModel** (modelo de tabla) controla los datos en si. Así es que para crear una **JTable** se deberá típicamente crear un **TableModel** primero. Se puede implementar totalmente la interfase **TableModel**, pero es usualmente mas simple heredar de la clase de ayuda **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.bruceekel.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);
    }
} / //:~

```

**DataModel** contiene un arreglo de datos, pero puede también obtener los datos de otra fuente como una base de datos. El constructor agrega un **TableModelListener** que imprime el arreglo cada vez que la tabla es modificada. El resto de los métodos siguen las convenciones de nombres de los Beans, y son utilizados por **JTable** cuando se quiere presentar la información en **DataModel**. **AbstractTableModel** proporciona los métodos por defecto para **setValue()** y **isCellEditable()** que previene los cambios en los datos, así es que si se quiere ser capas de editar los datos, se debe sobrescribir estos métodos.

Una vez que se tiene un **TableModel**, solo se necesita manejarla con el constructor de **JTable**. Se deben cuidar todos los detalles de despliegue edición, y actualización. Este ejemplo también coloca la **JTable** en un **JScrollPane**.

## Seleccionando un Look & Feel

Uno de los aspectos muy interesantes de Swing es el “Pluggable Look & Feel” (vista y sentido que se puede conectar). Esto permite que un programa emule el look & feel de varios ambientes operativos. Se puede a menudo hacer todos los tipos de adornos como cambiar dinámicamente el look & feel mientras el programa se esta ejecutando. Sin embargo, generalmente solo se quiere hacer una de dos cosas, seleccionar el look & feel “plataforma cruzada” (que es el “metal” de Swing), o seleccionar el look & feel para el sistema donde se está actualmente, de esta forma el programa Java se ve como si fuera creado específicamente para ese sistema. El código para seleccionar cualquiera de esos comportamientos es muy simple -pero de deba ejecutar *antes* de crear cualquier componente visual, dado que los componentes serán hechos basados en el look & feel actual y no cambiarán solo porque sucede que se cambió el look & feel a medio camino del programa (ese proceso es mas complicad y poco común, y es relegado a libros que trates específicamente de Swing).

Actualmente, si se quiere utilizar el look & feel de plataforma cruzada (“metal”) que es característico de los programas Swing, no se tiene que hacer nada -es por defecto. Pero si se quiere utilizar el look & feel del actual ambiente operativo, simplemente se inserta el siguiente código, típicamente en el comienzo de su **main()** pero de alguna manera antes que cualquier componente sea agregado:

```
try {
    UIManager.setLookAndFeel(UIManager.
        getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

No se necesita nada en la cláusula **catch** porque el **UIManager** por defecto cambiará a el look & feel por defecto de plataforma cruzada si el intento de configurar alguna de las alternativas falla. Sin embargo, durante un

depurado la excepción puede ser bastante útil si se quiere poner al menos una instrucción para imprimir algo en la cláusula de captura.

Aquí hay un programa que toma un argumento de la línea de comandos para seleccionar un look & feel, y muestra como muchos componentes diferentes se ven bajo el look & feel elegido:

```
//: c13:LookAndFeel.java
// Selecting different looks & feels.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceekel.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 la opción es para especificar explícitamente una cadena para un look & feel, como se ve con **MotifLookAndFeel**. Sin embargo, esa y el look & feel “metal” por defecto son solo los dos que pueden legalmente ser utilizados en cualquier plataforma; a pesar de que hay cadenas look & feel para Windows y Macintosh, estos pueden solo ser utilizados solo en sus respectivas plataformas (estas son producidas cuando se llama a **getSystemLookAndFeelClassName()** y se esta en una plataforma en particular).

Es también posible crear un paquete look & feel a medida, por ejemplo, si se esta creando un marco de trabajo para una compañía que quiere una apariencia distintiva. esto es un trabajo muy grande y esta mas allá del alcance de este libro (¡De hecho, esta mas allá del alcance de muchos libros dedicados a Swing!).

## El portapapeles

La JFC soporta operaciones limitadas con el sistema de portapapeles (en el paquete **java.awt.datatransfer**). De pueden copiar objetos String a el portapapeles como texto, y se puede pegar texto del portapapeles en objetos **String**. Claro, el portapapeles esta diseñado para almacenar cualquier tipo de datos, pero como estos datos están representados en el portapapeles esta por arriba del programa que esta cortando y pegando. La API del portapapeles de Java proporciona por extensibilidad el concepto de un “sabor”. Cuando un dato viene del portapapeles, está asociado a un grupo de sabores a los que puede ser convertido (por ejemplo, un gráfico puede ser representado como una cadena de números o como una imagen) y se puede ver si un portapapeles en particular soporta el sabor en el cual se esta interesado.

El siguiente programa es una demostración simple de corte, copia y pegado con datos del tipo **String** en un **JTextArea**. Una cosa que se notará es que las secuencias de teclado que normalmente se utilizan para cortar, copiar y pegar también trabajan. Pero si se observa en cualquier **JTextField** o **JTextArea** en cualquier otro programa, se encontrará que estos también

soportan las secuencias de teclado del portapapeles automáticamente. Este ejemplo simplemente agrega control programático del portapapeles, y se puede utilizar estas técnicas si se quiere capturar texto del portapapeles en alguna otra cosa que un **JTextComponent**.

```
//: c13:CutAndPaste.java
// Using the clipboard.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceekel.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 agregado del menú y del **JTextArea** deben por ahora parecer una actividad prosaica. Lo que es diferente es la creación del campo **Clipboard clipbd**, lo que es realizado a través de **Toolkit**.

Todas las acciones tienen lugar en los listeners. Los listeners **CopyL** y **CutL** son los mismos excepto por la última línea de **CutL**, que borra la línea que ha sido copiada. Las dos líneas especiales son la creación de un objeto **StringSelection** del **String** y la llamada a **setContents()** con este **StringSelection**. Todo lo que está ahí es colocar un **String** en el portapapeles.

En **PasteL**, los datos son quitados del portapapeles utilizando **getContents()**. Lo que regresa es un objeto **Transferable** bastante anónimo, y no se sabe realmente que contiene. Una forma de saberlo es llamando a **getTransferFlavors()**, que retorna un arreglo de objetos **DataFlavor** indicando que sabores son soportados por este objeto en particular. Se puede también preguntarle directamente con **isDataFlavorSupported()**, pasando el sabor en el cual se esta interesado. Aquí, sin embargo una estrategia osada es utilizado: **getTransferData()** es llamado asumiendo que el contenido soporta el sabor de **String**, y si no lo hace el problema es solucionado en el manejador de excepciones.

En el futuro se puede esperar que mas sabores sean aceptados.

# Empaquetando un applet en un fichero JAR

Un importante uso de la utilizad JAR es optimizar la carga del applet. En Java 1.0, las personas se inclinaban a abarrotar todo el código en una sola clase así en cliente necesitaría solo un llamado al servidor para bajar el código del applet. No solo resulta en programas desordenados y difícil de leer (y mantener), sino que el fichero `.class` seguía sin comprimirse así es que la carga era mas lenta de lo que podría ser.

Los ficheros JAR solucionan el problema comprimiendo todos los ficheros `.class` en un solo fichero que es bajado por el navegador. Ahora se puede crear el diseño adecuado sin preocuparse por cuantos ficheros `.class` se van a generar, y el usuario obtendrá un tiempo de bajada mucho mas rápido.

Considere el **TicTacToe.java**. Se ve como una sola clase, pero de echo contiene cinco clases internas, así es que son seis en total. Una ves que se ha compilado el programa se empaqueta en un fichero JAR con la linea:

```
| jar cd TicTacToe.jar *.class  
Esto asume que el único fichero .class en el directorio actual son los  
generados de TicTacToe.java (de otra forma se tendrá un equipaje extra).
```

Ahora se puede crear una página HTML con la nueva etiqueta **archive** para indicar el nombre del fichero JAR. Aquí esta la etiqueta utilizando la fiche forma de la etiqueta HTML, como ilustración:

```
<head><title>TicTacToe Example Applet  
</title></head>  
<body>  
<applet code=TicTacToe.class  
archive=TicTacToe.jar  
width=200 height=100>  
</applet>  
</body>
```

Se deberá colocar dentro de la nueva (desordenada, complicada) forma mostrada temprano en este capítulo para hacer que trabaje.

## Técnicas de programación

Dado que la programación GUI en Java ha evolucionado tecnológicamente en algunos puntos significantes entre Java 1.0/1.1 y la librería Swing de Java 2, hay algunos idiomas de programación viejos que se han colado en los ejemplos que se pueden ver para Swing. Además, Swing permite programar en mas y mejores formas que las permitidas en los viejos modelos. En esta

sección, algunos de estos temas serán demostrados introduciendo y examinando algunos lenguajes de programación.

## Enlazando eventos dinámicamente

Unos de los beneficios del modelo de eventos de Swing es la flexibilidad. Se puede agregar y quitar comportamientos con una simple llamada a métodos. El siguiente ejemplo demuestra esto:

```
//: 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.bruceekel.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);
}
} //:~

```

Las cosas nuevas en este ejemplo son:

1. Hay mas de un listener enganchado a cada **Button**. Usualmente, el manejo de eventos de los componentes son *multicast*, lo que significa que se pueden registrar muchos listener para un solo evento. En los componentes especiales en los cuales un evento es manejado como *unicast*, se obtendrá un **TooManyListenersException**.
2. Durante la ejecución del programa, los listener son dinámicamente agregados y quitados del **Button b2**. Para agregar se utiliza lo que hemos visto anteriormente, pero cada componente tiene también un método **removeXXXListener()** para quitar cada tipo de listener.

Este tipo de flexibilidad proporciona mucho mas poder en su programación.

Se debe notar que no esta garantido que los listener de eventos sean llamados en el orden en que fueron agregados (a pesar de que la mayoría de las implementaciones hacen el trabajo de esta forma).

## Separando la lógica de negocios de la lógica UI

En general se querrá diseñar las clases de tal forma que ellas hagan “solo una cosa”. Esto es particularmente importante cuando afecta a el código de la interfase de usuario, dado que es fácil atar “lo que se esta haciendo” con “como se esta mostrando”. Este tipo de asociación impide la reutilización del código. Es mucho mas deseable separar la “lógica de negocios” de la GUI. De esta forma, no solo se puede reutilizar la lógica de negocios mucho mas fácilmente, también es mucho mas fácil de reutilizar la GUI.

Otro tema son los sistemas *multitiered*, donde los “objetos de negocios” residen en una máquina completamente separada. Esta localización centras de las reglas de negocios permiten que los cambios sean automáticamente efectivos para todas las nuevas transacciones, y de esta manera una imposición de configurar un sistema. Sin embargo, estos objetos de negocios pueden ser utilizados en muchas aplicaciones diferentes y no están atados a una forma particular de desplegarse. Estos deben realizar las operaciones de negocios y nada mas.

El siguiente ejemplo muestra cuan fácil es separar la lógica de negocios del código GUI:

```
//: 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 **BusinessLogic** es una clase directa que realiza las operaciones sin rastros de que pueda ser utilizadas en un ambiente GUI. Solo hace su trabajo.

**Separation** mantiene la pista sobre todos los detalles UI, y se comunica con **BusinessLogic** solo a través de su interfase pública. Todas las operaciones con centradas alrededor de obtener la información atrás y adelante a través de la UI y el objeto **BusinessLogic**. Así es que **Separation**, de hecho, solo

hace su trabajo. Dado que **Separation** conoce solo que se está comunicando con un objeto **BusinessLogic** (el cual, no está altamente asociado), se le pueden enviar mensajes para comunicarse con otros tipos de objetos sin muchos problemas.

Pensando en términos de separación de UI de la lógica de negocios también se hace más fácil cuando se adapta código legado para trabajar con Java.

## Una forma canónica

Las clases internas, el evento de modelos de Swing, y el hecho de que el viejo modelo de eventos sigue siendo soportado junto con las características de la nueva librerías que confía en el viejo estilo de programación ha agregado un nuevo elemento de confusión en el proceso de diseño de código.

Excepto en circunstancias extenuantes se puede siempre utilizar la más simple y clara estrategia: clases listener (típicamente escritas como clases internas) para solucionar las necesidades de manejo de eventos. Esta es la forma utilizada en la mayoría de los ejemplos en este capítulo.

Siguiendo este modelo se debería de ser capas de reducir las instrucciones en los programas que dicen: “Me pregunto que ha causado este evento”. Cada parte del código está preocupado por *hacer*, no por la verificación de tipo. Esta es la mejor forma de escribir el código; no solo es más fácil de visualizar, es mucho más fácil de leer y mantener.

# Programación visual y Beans

Mucho antes en este libro se ha visto que tan valioso es java para crear partes de código valiosas. La unidad de código “sumamente utilizable” había sido la clase, dado que comprende una consistente unidad de características (campos) y comportamientos (métodos) que pueden ser reutilizado directamente mediante composición o herencia.

Herencia y polimorfismo son partes esenciales de la programación orientada a objetos, pero en la mayoría de los casos cuando se están agrupando una aplicación, lo que realmente se quiere son componentes que hagan exactamente lo que se necesita. Se querrán tirar esas partes en el diseño como el ingeniero electrónico agrupa circuitos integrados y placas de circuitos. Parece, también, que debería haber alguna forma de acelerar este estilo de programación de “ensamblado modular”.

La “programación visual” es útil -muy útil- con Microsoft Visual Basic (VB), seguido de una segunda generación con Delphi de Borland (la inspiración primaria para el diseño de JavaBeans). Con estas herramientas de programación los componentes son representados visualmente, lo que tiene sentido dado que usualmente representan algún tipo de componente visual como un botón o un campo de texto. La representación visual, de hecho, es a menudo exactamente la forma en que el componente se verá en el programa que se ejecuta. Así es que parte del proceso de programación visual, involucra arrastrar un componente de una paleta y dejarlo en la hoja. La herramienta para armar aplicaciones escribe código mientras se hace esto, y este código producirá el componente a ser creado en el programa que se ejecuta.

Dejar el componente en la hoja usualmente no es suficiente para completar el programa. A menudo, se debe cambiar las características de un componente, como de que color es, que texto esta en el, que base de datos está conectada a él, etc. Las características que pueden ser modificadas en tiempo de diseño son referidas como *propiedades*. Se pueden manipular las propiedades del componente dentro de la herramienta para crear aplicaciones, y cuando se crea el programa esta configuración es guardada así es que puede ser cobrar nueva vida cuando el programa es ejecutado.

A esta altura probablemente se maneje la idea de que un objeto es más que estas características; es también un grupo de comportamientos. En tiempo de diseño, los comportamientos de los objetos visuales son parcialmente representadas por *eventos*, lo cual significa “Aquí hay algo que puede suceder con el componente”. Comúnmente, se decide que es lo que se quiere que suceda cuando un evento ocurre ligando código a ese evento.

Aquí hay una parte crítica: las herramientas para armar aplicaciones utilizan reflexión para interrogar al componente y encontrar qué propiedades y eventos soporta el componente. Una vez que conoce qué es, puede desplegar en pantalla las propiedades y permitir que se cambien (guardando el estado cuando se arma el programa), y también desplegar los eventos. En general, se hace algo como doble clic en un evento y la herramienta para armar aplicaciones crea un cuerpo de código y lo enlaza a ese evento en particular. Todo lo que se tiene que hacer en este punto es escribir el código que se ejecuta cuando del evento se sucede.

La herramienta de creación de aplicación agrega un montón de trabajo terminado a el trabajo. Como resultado nos podemos enfocar en como se ve el programa y qué supuestamente hace, y confiar en que la herramienta de creación de aplicaciones maneje los detalles de conexión por nosotros. La razón por la cual las herramientas de programación visual han sido tan exitosas es que aceleran dramáticamente el proceso de creación de una aplicación -claro que la interfase de usuario, pero a menudo también otras partes de la aplicación.

## ¿Que es un Bean?

Luego de aclarar el ambiente, entonces, un componente es realmente solo un bloque de código, típicamente personificado en una clase. El tema clave es la habilidad de la herramienta de creación de aplicaciones para descubrir las propiedades y eventos para ese componente. Para crear un componente de VB, el programador tiene que escribir un pedazo de código bastante complicado siguiendo ciertas convenciones para exponer las propiedades y eventos. Delphi fue una segunda generación de herramienta de programación visual y el lenguaje fue activamente diseñado girando en torno a la programación visual así es que es mucho mas fácil crear un componente. Sin embargo, Java trae la creación de componentes visuales a un estado mas avanzado con JavaBeans, dado que un Bean es simplemente una clase. No se tiene que escribir código extra o utilizar extensiones especiales del lenguaje para hacer que algo sea un Bean. La única cosa que se necesita hacer, de hecho, es modificar ligeramente la forma en que se nombran los métodos. Es el nombre del método el que le indica a la herramienta de creación de aplicaciones cuando es una propiedad, un evento, o solo un método común.

En la documentación de Java, esta convención de nombres es erróneamente llamada “patrones de diseño”. esto es desafortunado, dado que los diseños de patrones (vea *Thinking in Patterns with Java* que se puede bajar en [www.BruceEckel.com](http://www.BruceEckel.com)) son suficientemente provocativos sin este tipo de confusión. Esto no es un diseño de patrones, solo en una convención de nombres y es bastante simple:

1. Para una propiedad llamada **xxx**, típicamente se crean dos métodos: **getXxx()** y **setXxx()**. Debe notarse que la primera letra luego del “get” o “set” es automáticamente llevada a minúsculas para producir el nombre de la propiedad. El tipo producido por el método “get” es el mismo que el tipo de al argumento del método “set”. El nombre de la propiedad y el tipo del “get” y del “set” no están relacionados.
2. Para una propiedad **boolean**, se pueden utilizar el la estrategia “get” y el “set” mas arriba, pero se puede utilizar “is” en lugar de “get”.
3. Comúnmente los métodos del Bean no se amoldan a las convenciones de nombres arriba, pero son públicos.
4. Para eventos, se usa la estrategia del “listener” Swing. Esto es exactamente la misma que se ha visto:  
**addFooBarListener(FooBarListener)** y  
**removeFooBarListener(FooBarListener)** para manejar un evento **FooBarEvent**. La mayoría del tiempo los eventos y listener incluidos en el lenguaje satisfacen las necesidades, pero se puede también crear sus propios eventos e interfaces listener.

El punto 1 mas arriba responde una pregunta acerca de algo que se nota cuando se observa el viejo código contra el nuevo código: una cantidad de nombres de métodos habían tenido pequeños, cambios aparentemente sin sentido. Ahora se puede ver que la mayoría de los cambios se tuvieron que hacer para adaptarse a las convenciones de nombres para los “get” y “set” y convertir un componente en particular en un Bean.

Se pueden utilizar estas guías para crear un simple Bean:

```
//: 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 jmps;
    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 jmps; }
    public void setJumper(boolean j) { jmps = 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!");
    }
} ///:~
```

Primero se puede ver que es simplemente una clase. Usualmente, todos los campos serán privados, y accesibles solo a través de los métodos. Siguiendo la convención de nombres, las propiedades son **jumps**, **color**, **spots**, y **jumper** (debe notarse el cambio de mayúscula a minúscula en la primer letra del nombre de propiedad). A pesar de que el nombre del identificador interno es el mismo que el nombre de la propiedad en los primeros tres casos, **en jumper** se puede ver que el nombre de la propiedad no está forzada a utilizar ningún identificador en particular para variables internas (o, ciertamente, incluso *tener* alguna variable interna para esa propiedad).

Los eventos manejados en este Bean son **ActionEvent** y **KeyEvent**, basado en el nombre de los métodos “add” y “remove” para el listener asociado. Finalmente, se puede ver que el método común **croak()** sigue siendo parte del Bean simplemente porque es un método público, no porque conforme ningún esquema de nombres.

## Extrayendo **BeanInfo** con el **Introspector**

Una de las partes más críticas del esquema de un Bean sucede cuando se arrastra un Bean de una paleta y se coloca dentro de una hoja. La herramienta para armar aplicaciones debe ser capaz de crear el Bean (lo que puede hacer si hay un constructor por defecto) y entonces, sin acceso a el código fuente del Bean, extraer toda la información necesaria para crear la tabla de propiedades y los manejadores de eventos.

Parte de la solución ya es evidente desde el final del capítulo 12: La *reflexión* de Java permite que todos los métodos de una clase anónima sean descubiertos. Esto es perfecto para solucionar el problema de los Beans sin requerir que se utilice palabras extras del lenguaje como se requieren en otros lenguajes de programación visuales. De hecho, una de las principales razones por las cuales fue agregada la reflexión a Java fue para soportar Beans (aunque la reflexión soporta también la serialización de objetos y la invocación remota de métodos). Así es que se puede esperar que el creador de la herramienta para crear aplicaciones tenga que reflexionar cada Bean y buscar entre los métodos para encontrar las propiedades y los eventos para cada Bean.

Naturalmente esto es posible, pero los diseñadores de Java quisieron proporcionar una herramienta estándar, no solo para hacer los Beans más simples de utilizar, también para proporcionar una puerta estándar para la creación de Beans más complejos. Esta herramienta es la clase **Introspector**, y el método más importante de esta clase es el **static getBeanInfo()**. Se pasa una referencia a una clase a este método y esta interroga totalmente la clase

y retorna un objeto **BeanInfo** que se puede entonces analizar detenidamente para encontrar las propiedades, métodos y eventos.

Usualmente no hay que preocuparse por esto -probablemente la mayoría de los Bean salgan de estanterías de vendedores, y no se necesita conocer toda la magia atrás. Simplemente se arrastran los Bean en la hoja, luego se configuran las propiedades y se escriben manejadores para los eventos en los cuales se está interesado. Sin embargo, es un ejercicio interesante y educacional utilizar el **Introspector** para desplegar información acerca de un Bean, así es que aquí hay una herramienta que hace esto:

```
//: 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.bruceekel.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].get.PropertyType();
            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(p, BorderLayout.NORTH);
}

```

```

        cp.add(new JScrollPane(results));
        Dumper d_mpr = 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);
    }
} //:~

```

**BeanDumper.dump()** es el método que hace todo el trabajo. Primero trata de crear un objeto **BeanInfo**, y si lo logra llama a los métodos de **BeanInfo** que producen información acerca de las propiedades, métodos y eventos. En **Instrospector.getBeanInfo()**, se podrá ver que hay un segundo argumento. Esto le indica a el **Instrospector** donde parar en la jerarquía de herencia. Ahí, se detiene antes de que analice todos los métodos de **Object**, dado que no estamos interesados en verlos a todos.

Para las propiedades, **getPropertyDescriptors()** retorna un arreglo de **PropertyDescriptors**. Para cada **PropertyDescriptor** se puede llamar a **get.PropertyType()** para encontrar la clase de objeto que se esta pasando mediante los métodos de propiedades. Entonces, para cada propiedad se puede obtener su seudónimo (extraido de los nombres de métodos) con **getName()**, el método para leer con **getReadMethod()**, y el método para escribir con **getWriteMethod()**. Estos dos métodos retornan un objeto **Method** que puede de hecho ser utilizado para invocar a el método correspondiente en el objeto (esto es parte de la reflexión).

Para los métodos públicos (incluyendo los métodos de propiedades), **getMethodDescriptors()** retornan un arreglo de **MethodDescriptors**. Para cada uno se puede tener el objeto **Method** asociado e imprimir su nombre.

Para los eventos, **getEventSetDescriptors()** retornan un arreglo de (¿que mas?) **EventSetDescriptors**. Cada uno de estos puede ser consultado para encontrar la clase del listener, los métodos de esa clase listener, y los métodos para agregar y quitar listener. El programa **BeanDumper** imprime toda esta información.

Luego de arrancar, el programa fuerza la evaluación de **frogbean.Frog**. La salida, luego de quitar los detalles extra que son innecesarios aquí, 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 el **Instrospector** ve cuando produce un objeto **BeanInfo** del Bean. Se puede ver que el tipo de propiedad y su nombre son independientes. Deben notarse las minúsculas del nombre de la propiedad (El único momento donde no sucede esto es cuando el nombre de propiedad comienza con mas que una letra mayúscula en una fila). Y se debe recordar que los nombres de los métodos que se están viendo aquí (como los métodos de lectura y escritura) son producidos actualmente de un objeto **Method** que puede ser utilizado para invocar el método asociado a el objeto.

La lista de métodos públicos incluyen los métodos que no están asociados con una propiedad o evento, como lo es **croak()**, de la misma forma que aquellas que lo están. Estos son todos los métodos que se puede llamar desde un programa para un Bean, y la herramienta para crear aplicaciones puede elegir una lista de todos estos mientras se están haciendo las llamadas a métodos, para alivianar su tarea.

Finalmente, se puede ver que el análisis de los eventos son mostrados en su totalidad en el listener, sus métodos, y los métodos para agregar y quitar métodos. Básicamente, una vez que se tiene el **BeanInfo**, se puede encontrar todo lo de importancia para el Bean. Se puede también llamar a los métodos para ese Bean, a pesar de que no se tenga ninguna otra información que no sea el objeto (nuevamente, una característica de la reflexión).

## Un Bean mas sofisticado

El siguiente ejemplo es ligeramente mas sofisticado, a pesar de su frivolidad. Es un **JPanel** que dibuja un pequeño círculo alrededor del ratón cuando el ratón es movido. Cuando se presiona el ratón, la palabra “Bang!” aparece en el medio de la pantalla, y un listener de acción es disparado.

Las propiedades que se pueden cambiar son el tamaño del círculo así como el color, tamaño y texto de la palabra que es mostrada cuando se presiona el ratón. Un **BangBean** también tiene su propio **addActionListener()** y **removeActionListener()** así es que se puede enganchar su propio listener que será disparado cuando el usuario haga un clic en el **BangBean**. Será posible reconocer la propiedad y el evento soportado:

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

```

La primera cosa que se notará es que **BangBean** implementa la interfase **Serializable**. Esto significa que la herramienta para crear aplicaciones puede “conservar en vinagre” toda la información para el **BangBean** utilizando serialización luego de que el diseñador del programa ha ajustado los valores de las propiedades. Cuando el Bean es creado como parte de la aplicación ejecutable, estas propiedades “conservadas en vinagre” son almacenadas así es que se tiene exactamente lo que se ha diseñado.

Se puede ver que todos los campos son privados, lo que es normalmente hecho con un Bean -permitir el acceso a través de métodos, usualmente utilizando el esquema de “propiedad”.

Cuando se observa la firma para **addActionListener()**, se verá que puede lanzar un **TooManyListenersException**. Esto indica que es *unicast*, lo que significa que solo alerta a un listener cuando el evento se sucede.

Normalmente, se utilizarán eventos *multicast* así es que muchos listeners pueden ser alertados cuando con un evento. Sin embargo, esto entra en temas para los cuales no estaremos listos hasta el siguiente capítulo, así es que será vuelto a ver allí (bajo el título “revisión de JavaBeans”). Un evento unicast soslaya el problema.

Cuando se hace clic en el ratón, el texto es colocado en el medio del **BangBean**, y si el campo **actionListener** no es **null**, **actionPerformed()** es llamado, creando un objeto **ActionEvent** en el proceso. Dondequiero que el ratón sea movido, sus nuevas coordenadas son capturadas y el lienzo es vuelto a dibujar (borrando texto prueba en el lienzo, como es verá).

Aquí esta la clase **BangBeanTest** que permite verificar el Bean como applet o como 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 se encuentra en un ambiente de desarrollo, esta clase no será utilizado, pero es útil para proporcionar un método rápido de prueba para cada uno de sus Beans. **BangBeanTest** coloca un **BangBean** dentro del applet, enganchando un **ActionListener** simple a el **BangBean** para imprimir un contador de evento a el **JTextField** donde sea que un **ActionEvent** se sucede. Usualmente claro, la herramienta creadora de aplicación creará la mayoría del código que utiliza el Bean.

Cuando se ejecute el **BangBean** a través de **BeanDumper** o colocando el **BangBean** dentro de un ambiente de desarrollo que habilite Bean, se notará que hay muchas mas propiedades y acciones de las que son evidentes den el código anterior. Esto es porque **BangBean** es heredado de **JPanel**, y **JPanel** es también un Bean, así es que se están viendo sus propiedades y eventos también.

## Empaquetando un Bean

Antes de que se pueda traer un Bean a una herramienta de desarrollo visual habilitada para Beans, este debe ser colocado en un contenedor estándar para Beans, que es un fichero JAR que incluye todas las clases del Bean de la misma forma que un fichero "manifiesto" que dice "Esto es un Bean". Un fichero manifiesto es simplemente un fichero de texto que sigue una forma particular. Para en **BangBean**, el fichero de manifiesto se ve como esto (sin las primera y última linea):

```
/:! :BangBean.mf
Manifest-Version: 1.0
Name: bangbean/BangBean.class
Java-Bean: True
//:~
```

La primer linea indica la versión del esquema de manifiesto, el cual hasta nuevo aviso de Sun es 1.0. La segunda linea (las líneas vacías son ignoradas) nombra la clase **BangBean.class**, y la tercera dice, "Es un Bean". Sin tercera linea, la herramienta para armar programas no reconocerá la clase como un Bean.

La única parte trampa es que hay que asegurarse que se tiene la ruta apropiada en el campo "Name:". Si no se mira hacia atrás en **BangBean.java**, se verá que esta en un **package bangbean** (y de esta forma en un subdirectorío llamado "bangbean" que esta fuera de la ruta de clases), y el nombre en el fichero manifiesto debe incluir esta información de paquete. Además, se debe colocar el fichero manifiesto en el directorio *arriba* del raíz de su ruta de paquete, que en este caso significa colocar el fichero en el directorio arriba del subdirectorío "bangbean". Entonces se debe invocar **jar** de el mismo directorio que el fichero manifiesto, como sigue:

```
| jar cfm BangBean.jar BangBean.mf bangbean
```

Esto asume que se quiere nombrar el fichero JAR resultante como **BangBean.jar** y que se ha puesto el manifiesto en un fichero llamado **BanBean.mf**.

Nos preguntaremos “¿Que sucede con todas las otras clases que fueron generadas cuando se compilo **BangBean.java**?” Bueno, todas ellas terminaron dentro del subdirectorio **bangbean**, y se vera que el último argumento para el **jar** en la línea de comandos es el subdirectorio **bangbean**. Cuando se da el nombre de un subdirectorio a el **jar**, este empaqueta el subdirectorio entero dentro del fichero jar (incluyendo, en este caso, el fichero fuente original de **BangBean.java** -se puede no elegir incluir el código fuente con el Bean). Además, si se vuelve a desempacar el fichero JAR que se ha creado, se descubrirá que el fichero manifiesto no esta dentro, pero que **jar** ha creado su propio fichero manifiesto (basado en parte en el original) llamado **MANIFEST.MF** y lo ha colocado dentro del subdirectorio **META-INF** (para “meta-información”). Si se abre este fichero manifiesto se notará que la información de firma digital ha sido agregada por **jar** para cada fichero, de la forma:

```
Digest-Algorithms: SHA MD5  
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=  
MD5-Digest: O4Ncs1hE3SmnZlp2hj6qeg==
```

En general no necesitamos preocuparnos de esto, y si se hacen cambios se puede simplemente modificar es manifiesto original y llamar nuevamente a **jar** para crear un nuevo fichero JAR para su Bean. Se puede también agregar otros Beans a el fichero JAR simplemente agregando la información a su manifiesto.

Una cosa para notar es que probablemente se quiera colocar cada Bean en su propio subdirectorio, dado que cuando se crea el fichero JAR se pasa a la utilidad **jar** el nombre de un subdirectorio y este coloca todo en ese subdirectorio dentro del fichero JAR. SE puede ver que **Frog** y **BangBean** están en sus propios subdirectorios.

Una ves que se tiene un Bean propiamente dentro de un fichero JAR se puede traer a un ambiente para crear programas habilitado para Beans. La forma para hacer esto varía de una herramienta a la otra, pero Sun proporciona una mesa de prueba disponible libremente para sus JavaBeans en su “Equipo de desarrollo de Beans” (BDK Beans Development Kit) llamada o el “benbox” (Se puede bajar el BDK de [java.sun.com/beans](http://java.sun.com/beans)). Para colocar su Bean en el beanbox, copie el fichero JAR dentro del subdirectorio BDK’s “jars” antes de comenzar el beanbox.

## Soporte de Bean mas complejo

Se puede ver lo notablemente simple que es crear un Bean. Y no estamos limitados a lo que hemos visto aquí. La arquitectura proporciona un simple

punto de entrada y se puede también subir a situaciones mas complejas. Estas situaciones están mas allá del alcance este libro, pero serán amigablemente introducidas aquí. Se pueden encontrar mas detalles en [java.sun.com/beans](http://java.sun.com/beans)

Un lugar donde se pueden agregar sofisticaciones es con propiedades. Los ejemplos anteriores mostraron propiedades simples, pero es posible también representar múltiples propiedades en un arreglo. Esto es llamado una *propiedad indexada*. Simplemente se proporciona los métodos apropiados (nuevamente seguido de una convención para los nombres de métodos) y el **Instrospector** reconoce una propiedad indexada así su herramienta creadora puede responder apropiadamente.

Las propiedades pueden ser *ligadas* lo que significa que pueden notificar a otros objetos mediante un **PropertyChangeEvent**. Los otros objetos pueden entonces elegir cambiar ellos mismos basados en el cambio del Bean.

Las propiedades pueden ser *forzadas* lo que significa que otros objetos pueden vedar un cambio a esa propiedad si esto es inaceptable. Los otros objetos son notificados utilizando un **PropertyChangeEvent**, y estos pueden lanzar una **PropertyVetoException** para prevenir que el cambio suceda y restaurar los viejos valores.

Se puede también cambiar la forma en que su Bean es representado en tiempo de diseño:

1. Se puede proporcionar una hoja de propiedad a medida para un Bean en particular. La hoja propiedad común será utilizada para todos los otros Beans, pero una a medida puede ser automáticamente invocada cuando su Bean es seleccionado.
2. Se puede crear un editor a medida para una propiedad en particular, así es que la hoja de propiedad común es utilizada, pero cuando la propiedad especial es editada, el editor a medida será automáticamente invocado.
3. Se puede proporcionar una clase **BeanInfo** a medida para su Bean que produzca información diferente del creado por defecto por el **Instrospector**.
4. Es también posible cambiar a modo “experto” y activar o desactivar todas las **FeatureDescriptor** para distinguir entre características básicas y mas complicadas.

## Mas de Beans

Hay otro tema que no puede ser hablado aquí. Cuando quiera que se cree un Bean, se puede esperar que se ejecute en un ambiente multitarea. Esto

significa que se debe entender los temas de los hilos, que serán introducidos en el capítulo 14. Se encontrará una sección llamada “revisión de JavaBeans” en los cuales se verá el problema y su solución.

Hay un gran número de libros acerca de JavaBeans; por ejemplo, *JavaBeans* de Elliotte Rusty Harold (IDG, 1998).

## Resumen

De todas las librerías en Java, la librería GUI ha presenciado el cambio más dramático de Java 1.0 a Java 2. La AWT de Java 1.0 fue criticada sin rodeos como uno de los peores diseños vistos, y a pesar de que permita crear programas portátiles, la GUI resultante fue “igualmente mediocre en todas las plataformas”. Esto fue también limitante, complicado, y desagradable de utilizar comparado con las herramientas de desarrollo nativas existentes en una plataforma en particular.

Cuando Java 1.1 introdujo el nuevo modelo de eventos y los JavaBeans, la escena fue asentada -ahora es posible crear componentes GUI que se pueden arrastrar y dejar caer fácilmente dentro de herramientas visuales para crear aplicaciones . Además, el diseño del modelo de eventos y Beans, claramente muestra una fuerte consideración por facilitar la programación y el mantenimiento del código (a veces esto no era evidente en la AWT 1.9), Pero no fue hasta que apareció la JFC/Swing que el trabajo haya sido terminado. Con los componentes Swing, la programación GUI multiplataforma puede ser una experiencia civilizada.

Actualmente, la única cosa que falta es la herramienta para crear GUI, y ahí es donde la verdadera revolución descansa. Visual Basic de Microsoft y Visual C++ necesitan de herramientas creadoras de aplicaciones, y se ocuparon de Delphi de Borland y el creador de C++. Si se quiere una herramienta de creación de aplicaciones mejore, se tiene que cruzar los dedos y esperar que el vendedor le de lo que quiere. Pero Java es un ambiente abierto, y de esta forma no solo permite la competencia de ambientes de creación, los mejora. Y para que esas herramientas sean tomadas seriamente, deben soportar JavaBeans. Esto significa un campo de juego parejo: si una mejor herramienta para crear aplicaciones está saliendo, no se está atado a la que ha estado utilizando -se puede adquirir y mover a la nueva e incrementar la productividad. Este tipo de ambiente competitivo para herramientas para crear aplicaciones GUI no ha sido visto antes, y el mercado resultante puede generar solo resultados positivos para la productividad del programador.

Este capítulo trata de dar solo una introducción a el poder de Swing y como iniciación así se puede ver que tan simple es encontrar el camino a través de las librerías. Lo que se ha visto hasta ahora probablemente satisfaga una

gran parte de sus necesidades de diseño UI. Sin embargo, hay mucho más de Swing -pretende ser un poderoso conjunto de herramientas de diseño UI completo. Hay una forma de lograr exactamente todo lo que se puede imaginar.

Si no se ve lo que se necesita aquí, explore la documentación en línea de Sun y busque en la Web, y si no es suficiente entonces encuentre un libro dedicado a Swing -un buen lugar para comenzar e *The JFC Swing Tutorial* de Walrath & Campione (Addison Wesley, 1999).

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Cree un applet/aplicación utilizando clase **Console** como es mostrado en este capítulo. Incluya un campo de texto y tres botones. Cuando se presione cada botón, haga que algún texto diferente aparezca en campo de texto.
2. Agregue una casilla de verificación para el applet creado en el Ejercicio 1, capture el evento, e inserte un texto diferente dentro del campo de texto.
3. Cree un applet/aplicación utilizando **Console**. En la documentación HTML en *java.sun.com*, encuentre el **JPasswordField** y agregue esto a el programa. Si el usuario escribe la palabra clave correcta, use **JOptionPane** para proporcionar un mensaje para indicarle esto al usuario.
4. Cree un applet/aplicación utilizando **Console**, y agregue todos los componentes que tienten un método **addActionListener()** (Busque estos en la documentación HTML en *java.sun.com*. Sugerencia: utilice el índice). Capture estos eventos y despliegue un mensaje apropiado para cada uno dentro de un campo de texto.
5. Cree un applet/aplicación utilizando **Console**, con un **JButton** y un **JTestFiel**. Escriba y enganche el listener apropiado de tal forma que si el botón tiene el foco, los caracteres escritos en el aparecerán en el **JTextField**.
6. Cree una applet/aplicación utilizando **Console**. Agregue en el marco principal todos los componentes descritos en este capítulo, incluyendo los menús y una caja de diálogo.
7. Modifique **TextFields.java** para que los caracteres en **t2** retengan las mayúsculas y minúsculas originales de cuando fueron escritos, en lugar de automáticamente sean forzadas a mayúsculas.

8. Encuentre y baje uno o mas ambientes de desarrollo GUI disponibles libremente en la Internet, o compre un producto comercial. Descubra que es necesario para agregar **BangBean** a este ambiente y que es necesario para utilizarlo.
9. Agregue **Frog.class** a el fichero manifiesto como se muestra en este capítulo y ejecute **jar** para crear un fichero JAR que contenga **Frog** y **BangBean**. Ahora baje e instale la BDK de Sun o use su propia herramienta creadora de programas con Beans habilitados y agregue el fichero JAR a su ambiente de tal forma que pueda probar los dos Beans.
10. Cree su propio JavaBean llamado **Valve** que contenga dos propiedades: un **boolean** llamado “encendido” y un **int** llamado “nivel”. Cree un fichero manifiesto, utilice **jar** para empaquetar el Bean, luego cárguelo dentro de la caja de Beans o dentro de la herramienta para crear programas con los Beans habilitados así se puede probar.
11. Modifique **MessageBoxes.java** de tal forma que haya un **ActionListener** individual para cada botón (en lugar de hacer coincidir el texto del botón).
12. Realice un nuevo monitoreo de un evento en **TrackEvent.java** agregando el código nuevo de manejo de eventos. Necesitará descubrir por su mismo el tipo de evento que se quiere monitorear.
13. Herede un nuevo tipo de botón de **JButton**. Cada ves que se presione este botón, deberá cambiar su color a un valor seleccionado al azar. Vea **ColorBoxes.java** en el capítulo 14 por un ejemplo de como generar un valor de color aleatorio.
14. Modifique **TextPane.java** para utilizar un **JTextArea** en lugar de un **JTextPane**.
15. Modifique **Menus.java** para utilizar botones de radio en lugar de casillas de verificación en los menús.
16. Simplifique **List.java** pasando el arreglo a el constructor y eliminando el agregado dinámico de elementos a la lista.
17. Modifique **SineWave.java** para convertir a **SineDraw** en un JavaBean agregando los métodos “getter” y “setter”.
18. ¿Recuerda el juego “para hacer bocetos” con dos perillas, una que controla el movimiento vertical del punto de dibujo, y una que controla el movimiento horizontal? Cree una de estas, utilizando **SineWave.java** como inicio. En lugar de perillas, utilice deslizadores. Agregue un botón que borre el bosquejo entero.

19. Cree un “indicador de progreso asintótico” que se torne mas lento a medida que se aproxime a el punto de finalización. Agregue un comportamiento aleatorio errático así se puede periódicamente ver como comienza a acelerarse.
20. Modifique **Progress.java** de tal forma que no comparta modelos, pero en lugar de eso utilice un listener para conectar al deslizador y la barra de progreso.
21. Siguiendo las instrucciones de la sección titulada “Empaquetando un applet en un fichero JAR” para colocar **TicTacToe.java** dentro de un fichero JAR. Cree una página HTML con la (desordenada y complicada) versión de la etiqueta applet, y modifíquela para que utilice la etiqueta de archivo para el fichero JAR. (Sugerencia: comience con la página HTML, para **TicTacToe.java** que viene con el código fuente distribuido con este libro).
22. Cree un applet/aplicación utilizando **Console**. Este debe tener tres deslizadores, uno para cada uno de los valores rojo, verde y azul en **java.awt.Color**. El resto de la hoja debe ser un **JPanel** que muestre el color determinado por los tres deslizadores. También incluya un campo de texto que no se pueda editar que muestre los valores RGB actuales.
23. En la documentación HTML para **java.swing**, busque el **JColorChooser**. Escriba un programa con un botón que traiga el color elegido como diálogo.
24. Al menos cada componente Swing esta derivado de **Component**, que tiene un método **setCursor()**. Búsquelo en la documentación HTML. Cree un applet y cambie el cursor a uno de los cursos existentes en la clase **Cursor**.
25. Comience con **ShowAddListeners.java**, cree un programa con la funcionalidad completa de **ShowMethodsClean.java** del capítulo 12.

# 14: Hilos múltiples

Los objetos proporcionan una forma de dividir un programa en secciones independientes. A menudo, se puede necesitar separar un programa en tareas secundarias que se ejecuten independientemente.

Cada una de estas tareas secundarias es llamada *hilo*, y se programan como si cada hilo se ejecuta solo y como si la CPU fuera una sola. Algun mecanismo de capas mas bajas esta dividiendo el tiempo de la CPU para nosotros, pero en general, no hay que pensar en esto, lo que hace la programación de múltiples hilos una tarea mucho mas fácil.

Un *proceso* es un programa que se contiene a si mismo ejecutándose con su propio espacio de direcciones. Un sistema operativo *multitarea*es capas de ejecutar mas de un proceso (programa) a la vez, haciendo que parezca que cada uno esta por su cuenta, proporcionando periódicamente ciclos de CPU para cada proceso. Un hilo es un solo flujo de control dentro de un proceso. Un solo proceso puede tener múltiples hilos ejecutándose al mismo tiempo.

Hay muchos usos posibles para la multitarea, pero en general, se tendrá una parte de un programa colgado por un evento en particular o recurso, y no se quiere colgar el resto del programa a causa de esto. Así es que se crea un hilo asociado con ese evento o recurso y se deja correr independientemente del programa principal. Un buen ejemplo es un botón de “salida” -no se quiere estar forzado a sondear el botón de salida en cada parte de código que se escribe y de la misma forma se quiere que tenga una buena respuesta, como si *fuerá*verificado regularmente. De hecho, uno de las razones convincentes de forma inmediata para la multitarea es producir una interfase de buena respuesta.

## Interfases de usuario de buena respuesta

Como punto de partida, considere un programa que realice alguna operación de CPU intensiva y de esta forma termina ignorando la entrada de usuario y dejando de reaccionar. Esta, un applet/aplicación, simplemente desplegará el resultado de un contador ejecutándose:

```
//: 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.bruceekel.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 referente a el applet y a Swing debe ser razonablemente familiar del Capítulo 13. El método **go()** es donde el programa se queda ocupado: se coloca el valor actual de **count** en el **JTextField t**, luego se incrementa **count**.

Parte del bucle infinito dentro de **go()** es llamar a **sleep()**. **sleep()** debe ser asociado con un objeto **Thread**, y nos muestra que toda aplicación tiene

*algún* hilo asociado con ella (Efectivamente, Java está basado en hilos y siempre hay alguno ejecutándose con la aplicación). Así es que independientemente de si se está explícitamente utilizando hilos, se puede obtener el hilo utilizado por el programa con **Thread** y el método estático **sleep()**.

Debe notarse que **sleep()** puede lanzar una **InterruptedException**, apesar de que lanzar algo como una excepción es considerado una forma hostil de cortar un hilo y debe ser desalentado (Una vez mas las excepciones son para condiciones excepcionales, no para el control normal del flujo). Interrumpir un hilo dormido está incluido para soportar una característica futura del lenguaje.

Cuando el botón **start** es presionado, **go()** es invocado. Examinando **go()**, se puede pensar inocentemente (como he hecho) que permitirá multitarea porque se pone a dormir. Esto es, mientras el método es dormido, parece como que la CPU pueda estar ocupada realizando el monitoreo de acciones sobre otros botones. Pero vemos que el problema es que **go()** nunca retorna, dado que es un bucle infinito, y esto significa que **actionPerformed()** nunca retorna. Dado que esta trancado dentro de **actionPerformed()** por haber presionado la tecla por primera vez, el programa no puede manejar ningún otro evento (Para salir, de debe de alguna forma matar el proceso; la forma mas fácil de hacer esto es presionando las teclas Control-C en la ventana de consola, si se ha iniciado desde la consola. Si se ha iniciado desde el navegador, se tendrá que matar la ventana del navegador).

Básicamente el problema aquí es que **go()** necesita continuar realizando sus tareas, y al mismo tiempo se necesita retornar así **actionPerformed()** puede completarse y la interfase de usuario puede continuar respondiendo al usuario. Pero en un método convencional como **go()** no se puede continuar y al mismo tiempo retornar el control a el resto del programa. Esto suena como una cosa imposible de lograr, como si la CPU puede estar en dos lugares a la vez, y esto es precisamente la ilusión que la multitarea proporciona.

El modelo de hilado (y su soporte de programación en Java) es una conveniencia de programación para simplificar el hacer malabares con muchas operaciones en el mismo momento con un solo programa. Con el hilado, la CPU va saltando de un lado a otro y le da a cada hilo algo de su tiempo. Cada hilo tiene el sentido de constantemente tener la CPU para si solo, pero el tiempo de CPU es dividido entre todos los hilos. La excepción de esto es si el programa está ejecutando en múltiples CPUs. Pero una de las grandes cosas acerca del hilado es que uno se abstrae de esta capa, así es que su código no necesita saber cuando está ejecutándose en una sola CPU o en muchas. De esta forma, los hilos son una forma de crear programas escalables transparentemente.

El hilado reduce un poco la eficiencia de computación, pero la mejora obtenida en el diseño del programa, el balance de recursos, y la conveniencia del usuario es realmente valioso. Claro, si se tiene mas de una CPU, el sistema operativo puede dedicar cada CPU a un grupo de hilos o incluso a un solo hilo y la totalidad del programa puede correr mucho mas rápido. La multitarea y el multihilado tiende a ser las formas mas razonables de utilizar sistemas con múltiples procesadores.

## Heredando de **Thread**

La forma mas simple de crear un hilo es heredando de la clase **Thread**, que tiene todo lo necesario para crear y ejecutar hilos. El método mas importante para **Thread** es **run()**, el que se debe sobrecargar para hacer que el hilo haga lo que se quiere. De esta forma, **run()** es el código que se ejecutará “simultáneamente” con los otros hilos en un programa.

El siguiente ejemplo crea cualquier número de hilos que son rastreados asignándole a cada hilo un número único, generado con una variable estática. El método **run()** de **Thread** es sobrecargado para realizar una cuenta regresiva cada vez que se pasa su bucle y para finalizar cuando el contador es cero (en el punto en que **run()** retorna, el hilo es terminado).

```
//: 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()** virtualmente siempre tiene un tipo de bucle que continúa hasta que el hilo no es mas necesario, así es que se debe establecer la condición en donde se sale de el bucle (o, en el caso anterior, simplemente se retorna de **run()** con una instrucción **return**). A menudo, **run()** se da en la

forma de un bucle infinito, que significa que, salvo que algún factor externo cause que se termine, continuará para siempre.

En el **main()** se puede ver una gran cantidad de hilos que son creados y ejecutados. El método **start()** en la clase **Thread** realiza inicializaciones especiales para el hilo y luego llama a **run()**. Así es que los pasos son: el constructor es llamado para crear el objeto, luego **start()** configura el hilo y llama a **run()**. Si no se llama a **start()** (lo que se puede hacer en el constructor, si es apropiado) el hilo nunca comenzará.

La salida para una corrida de este programa (será diferente de una corrida a otra) 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 notará que en ninguna parte de este ejemplo se llama a **sleep()**, y a pesar de eso la salida indica que cada hilo obtiene una porción de tiempo de CPU en donde ejecutarse. Esto muestra que **sleep()**, mientras confía en la existencia de un hilo para ejecutarse, no está involucrado con la habilitación o inhabilitación del hilado. Es simplemente otro método.

Se puede también ver que los hilos no se ejecutan en el orden que fueron creados. Efectivamente, el orden que la CPU atiende a un grupo de hilos

existentes es indeterminado, a no ser que se ajusten las prioridades utilizando el método **setPriority()** de **Thread**.

Cuando **main()** crea los objetos **Thread** no esta capturando las referencias a ninguno de ellos. Un objeto común puede ser un objeto de risa para la recolección de basura, pero no un **Thread**. Cada **Thread** se “registra” a si mismo así es que hay una referencia de el en alguna parte y el recolector de basura no puede limpiarlo.

## Hilado para una interfase que responda

Ahora es posible solucionar el problema de **Counter1.java** con un hilo. El truco es localizar la tarea -esto es, el bucle que esta dentro de **go()**- dentro del método **run()** de un hilo. Cuando el usuario presiona el botón **start**, el hilo es iniciado, pero luego la *creación* del hilo se completa, así es que a pesar de que el hilo esta ejecutándose, el trabajo principal del programa (observar por y responder a los eventos de la interfase de usuario) puede continuar. Aquí esta 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.bruceekel.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);
    }
} //:~

```

**Counter2** es un programa directo, cuyo trabajo solo es configurar y mantener la interfase de usuario. Pero ahora, cuando el usuario presiona el botón **start**, el código de manejo de eventos no llama un método. En lugar de eso un hilo de la clase **SeparateSubtask** es creado, y entonces el bucle de evento **Counter2** continúa.

La clase **SeparateSubTask** es una simple extensión de **Thread** con un constructor que ejecuta el hilo llamando a **start()**, y un **run()** que esencialmente contiene el código “**go()**” de **Counter1.java**.

Dado que **SeparateSubTask** es una clase interna, puede directamente acceder a **JTextField t** en **Counter2**; se puede ver que esto sucede dentro de **run()**. El campo **t** en la clase externa es privada dado que **SeparateSubTask** puede acceder a ella sin ningún permiso especial -y esto siempre es bueno para hacer los campos “tan privados como sea posible” así no pueden ser cambiados accidentalmente por fuerzas fuera de la clase.

Cuando se presiona el botón **onOff** cambia la **runFlag** dentro del objeto **SeparateSubTask**. Este hilo (cuando se fija en la bandera) puede entonces comenzar y parar solo. Presionando el botón **onOff** produce una respuesta aparentemente instantánea. Claro, la respuesta no es realmente instantánea, no como un sistema que se maneja con interrupciones. El contador se detiene solo cuando el hilo tiene la CPU y se ha enterado de que la bandera ha cambiado.

Se puede ver que la clase interna **SeparateSubTask** es privada, lo que significa que sus campos y métodos se le pueden dar acceso por defecto (excepto para **run()**, que debe ser pública dado que es pública en las clases base). La clase interna privada no es accesible para nadie que no sea **Counter2**, y las dos clases estén estrechamente asociadas. En cualquier momento se puede advertir de clases que aparentan tener una alta asociación con otras, considere la mejora del código y el mantenimiento que de obtiene utilizando clases internas.

## Combinando el hilo con la clase principal

En el ejemplo mas arriba se puede ver que la clase que crea el hilo esta separada de la clase principal. Esto hace que tenga mucho sentido y es relativamente fácil de entender. Hay sin embargo, una forma alternativa que a menudo se ve utilizada que no es tan clara pero es usualmente mas conciso (que probablemente cuenta por su popularidad). Esta forma combina la clase del programa principal con la clase que genera el hilo haciendo la clase del programa principal un hilo. Dado que para un programa GUI la clase principal del programa debe ser heredada de **Frame** o **Applet**, una interfase puede ser utilizada para pegar la funcionalidad adicional. Esta interfase es llamada **Runnable**, y contiene los mismos métodos básicos que tiene **Thread**. De hecho, **Thread** también implementa **Runnable**, que especifica que hay ahí un método **run()**.

La *utilización* de la combinación de programa e hilo no es tan obvia. Cuando se inicia el programa, se crea un objeto que es **Runnable**, pero no se inicia el hilo. Esto debe ser hecho explícitamente. Se puede ver esto en el siguiente programa, que reproduce la funcionalidad de **Counter2**:

```
//: 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, **run()** esta dentro de la clase, pero sigue inactivo después de que el **init()** se completa. Cuando se presiona el botón **start**, el hilo es creado (si no existe todavía) el la expresión un tanto confusa:

```
| new Thread(Counter3.this);
```

Cuando algo tiene una interfase **Runnable**, simplemente significa que tiene un método **run()**, pero no hay nada especial acerca de eso -esto no produce habilidades de hilado innatas, como aquellas de una clase heredada de **Thread**. Así es que para producir un hilo de un objeto **Runnable**, se debe crear un objeto **Thread** separado como se muestra arriba, pasando el objeto **Runnable** a el constructor especial de **Thread**. Se puede llamar a **start()** de ese hilo:

```
| selfThread.start();
```

Esto realiza las inicializaciones y luego llama a **run()**.

El aspecto conveniente de la interfase **Runnable** es que todo pertenece a la misma clase. Si se necesita acceder a algo, simplemente se hace yendo a través de un objeto separado. Sin embargo, como se vio en el ejemplo anterior, este acceso es así de fácil solamente utilizando una clase interna<sup>1</sup>.

## Creando varios hilos

Considere la creación de varios hilos diferentes. No se puede hacer esto con el ejemplo anterior, así es que se debe volver a tener clases separadas heredadas de **Thread** para encapsular **run()**. Pero esta es una solución mas general y mas fácil de entender, así es que mientras en el ejemplo anterior muestra un estilo de codificación a menudo se verá, no puedo recomendarlo para la mayoría de los casos porque es solo un poco mas confuso y menos flexible.

El siguiente ejemplo repite la forma de los ejemplos anteriores con contadores e botones interruptores. Por ahora toda la información de un contador en particular, incluyendo el botón y el campo de texto, esta dentro de su propio objeto que es heredado de **Thread**. Todos los campos en **Ticker** son privados, lo que significa que la implementación de **Ticker** puede ser cambiada a gusto, incluyendo la cantidad y tipos de los componentes que se procuran y despliegan información. Cuando un objeto **Ticker** es creado, el constructor agrega los componentes visuales a el cuadro de contenido del objeto exterior.

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

---

<sup>1</sup> **Runnable** fué en Java 1.0, mientras que las clases internas no fueron introducidas hasta Java 1.1, las cuales pueden en parte justificar la existencia de **Runnable**. Ademas, las arquitecturas tradicionales de hilado múltiple se enfocan en una función sea ejecutada antes que un objeto. Mi preferencia es siempre heredar de **Thread** si se puede; es mas claro y mas flexible para mi.

```

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());
    // Obtiene los parametros "size" de la página Web:
    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();
    // No es un applet, así es que configuramos la bandera y
    // producimos los valores de parametros de los argumentos:
}

```

```

        applet.isApplet = false;
        if(args.length != 0)
            applet.size = Integer.parseInt(args[0]);
        Console.run(applet, 200, applet.size * 50);
    }
} //:~
```

**Ticker** contiene no solo su equipamiento para hilado también la forma de controlar y desplegar el hilo. Se puede crear tantos hilos como se quiera sin crear explícitamente los componentes de las ventanas.

En **Counter4** hay un arreglo de objetos **Ticker** llamado **s**. Para lograr una flexibilidad máxima, el tamaño de este arreglo es iniciado extendiéndose dentro de la pagina Web utilizando los parámetros **applet**. Aquí es donde el parámetro tamaño se ve parecido en la página, insertado dentro de la etiqueta **applet**:

```
<param name=size value="20">
```

El **param**, **name**, y **value** son palabras clave HTML, **name** es a lo que hacemos referencia dentro del programa, y **value** puede ser cualquier cadena, no simplemente algo que se resuelve como un número.

Se notará que la determinación del tamaño del arreglo **s** se termina dentro de **init()**, y no es parte de una definición en línea de **s**. Esto es, *no se puede* expresar como parte de la definición de clase (fuera de cualquier método):

```
int size = Integer.parseInt(getParameter("size"));
Ticker[] s = new Ticker[size];
```

Se puede compilar esto, pero se obtendrá una extraña “excepción de puntero nulo” en tiempo de ejecución. Trabaja bien si se mueve la asignación dentro del **init()**. El marco de trabajo de applet realiza las cargas necesarias para tomar los parámetros antes d entrar al **init()**.

Además, este código está configurado para ser un applet y una aplicación. Cuando es una aplicación el argumento **size** es extraído de la línea de comandos (o un valor por defecto es proporcionado).

Una vez que el tamaño del arreglo es establecido, nuevos objetos **Ticker** son creados; como parte del constructor **Ticker** el botón y el campo de texto para cada **Ticker** es agregado a el applet.

Presionar el botón **start** significa recorrer con un bucle el arreglo entero de **Tickers** y llamar a **start** por cada uno. Recuerde, **start()** realiza las tareas necesarias de iniciación y luego llama a **run()** para ese hilo.

El listener **ToggleL** simplemente invierte la bandera en el **Ticker** y cuando el hilo asociado después toma nota puede reaccionar adecuadamente.

Algo para valorar en este ejemplo es que permite crear grandes grupos de tareas independiente y monitorear su comportamiento. En este caso, se verá que a medida que el número de tareas se va haciendo mas largo, su máquina

probablemente muestre mas divergencia al desplegar los números dada la forma en que los hilos son servidos.

Se puede también experimentar descubrir que tan importante es el **sleep(100)** que esta dentro de **Ticker.run()**. Si se quita el **sleep()**, las cosas trabajarán bien hasta que se presione el botón. Entonces ese hilo particular tiene una **runFlag** false y **run()** simplemente esta trabado en un bucle infinito estrecho, lo que se hace difícil de desbaratar con hilos múltiples, así es que la respuesta y velocidad del programa realmente se caerá.

## Hilos demonio

Un hilo “demonio” es uno que se supone proporciona un servicio general de fondo mientras el programa se está ejecutando, pero no es parte de la esencia programa. De esta forma, cuando todos los otros hilos que no son demonios se completan el programa es terminado. Inversamente, si hay algún hilo que no es demonio ejecutándose, el programa no terminará (Esto es, por ejemplo, un hilo que corre **main()**).

Se puede encontrar si un hilo es un demonio llamando a **isDaemon()**, y cambiar esta característica con **setDaemon()**. Si un hilo es un demonio, entonces todos los hilos que se creen a partir de este serán un demonio.

El siguiente ejemplo muestra los hilos demonios:

```
//: 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 **Daemon** configura su bandera de demonio a “verdadero” y luego desova un racimo de otros hilos para mostrar que son solo demonios. Entonces entra en un bucle infinito que llama a **yield()** para darle el control a otros procesos. En una versión mas joven de este programa, los bucles infinitos incrementaban contadores **int**, pero llevaba e ala totalidad del programa a una parada. El utilizar **yield()** hace que el programa este bastante lleno de vida.

No hay nada que mantenga funcionando el programa una vez que se **main()** termina el trabajo, dado que no hay nada excepto demonios ejecutándose. De esta forma se puede ver los resultados de ejecutar todos los hilos demonios, **System.in** esta configurado para lectura así es que el programa espera a que una tecla se presione antes de terminar. Sin esto se verían solo algunos de los resultados de la creación de los demonios (trate de reemplazar el código de **read()** con llamadas **sleep()** de varios largos para ver este comportamiento).

## Compartiendo recursos limitados

Se puede pensar un programa con un hilo solo como una entidad solitaria moviéndose a través de su espacio de problema y haciendo una sola cosa a la vez. Dado que solo hay una entidad, no se tiene que pensar nunca acerca del problema de dos entidades tratado de utilizar el mismo recurso en el mismo momento, como dos personas tratando de estacionar en el mismo lugar, pasar a través de una puerta en el mismo momento, o incluso hablar a la misma vez.

Con el hilado múltiples las cosas no estás solas nunca mas, pero se tiene ahora la posibilidad de que dos o mas hilos intenten utilizar al mismo recurso limitado a la vez. La colisión en un recurso debe ser prevenido o de otra manera se pueden tener dos hilos tratando de acceder a la misma cuenta de banco en el mismo momento, imprimir en la misma impresora, o ajustar la misma válvula, etc.

## Acceso inapropiado a los recursos

Considere una variación en los contadores que están siendo utilizados antes en este capítulo. En el siguiente ejemplo, cada hilo contiene dos contadores que son incrementados y desplegados dentro de **run()**, además, hay otros hilos de la clase **Watcher** que están observando los contadores para ver si son siempre equivalentes. Esto parece como una actividad innecesaria, dado que buscar en el código parece obvio que los contadores siempre serán los mismos. Pero esto es donde la sorpresa aparece. He aquí una primer 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("Unsynced");
}
}
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 contiene sus propios componentes para desplegar; dos campos de texto y una etiqueta que inicialmente indica que los contadores son equivalentes. Estos componentes son agregados a el cuadro de contenido del objeto clase externa en el constructor **TwoCounter**.

Dado que un hilo **TwoCounter** es iniciado con las teclas presionadas por el usuario, es posible que **start()** pueda ser llamado mas de una vez. Esto es ilegal que **Thread.start()** sea llamado mas de una vez para un hilo (una excepción es lanzada). Se puede ver la maquinaria para prevenir esto en la bandera **started** y el método sobrecargado **start()**.

En **run()**, **count1** y **count2** son incrementados y desplegados de una forma que parecerá mantenerlos idénticas. El **sleep()** es llamado; sin esta llamada

el programa se frustra porque comienza a ser duro para la CPU alternar las tareas.

El método **synchTest()** realiza la actividad aparentemente inútil de verificar si **count1** es equivalente a **count2**; si ellos no son equivalentes se configura la etiqueta a "Unsynced" para indicar esto. Pero primero, llama a un miembro estático de la clase **Sharing1** que incrementa y despliega un contador de acceso para mostrar cuantas veces se ha verificado que ha ocurrido exitosamente (La razón para esto se hace aparente en variaciones mas posteriores en este ejemplo).

La clase **Watcher** es un hilo cuyo trabajo es llamar a **synchTest()** para todos los objetos **TwoCounter** activos. Hace esto yendo elemento por elemento en el arreglo que contiene el objeto **Sharing1**. Se puede pensar en el **Watcher** como fisgoneando constantemente sobre los hombros de objetos **TwoCounter**.

**Sharing1** contiene un arreglo de objetos **TwoCounter** esto es inicializado en **init()** y comienzan como hilos cuando se presiona el botón "start". Mas adelante cuando se presiona el botón "Watch", uno o mas observadores son creados y liberados sobre los incrédulos hilos **TwoCounter**.

Debe notarse que para ejecutar esto como un applet en un navegador, la etiqueta de applet necesitara contener las líneas:

```
| <param name=size value="20">
| <param name=watchers value="1">
```

Se puede experimentar cambiando el ancho, y alto, y los parámetros para satisfacer gustos. Cambiando el **size** y los **watchers** cambien el comportamiento del programa. Este programa es configurado y ejecutado como aplicación independiente colocando los argumentos en la línea de comandos (o proporcionando valores por defecto).

He aquí la parte sorprendente. En **TwoCounter.run()**, el bucle infinito solo pasa repetidamente por las líneas adyacentes:

```
| t1.setText(Integer.toString(count1++));
| t2.setText(Integer.toString(count2++));
```

(de la misma forma que durmiendo, pero esto no es lo importante aquí). ¡Cuando se ejecuta el programa, sin embargo, se descubrirá que **count1** y **count2** serán observados (por los **Watchers**) para ser diferentes por momentos! Esto a causa de la naturaleza de los hilos -estos pueden ser suspendidos en cualquier momento. Así es que por momentos, la suspensión ocurre *entre* la ejecución de las anteriores dos líneas, y el hilo **Watcher** se sucede para salir y realizar la comparación y solo este momento, de esta forma encontrar los dos contadores diferentes.

Este ejemplo muestra un problema fundamental con la utilización de hilos. Nunca se sabe cuando un hilo puede ser ejecutado. Imagíñese sentado en una mesa con un tenedor, cerca de pinchar el último pedazo de comida en

su plato y cuando el tenedor lo alcanza, la comida de pronto desaparece (porque su hilo fue suspendido y otro hilo entró y robó la comida). Este es el problema con el que se está tratando.

A veces no importa si un recurso es accedido a la misma vez si esta tratando de utilizarlos (la comida está en otro plato). Pero para que el hilado múltiple trabaje, se necesita alguna forma de prevenir que dos hilos accedan al mismo recurso, al menos durante períodos críticos.

Previniendo este tipo de colisión es simplemente un tema de colocar un bloqueo en un recurso cuando un hilo lo esta utilizando. El primer hilo que accede a un recurso lo bloquea, y el otro hilo no puede acceder a ese recurso hasta que el bloqueo sea quitado, en el momento en que otro hilo lo bloquea y lo utiliza, etc. Si el asiento delantero de un auto es el recurso limitado, el niño que clama a gritos “¡Mío!” tranca la puerta.

## Como comparte recursos Java

Java tiene incorporado soporte para prevenir colisiones sobre un tipo de recurso: la memoria en un objeto. Dado que típicamente hace los elementos de datos en una clase privada y accede a la memoria solo a través de sus métodos, se pueden prevenir colisiones haciendo un método particular **synchronized**. Solo un hilo a la vez puede llamar un método **synchronized** para un objeto en particular (a pesar de que ese hilo puede llamar a mas de uno de los métodos sincronizados de un objeto). Aquí hay métodos sincronizados simples:

```
| synchronized void f() { /* ... */ }  
| synchronized void g(){ /* ... */ }
```

Cada objeto contiene un bloqueo simple (también llamado *monitor*) esto es automáticamente parte de un objeto (no se debe escribir ningún código especial). Cuando se llama un método **synchronized**, este objeto es bloqueado y ningún otro método **synchronized** de este objeto puede ser llamado hasta que el primero termine y libere el bloqueo. Es el ejemplo anterior, si **f()** es llamado para un objeto, **g()** no puede ser llamado para el mismo objeto hasta que **f()** es completado y libere el bloqueo. De esta forma, hay un bloqueo simple que es compartido por un método **synchronized** de un objeto en particular, y este bloqueo previene memoria en común sea escrita y mas de un método a la vez (i.e. mas de un hilo a la vez).

Hay también un solo bloqueo por clase (como parte del objeto **Class** para cada clase), así es que ese método estático **synchronized** puede bloquear cada uno de los otros de accesos simultáneos a los datos estáticos de la totalidad de la clase básica.

Debe notarse que si se quiere cuidar otros recursos de accesos simultáneos de múltiples hilos, se puede hacer forzando el acceso a estos recursos a través de métodos sincronizados.

## Sincronizando los contadores

Armados con esta nueva palabra clave parece que la solución esta en la mano: simplemente utilizaremos la palabra clave **synchronized** para los métodos en **TwoCounter**. El siguiente ejemplo es el mismo que el anterior, con el agregado de la nueva palabra clave:

```
//: 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.bruceekel.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 notará que *ambos, run() y synchTest()* son **synchronized**. Si se sincroniza solo uno de los métodos, entonces el toro es libre de ignorar el bloqueo y puede ser llamado con impunidad. Este es un punto importante: Cada método que accede a un recurso crítico compartido debe ser **synchronized** o no trabajará correctamente.

Ahora un nuevo tema se origina. El **Watcher** nunca puede ver que esta sucediendo porque el método **run()** entero ha sido **synchronized**, y dado que **run()** siempre está ejecutándose para cada objeto el bloqueo está siempre inmovilizado y **synchTest()** puede que nunca sea llamado. Se puede ver esto porque el **accessCount** nunca cambia.

Lo que nos gustaría para este ejemplo es una forma de aislar solo la *parte* del código dentro de **run()**. La sección del código que se quiere aislar de esta forma es llamada una *sección crítica* y se usa la palabra clave **synchronized** en una forma diferente de configurar una sección crítica. Java soporta secciones críticas con el *bloque sincronizado*; en este momento **synchronized** es utilizado para especificar el objeto cuyo bloqueo es utilizado para sincronizar el código encerrado:

```
| synchronized(syncObject) {
```

```

    // This code can be accessed
    // by only one thread at a time
}

```

Antes que el bloqueo sincronizado puede ser entrado, el bloqueo debe ser adquirido en **syncObject**. Si algún otro hilo ya tiene este bloqueo, entonces el bloqueo no puede ser entrado hasta que el bloqueo sea abandonado.

El ejemplo **Sharing2** puede ser modificado removiendo la palabra clave **synchronized** del método **run()** entero y en lugar de colocar un bloqueo **synchronized** alrededor de dos líneas críticas. ¿Pero qué objeto debe ser utilizado como cerradura? El único que ya es respetado por **synchTest()**, que es el objeto actual (**this**)! Así es que el **run()** modificado se vería como esto:

```

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");
        }
    }
}

```

Esto es el único cambio que debe ser hecho en **Sharing2.java**, y se verá que mientras los dos contadores no se salen de sincronismo (conforme a cuando se le permite a el **Watcher** observarlos), sigue siendo adecuado el acceso proporcionado a el **Watcher** durante la ejecución de **run()**.

Claro, toda la sincronización depende de la diligencia del programador: cada parte del código que puede acceder a un recurso compartido debe ser envuelto en un bloqueo sincronizado apropiado.

## Eficiencia sincronizada

Dado que tener dos métodos escribiendo en la misma porción de datos no suena *nunca* como una particularmente buena idea, puede parecer que tiene mas sentido para todos los métodos ser automáticamente **synchronized** y eliminar la palabra clave **synchronized** del todo (claro, el ejemplo con un **run()** sincronizado muestra que esto pueden no trabajar de todas formas). Pero esto nos quita de dudas que obtener un bloqueo no es una operación barata -esta multiplica el costo de una llamada a método (esto es, entrando y saliendo del método, no ejecutando el cuerpo del método) al menos de cuatro veces, y puede ser mucho mas dependiente de la implementación. Así es que si se sabe que un método en particular no causará problemas de enfrentamientos es conveniente dejar fuera la palabra clave **synchronized**. Por otro lado, dejar fuera la palabra clave **synchronized** porque se piensa

que es un cuello de botella en el rendimiento, y esperar que no haya colisiones es una invitación a el desastre.

## Revisión de JavaBeans

Ahora que se entiende la sincronización, de puede ver de otro punto de vista JavaBeans. Siempre que se cree un Bean, se debe asumir que se ejecutará en un ambiente multitarea. Esto significa que:

1. Siempre que sea posible, todos los método públicos de un Bean debe ser **synchronized**. Claro, esto incurre en la sobrecarga de tiempo de la sincronización. Si esto es un problema, los métodos que no causen problemas en las secciones críticas pueden dejarse sin sincronización, pero se debe tener presente que esto no es siempre obvio. Los métodos que claramente tienden a ser pequeños (como **getCircleSize()** en el siguiente ejemplo) y/o “atómico”, esto es, la llamada al método se ejecuta en una pequeña cantidad de código que el objeto no pueda ser cambiado durante la ejecución. Hacer estos métodos no sincronizados puede no tener un efecto significante en la velocidad del programa. Se puede, de la misma forma hacer todos los métodos públicos de un Bean sincronizados y quitar las palabras claves **synchronized** solo cuando se sepa con seguridad que es necesario y que harán una diferencia.
2. Cuando se dispare un evento que se difunda en forma múltiple a un montón de listeners interesados en ese evento, se debe asumir que estos listener pueden ser agregados o quitados mientras se mueven a través de la lista.

Es bastante fácil tratar con el primer punto, pero el segundo punto requiere un poco mas de reflexión. Considere el ejemplo **BangBean.java** presentado en el capítulo anterior. Este se escabulle de la pregunta de hilado múltiple ignorando la palabra clave **synchronized** (que no había sido introducido todavía) y haciendo el evento de difusión única. He aquí un ejemplo modificado para trabajar en un ambiente multitarea y para utilizar la difusión múltiple para los eventos:

```
//: c14:BangBean2.java
// Se deberá escribir los Beans de esta forma así
// pueden ejecutarse en hambientes de hilado múltiple.
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; // Tamaño del círculo
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);
}
// Esto es un listener de emisión múltiple, que es
// mas utilizado que la estrategia de emisión única
// tomada en BangBean.java:
public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
}
public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
}

```

```

// Note que esto no es sincronizado:
public void notifyListeners() {
    ActionEvent a =
        new ActionEvent(BangBean2.this,
                        ActionEvent.ACTION_PERFORMED, null);
    ArrayList lv = null;
    // Hacer una copia superficial de la lista en caso de que
    // alguien agregue un listener mientras estamos
    // llamando a los listeners:
    synchronized(this) {
        lv = (ArrayList)actionListeners.clone();
    }
    // Llamar a todos los métodos listener:
    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);
}
} //:~

```

Agregando la sincronización a los métodos es un cambio fácil. Sin embargo, note en **addActionListener()** y en **removeActionListener()** que los **ActionListeners** son ahora agregados y quitados de una **ArrayList**, así es que se puede tener tantos como se quiera.

Se puede ver que el método **notifyListeners()** no es **synchronized**. Este puede ser llamado de mas de un hilo a la vez. Es también posible para **addActionListener()** o **removeActionListener()** ser llamado en el medio de una llamada a **notifyListeners()**, que es un problema dado que este atraviesa el **actionListeners()** del **ArrayList**. Para aligerar el problema, el **ArrayList** es clonado dentro de una cláusula **synchronized** y el clon es atravesado (vea el Apéndice A por detalles de clonado). De esta forma el **ArrayList** original puede ser manipulado sin ningún impacto en **notifyListeners()**.

El método **paintComponent()** también no es sincronizado. Decidir si sincronizar o no métodos sobrescritos no es tan claro como cuando se agrega métodos propios. En este ejemplo se descarta que **paint()** parece trabajar OK sea o no sincronizado. Pero los temas que se deben considerar son:

1. ¿Modifica el método el estado de variables “críticas” dentro del objeto? Para descubrir cuando las variables son “críticas” se debe determinar cuando serán leídas o asignadas por otros hilos en el programa (En ese caso, la lectura o asignación son virtualmente llevadas a cabo siempre por métodos sincronizados, así es que se puede solo examinar estos). En el caso de **paint()**, ninguna modificación se lleva a cabo.
2. ¿Depende el método del estado de estas variables “críticas”? Si un método sincronizado modifica una variable que un método utiliza, entonces se puede querer con razón hacer el método sincronizado también. Basado en esto, se debe observar que **cSize** es cambiado por un método sincronizado y por consiguiente **paint()** debe ser sincronizado. Aquí sin embargo, se puede preguntar “Que es la peor cosa que puede suceder si **cSize** es modificado durante un **paint()**? Cuando se ve que no es nada tan malo, y que es un efecto pasajero, se puede decidir dejar **paint()** sin sincronizar para prevenir la sobrecarga de la llamada al método sincronizado.
3. Una tercer pista es notar cuando la clase base de la versión de **paint()** es **synchronized**, lo que no es. Esto no es un argumento hermético, solo una pista. En este caso, por ejemplo, un campo que es cambiado mediante un método sincronizado (que es **cSize**) ha sido mezclado dentro de la fórmula **paint()** y puede cambiar la situación. Debe

notarse, sin embargo, que **synchronized** no se hereda -esto es, si un método es sincronizado en la clase base entonces no *es* automáticamente sincronizada la versión sobrescrita de su clase derivada.

En código de prueba en **TestBangBean2** del capítulo anterior ha sido modificado para demostrar la habilidad de emisión simultánea de **BangBean2** agregando listeners extra.

## Bloqueo

Un hilo puede estar en uno de cuatro estados:

1. *New - Nuevo*: El nuevo objeto hilo ha sido creado pero no ha sido iniciado todavía así es que no puede ejecutarse.
2. *Runnable - Capaz de ejecutarse*: Esto significa que el hilo *puede* correr cuando el mecanismo de división de tiempo de CPU disponga para el hilo. De esta forma, el hilo puede o no estar ejecutándose, pero no hay nada para prevenir que se ejecute si el organizador puede disponerlo; este no está muerto o bloqueado.
3. *Dead - Muerto*: La forma normal para un hilo para morir es retornando de su método **run()**. Se puede también llamar a **stop()**, pero esto lanza una excepción que es subclase de **Error** (lo que significa que no se está forzado a colocar la llamada en un bloque **try**). Recuerde que lanzar una excepción debe ser un evento especial y no parte de la ejecución normal de un programa; de esta forma el uso de **stop()** está desaprobado en Java 2. También está el método **destroy()** (que nunca ha sido implementado) que no se debería llamar nunca si se puede evitar dado que es drástico y no libera bloqueos de objetos.
4. *Blocked - Bloqueado*: El hilo puede ejecutarse pero hay algo que lo evita. Mientras que un hilo está en el estado de bloqueado el organizador simplemente saltará sobre él y no le dará ningún tiempo de CPU. Hasta que el hilo entre al estado *Runnable* no realizará ninguna operación.

## Convertirse en bloqueado

El estado bloqueado es el más interesante, y es valioso examinarlo en profundidad. Un hilo puede convertirse en bloqueado por cinco razones:

1. Se colocó el hilo a dormir llamando a **sleep(milliseconds)**, en tal caso no correrá por el tiempo especificado.

2. Se ha suspendido la ejecución del hilo con **suspend()**. No tendrá capacidad para ejecutarse nuevamente hasta que se le envíe al hilo el mensaje **resume()** (este ha sido desaprobado en Java 2, y será examinado mas adelante).
3. Se ha suspendido la ejecución del hilo con **wait()**. No se volverá ejecutable nuevamente hasta que el hilo obtenga los mensajes **notify()** o **notifyAll()** (Si, esto se ve como la segunda razón, pero hay una diferencia distinguiible que será revelada).
4. El hilo esta esperando que alguna E/S se complete.
5. El hilo esta tratando de llamar a un método sincronizado en otro objeto, y el bloqueo de ese objeto no está disponible.

Se puede llamar también a **yield()** (un método de la clase **Thread**) para voluntariamente ceder la CPU así otros hilos pueden ejecutarse. Sin embargo, la misma cosa sucede si el organizador decide que el hilo ha tenido suficiente tiempo y salta a otro hilo. Esto es, nada evita que el organizador mueva el hilo y le de el tiempo a algún otro hilo. Cuando un hilo es bloqueado, hay alguna razón por la que no puede continuar ejecutándose.

El siguiente ejemplo muestra las cinco formas de convertirse en bloqueado. Todas existen en un solo fichero llamado **Blocking.java**, pero serán examinadas aquí en partes discretas (Se habrá notado que las etiquetas "Continued" y "Continuing" que permiten que la herramienta de extracción de código colocar todo junto).

Dado que este ejemplo demuestra algunos métodos desaprobados, se *tendrán* mensajes de desaprobación cuando sea compilado.

Primero, el marco de trabajo básico:

```
//: c14:Blocking.java
// Demuestra las distintas formas de que un hilo
// puede ser bloqueado.
// <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.*;
////////// El marco de trabajo básico ///////////
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(); Desaprobado en 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 **Blockable** es trata de dar a entender que es una clase base para todas las clases en este ejemplo que demuestra bloqueo. Un objeto **Blockable** contiene un **JTextField** llamado **state** que es utilizado para desplegar información acerca del objeto. El método que despliega esta información es **update()**. Se puede ver que utiliza **getClass().getName()** para producir el nombre de la clase en lugar de solo imprimirla; esto es porque **update()** no puede conocer el nombre exacto de la clase que la llama, dado que es una clase derivada de **Blockable**.

El indicador de cambio en la **Blockable** es un **int i**, que será incrementado por el método **run()** de la clase derivada.

Hay aquí un hilo de la clase **Peeker** que es iniciado por cada objeto **Blockable**, y el trabajo de **Peeker** es observar el objeto asociado **Blockable** para ver los cambios en **i** llamando a **read()** y reportándolos en el estado **JTextField**. Esto es importante: Debe notarse que **read()** y **update()** son ambos sincronizados, lo que significa que requiere que el bloqueo del objeto esté libre.

## Sleeping

La primer prueba en este programa es con **sleep()**:

```
///:Continuing
////////// Bloqueando mediante 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 **Sleeper1** el método **run()** entero es sincronizado. Se verá que el **Peeker** asociado con este objeto no se ejecutara alegremente *hasta* que se inicie el hilo, y entonces el **Peeker** deja de estar frío. Esta es una forma de bloqueo: dado que **Sleeper1.run()** es sincronizado, y una vez que el hilo se inicia se esta siempre dentro de **run()**, el método nunca abandona el bloqueo del objeto y el **Peeker** esta bloqueado.

**Sleeper2** proporciona una solución haciendo que **run()** no sea sincronizado. Solo el método **change()** es sincronizado, lo que significa que mientras **run()** esta en el **sleep()**, el **Peeker** puede acceder a el método sincronizado que necesita, llamado **read()**. Aquí se verá que el **Peeker** continua ejecutándose cuando se inicia el hilo **Sleeper2**.

## Suspendiendo y reanudando

La siguiente parte del ejemplo introduce el concepto de suspensión. La clase **Thread** tiene un método **suspend()** para temporalmente detener el hilo y **resume()** que comienza de nuevo en el punto en el que fue detenido. **resume()** debería ser llamado por algún hilo fuera del suspendido, y en este caso hay una clase separada llamada **Resumer** que hace exactamente eso. Cada uno de las clases que demuestran la suspensión y la reanudación tienen una clase **Resumer** asociado:

```
///:Continuing
////////// Bloqueando mediante 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(); // Desaprobado en Java 1.2
        }
    }
}
class SuspendResume2 extends SuspendResume {
    public SuspendResume2(Container c) { super(c); }
    public void run() {
        while(true) {
            change();
            suspend(); // Desaprobado en 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

```

**SuspendResume1** también tiene un método **run()** sincronizado.

Nuevamente, cuando se inicia este hilo se verá que su **Peeker** asociado se bloquea esperando que el bloqueo este disponible, lo que nunca sucede. Esto es solucionado como antes en **SuspendResume2**, que no define la totalidad del método **run()** como sincronizado y en lugar de eso utiliza un método separado **synchronized change()**.

Se debe ser cuidadoso con que Java 2 desaprueba el uso de **suspend()** y **resume()**, dado que **suspend()** sujet a el bloqueo del objeto y de esta forma es propenso a estancarse en un punto muerto. Esto es, se puede fácilmente obtener un montón de objetos bloqueados esperando por los otros, y esto causará que el programa se congele. A pesar de esto se puede ver utilizado en viejos programas no se deberán utilizar **suspend()** y **resume()**. La solución apropiada esta descrita mas tarde en este capítulo.

## Espera y notificación

En los primeros dos ejemplos, es importante entender que **sleep()** y **suspend()** *no* liberan el bloqueo cuando son llamados. Debe tener cuidado de esto cuando se trabaja con bloqueos. Por el otro lado, el método **wait()** *libera* el bloqueo cuando es llamado, lo que significa que otros métodos sincronizados en el objeto hilo pueden ser llamados durante un **wait()**. En las siguientes dos clases, se verá que el método **run()** es totalmente sincronizado en ambos casos, sin embargo, el **Peeker** sigue teniendo acceso total a los métodos sincronizados durante un **wait()**. Esto es porque **wait()** libera el bloqueo del objeto aunque suspende el método de donde fue llamado.

Se podrá ver también que hay dos formas de **wait()**. La primera toma un argumento en milisegundos que tiene el mismo significado que en **sleep()**: detiene por ese período de tiempo. La diferencia es que en **wait()**, el bloqueo del objeto es liberado y se puede salir del **wait()** a causa de un **notify()** se puede tener el reloj liberado.

La segunda forma no toma argumentos, y significa que **wait()** continuará hasta que un **notify()** venga y no terminará después de tiempo.

Un ligeramente exclusivo aspecto de **wait()** y **notify()** es que ambos métodos son parte de la clase base **Object** y no parte de **Thread** como lo es **sleep()**, **suspend()**, y **resume()**. A pesar de que esto parezca un porco extraño al principio - tener algo exclusivamente para hilado múltiple como parte de la clase base universal- es esencial dado que manipula el bloqueo que es también parte de todo objeto. Como resultado, se puede colocar un **wait()**

dentro de cualquier método sincronizado, sin importar si hay algún hilo ejecutándose dentro de una clase en particular. De hecho, el *único* lugar donde se puede llamar a **wait()** es dentro de un método sincronizado o bloqueo. Si se llama a **wait()** o **notify()** dentro de un método que no es sincronizado, el programa compilará, pero cuando se ejecute se obtendrá un **IllegalMonitorStateException** con el mensaje poco intuitivo “current thread not owner” (El hilo actual no tiene dueño). Debe notarse que **sleep()**, **suspend()**, y **resume()** pueden todos ser llamados dentro de métodos que no sean sincronizados dado que no manipulan el bloqueo.

Se puede llamar a **wait()** o **notify()** solo para su propio bloqueo. Nuevamente, se puede compilar código que trata de utilizar el bloqueo equivocado, pero producirá el mismo mensaje de antes

**IllegalMonitorStateException**. No se puede engañar con algún otro bloqueo, pero se puede preguntar a otro objeto para realizar alguna operación que manipula su propio bloqueo. Así es que una estrategia es crear un método sincronizado que llame a **notify()** para su propio objeto. Sin embargo, en **Notifier** se verá la llamada **notify()** dentro de un bloque **synchronized**:

```
| synchronized(wn2) {  
|     wn2.notify();  
| }
```

Donde **wn2** es el objeto del tipo **WaitNotify2**. Este método, que no es parte de **WaitNotify2**, obtiene el bloqueo sobre el objeto **wn2**, punto en el cual es legal para este llamar a **notify()** para **wn2** y no se obtendrá la excepción **IllegalMonitorStateException**.

```
///:Continuing  
////////// Bloqueo mediante 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()** es típicamente utilizado cuando se logra el punto donde se está esperando por alguna otra condición, bajo el control de forzar la salida del hilo, para cambiar y si no se quiere esperar inactivamente dentro del hilo. Así es que **wait()** permite colocar el hilo a dormir mientras se espera que el mundo cambie, y solo cuando un **notify()** o **notifyAll()** se sucede el hilo se despierta y verifica los cambios. De esta forma, se proporciona una forma de sincronismo entre hilos.

## Bloque de E/S

Si un flujo está esperando por alguna actividad de E/S, automáticamente bloquea. En la siguiente porción del ejemplo, las dos clases trabajan con objetos **Reader** y **Writer** genéricos, pero en el marco de trabajo de prueba un flujo en tubo será configurado para permitir que los dos hilos de forma segura pasen datos de unos a otros (que es el propósito de los flujos en tubos).

El **Sender** coloca datos dentro del **Writer** y duerme por una cantidad aleatoria de tiempo. Sin embargo, **Receiver** no tiene **sleep()**, **suspend()**, o **wait()**. Pero cuando realiza un **read()** automáticamente bloquea cuando 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++; // Muestra si peeker esta vivo
                // Bloquea hasta que los caracteres esten ahí:
                state.setText("Receiver read: "
                    + (char)in.read());
            }
        } catch(IOException e) {
            System.err.println("IO problem");
        }
    }
}
} ///:Continued

```

Ambas clases también colocan información en sus campos **state** y cambian **i** así es que el **Peeker** puede ver que el hilo se está ejecutando.

## Probando

El la clase principal del applet es sorprendentemente simple porque la mayoría del trabajo ha sido colocada en el marco de trabajo **Blockable**. Básicamente, un arreglo de objetos **Blockable** es creado, y dado que cada uno es un hilo, realizan sus propias actividades cuando se presiona el botón "start". Hay también un botón y una cláusula **actionPerformed()** para

detener todo los objetos **Peeker**, que proporciona una demostración de la alternativa a el método **stop()** de **Thread** desaprobado en Java 2.

Para establecer una conexión entre los objetos **Sender** y el **Receiver**, un **PipedWriter** y **PipedReader** son creados. Debe notarse que **PipedWriter in** debe ser conectado a **PipedWriter out** mediante un argumento en el constructor. Luego de eso, todo lo que es colocado en **out** puede mas tarde ser extraído en **in**, como si fuera pasado a través de un tubo (de aquí en nombre). Los objetos **in** y **out** son entonces pasados a los constructores **Reciver** y **Sender**, respectivamente, que negocian como objetos **Reader** y **Writer** de cualquier tipo (esto es, se les realiza una conversión ascendente).

El arreglo de referencias **Blockable b** no es inicializado en este punto de la definición porque los flujos de tubo no pueden ser configurados antes que la definición toma lugar (la necesidad del bloque **try** previene esto).

```
///:Continuing
////////// Probando todo ///////////
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) {
            // Demonstracion de la alternativa
            // preferida a 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);
}
} //:~

```

En **init()**, se debe notar el bucle que se mueve a través del arreglo entero y almacena los campos de texto **state** y **peeker.status** a la página.

Cuando el hilo **Blockable** es inicialmente creado, cada uno crea automáticamente y e inicia su propio **Peeker**. Así es que se verá los **Peekers** ejecutándose antes que los hilos **Blockable** son iniciados. Esto es importante, casi tanto como los **Peekers** sean bloqueados y detenidos cuando el hilo **Blockable** inicia, y es esencial ver esto para entender este aspecto particular del bloqueo.

## Bloqueo muerto

Dado que los hilos pueden bloquearse y dado que los objetos pueden tener métodos **synchronized** que previenen que los hilos accedan a el objetos hasta que el bloqueo de sincronización sea liberado, es posible para un hilo quede estancado esperando por otro hilo, que a su vez espera por otro hilo, etc., hasta que la cadena llega a un hilo que espera por el primero. Se obtiene un bucle continuo de hilos esperando uno por cada uno de los otros y ninguno puede moverlo. Esto es llamado *deadlock* (*bloqueo muerto*). Esto no sucede a menudo, pero cuando sucede es frustrante de depurar.

No hay ningún soporte del lenguaje para ayudar a prevenir el bloqueo muerto; es el programador el que tiene que evitarlo con un cuidadoso diseño. Estas no son palabras confortantes para la persona que esta tratando de depurar un programa con un bloqueo muerto.

## La desaprobación de **stop()**, **suspend()**, **resume()**, y **destroy()** en Java 2

Un cambio que ha sido hecho en Java 2 para reducir la posibilidad de bloqueos muertos es la desaprobación de los métodos **stop()**, **suspend()**, **resume()** y **destroy()** de la clase **Thread**,

La razón para que el método **stop()** sea deprecado es porque no libera los bloqueos que el hilo ha adquirido, y si los objetos están en un estado inconsistente ("dañado") otros hilos pueden verse y modificarse en este estado. El problema resultante puede ser sutil difícil de detectar. En lugar de utilizar **stop()**, se debería seguir el ejemplo en **Blocking.java** y utilizar una bandera para indicarle al hilo a terminar por si mismo saliendo de su método **run()**.

Hay veces cuando un hilo se bloquea -como cuando se espera una entrada- y no puede consultar una bandera como se hace en **Blocking.java**. En este caso, no se debería utilizar **stop()**, en lugar de esto se puede utilizar el método **interrupt()** de **Thread** para salir del código bloqueado:

```
//: c14:Interrupt.java
// La estrategia alternativa a utilizar
// stop() cuando un hilo es bloqueado.
// <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(); // Bloquea
        } 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()** dentro de **Blocked.run()** produce el hilo bloqueado. Cuando se presiona el botón, la referencia a **blocked** se establece a **null** para que el recolector de basura lo limpie, y entonces el método **interrupt()** del objeto es llamado. La primera vez que se presiona el botón se verá que el hilo salie, pero luego no hay hilo para matar así es que se verá simplemente que el botón es presionado.

Los métodos **suspend()** y **resume()** resultan ser esencialmente propensos a bloqueos muertos. Cuando se llama a **suspend()**, el hilo objetivo se detiene pero sigue almacenando los bloqueos que ha adquirido en ese punto. Así es que ningún otro hilo puede acceder a los recursos bloqueados hasta que el hilo sea reanudado. Cualquier hilo que quiera reiniciar el hilo objetivo y también tratar de utilizar cualquiera de los recursos bloqueados produce un bloqueo muerto. No se debería de utilizar **suspend()** y **resume()**, pero en lugar de colocar una bandera en su clase **Thread** para indicar cuando el hilo debe ser activado o suspendido, el hilo entra en una espera utilizando **wait()**. Cuando la bandera indica que el hilo debe ser reanudado, esto se realiza con **notify()**. Un ejemplo puede ser hecho modificando **Counter2.java**. A pesar de que el efecto es similar, se debe notar que la organización del código es diferente - las clases internas anónimas son utilizadas para todos los listener y el **Thread** es una clase interna, que hace la programación ligeramente mas conveniente dado que elimina algo de la contabilidad necesaria en **Counter2.java**.

```

//: c14:Suspend.java
// La estrategia alternativa a la utililizaciónde suspend()
// y resume(), que son desaprobadas en 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);
}
} //:~

```

La bandera **suspended** dentro de **Suspendable** es utilizado para activar y desactivar la suspensión. Para suspender, la bandera es colocada en **true** llamando a **fauxSuspend()** y este es detectado dentro de **run().wait()**, como

se describió antes en este capítulo, debe ser **synchronized** así es que es el que tiene el bloqueo del objeto. En **fauxResume()**, la bandera **suspended** es colocada en **false** y **notify()** es llamado -dado que despierta a **wait()** dentro de una cláusula **synchronized** el método **fauxResume()** debe ser también **synchronized** así es que obtiene un bloqueo antes llamando **notify()** (de esta forma el bloqueo esta disponible para el **wait()** para levantarla). Si sigue el estilo mostrado en este programa se puede evitar la utilización de **suspend()** y **resume()**.

El método **destroy()** del **Thread** nunca han sido implementado; este es como un **suspend()** que no puede reiniciarse, así es que es el mismo tema de bloqueo muerto tiene **suspend()**. Sin embargo, esto no es un método desaprobado y puede ser implementado en una versión futura de Java (luego de a 2) para situaciones especiales en donde el riesgo de un bloqueo muerto es aceptable.

Se puede preguntar por que estos métodos, ahora desaprobados, fueron introducidos en Java en un principio. Parece que simplemente quitarlo es una admisión de un error significante (y agujonea con otro agujero en los argumentos para el excepcional diseño e infalibilidad pregonada por las personas de marketing de Sun). La parte que alienta acerca del cambio es que claramente indica que los técnicos y no la gente de marketing dirigen los asuntos -ellos descubrieron un problema y lo están corrigiendo. Encuentro esto mucho mas prometedor y esperanzador que dejar el problema en "corregirlo es admitir un error". Esto significa que Java continuará mejorando, aún si significa un poco de incomodidad por parte de los programadores de Java. Debería tratar con la incomodidad en lugar de mirarlo estancado.

## Prioridades

La *prioridad* de un hilo indica al organizador que tan importante el hilo es. Si hay una gran cantidad de hilos bloqueados y esperando a ser ejecutados, el organizador ejecutara el que tenga la prioridad mas alta primero. Sin embargo, esto no significa que los hilos con prioridad mas baja no serán ejecutados (esto es, no se puede obtener un bloqueo muerto por un tema de prioridades). Los hilos de prioridad mas baja tienden a ejecutarse menos a menudo.

A pesar de que las prioridades son interesantes de conocer acerca de como desempeñarse con ellas, en la práctica al menos nunca se tendrán que establecer las prioridades manualmente. Así es que siéntase libre de saltar el resto de esta sección si las prioridades no le son interesantes.

## Leyendo y estableciendo prioridades

Se puede leer la prioridad de un hilo con **getPriority()** y cambiarlo con **setPriority()**. La forma del ejemplo del “counter” anterior puede ser utilizado para mostrar el efecto de cambiar las prioridades. En este applet se verá que los contadores disminuyen la velocidad cuando los hilos asociados tienen sus prioridades bajas:

```
//: c14:Counter5.java
// Ajustando las prioridades de los hilos.
// <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); // Despliega la prioridad
    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();
        // Despliega recusivamente los grupos padres de hilos:
        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);
    }
} //:~

```

**Ticker2** sigue la forma establecida anteriormente en este capítulo, pero hay un **JTextField** extra para desplegar la prioridad de el hilo y dos botones mas para incrementar y disminuir la prioridad.

También debe notarse en uso de **yield()**, que voluntariamente devuelva el control al organizador. Sin esto el mecanismos de lectura múltiple sigue trabajando, pero se puede notar que se ejecuta mas lento (trate de quitar la llamada a **yield()** para ver esto). Se puede también llamar a **sleep()**, pero entonces la taza de conteo será controlada por la duración de **sleep()** en lugar de la prioridad.

El **init()** en **Counter5** crea un arreglo de diez **Ticker2**s; sus botones y campos son colocados en la forma por el constructor de **Ticker**. **Counter5** agrega botones para iniciar al igual que incrementar y disminuir la prioridad máxima en el grupo de hilos. Además, estas son etiquetas que despliegan 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, las prioridades de los grupos de hilos padres son también desplegados como etiquetas.

Cuando se presiones un botón “up” o “down”, esa prioridad de **Ticker2** es traída y incrementada o disminuida respectivamente.

Cuando se ejecute este programa, se podrá notar varias cosas. La primera de todas es que la prioridad por defecto del grupo de hilos es cinto. Aún si se disminuye la prioridad máxima debajo de cinco antes de comenzar los hilos (o después de crear los hilos, lo que requiere un cambio de código), cada hilo tendrá una prioridad por defecto de cinco.

La prueba simple es tomar un contador y disminuir su prioridad a uno y observar que cuenta mucho mas lentamente. Pero ahora trate de incrementarla nuevamente. Se puede regresar a la prioridad del grupo, pero no mas. Ahora disminuya la prioridad del grupo un par de veces. Las prioridades de los hilos no son cambiadas, pero si se tratan de modificar arriba o abajo se vera que son automáticamente colocadas a la prioridad del grupo. También, los hilos nuevos le serán dadas prioridades por defecto, aun si es mayor que la prioridad del grupo (De esta forma la prioridad de grupo no es una forma de prevenir que los nuevos hilos tengan prioridades mayores que los existentes).

Finalmente, trate de incrementar la prioridad máxima del grupo. No puede hacerse. Se puede solo reducir la prioridad máxima del grupo, no incrementarla.

## Grupos de hilos

Todos los hilos pertenecen a un grupo de hilos. Esto puede ser el grupo de hilos por defecto o un grupo que explícitamente se especifica cuando se crea el hilo. En la creación, el hilo es destinado a un grupo y no puede cambiarse a uno diferente. Cada aplicación tiene al menos un grupo de hilos que pertenecen al grupo de hilos del sistema. Si se crea mas hilos sin especificar un grupo, estos pertenecerán al grupo de hilos del sistema.

Los grupos de hilos deben pertenecer también a otro grupo de hilos. El grupo de hilos al cual debe pertenecer el nuevo debe ser especificado en el constructor. Si se crea un grupo de hilos sin especificar un grupo de hilos para el cual pertenezca, será colocado en el grupo de hilos del sistema. De esta forma, todos los grupos de hilos en su aplicación al final de cuantas tienen el grupo de hilos del sistema como parente.

La razón para la existencia de grupos de hilos es difícil de determinar de la literatura, la que tiende a ser confusa en este tema. A menudo se citan

“razones de seguridad”. De acuerdo a Arnold & Gosling<sup>2</sup>, “Los hilos dentro de un grupo de hilos pueden modificar los otros hilos en el grupo, incluyendo cualquiera mas abajo en la jerarquía. Un hilo no puede modificar hilos fuera de su propio grupo o grupos que contiene”. Es difícil de saber que se supone que significa “modificar” aquí. El siguiente ejemplo muestra un hilo en un subgrupo “rama” modificando las prioridades de todos los hilos en el árbol de grupos de hilos así como llamando un método para todos los hilos en ese árbol.

```
//: c14:TestAccess.java
// Como los hilos pueden acceder a otros hilos
// en un grupo de hilos padre.
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++; // modificar este hilo
            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();
        }
    }
}
```

---

<sup>2</sup> The Java Programming Language, por Ken Arnold y James Gosling, Addison-Wesley 1996 pp179.

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

En el **main()**, múltiples **ThreadGroups** son creados, ramificando de cada uno: **x** solo tiene como argumento su nombre (un **String**), así es que es automáticamente colocado en el grupo de hilos “sistema”, mientras que **y** está bajo **x** y **z** esta bajo **y**. Debe notarse que esa inicialización se sucede en orden textual así es que este código es legal.

Dos hilos son creados y colocados en diferentes grupos de hilos.

**TestThread1** no tiene un método **run()** pero tiene un método **f()** que modifica el hilo e imprime algo así es que se puede ver que fue llamado.

**TestThread2** es una subclase de **TestThread1** y su **run()** es ligeramente elaborado. Primero toma el grupo de hilos del hilo actual, luego se mueve en el árbol de herencia dos niveles utilizando **getParent()** (Esto fue ingeniado dado que he colocado a propósito el objeto **TestThread2** dos niveles mas abajo en la jerarquía). En este punto, un arreglo de referencias a **Threads** es creada utilizando el método **activeCount()** para preguntar cuantos hilos hay en este grupo y en todos los grupos de hilos hijos. El método **enumerate()** coloca referencias a todos estos hilos en el arreglo **gAll**, luego simplemente se mueve a través del arreglo entero llamando en método **f()** para cada hilo, así como también modificando la prioridad. De esta forma, un hilo en una “hoja” del grupo de hilos modifica los hilos en un grupo de hilos padre.

El método de depuración **list()** imprime toda la información acerca de un grupo de hilos a la salida estándar y es útil cuando se investiga el comportamiento del grupo 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]
```

**list()** no solo imprime el nombre de la clase de **ThreadGroup** o **Thread** también imprime el nombre del grupo de hilos y su máxima prioridad. Para los hilos, el nombre del hilos es impreso, seguido de la prioridad y del grupo al que pertenece. Note que **list()** requisa los hilos y los grupos de hilos para indicar que son hijos de los grupos de hilos no requisados.

Se puede ver que **f()** es llamado por el método **TestThread2 run()**, así es que es obvio que todos los hilos en un grupo son vulnerables. Sin embargo, se puede acceder solo a los hilos que son ramificaciones de su propio árbol de grupo de hilos **system**, y tal vez esto es lo que significa “seguridad”. No se puede acceder a ningún otro árbol de grupo de hilos de sistema.

## Controlando grupos de hilos

Dejando de lado el tema de seguridad, una cosa de los grupos de hilos parece ser útil para su control: se puede realizar ciertas operaciones en un grupo de hilos entero con un solo comando. El siguiente ejemplo demuestra esto, y las restricciones sobre las prioridades dentro de los grupos de hilos. Los números comentados entre paréntesis proporcionan una referencia para comparar con la salida.

```
//: c14(ThreadGroup1.java
// Como los grupos de hilos controlan las prioridades
// de los hilos dentro de ellos.
public class ThreadGroup1 {
    public static void main(String[] args) {
        // Tomar el hilo del sistema & imprimirl su información:
        ThreadGroup sys =
            Thread.currentThread().getThreadGroup();
        sys.list(); // (1)
        // Reduce la prioridad del grupo de hilos sistema:
        sys.setMaxPriority(Thread.MAX_PRIORITY - 1);
        // Incrementa la prioridad del hilo principal:
        Thread curr = Thread.currentThread();
        curr.setPriority(curr.getPriority() + 1);
        sys.list(); // (2)
        // Intenta configurar un nuevo grupo a el máximo:
        ThreadGroup g1 = new ThreadGroup("g1");
        g1.setMaxPriority(Thread.MAX_PRIORITY);
        // Intenta configurar un nuevo hilo a el máximo:
        Thread t = new Thread(g1, "A");
        t.setPriority(Thread.MAX_PRIORITY);
        g1.list(); // (3)
        // Reduce la prioridad máxima de g1's, luego intenta
        // incrementarla:
        g1.setMaxPriority(Thread.MAX_PRIORITY - 2);
        g1.setMaxPriority(Thread.MAX_PRIORITY);
        g1.list(); // (4)
        // Intenta configurar un nuevo hilo a el máximo:
        t = new Thread(g1, "B");
        t.setPriority(Thread.MAX_PRIORITY);
        g1.list(); // (5)
        // Baja la prioridad máxima por abajo de la prioridad
        // por defecto:
        g1.setMaxPriority(Thread.MIN_PRIORITY + 2);
        // Mira la nueva prioridad del los hilos antes
        // y despues de cambiarla:
        t = new Thread(g1, "C");
        g1.list(); // (6)
        t.setPriority(t.getPriority() -1);
        g1.list(); // (7)
        // Hace a g2 un grupo de hilos hijo de g1 y
        // trata de incrementar su prioridad:
        ThreadGroup g2 = new ThreadGroup(g1, "g2");
        g2.list(); // (8)
        g2.setMaxPriority(Thread.MAX_PRIORITY);
```

```

        g2.list(); // (9)
        // Agrega un montón de hilos nuevos a g2:
        for (int i = 0; i < 5; i++)
            new Thread(g2, Integer.toString(i));
        // Muestra información acerca de todos los grupos de hilos
        // e hilos:
        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();
        // Suspende & Detiene todos los hilos en
        // este grupo y sus subgrupos:
        System.out.println("All threads started");
        sys.suspend(); // Desaprobado en Java 2
        // Nunca llega aquí...
        System.out.println("All threads suspended");
        sys.stop(); // Deprecated in Java 2
        System.out.println("All threads stopped");
    }
}
} //:~
```

La salida que sigue ha sido editada para permitir que entre en la página (el **java.lang.** ha sido quitado) y para agregar números que correspondan con los números de comentario en la lista 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 ejecutándose, y la primer acción en el **main()** es llamar a el método **static** de **Thread** llamando a **currentThread()**. De este hilo, el grupo de hilos es producido y **list()** es llamado por un resultado. La salida es:

```
(1) ThreadGroup[name=system,maxpri=10]
    Thread[main,5,system]
```

Se puede ver que el nombre del grupo de hilos principal es **system**, y el nombre del hilo principal es **main**, y este pertenece a el grupo de hilos **system**.

El segundo ejercicio muestra que la prioridad máxima del grupo **system** puede ser reducida y el hilo **main** puede incrementar su prioridad:

```
(2) ThreadGroup[name=system,maxpri=9]
    Thread[main,6,system]
```

El tercer ejercicio crea un nuevo grupo de hilos, **g1**, que automáticamente pertenece al grupo de hilos **system** dado que no es especificado de otra forma. Un nuevo hilo **A** es colocado en **g1**. Después de asignar la máxima prioridad del grupo al nivel mas alto y la prioridad de **A** al nivel mas alto, el resultado es:

```
(3) ThreadGroup[name=g1,maxpri=9]
    Thread[A,9,g1]
```

De esta manera, no es posible cambiar la prioridad máxima del grupo de hilos a ser mayor que su grupo de hilos padre.

El cuarto ejercicio reduce la prioridad máxima de **g1** en dos y luego trata de incrementarla a **Thread.MAX\_PRIORITY**. El resultado es:

```
(4) ThreadGroup[name=g1,maxpri=8]
    Thread[A,9,g1]
```

Se puede ver que incrementar la prioridad máxima no funciona. Se puede solo reducir la prioridad máxima del grupo de hilos, no incrementarla. También, note que la prioridad del hilo **A** no cambia, y ahora es mayor la prioridad máxima del grupo de hilos. Cambiar la prioridad máxima del grupo de hilos no afecta a los hilos existentes.

El quinto ejercicio intenta configurar una nueva hilo a la máxima prioridad:

```
(5) ThreadGroup[name=g1,maxpri=8]
    Thread[A,9,g1]
    Thread[B,8,g1]
```

El nuevo hilo no puede ser cambiado a nada mas alto que la máxima prioridad de grupo de hilos.

La prioridad por defecto para este programa es seis; esto es la prioridad de un nuevo hilo que será creado y donde se quedará si no se manipula la prioridad. El ejercicio 6 baja la prioridad máxima del grupo de hilos por abajo de la prioridad por defecto para ver que sucede cuando se crea un nuevo hilo bajo estas condiciones:

```
(6) ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,6,g1]
```

Aún cuando la prioridad máxima del grupo de hilos es tres, el nuevo hilo sigue siendo creado utilizando la prioridad por defecto de seis. De esta forma, la prioridad máxima del grupo de hilos no afecta la prioridad por defecto (De hecho, parece que no hay forma de asignar una prioridad por defecto para nuevos hilos).

Luego de cambiar la prioridad, se intenta disminuir en uno, el resultado es:

```
(7) ThreadGroup[name=g1,maxpri=3]
    Thread[A,9,g1]
    Thread[B,8,g1]
    Thread[C,3,g1]
```

Solo cuando se intenta cambiar la prioridad es la prioridad máxima del grupo de hilos forzada.

Un experimento similar es realizado en (8) y (9), en donde un nuevo grupo de hilos **g2** es creado como hijo de **g1** y su prioridad máxima es cambiada. Se puede ver que es imposible para el máximo de **g2** ir mas allá de **g1**:

```
(8) ThreadGroup[name=g2,maxpri=3]
(9) ThreadGroup[name=g2,maxpri=3]
```

También debe notarse que **g2** es automáticamente configurada a la prioridad máxima de **g1** cuando **g2** es creado.

Después de todo estos experimentos, el sistema entero de grupos de hilos y sus hilos son impresos:

```
(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]
```

Dadas entonces estas reglas para los grupos de hilos, un grupo hijo debe siempre tener una prioridad máxima que sea menor o igual que la prioridad máxima del grupo padre.

La última parte de esta programa demuestra métodos para un grupo entero de hilos. Primero el programa se mueve a través del árbol entero de hilos e inicia los que no han sido iniciados. Por dramatismo, el grupo **system** es entonces suspendido y finalmente detenido (A pesar de que es interesante ver que **suspend()** y **stop()** funcionan en el grupo de hilos entero, se debe tener en mente que estos métodos son desaprobados en Java 2). Pero cuando se suspende el grupo **system** también se suspende el hilo **main** y la totalidad del programa de detiene, así es que nunca llega al punto donde los hilos son detenidos. Actualmente, si se detiene el hilo **main** se lanza una excepción **ThreadDeath**, así es que esto no es una cosa común de hacer. Dado que **ThreadGroup** es heredada de **Object**, que contiene el método **wait()**, se puede también elegir suspender el programa por unos segundos llamando a **wait(segundos \* 1000)**. Esto debe obtener el bloqueo dentro de un bloque sincronizado, claro.

La clase **ThreadGroup** de la misma forma que los métodos **suspend()** y **resume()** así es que se puede detener y arrancar un grupo entero de hilos y todos su hilos y subgrupos con un solo comando (Nuevamente, **suspend()** y **resume()** son desaprobadas en Java 2)

Los grupos de hilos pueden parecer un poco misteriosos al inicio, pero hay que tener presente que probablemente no se utilizarán directamente muy a menudo.

## Revisión de **Runnable**

Anteriormente en este capítulo, sugerí que se piense muy cuidadosamente antes de hacer un applet o **Frame** principal como una implementación de **Runnable**. Claro, si se debe heredar de una clase y se quiere agregar comportamiento de hilado en la clase, **Runnable** es la solución correcta. El ejemplo final en este capítulo saca provecho de hacer una clase **JPanel** **Runnable** que pinta diferentes colores en si misma. Esta aplicación es configurada para tomar valores de la línea de comandos para determinar que tan grande es la grilla de colores y cuanto tiempo se duerme entre los cambios de color. Jugando con estos valores se puede descubrir algunas características interesante y posiblemente inexplicables de los hilos:

```
//: c14:ColorBoxes.java
// Utilización de la interfase Runnable.
// <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() {
        // Tomo los parametros de la pagina Web:
        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);
    }
} //:~

```

**ColorBoxes** es el applet/aplicación usual con un **init()** que configura la GUI. Este asigna la **GridLayout** así es que tiene celdas **grid** en cada dimensión. Entonces se agrega la cantidad apropiada de objetos **CBox** para llenar la grilla, colocándoles el valor de **pause** a cada uno. En **main()** se puede ver como **pause** y **grid** tienen los valores por defecto que pueden ser cambiados pasándolos en la línea de comandos, o utilizando los parámetros del applet.

**CBox** es donde todo el todo el trabajo se realiza. Este es heredado de **JPanel** y implementa la interfase **Runnable** así es que cada **JPanel** puede también ser un **Thread**. Recuerde que cuando se implementa **Runnable**, no crea un objeto **Thread**, solo una clase que tiene un método **run()**. De esta forma, se debe explícitamente crear un objeto **Thread** y manejar el objeto **Runnable** en el constructor, entonces llamar a **start()** (esto sucede en el constructor). En **CBox** este hilo es llamado **t**.

Note el arreglo de **colores**, que es una enumeración de todos los colores en la clase **Color**. Esto es utilizado en **newColor()** para producir un color seleccionado de forma aleatoria. La celda actual de color es **cColor**.

**paintComponent()** es bastante simple -simplemente configura el color a **cColor** y llena el **JPanel** entero con ese color.

En **run()**, se puede ver el bucle infinito que configura el **cColor** a un nuevo color aleatorio y entonces llama a **repaint()** para mostrarlo. Entonces el hilo es dormido con **sleep()** por un tiempo determinado especificado en la línea de comandos.

Precisamente porque este diseño es flexible y el hilado es atado a cada elemento **JPanel**, se puede experimentar haciendo tantos hilos como se quiera (En realidad, hay una restricción impuesta por el número de hilos que su JVM puede manejar confortablemente).

Este programa puede hacer un estándar de comparación, dado que puede mostrar diferencias dramáticas de rendimiento entre una implementación de hilado de la JVM y otra.

## Demasiados hilos

En algún punto, se encontrara que **ColorBoxes** se atasca. En mi máquina esto ocurre en algún lugar mas allá de la grilla de 10 x 10. ¿Por que sucede esto? Naturalmente se puede sospechar que Swing puede tener algo que hacer con esto, así es que aquí hay un ejemplo que prueba esa premisa de hacer pocos hilos. El siguiente código es organizado nuevamente así es que un **ArrayList implements Runnable** y esa **ArrayList** almacena los bloques de colores y de forma aleatoria elige uno para actualizar. Entonces una cantidad de estos objetos **ArrayList** son creados, dependiendo a grandes rasgos en la dimensión de la grilla que se ha elegido. Como resultado, se tiene algunos pocos hilos mas que bloques de color, así es que si hay un aumento de velocidad sabremos que es porque había demasiados hilos en el ejemplo anterior:

```
//: c14:ColorBoxes2.java
// Uso balanceado de hilos.
// <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.bruceekel.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;
    // Pausa por defecto mas corta que ColorBoxes:
    private int pause = 50;
    private CBoxList[] v;
    public void init() {
        // Toma los parametros de la página Web:
        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 **ColorBoxes2** un arreglo de **CBoxList** es creado e inicializado para almacenar **grid CBoxLists**, cada una de las cuales sabe cuánto tiempo dormir. Un número igual de objetos **CBox2** es agregado a cada **CBoxList**, y cada lista es informada que arranque mediante **go()**, que arranca su hilo.

**CBox2** es similar a **CBox**: este se dibuja a sí mismo con un color elegido de forma aleatoria. Pero eso es *todo* lo **CBox** hace. Todo el hilado ha sido movido dentro de **CBoxList**.

El **CBoxList** puede también haber sido heredado de **Thread** y tener un objeto miembro del tipo **ArrayList**. Este diseño tiene la ventaja de que los métodos **add()** y **get()** pueden entonces ser determinados por argumentos específicos y retornar tipos de valores en lugar de objetos genéricos **Objects** (Sus nombres pueden también ser cambiados a algo más corto). Sin embargo, el diseño utilizado aquí parece a primera vista requerir menos código. Además, automáticamente retiene todos los otros comportamientos de una **ArrayList**. Con todas las conversiones y paréntesis necesarios para el **get()**, este puede no ser el caso cuando el cuerpo del código crezca.

Como antes, cuando se implementa **Runnable** no se tiene todo el equipamiento que viene con **Thread**, así es que se tiene que crear un nuevo **Thread** y uno mismo debe manejar su constructor para tener algo para iniciar con **start()**, como se puede ver en el constructor de **CBoxList** y en **go()**. El método **run()** simplemente elige un elemento de forma aleatoria dentro de la lista y llama a **nextColor()** de ese elemento para causar que elija un nuevo color seleccionado de forma aleatoria.

Ejecutando este programa, se puede ver que ciertamente corre más rápido y responde más rápidamente (por ejemplo, cuando se interrumpe, se detiene más rápidamente), y no parece caerse con grillas grandes. De esta forma un nuevo factor es agregado dentro de la ecuación de hilado: de debe mirar para ver que no se tienen “demasiados hilos” (cualquier cosa que se escape del significado del programa en particular y de la plataforma -aquí, la disminución de velocidad en **ColorBoxes** parece ser causado por el hecho de que hay solo un hilo que es responsable por todas las operaciones de dibujado, y se cae porque hay demasiadas peticiones). Si se tienen muchos hilos, se debe tratar de utilizar técnicas como la anterior para “balancear” el número de hilos en el programa. Si se ven problemas de rendimiento en un programa con hilos múltiples ahora se tienen un montón de temas para examinar:

1. ¿Se tienen muchas llamadas a **sleep()**, **yield()** y/o **wait()**?
2. ¿Son llamadas a **sleep()** suficientemente largas?

3. ¿Se están ejecutando muchos hilos?
5. Ha intentado con diferentes plataformas y JVMs?

Temas como estos son una de las razones por los cuales la programación con hilos múltiples es a menudo considerada un arte.

## Resumen

Es vital aprender cuando usar hilado múltiple y cuando evitarlo. La razón principal para usarlo es para manejar un montón de tareas que se entremezclan y hacen mas eficiente el uso de la computadora (incluyendo la habilidad de transparentemente distribuir las tareas entre múltiples CPUs) o mas conveniente para el usuario. El ejemplo clásico de balance de recursos es la utilización de la CPU durante las esperas de E/S. El ejemplo clásico de conveniencia del usuario es monitorear un botón de “parada” durante descargas largas.

Los principales inconvenientes del hilado múltiple son:

1. Desaceleración mientras se espera por recursos compartidos
2. Se requieren costos operativos adicionales para manejar hilos.
3. Complejidad no recompensada, como la idea tonta de tener un hilo separado para actualizar cada elemento de un arreglo.
4. Patologías que incluyen muerte por hambre, carreras y bloqueos muertos.

Como una ventaja adicional para los hilos es que ellos substituyen la conmutación de contextos de ejecución “livianos” (en el orden de las 100 instrucciones) por conmutación de contextos de ejecución “pesados” (en el orden de las 1000 instrucciones). Dado que todos los hilos en un proceso dado comparten el mismo espacio de memoria, una conmutación de contexto liviana solo se dan los cambios en la ejecución del programa y en las variables locales. Por el otro lado, un cambio en un proceso -en una conmutación de contexto pesada- debe intercambiar el espacio de memoria completo.

El hilado es como caminar dentro de un mundo enteramente nuevo y aprender un lenguaje de programación totalmente nuevo, o al menos un nuevo grupo de conceptos de lenguaje. Con la apariencia de soporte de hilado en muchos sistemas operativos de microcomputadoras, las extensiones para hilado han aparecido también en lenguajes de programación o bibliotecas. En todos los casos, la programación de hilos (1) parece misteriosa y requiere un desplazamiento en la forma en que se piensa acerca de la programación; y (2) se ve similar a el soporte de hilado en otros

lenguajes, así es que cuando entienda hilos, entenderá una lengua en común. Y a pesar de que el soporte para hilos puede hacer que Java parezca un lenguaje más complicado, no hay que culpar a Java. Los hilos son difíciles.

Una de las mayores dificultades con hilado ocurre porque más de un hilo puede estar compartiendo un recurso -como la memoria en un objeto- y hay que asegurarse que los múltiples hilos no traten de leer y cambiar el recurso al mismo tiempo. Esto requiere un juicioso uso de la palabra clave **synchronized**, que es una herramienta muy útil pero debe ser entendida a fondo porque puede calladamente introducir situaciones de bloqueos muertos.

Además, hay cierto arte en la aplicación de hilos. Java está diseñado para permitir crear tantos objetos como se necesite para solucionar un problema -al menos en teoría (Crear millones de objetos para un análisis finito de ingeniería, por ejemplo, puede no ser práctico en Java). Sin embargo, parece que hay un límite superior del número de hilos que se querrá crear, porque en algún punto un gran número de hilos parece convertirse en difícil de manejar. Este punto crítico no está en muchos miles como podría ser con objetos, más bien en pocos cientos, a veces menos de 100. Como a menudo se crean solo unos pocos hilos para solucionar un problema, esto no es típicamente un límite, aún en un diseño más general se convierte en una limitación.

Un tema significante no intuitivo es que, a causa de el organizador de hilos, se puede típicamente hacer que las aplicaciones se ejecuten más *rápido* insertando llamadas a **sleep()** dentro del bucle principal de programa en **run()**. Esto definitivamente hace que se sienta como un arte, en particular cuando los largos retrasos parecen aumentar el rendimiento. Claro, la razón para que esto suceda es que los retrasos cortos pueden causar que la interrupción del organizador para el final-del-**sleep()** suceda antes de que el hilo que se está ejecutando este listo para ir a dormir, forzando el organizador a pararlo y reiniciarse más tarde así puede terminar lo que estaba haciendo y entonces ir a dormir. Toma una reflexión extra darse cuenta cuán complicadas las cosas se pueden volver.

Una cosa que se puede notar perdida en este capítulo es un ejemplo de animación, que es uno de las cosas más populares para hacer con applets. Sin embargo, una solución completa (con sonido) a este problema viene con la JDK (disponible en [java.sun.com](http://java.sun.com)) en la sección de demostración. Además, podemos esperar un mejor soporte de animación venga como parte de futuras versiones de Java, mientras que soluciones que no son Java, que no son programables, completamente diferentes para animación Web aparecerán que probablemente sean superiores a las estrategias tradicionales. Por explicaciones acerca de como la animación de Java trabaja, vea *Core Java 2de* Horstmann & Cornell, Prentice-Hall, 1997. Por mas

discusiones sobre las ventajas del hilado, vea *Concurrent Programming in Java* por Doug Lea, Addison-Wesley, 1997, o *Java Threads* por Oaks & Wong, O'Reilly, 1997.

# Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Herede una clase de **Thread** y sobrecargue el método **run()**. Dentro de **run()**, imprima un mensaje, y entonces llame a **sleep()**. Repita esto tres veces, luego retorne de **run()**. Coloque un mensajes de inicio en el constructor y sobrescriba **finalize()** para imprimir un mensaje de detención. Cree un hilo separado que llame a **System.gc()** y a **System.runFinalization()** dentro de **run()**, imprima un mensaje mientras se haga. Cree muchos objetos hilos de ambos tipos y ejecútelo para ver que sucede.
2. Modifique **Sharing2.java** para agregar un bloque **synchronized** dentro del método **run()** de **TwoCounter** en lugar de sincronizar el método **run()** entero.
3. Cree dos subclases **Thread**, una con un **run()** que inicie, capture la referencia del segundo objeto **Thread** y luego llame a **wait()**. El otro **run()** de la clase debe llamar luego de que algunos segundos hayan transcurridos a **notifyAll()** para que el primer hilo pueda imprimir un mensaje.
4. En **Counter5.java** dentro de **Ticker2**, remueva el **yield()** y explique los resultados. Reemplace el **yield()** con un **sleep()** y explique los resultados.
5. En **ThreadGroup1.java**, reemplace la llamada a **sys.suspend()** con una llamada a **wait()** para el grupo de hilos, causando que espere por dos segundos. Para que esto trabaje correctamente se debe lograr el bloqueo para **sys** dentro de un bloque sincronizado.
6. Cambie **Deamons.java** así es que **main()** tiene un **sleep()** en lugar de un **readLine()**. Experimente con diferentes tiempos de estar durmiendo para ver que sucede.
7. En el capítulo 8, localice el ejemplo **GreenhouseControls.java**, que consiste en tres ficheros. En **Event.java**, la clase **Event** esta basada en observar el tiempo. Cambie **Event** para que sea un **Thread**, y cambie el resto del diseño de tal forma que este trabaje con este **Event** basado en **Thread**.

8. Modifique el ejercicio 7 para que la clase **java.util.Timer** que se encuentra en JDK 1.3 sea usada para correr el sistema.
9. Comience con **SincWave.java** del capítulo 13, cree un programa (un applet/aplicación utilizando la clase **Console**) que dibuja una forma de onda seno animada que parece desplazarse a través de la ventana de visualización como un osciloscopio, controle la animación con un **Thread**. La velocidad de la animación debe ser controlada con un control **java.swing.JSlider**.
10. Modifique el ejercicio 9 de tal forma que muchos paneles con formas de onda seno sean creados dentro de la aplicación. En número de paneles de ondas seno deben ser controlados por etiquetas HTML o parámetros de línea de comando.
11. Modifique el ejercicio 9 de tal forma que la clase **java.swing.Timer** sea usada para controlar la animación. Note la diferencia entre esto y **java.util.Timer**.

# 15: Computación distribuida

Históricamente, la programación a través de múltiples máquinas ha sido propenso a errores, dificultoso y complejo.

El programador tiene que conocer muchos detalles acerca de la red de trabajo y muchas veces del hardware. Usualmente se necesita entender las diferentes “capas” del protocolo de redes, y es ahí donde un montón de diferentes funciones en cada una de las diferentes librerías de redes concernientes a conexión, empaquetado y desempaquetado de bloques de información; distribución de esos bloques atrás y adelante; y handshaking. Era una tarea desalentadora.

Sin embargo, la idea básica de computación distribuida no es tan difícil, y es abstraída de forma muy bonita en las librerías de Java. Se quiere:

- Obtener cierta información de la máquina ahí y moverla a la máquina allí, o viceversa. Esto es logrado con programación básica de redes.
- Conectar a una base de datos, que puede estar a través de la red de trabajo. Esto es logrado con *Java DataBase Connectivity*(JDBC), que es una abstracción fuera de los complicados, detalles específicos de la plataforma de SQL (el *lenguaje de consultas estructuradas* utilizado por la mayoría de las transacciones de bases de datos).
- Proporcionar servicios mediante un servidor Web. Esto es logrado con los *servlets* de Java y las *Java Server Pages*(JSPs)
- Ejecutar métodos en objetos que están en máquinas remotas de forma transparente, como si estos objetos residieran en las máquinas locales. Esto se logra con el *método de invocación remota* (*Remote Method Invocation*RMI) de Java.
- Utilizar código escrito en otros lenguajes, ejecutándose en otras arquitecturas. Esto se logra utilizando el *Common Object Request Broker Architecture*(CORBA), que es directamente soportado por Java.
- Aislar lógica de negocios de los temas de conectividad, especialmente las conexiones con bases de datos incluyendo manejo de transacciones y seguridad. Esto se logra utilizando *Enterprise JavaBeans*(EJBs).

EJBs no son actualmente una arquitectura distribuida, pero las aplicaciones resultantes son usualmente utilizadas en un sistema de redes de trabajo cliente servidor.

- De forma fácil y dinámica, agregar y quitar dispositivos de una red representando un sistema local. Esto se logra con Jini de Java.

A cada tema se le dará una introducción liviana en este capítulo. Por favor note que cada tema es voluminoso y por si solo el tema de libros enteros, así es que este capítulo es solo trata de que nos familiaricemos con los temas, no hacerlo un experto (sin embargo, se puede recorrer un camino largo con la información presentada aquí de programación en redes, servlets y JSPs).

## Programación de redes

Una de las mas grandes fortalezas de Java es el penoso trabajo en redes. Los diseñadores de la librería de redes de Java las han hecho bastante similares a la lectura y escritura de ficheros, excepto que el “fichero” existe en una máquina remota y la máquina remota puede decidir exactamente que quiere hacer con la información que se esta solicitando o enviando. Tanto como sea posible, los detalles de las capas bajas del trabajo en redes ha sido abstraído fuera y se ha tenido cuidado dentro de la JVM y de la instalación de Java en la máquina local. El modelo de programación que se utiliza es el del fichero; de echo, actualmente se envuelve la conexión de redes (un “socket”) con objetos de flujo, así es que se termina utilizando las mismas llamadas que se han hecho con otros flujos. Además, el hilado múltiple incluido es excepcionalmente conveniente cuando se trata con otro tema de redes: manejar múltiples conexiones a la vez.

Esta sección introduce al soporte de redes de trabajo de Java utilizando ejemplos fáciles de entender.

### Identificando una máquina

Claro, para distinguir una máquina de otra y asegurarse que se esta conectado con una máquina en particular, debe haber alguna forma de identificar de forma única las máquinas de una red. La primeras redes estaban satisfechas proporcionando nombres únicos para las máquinas dentro de la red local. Sin embargo, Java trabaja con la Internet, que requiere una forma de identificar de forma única una máquina de todas las otras *en el mundo*. Esto es logrado con las direcciones IP (Internet protocol) las cuales pueden existir en dos formas:

1. La forma familiar DNS (*Domain Name System*). Mi nombre es **bruceeckel.com**, y si tengo una computadora llamada **Opus** en mi

dominio, este nombre de dominio sería **Opus.bruceeeckel.com**. Esto es exactamente el tipo de nombre que se utilizará cuando se envíe correo a las personas, y es a menudo incorporado dentro de una dirección World Wide Web.

2. Alternativamente, se puede utilizar la forma “dotted quad” (cuaterna punteada), que son cuatro números separados por puntos, como **123.255.28.120**.

En ambos casos la dirección IP, es representada internamente como un número de 32 bits<sup>1</sup> (así es que cada uno de los cuatro números no pueden exceder los 255), y se puede obtener un objeto especial de Java para representar este número en cualquiera de las formas anteriores utilizando el método **static InetAddress.getByName()** que está en **java.net**. El resultado es un objeto del tipo **InetAddress** que se puede utilizar para crear un “socket”, como se verá más tarde.

Como un simple ejemplo de utilización de **InetAddress.getByName()**, considere que sucede si se tiene un proveedor de servicio discado a la Internet (ISP). Cada vez que se disca, se asigna una IP temporal. Pero mientras se está conectado, la dirección IP tiene la misma autenticidad que cualquier otra dirección IP en la Internet. Si alguien realiza una conexión a esta máquina utilizando esta dirección IP se pueden conectar entonces a un servidor Web o un servidor FTP que se esté ejecutando en la máquina. Claro, se necesita saber la dirección IP y dado que una nueva es asignada cada vez que se disca. ¿Cómo se puede encontrar cuál es?

El siguiente programa utiliza **InetAddress.getByName()** para producir la dirección IP de la máquina local. Para usarla, se debe conocer el nombre del computador. En Windows 95 o 98, hay que ir a “Configuración”, “Panel de control”, “Red”, y entonces seleccionar la lengüeta de “Identificación”. “Nombre de PC” es el nombre para colocar en la línea de comandos.

```
//: c15:WhoAmI.java
// Encuentre su dirección de red cuando
// esté conectado a 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 =
```

---

<sup>1</sup> Esto significa un máximo de solo cerca de cuatro billones de números, lo que se está terminando rápidamente. El nuevo estándar para direcciones IP utilizará un número de 128 bits, que puede producir suficientes direcciones IP para el futuro predecible.

```
|     InetAddress.getByName(args[0]);  
|     System.out.println(a);  
| }  
| } //:/~
```

En este caso, la máquina es llamada “peppy”. Así es que una vez que se esté conectado a el ISP se ejecuta este programa.

```
| java WhoAmI peppy
```

Se obtiene un mensaje como este (claro, la dirección es diferente cada vez):

```
| peppy/199.190.87.75
```

Si le digo a un amigo esta dirección y tengo un servidor Web ejecutándose en mi computadora, el puede conectarse a el utilizando la URL <http://199.190.87.75> (solo mientras mi computadora permanezca conectada durante esa sesión). Esta puede ser muchas veces una conveniente de distribuir información a alguien mas, o probar la configuración del sitio antes de colocarla en un servidor “real”.

## Servidores y clientes

El único propósito de una red es permitir que dos máquinas se conecten y se comuniquen entre ellas. Una vez que dos máquinas han encontrada cada una de las otras puede tener una bonita, conversación en los dos sentidos. ¿Pero como se encuentran una a la otra? Es como perderse en un parque de diversiones: una máquina se queda en un lugar y escucha cuando la otra máquina dice, “Hola, donde estas?”

La máquina que “se queda en un lugar” es llamada el *servidor*, y la que busca es llamada el *cliente*. Esta distinción es importante solo mientras el cliente este tratando de conectarse al servidor. Una ves que se ha conectado, comienza un proceso de comunicación en dos sentidos y no importa mas que uno tome el rol de servidor y la otra tome el rol de cliente.

Así es que el trabajo del servidor es escuchar por una conexión, y esta es realizado por el objeto deservidor especial que se crea. El trabajo del cliente es tratar de crear una conexión al servidor, y esto es realizado por el objeto especial cliente que se ha creado. Una vez que la conexión es hecha, se verá que en el final de ambos, el cliente y el servidor, la conexión se convierte mágicamente en un objeto de flujo de E/S, y de donde se puede tratar la conexión como si se estuviera leyendo y escribiendo en un fichero. De esta forma, luego de que la conexión es hecha se pueden utilizar los comandos de E/S familiares del capítulo 11. esto es una de las características mas bonitas de el trabajo en redes de Java.

## Probando programas sin una red

Por muchas razones, de puede no tener una máquina cliente, una máquina servidor, y una red disponible para probar sus programas. Se pueden

realizar los ejercicios en una situación de salón de clase, o se pueden escribir programas que no sean todavía suficientemente estables para colocarlos en la red. Los creadores del protocolo de la Internet fueron cuidadosos con ese tema, y crearon una dirección especial llamada **localhost** para ser el la dirección IP llamada “local loopback” para probar sin una red. La forma genérica de producir esta dirección en Java es:

```
| InetAddress addr = InetAddress.getByName(null);  
Si se maneja un null con getByName(), por defecto utiliza la localhost. La InetAddress es lo que se utilizará para referirse a esa máquina en particular, y se debe producir esto antes de que se siga adelante. No se puede manipular el contenido de una InetAddress (pero se puede imprimir, como se verá en el siguiente ejemplo). La única forma que se puede crear un InetAddress es a través de los métodos miembro estáticos sobrecargados getByName() (que es lo que usualmente utilizará), getAllByName, o getLocalHost() de la clase.
```

Se puede también producir la dirección loopback manejando la cadena **localhost**:

```
| InetAddress.getByName("localhost");  
asumiendo que “localhost” esta configurada en la tabla de hosts” de la máquina, o utilizando la forma de cuaterna punteada reservada para nombrar el número para el loopback:  
| InetAddress.getByName("127.0.0.1");  
Todas las formas producen el mismo resultado.
```

## Puerto: un único lugar dentro de la máquina

Una dirección IP no es suficiente para identificar un único servidor, dado que muchos servidores pueden existir en una máquina. Cada IP de una máquina puede contener también *puertos*, y cuando se esta configurando un cliente o un servidor se debe elegir un puerto donde el cliente y el servidor estén de acuerdo a conectar; si se esta comunicando con alguien, la dirección IP es el vecindario y el puerto es el bar.

El puerto no es una ubicación física en una máquina, pero una abstracción de software (mas que nada con propósitos contables). El programa cliente sabe como conectarse a la máquina mediante su dirección IP, pero como se conecta a el servicio deseado (potencialmente uno de muchos en la máquina)? Esto es donde los números de puerto llegan como un segundo nivel de direccionamiento. La idea es que si se pregunta por un puerto en particular, se esta pidiendo el servicio que esta asociado a el número de puerto. La hora es un ejemplo simple de un servicio. Típicamente cada servicio esta asociado con un número de puerto único en la máquina servidor dada. El cliente debe saber con anterioridad en que número de puerto el servicio deseado se está ejecutando.

Los servicios del sistema reservan el uso de los puertos que van desde el 1 hasta el 1024, así es que no debería utilizar estos o estos o cualquier otro puerto que sepa que esta en uso. La primera elección para los ejemplos de este libro será el puerto 8080 (en memoria del venerable chip Intel de 8 bits 8080 en mi primer computadora, una máquina CP/M).

## Sockets

El *socket* es la abstracción de software utilizado para representar los “terminales” de una conexión entre dos máquinas, y se puede imaginar un “cable” hipotético ejecutándose entre las dos máquinas con cada final de “cable” enchufado en un socket. Claro, el hardware físico y cableado entre las máquinas es completamente desconocido. El punto fundamental de la abstracción es que no se tienen mas conocimiento del que es necesario.

En Java, se crea un socket para realizar una conexión a otra máquina, luego de obtiene un **InputStream** y un **OutputStream** (o, con los convertidores adecuados, **Reader** y **Writer**) del socket para ser capaz de tratar la conexión como un objeto de flujo E/S. Hay dos clases de socket basados en flujo: un **ServerSocket** que un servidor utiliza para “escuchar” por conexiones entrantes y un **Socket** que un cliente utiliza para iniciar una conexión. Una vez que un cliente crea un socket conexión, el **serverSocket** retorna (mediante el método **accept()**) el **Socket** correspondiente a través de donde las comunicaciones tendrán lugar en el lado del servidor. Desde este momento, se tiene un **socket** verdadero para una conexión de socket y se tratan ambos finales de la misma forma porque *son* lo mismo. En este punto, se utilizan los métodos **getInputStream()** y **getOutputStream()** para producir los correspondientes objetos **InputStream** y **OutputStream** de cada **Socket**. Esto debe ser envuelto dentro de buffers y clases para formatear exactamente como cualquier otro objeto de flujo descrito en el capítulo 11.

El uso del término **ServerSocket** puede parecer otro ejemplo de esquema de nombres confusos en las librerías Java. Se puede pensar que **ServerSocket** puede ser mejor llamado “ServerConnector” o algo sin la palabra “Socket” en ella. De puede pensar también que **ServerSocket** y **Socket** deberían ser heredados ambos de la misma clase base. Efectivamente, las dos clases tienen muchos métodos en común, pero no los suficientes como para darles una clase base en común. En lugar de eso, el trabajo de **ServerSocket** es esperar hasta que alguna otra máquina se conecte a el, entonces para retornar el **Socket** actual. Esta es por lo cual **ServerSocket** parece ser un poco mal nombrada, dado que su trabajo no es realmente ser un socket, en lugar de eso crea un objeto **Socket** cuando alguien se conecta a el.

Sin embargo, el **ServerSocket** crea un “servidor” físico o escucha un socket en la máquina host. Este socket escucha por conexiones entrantes y entonces retorna un socket “establecido” (con los finales local y remoto

definido) mediante el método **accept()**. La parte confusa es que ambos sockets (es que escucha y el establecido) están asociados con el mismo socket servidor. El socket que escucha puede aceptar solo peticiones de nuevas conexiones y no paquetes de datos. Así es que mientras **ServerSocket** no tiene mucho sentido en la programación, lo tiene “físicamente”.

Cuando se crea un **ServerSocket**, se da solo un número de puerto. No se tiene que dar una dirección IP porque ya está en la máquina que representa. Cuando se crea un socket, sin embargo, se debe dar la dirección IP y el número de puerto a donde se está tratando de conectar (Sin embargo, el **Socket** que regresa de **ServerSocket.accept()** ya contiene toda esta información).

## Un simple servidor y cliente

Este ejemplo integra la simplicidad del uso de servidores y clientes utilizando sockets. Todos los servidores hacen su espera por una conexión, luego utilizan **Socket** producidos por esa conexión para crear un **InputStream** y **OutputStream**. Estos son convertidos en un **Reader** y un **Writer**, entonces envueltos en un **BufferedReader** y un **PrintWriter**. Después de eso, todo lo que se lee del **BufferedReader** se repite a el **PrintWriter** hasta que se recibe el “fin” de línea, momento en que se cierra la conexión.

El cliente establece la conexión al servidor, entonces se crea un **OutputStream** y se realiza la misma envoltura que en el servidor. Las líneas de texto son enviadas a través del **PrintWriter** resultante. El cliente también crea un **InputStream** (nuevamente, con las conversiones apropiadas y envolturas) para escuchar que está diciendo el servidor (que, en este caso, es solo el eco de las palabras que regresan).

El servidor y el cliente utilizan el mismo número de puerto y el cliente utiliza la dirección del loopback local para conectarse al servidor en la misma máquina así es que no hay que probarlo a través de una red (Para algunas configuraciones, se puede necesitar estar *conectado* a una red para que los programas trabajen, aún si no se está *comunicando* a esa red).

He aquí el servidor:

```
//: c15:JabberServer.java
// Servidor muy simple que simplemente
// regresa un eco de lo que el cliente envía.
import java.io.*;
import java.net.*;
public class JabberServer {
    // Elija un puerto fuera del rango de 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);
```

Se puede ver que el **ServerSocket** solo necesita un número de puerto, no una dirección de IP (dado que se está ejecutando en *esta* máquina!). Cuando se llame a **accept()**, el método bloquea hasta que algún cliente trate de conectarse. Esto es, aquí es donde se espera por una conexión, pero otros procesos pueden ejecutarse (vea el Capítulo 14). Cuando una conexión es hecha, **accept()** retorna con un objeto **Socket** representando esa conexión.

La responsabilidad por la limpieza de los socket aquí es un cuidadoso trabajo artesanal. Si el constructor de **ServerSocket** falla, el programa simplemente sale (debe notarse que debemos asumir que el constructor para **ServerSocket** no deja abierto ningún socket descansando por ahí si falla). Para este caso, **main() throws IOException** así es que un bloque **try** no es necesario. Si el constructor de **ServerSocket** es exitoso entonces todas las otras llamadas a métodos deben ser protegidas en un bloque **try-finally** para asegurarse que, no importa como el bloque es dejado, el **ServerSocket** es cerrado adecuadamente.

La misma lógica es utilizada para el **Socket** returned por **accept()**. Si **accept()** falla, entonces debemos asumir que el **Socket** no existe o retiene

algún recurso, así es que no es necesario que sea limpiado. Si se sucede, sin embargo, la siguiente instrucción deben estar en un bloque **try - finally** así es que si fallan el **Socket** será a pesar de todo limpiado. Se requiere cuidado aquí porque los sockets utilizan recursos que no son memoria importantes, así es que se debe ser diligente para limpiarlos (dado que no hay destructor en Java que lo haga por usted).

El **ServerSocket** y el **Socket** producidos por **accept()** son impresos en **System.out**. Esto significa que sus métodos **toString** son automáticamente llamados. Estos producen:

```
| ServerSocket [addr=0.0.0.0,PORT=0,localport=8080]  
| Socket [addr=127.0.0.1,PORT=1077,localport=8080]
```

Brevemente, se puede ver como estos encajan juntos con lo que el cliente esta haciendo.

La siguiente parte del programa de ve simplemente como ficheros abiertos para lectura y escritura exceptuando que **InputStream** y **OutputStream** son creados del objeto **Socket**. Ambos objetos **InputStream** y **OutputStream** son convertidos a objetos **Reader** y **Writer** utilizando las clases “convertidoras” **InputStreamReader** y **OutputStreamWriter**, respectivamente. Se puede también disponer de las clases de Java 1.0 **InputStream** y **OutputStream** directamente, pero con la salida es una ventaja aparte a utilizar la estrategia de **Writer**. Esta aparece con **PrintWriter**, que tiene un constructor sobrecargado que toma un segundo argumento, una bandera **boolean** que indica cuando automáticamente limpiar la salida en el final de cada instrucción **println()** (pero *no print()*) Cada vez que se escriba a **out**, su buffer debe ser limpiado así es que la información viaja a través de la red. La limpieza es importante para este ejemplo en particular porque tanto el cliente como el servidor esperan una línea de la otra parte antes de proceder. Si la limpieza no se sucede, la información no será colocada en la red hasta que el buffer este lleno, lo que produce un montón de problemas en este ejemplo.

Cuando se escriben programas de red se necesita ser cuidadoso acerca de la utilización de limpieza automática. Cada vez que se limpia el buffer un paquete debe ser creado y enviado. En este caso, esto es exactamente lo que queremos, dado que si el paquete que contiene la línea no es enviado entonces el abrazo de ida y vuelta entre el servidor y el cliente se detendrá. Puesto de otra forma, el final de una línea es el final del mensaje. Pero en muchos casos, los mensajes no son limitados por líneas así es que es mucho mas eficiente no utilizar limpieza automática y en lugar de eso el buffer realizado incluido decide cuando crear y enviar un paquete. de esta forma, los paquetes grandes pueden ser enviados y el proceso será mas rápido.

Note que, como virtualmente todos los flujos que se abren, estos tienen un buffer. Hay un ejercicio al final de este capítulo para mostrar lo que sucede si no se coloca un buffer para los flujos (las cosas se tornan lentas).

El bucle **while** lee líneas de **BufferedReader in** y escribe información a **System.out** y a **PrintWriter out**. Note que **in** y **out** pueden ser cualquier flujo, solo sucede que están conectados a la red.

Cuando el cliente envía la línea consistente de un “END”, el programa sale del bucle y cierra el **Socket**.

Aquí está el cliente:

```
///: c15:JabberClient.java
// Cliente muy simple que solo envía
// líneas a el servidor y lee líneas
// el servidor envía.
import java.net.*;
import java.io.*;
public class JabberClient {
    public static void main(String[] args)
        throws IOException {
        // Pasandole null a getByName() produce la
        // dirección IP especial "Local Loopback", para
        // verificar en una máquina w/o en red:
        InetAddress addr =
            InetAddress.getByName(null);
        // Alternativamente, se puede
        // utilizar la dirección o nombre:
        // 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);
        // Protege todo en un try-finally para
        // asegurarse que el socket es cerrado:
        try {
            System.out.println("socket = " + socket);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            // La slaida es automáticamente limpiada
            // por 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 **main()** se puede ver todas las formas de producir la **InetAddress** de la dirección IP del loopback local: utilizando **null**, **localhost**, o la dirección explícitamente reservada **127.0.0.1**. Claro, si se quiere conectar a una máquina a través de una red se debe sustituir esa dirección IP de máquina. Cuando la **InetAddress addr** es impresa (mediante la llamada automática para su método **toString()**) el resultado es:

```
| localhost/127.0.0.1  
| Manejando en getByName() un null, por defecto encuentra a localhost, y esto produce la dirección especial 127.0.0.1.
```

Note que el **Socket** llamado **socket** es creado con **InetAddress** y el número de puerto. Para entender que significa cuando imprime uno de estos objetos **Socket**, recuerde que una conexión está determinada únicamente por estos cuatro pedazos de datos: **clientHost**, **clientPortNumber**, **serverHost**, y **serverPortNumber**. Cuando el servidor levanta, toma su puerto asignado (8080) en el localhost (127.0.0.1). Cuando el cliente levanta, este localiza el siguiente puerto disponible en su máquina, 1077 en este caso, lo que también sucede estar en la misma máquina (127.0.0.1) como el servidor. Ahora, para poder mover datos entre cliente y servidor, cada lado tiene que conocer donde enviarlo. Sin embargo, durante el proceso de conectarse al servidor “conocido”, el cliente envía una “dirección de retorno” así es que el servidor sabe donde enviar sus datos. Esto es lo que se ve en la salida del ejemplo para el lado del servidor:

```
| Socket[addr=127.0.0.1,port=1077,localport=8080]  
| Esto significa que el servidor solo acepta una conexión desde 127.0.0.1 en el puerto 1077 mientras escuchaba en su puerto local (8080). Del lado del cliente:
```

```
| Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]  
| lo que significa que el cliente realiza una conexión a 127.0.0.1 en el puerto 8080 utilizando el puerto local 1077.
```

Se notará que cada vez que inicia el cliente nuevamente, el número de puerto local es incrementado. Comienza en 1025 (delante del bloque de puertos reservados) y sigue adelante hasta que se reinicie la máquina, en este punto inicia en 1025 nuevamente (En máquinas UNIX, una vez que el límite del rango de socket es alcanzado, los números regresarán al número mas bajo disponible nuevamente).

Una vez que el objeto **Socket** ha sido creado, el proceso de convertirlo en un **BufferedReader** y **PrintWriter** es el mismo como en el servidor (nuevamente, en ambos casos se comienza con un **Socket**). Aquí, el cliente inicia la conversación enviando la cadena “howdy” seguida de un número.

Debe notarse que el buffer debe nuevamente ser limpiado (lo que sucede automáticamente mediante el segundo argumento del constructor **PrintWriter**). Si el buffer no es limpiado, la totalidad de la conversación se colgará porque el inicial “howdy” nunca será enviado (el buffer no está lo suficientemente lleno para que el envío se suceda automáticamente). Cada línea que es enviada de vuelta del servidor es escrita a **System.out** para verificar que todo está trabajando correctamente. Para terminar la conversación, el acordado “END” es enviado. Si el cliente simplemente se cuelga, el servidor lanza una excepción.

Se puede ver que el mismo cuidado es tomado aquí para asegurarse que los recursos de red representados por el **Socket** son limpiados adecuadamente, utilizando un bloque **try-finally**.

Los socket producen una conexión “dedicada” que persiste hasta que sea explícitamente desconectada (La conexión dedicada puede aún ser desconectada implícitamente si uno de los lados, o un enlace intermedio, de la conexión se cae). Esto significa que las dos partes están trabados en comunicación y la conexión es constantemente abierta. Esto parece como una estrategia lógica para redes, pero agrega una carga extra en la red. Mas tarde en este capítulo se vera una estrategia diferente para redes, en donde las conexiones son solo temporales.

## Sirviendo a múltiples clientes

El **JabberServer** trabaja, pero puede manejar solo un cliente a la vez. En un servidor típico, se querrá ser capas de tratar con muchos clientes a la vez. La respuesta es el hilado múltiple, y en lenguajes que no soportan directamente el hilado múltiple esto significa todo tipo de complicaciones. En al capítulo 14 se pudo ver que el hilado múltiple en Java es tan simple como es posible, considerando que el hilado múltiple es un tema mas bien complejo. Dado de el hilado en Java es razonablemente directo, hacer un servidor que maneje múltiples clientes es relativamente fácil.

El esquema básico es hacer un único **ServerSocket** en el servidor y llamar a **accept()** para esperar una nueva conexión. Cuando **accept()** retorna, se toma el **Socket** resultante y se utiliza para crear un nuevo hilo cuyo trabajo es servir a un cliente en particular. Entonces se llama a **accept()** nuevamente para esperar por un nuevo cliente.

En el siguiente código de servidor, se puede ver que se ve similar a el ejemplo **JabberServer.java** exceptuando que todas las operaciones par servir a un cliente en particular ha sido movido dentro de una clase que es un hilo por separado:

```
//: c15:MultiJabberServer.java
// Un servidor que utiliza hilado múltiple
```

```

// para manejar cualquier número de clientes.
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()));
        // Habilita la limpieza automática:
        out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(
                        socket.getOutputStream()), true));
        // Si alguna de las siguientes llamadas lanza una
        // excepción, el que llama es responsable por
        // cerrar el socket. De otra forma el hilo
        // lo cerrará.
        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) {
                // Bloquea hasta que una conexión se sucede:

```

```

        Socket socket = s.accept();
        try {
            new ServeOneJabber(socket);
        } catch(IOException e) {
            // Si falla, cierra el socket,
            // de otra forma el hilo lo cerrará:
            socket.close();
        }
    }
} finally {
    s.close();
}
}
} //:~

```

El hilo **ServerOneJabber** toma el objeto **Socket** que es producido por **accept()** en **main()** cada vez que un nuevo cliente crea una conexión. Entonces, como antes, crea un **BufferedReader** y limpia automáticamente el objeto **PrintWriter** utilizando el **Socket**. Finalmente, se llama el método especial **start()** del **Thread**, que realiza la inicialización del hilo y luego llama a **run()**. Esto realiza el mismo tipo de acción que en el ejemplo previo: leyendo algo del socket y entonces lo devuelve hasta que se lea la señal “END” especial.

La responsabilidad de la limpieza del socket debe nuevamente ser cuidadosamente diseñada. En este caso, el socket es creado fuera del **ServeOneJabber** así es que la responsabilidad puede ser compartida. Si el constructor de **ServeOneJabber** falla, simplemente lanzara una excepción a el que llama, quien realizará la limpieza del hilo. Pero si el constructor se sucede, entonces el objeto **ServeOneJabber** toma la responsabilidad por la limpieza del hilo, en su **run()**.

Debe notarse la simplicidad del **MultiJabberServer**. Como antes, un **ServerSocket** es creado y **accept()** es llamado para permitir una nueva conexión. Pero esta vez, el valor retornado de **accept()** (un **Socket**) es pasado a el constructor para **ServeOneJabber**, que crea un nuevo hilo para manejar esa conexión. Cuando la conexión es terminado, el hilo simplemente se hace humo.

Si la creación del **ServerSocket** falla, la excepción es nuevamente lanzada a través del **main()**. Pero si la creación se sucede, el **try-finally** exterior garantiza su limpieza. El **try-catch** interno protege solo contra las fallas del constructor de **ServeOneJabber**; si el constructor se sucede, entonces el hilo **ServeOneJabber** cerrará el socket asociado.

Para probar que el servidor realmente maneja múltiples clientes, el siguiente programa crea muchos clientes (utilizando hilos) que se conectan a el mismo servidor. El máximo número de hilos permitidos esta determinado por la constante **final int MAX\_THREADS**.

```
| //: c15:MultiJabberClient.java
```

```

// Cliente que prueba el MultiJabberServer
// iniciando múltiples clientes.
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");
            // Si la creación del socket falla,
            // nada necesita ser limpiado.
        }
        try {
            in =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            // Habilita la limpieza automática:
            out =
                new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(
                            socket.getOutputStream()))), true);
            start();
        } catch(IOException e) {
            // El socket debería ser cerrado con
            // cualquier falla que las sucedidas
            // en el constructor del socket
            try {
                socket.close();
            } catch(IOException e2) {
                System.err.println("Socket not closed");
            }
        }
        // De otra forma el socket será cerrado por
        // el método run() del hilo.
    }
    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 {
    // Siempre se cierra:
    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 de **JabberClientThread** toma una **InetAddress** y la utiliza para abrir un **Socket**. Probablemente comience a ver el patrón: el socket es siempre utilizado para crear algún tipo de objeto **Reader** y/o **Writer** (o **InputStream** y/o **OutputStream**), que es la única forma de que el **socket** pueda ser utilizado (Se puede, claro, escribir una clase o dos para automatizar este proceso en lugar de hacer toda la escritura si se convierte en dolorosa). Nuevamente, **start()** realiza la inicialización del hilo y llama a **run()**. Aquí, los mensajes son enviados a el servidor y la información del servidor es reenviada a la pantalla. Sin embargo, el hilo tiene un tiempo de vida limitado y eventualmente se completa. Debe notarse que el socket es limpiado si el constructor falla luego del que el socket es creado pero antes de que se complete el constructor. De otra forma la responsabilidad de llamar a **close()** para el socket es relegada a el método **run()**.

El **threadcount** mantiene la pista de cuantos objetos **JabberClientThread** actualmente existen. Eso es incrementado como parte del constructor y disminuido cuando se sale de **run()** (lo que significa que el hilo es terminado). En **MultiJabberClient.main()**, se puede ver que el número de hilos es verificado, y si hay demasiados, no son creados mas. Entonces el método se duerme. De esta forma, algunos hilos terminarán eventualmente y mas pueden ser creados. Se puede experimentar con **MAX\_THREADS**

para ver cuando el sistema comienza a tener problemas con muchas conexiones.

## Datagramas

El ejemplo que hemos visto hasta ahora utiliza el *Transmission Control Protocol* (TCP, también conocido como *stream-based sockets*), que está diseñado para ser fiable y garantizar que los datos estén ahí. Este permite la retransmisión de los datos perdidos, proporciona muchos caminos a través de diferentes rutas en caso de que alguna caiga, y los bytes son distribuidos en el orden que son enviados. Todo este control y flexibilidad tiene un costo: TCP tiene un costo de operación alto.

Hay un segundo protocolo, llamado *User datagram Protocol* (UDP), que no garantiza que los paquetes sean distribuidos y no garantiza que los paquetes sean distribuidos y no garantiza que lleguen en el orden en que han sido enviados. Esto es llamado un “protocolo no confiable” (TCP es un “protocolo confiable”), lo que suena mal, pero dado que es mucho más rápido puede ser útil. Hay algunas aplicaciones, como en una señal de audio en que no es tan crítico si un par de paquetes son perdidos por aquí o por allí pero la velocidad es vital. O consideremos un servidor de hora, donde no importa si uno de los mensajes es perdido. También, algunas aplicaciones pueden ser capaces de disparar un mensaje UDP a un servidor y puede entonces asumir, si no hay respuesta en un período razonable de tiempo, que el mensaje se perdió.

Típicamente, se hará la mayoría de la programación directa de red con TCP, y solo ocasionalmente utilizará UDP. Hay aquí un tratamiento más completo de UDP, incluyendo un ejemplo, en la primera edición de este libro (disponible en el CD ROM que viene con este libro, o libremente en [www.BruceEckel.com](http://www.BruceEckel.com)).

## Utilizando URLs desde dentro de un applet

Es posible para un applet causar que se despliegue cualquier URL a través de un navegador Web donde el applet se está ejecutando. Se puede hacer esto con la siguiente línea:

```
| getAppletContext().showDocument(u);  
en donde u es el objeto URL. Aquí está un simple ejemplo que lo reenvía a otra página Web. A pesar de que solo se está reenviando a una página HTML, se puede también 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.bruceekel.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 {
                // Esto puede ser un programa CGI
                // en lugar de una página HTML.
                URL u = new URL(getDocumentBase(),
                    "FetcherFrame.html");
                // Despliega en la salida de la URL utilizando
                // el navegador Web, como una página común:
                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** es cuando protege. Se puede conectar a servidores Web sin conocer mucho de todo lo que se está haciendo detrás de bambalinas.

## Leyendo un fichero del servidor

Una variación del programa anterior lee un fichero localizado en el servidor. En este caso, el fichero 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.bruceekel.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 el punto de partida como antes, pero esta vez el nombre del fichero es leído desde un **JTextField**. Una vez que el objeto **URL** es creado, la versión **String** es colocada en el **JTextArea** así es que podemos ver como se ve. Entonces un **InputStream** es procurado de la **URL**, que en este caso simplemente produce un flujo de caracteres en el fichero. Luego se convierte en un **Reader** y con un buffer, cada línea es leída y agregada a el **JTextArea**. Debe notarse que el **JTextArea** ha sido colocado dentro de un **JScrollPane** así es que el desplazamiento es manejado automáticamente.

## Mas de redes

Hay actualmente un montón mas de redes que pueden ser cubierto en este tratamiento introductorio. El trabajo en redes de Java también proporciona un soporte bastante extenso para URLs, incluyendo manejadores de protocolos para diferentes tipos de contenidos que pueden ser descubiertos en un sitio de la Internet. Se puede encontrar otras características de Java

para trabajo en redes mas completos y cuidadosamente descritos en *Java Network Programming* por Elliotte Rusty Harold (O'Reilly, 1997).

## Conectividad a bases de datos de Java (JDBC)

Se ha estimado que la mitad de todos los desarrollos de software involucran operaciones cliente/servidor. Una gran promesa de Java ha sido la habilidad de crear aplicaciones cliente/servidor de bases de datos independientes de la plataforma. Esto viene para disfrutar con Java Data Base Connectivity (JDBC).

Uno de los mayores problemas con las bases de datos ha sido las guerras de características entre las bases de datos de las empresas. Hay un lenguaje de base de datos “estándar”, Structured Query Language (SQL-92), pero se debe conocer con que vendedor de bases de datos se esta trabajando a pesar del estándar. JDBC esta diseñado para ser independiente de la plataforma, así es que no se necesita preocuparse acerca de la base de datos que se esta utilizando mientras se esta programando. Sin embargo, sigue siendo posible hacer llamadas específicas del vendedor de JDBC así es que no se esta restringido a hacer lo que se debe.

Un lugar donde los programadores pueden necesitar utilizar nombres tipo SQL es en la instrucción de SQL TABLE CREATE cuando se esta creando una nueva tabla de bases de datos y definir el tipo SQL para cada columna. Desafortunadamente hay variaciones significantes entre los tipos SQL soportados por los diferentes productos de bases de datos. Bases de datos diferentes que soportan tipos SQL con la misma semántica y estructura pueden darle a estos tipos nombres diferentes. La mayoría de las bases de datos soportan un tipo de dato SQL para grandes valores binarios: en Oracle este tipo es llamado un LONG RAW, Sybase lo llama una IMAGE, Informix lo llama BYTE y DB2 lo llama un LONG VARCHAR FOR BIT DATA. Por consiguiente, si la portabilidad de las bases de datos es una meta se debe utilizar solo identificadores de tipo SQL.

La portabilidad es un tema cuando se escribe en un libro donde los lectores puedan querer probar los ejemplos con todos los tipos de almacenamientos de datos desconocidos. He tratado de escribir estos ejemplos para hacerlos lo mas portátil posible. Se debe también notar que el código específico de la base de datos ha sido aislado para centralizar cualquier cambio que se pueda necesitar hacer los ejemplos operacionales es el ambiente.

JDBC, como muchas de las APIs en Java, esta diseñada para que sea simple. Las llamadas a métodos que se hacen corresponden con las operaciones

lógicas que se pensaría hacer cuando se reúnen datos de una base de datos: conectarse a la base de datos, crear una instrucción y ejecutar la consulta, y ver el grupo de resultados.

Para permitir esta independencia de plataforma, JDBC proporciona un *manejador de controladores* que dinámicamente mantiene todos los objetos controladores que los pedidos a la base de datos necesitarán. Así es que si se tiene tres tipos diferentes de bases de datos a donde conectarse, necesitará tres objetos controladores diferentes. Los objetos controladores se registran a sí mismos con el manejador de controladores en el momento de la carga, y se puede forzar la carga utilizando **Class.forName()**.

Para abrir una base de datos, se debe crear una “URL de la base de datos” que especifica:

1. Que se está utilizando JDBC con “jdbc”.
2. El “sub protocolo”: el nombre del controlador o el nombre de un mecanismo de conectividad a la base de datos. Dado que el diseño de la JDBC fue inspirado por ODBC, el primer sub protocolo disponible es el “jdbc-odbc bridge”, especificado por “odbc”.
3. El identificador de base de datos. Este varía con el controlador de bases de datos utilizado, pero generalmente este proporciona un nombre lógico que es trazado por el administrador del software de las bases de datos a un directorio físico donde las tablas de las bases de datos son localizadas. Para que su identificador de bases de datos tenga algún significado, se debe registrar el nombre utilizando el software de administración de base de datos (El proceso de registro varía de una plataforma a otra).

Toda esta información es combinada en una cadena, la “URL de base de datos”. Por ejemplo, para conectarse a través del sub-protocolo ODBC a una base de datos identificada como “personas”, la URL de la base de datos puede ser:

```
| String dbUrl = "jdbc:odbc:personas";
```

Si se está conectando a través de una red, la URL de la base de datos contendrá la información de conexión identificando la máquina remota y puede tornarse un poco intimidante. He aquí un ejemplo de una base de datos CloudScape que es llamada de un cliente remoto utilizando RMI:

```
| jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

Esta base de datos URL es realmente dos llamadas jdbc en una. La primera parte “jdbc:rmi://192.168.170.27:1099/” utiliza RMI para hacer que la conexión a el motor de base de datos remota que está esperando en el puerto 1099 en la dirección IP 192.168.170.27. La segunda parte de la URL, “jdbc:cloudscape:db” comunica las configuraciones más típicas utilizando el

subprotocolo y el nombre de la base de datos y esto solo sucede luego de que la primer sección ha hecho la conexión mediante RMI a la máquina remota.

Cuando se este listo para conectarse a la base de datos, llama el método **static DriverManager.getConnection()** y le pasa la URL de la base de datos, el nombre de usuario y la palabra clave de acceso para entrar en la base de datos. Se devuelve un objeto **Connection** que se puede entonces utilizar para realizar consultas y manipular la base de datos.

El siguiente ejemplo abre una base de datos de información de contratos y busca una persona cuyo apellido esta dado en la línea de comandos. Este selecciona solo los nombre de las personas que tienen direcciones de correo electrónico, luego imprime todos los que corresponden con el apellido dado:

```
//: 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:personas";
        String user = "";
        String password = "";
        // Carga el controlador (registrandose solo)
        Class.forName(
            "sun.jdbc.odbc.JdbcOdbcDriver");
        Connection c = DriverManager.getConnection(
            dbUrl, user, password);
        Statement s = c.createStatement();
        // codigo SQL:
        ResultSet r =
            s.executeQuery(
                "SELECT FIRST, LAST, EMAIL " +
                "FROM personas.csv personas " +
                "WHERE " +
                "(LAST='" + args[0] + "') " +
                "AND (EMAIL Is Not Null) " +
                "ORDER BY FIRST");
        while(r.next()) {
            // Las mayúsculas no imptan:
            System.out.println(
                r.getString("Last") + ", "
                + r.getString("FIRST")
                + ":" + r.getString("EMAIL"));
        }
        s.close(); // Tambien cierra ResultSet
    }
} ///:~
```

Se puede ver la creación de una URL de base de datos como se ha descrito previamente. En este ejemplo, no hay protección de claves en la base de datos así es que el nombre de usuario y la palabra clave son cadenas vacías.

Una vez que la conexión es hecha con **DriverManager.getConnection()**, se puede utilizar el objeto **Connection** para crear un objeto **Statement** utilizando el método **createStatement()**. Con el resultante **Statement**, se puede llamar a **executeQuery()**, pasándole una cadena conteniendo una instrucción SQL estándar SQL-92 (Se verá en breve como se puede generar esta instrucción automáticamente, así es que no se tiene que conocer mucho acerca de SQL).

El método **ExecuteQuery()** retorna un objeto **ResultSet**, que es un iterator: el método **next()** mueve el iterator a el siguiente registro en la instrucción, o retorna **false** si el final del grupo de resultados ha sido alcanzado. Siempre se tendrá un objeto **ResultSet** de regreso de una **executeQuery()** incluso si el resultado de la consulta es un conjunto vacío (esto es, una excepción no es lanzada). Debe notarse que se deba llamar a **next()** una vez antes de tratar de leer cualquier registro de datos. Si el conjunto resultado es vacío, esta primer llamada a **next()** retornará **false**. Para cada registro en el grupo resultado, se pueden seleccionar los campos utilizando (entre otras estrategias) el nombre de campo como una cadena. También debe notarse que la capitalización de el nombre de campo es ignorada -esto no importa con una base de datos SQL. Se determina el tipo que se retornará llamando a **getInt()**, **getString()**, **getFloat()**, etc. En este punto, se tienten los datos de la base de datos en el formato nativo de Java y se puede hacer lo que se quiera con esto utilizando código común de Java.

## Haciendo que el ejemplo funcione

Con JDBC, entender el código es relativamente simple. La parte confusa es hacer que trabaje en un sistema en particular. La razón de que esto sea confuso es que hay que resolver como obtener el controlador JDBC apropiado para cargar, y como configurar su base de datos utilizando el software de administración.

Claro, este proceso puede variar radicalmente de máquina a máquina, pero el proceso que he utilizado para hacer que trabaje bajo Windows de 32 bits puede darle una pista para ayudarnos a atacar una situación en particular.

### Paso 1: Encuentre el controlador JDBC

El programa anterior contiene la instrucción:

```
| Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );  
Esto implica una estructura de directorio, que es engañosa. Con esta  
instalación en particular de JDK 1.1, no hay un fichero llamado  
JdbcOdbcDriver.class, así es que si se mira en este ejemplo y se busca por el  
se verá frustrado. Otros ejemplos publicados utilizan un seudo nombre,  
como "myDriver.ClassName", que es menos que útil. De hecho, la
```

instrucción de carga anterior para el controlador jdbc-odbc (el único que en la actualidad viene con la JDK) aparece solo en unos pocos lugares en la documentación en línea (en particular, una página etiquetada “JDBC-ODCB Bridge Driver”). Si la instrucción de carga anterior no trabaja, entonces el nombre debe haber cambiado como parte del cambio de versión, así es que se debe buscar en la documentación nuevamente.

Si la instrucción de carga esta mal, se tendrá una excepción en este punto. Para probar si la instrucción de carga esta trabajando correctamente, comente el código luego de la instrucción hasta la cláusula **catch**; si el programa no lanza excepciones significa que el controlador es cargado adecuadamente.

## Paso 2: Configure la base de datos

Nuevamente, esto es específico de Windows 32-bit; se deberá hacer algo de investigación para imaginarlo para otra plataforma.

Primero, abra el panel de control. De debe encontrar dos íconos que dicen “ODBC”. De debe utilizar uno que dice “32bit ODBC”, dado que los otros es para compatibilidad inversa con el software de 16-bit y pueden no producir resultados para JDBC. Cuando se abre el icono de “32bit ODBC”, se verá un dialogo con lengüetas con un montón de lengüetas, incluyendo “DSN de usuario”, “DSN de sistema”, “DSN de archivo”, etc, en donde “DSN” significa “Nombre de fuente de datos del inglés Data Source Name”. Esto lanza esto para el puente JDBC-ODBC, el único lugar donde es importante configurar su base de datos es en “DSN de sistema”, pero se querrá también probar la configuración y crear consultas, y para esto también se necesitará configurar su base de datos en “DSN de archivo”. Esto permitirá que la herramienta Microsoft Query (Que viene con Microsoft Office) encuentre la base de datos. Debe notarse que otras herramientas de consultas también están disponibles de otros vendedores.

La base de datos mas interesante es una que ya estamos utilizando. ODBC estándar soporta un montón de formatos de ficheros diferentes incluyendo algunos venerables caballos de tiros como DBase. Sin embargo, también incluye el formato simple de “ASCII separados por comas”, que virtualmente todas las herramientas de datos tienen la habilidad de escribir. En mi caso, solo tomo mi base de datos de “personas” que ha mantenido por años utilizando varias herramientas manejadoras de contactos y exportado como fichero ASCII separado por comas (típicamente tienen la extensión **.csv**). En la sección “DSN de sistema” elijo “Aregar”, elijo el manejador de texto para mi fichero ASCII separado por comas, y entonces quito la marca que tiene “usar directorio actual” para permitir que se especifique el directorio donde voy a exportar el fichero de datos.

Se notará que cuando se haga esto no se especifica un fichero, solo un directorio. Esto es porque una base de datos es típicamente representada por una colección de fichero bajo un solo directorio (a pesar de que puede ser representadas de otra forma igualmente). Cada fichero usualmente contiene una sola tabla, y las instrucciones SQL pueden producir resultados que son escogidos de múltiples tablas en la base de datos (esto es llamado *unión*). Una base de datos que contiene una sola tabla (como mi base de datos “personas”) es usualmente llamada una *base de datos de fichero plano*. Muchos problemas que van mas allá del simple almacenamiento y recuperación de datos generalmente requieren múltiples tablas que deben ser relacionadas por uniones para producir los resultados deseados, y estas son llamadas bases de datos *relacionales*

### Paso 3: Probar la configuración

Para probar la configuración se necesitar una forma de descubrir cuando la base de datos es visible de un programa que le realizará consultas. Claro, se puede simplemente ejecutar el programa JDBC de ejemplo mas arriba, hasta e incluyendo la instrucción:

```
| Connection c = DriverManager.getConnection(  
| dbUrl, user, password);  
| si una excepción es lanzada, su configuración es incorrecta.
```

Sin embargo, es útil obtener una herramienta generadora de consultas involucradas en este punto. He utilizado Microsoft Query que viene con Microsoft Office, pero se puede preferir algo mas. La herramienta de consultas debe saber donde está la base de datos, y Microsoft Query requiere que se vaya hasta la lengüeta “DSN de fichero” del administrador ODBC y agregue una nueva entrada ahí, nuevamente especificando el controlador de texto y el directorio donde la base de datos está. Se puede nombrar la entrada de la forma que se quiera, pero es útil utilizar el mismo nombre que en “DSN de sistema”.

Una vez que se ha hecho esto, se verá que su base de datos esta disponible cuando se crea una nueva consulta utilizando la herramienta de consultas.

### Paso 4: Generar la consulta SQL

La consulta que he creado utilizando Microsoft Query no solo me muestra que mi base de datos esta ahí correctamente, también crea automáticamente el código SQL que necesito para insertar en mi programa Java. Quiero una consulta que busque registros que tengan el apellido que he escrito en la línea de comandos cuando he iniciado el programa Java. Así es que como punto de partida, he buscado por un nombre específico, “Eckel”. Quiero desplegar también solo aquellos nombres que tienen direcciones de correo asociadas a ellos. El paso que he tomado para crear esta consulta fue:

1. Comienzo una nueva consulta y utilizo el Asistente de consultas. Selecciono la base de datos “personas” (Esto es un equivalente a abrir la conexión a la base de datos utilizando la URL apropiada).
2. Selecciono la tabla “personas” dentro de la base de datos. Desde dentro de la tabla, elijo las columnas NOMBRE, APELLIDO, y CORREO.
3. Bajo “Filtro de datos”, elijo APELLIDO y selecciono “igual” con un argumento “Eckel”. Pico en el botón de radio “y”.
4. Elijo CORREO y selecciono “no es nulo”.
5. Bajo “Ordenar por”, elijo NOMBRE.

El resultado de esta consulta mostrará si ha obtenido lo que quiere.

Ahora se puede presionar el botón SQL y sin ninguna investigación de su parte, aparecerá el código SQL correcto, listo para que se pueda cortar y pegar. Para esta consulta, se ve así:

```
SELECT personas.FIRST, personas.LAST, personas.EMAIL
FROM personas.csv personas
WHERE (personas.LAST='Eckel') AND
(personas.EMAIL Is Not Null)
ORDER BY personas.FIRST
```

Especialmente con consultas mas complicadas es fácil tener cosas mal, pero utilizando una herramienta de consultas se puede interactivamente probar sus consultas y automáticamente generar el código correcto. Es difícil de debatir el caso para hacer esto a mano.

## Paso 5: Modificar y pegar en su consulta

Se habrá notado que el código arriba de ve diferente del que hemos utilizado en el programa. Esto es porque la herramienta de consulta utiliza una calificación completa para todos los nombres, incluso donde solo hay una tabla involucrada (Cuando mas de una tabla esta involucrada, la calificación previene colisiones entre columnas de diferentes tablas que tienen los mismos nombres). Dado que esta petición involucra solo una tabla, se puede opcionalmente remover el calificador “personas” de los nombres, como aquí:

```
SELECT FIRST, LAST, EMAIL
FROM personas.csv personas
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

Además, no se quiere que este programa sea altamente codificado para buscar solo un nombre. En lugar de eso, se podría buscar por el nombre dado como argumento en la línea de comandos. Haciendo estos cambios y

convirtiendo la instrucción SQL en una cadena creada dinámicamente se 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 llamada *procedimientos almacenados* que es utilizado por un tema de velocidad. Pero para experimentar con la base de datos y como primer corte, crear sus propias cadenas de consultas en Java esta bien.

Se puede ver en este ejemplo que utilizando las herramientas disponibles actualmente -en particular la herramienta de creación de consultas- la programación de bases de datos con SQL y JDBC puede ser bastante directa.

## Una versión GUI del programa de búsqueda

Es mas útil dejar la búsqueda del programa corriendo todo el tiempo y simplemente escribir en el nombre cuando se quiera encontrar algo. El siguiente programa crea un programa de búsqueda como una aplicación/applet, y agrega también la terminación así es que los datos serán mostrados sin forzar la escritura del apellido completo:

```
//: c15:jdbc:VLookup.java
// Versión GUI de 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:personas";
    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 {
    // Carga el controlador (registrarse a si mismo)
    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 {
        // Terminación de nombre:
        r = s.executeQuery(
            "SELECT LAST FROM personas.csv personas " +
            "WHERE (LAST Like '" +
            searchFor.getText() +
            "%') ORDER BY LAST");
        if(r.next())
            completion.setText(
                r.getString("last"));
        r = s.executeQuery(
            "SELECT FIRST, LAST, EMAIL " +
            "FROM personas.csv personas " +
            "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 las bases de datos es la misma, pero se pude ver que un **DocumentListener** es agregado para escuchar el **JTextField** (vea la entrada **javax.swing.JTextField** en la documentación HTML de Java de *java.sun.com* por detalles), así es que cuando quiera que se escriba un nuevo carácter, primero trata de completar en nombre buscando por el último nombre en la base de datos y utilizando el primero que se muestra (Se coloca en la **JLabel completion**, y se utiliza como el texto de búsqueda). De esta forma, en el momento en que se han escrito suficientes caracteres para el programa encuentre de forma única el nombre que se está buscando, se puede parar.

## Por que la API JDBC parece tan complicada

Cuando se busca documentación en línea para JDBC puede parecer desalentadora. En particular, en la interfase **DatabaseMetaData** -que es simplemente enorme, contrariamente a la mayoría de las interfaces que se ha visto en Java- hay métodos como **dataDefinitionCausesTransactionCommit()**, **getMaxColumnNameLength()**, **getMaxStatementLength()**, **storesMixedCaseQuotedIdentifiers()**, **supportsANSI92IntermediateSQL()**, **supportsLimitedOuterJoins()**, y mucho mas. ¿Que es todo esto?

Como se ha mencionado antes, las bases de datos parecían ser en sus comienzos en constante estado de agitación, primariamente porque la demanda por aplicaciones de bases de datos, y sus herramientas de bases de datos, era muy grande. Solo recientemente ha habido una convergencia en el lenguaje en común SQL (y hay abundancia de otros lenguajes de bases de datos en uso regular). Pero aún con un “estándar” SQL hay tantas variaciones en ese tema que JDBC debe proporcionar una gran interfase

**DatabaseMetaData** así es que el código puede descubrir las posibilidades de la base de datos SQL “estándar” a la cual esta actualmente conectada. Brevemente, se puede escribir simple, transportable SQL, pero si se quiere optimizar velocidad de codificación multiplicará tremadamente a medida que investigue las capacidades de una base de datos de un proveedor en particular.

Esto, claro, no es una falla de Java. Las discrepancias entre los productos de bases datos solo es algo que JDBC trata de ayudar a compensar. Pero recuerde que la vida puede ser mas fácil si puede escribir código genérico de consultas y no preocuparse mucho del rendimiento, o, si se debe ajustar el rendimiento, conozca la plataforma en la que se está escribiendo así no necesita escribir todo ese código de investigación.

## Un ejemplo mas sofisticado

Un ejemplo<sup>2</sup> mas interesante involucra una base de datos de múltiples tablas que reside en un servidor. Aquí, la base de datos tiene la intención de proporcionar un depósito para actividades de la comunidad y permitir que las personas firmen por estos eventos, así es que es llamada la *Community Interests Database*(CID Base de datos de intereses comunitarios). Este ejemplo no solo proporciona un vistazo general de la base de datos y su implementación, no intenta ser un manual de instrucción profundo de desarrollo de bases de datos. Hay muchos libros, seminarios, y paquetes de software que lo ayudarán en el diseño y desarrollo de una base de datos.

Además, este ejemplo supone que se tiene una instalación previa de una base de datos SQL en un servidor (a pesar de que puede ser ejecutada en la máquina local), y la indagación y descubrimiento de un manejador JDBC apropiado para la base de datos. Muchas bases de datos libres SQL están disponibles, y algunas son incluso automáticamente instaladas con muchos distribuciones de Linux. Se es responsable por realizar la elección de la base de datos y localizar el manejador JDBC; el ejemplo aquí esta basado en un sistema de bases de datos SQL llamado “Cloudscape”.

Para conservar la información de conexión simple, el manejador de base de datos, URL de la base de datos, nombre de usuario y palabra clave son colocados en una clase separada:

```
//: c15:jdbc:CIDConnect.java
// Información de conexión de la base de datos para
// la base de datos de intereses comunitarios (CID).
public class CIDConnect {
    // Toda la información específica para CloudScape:
    public static String dbDriver =
        "COM.cloudscape.core.JDBCDriver";
```

<sup>2</sup> Creado por Dave Bartlett

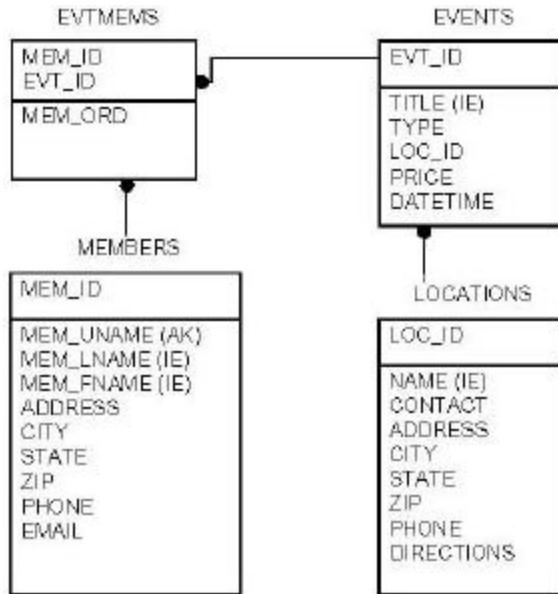
```

    public static String dbURL =
        "jdbc:cloudscape:d:/docs/_work/JsapienDB";
    public static String user = "";
    public static String password = "";
} //:~

```

En este ejemplo, no hay protección de palabra clave así es que el nombre de usuario y la palabra clave son cadenas vacías.

La base de datos consiste en un grupo de tablas que tienen una estructura como la mostrada aquí:



“Members” contienen la información de los miembros de la comunidad, “Events” y “Locations” contienen información acerca de las actividades y donde toman lugar, y “Evtmems” conecta entre eventos y miembros que quieren acudir a ese evento. Se puede ver que un miembro dato en una tabla produce una clave en otra tabla.

La siguiente clase contiene las cadenas SQL que crean estas tablas de base de datos (por explicaciones de código SQL hay que referirse a una guía SQL):

```

//: c15:jdbc:CIDSQl.java
// La cadena SQL para crear las tablas para el CID.
public class CIDSQl {
    public static String[] sql = {
        // Crear la tabla MEMBERS:
        "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)",
// Crear la tabla EVTMEMS
"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)",
// Crear la tabla LOCATIONS
"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 siguiente programa usa la información de **CIDConnect** y **CIDSQL** para cargar el manejador JDBC , realizando una conexión a la base de datos, y entonces crear la estructura de la base de datos diagramadas mas arriba. PAra conectarse con la base de datos, se llama a el método estático **DriverManager.getConnection()**, y se le pasa la URL de la base de datos, el nombre de usuario, y la palabra clave para entrar en la base de datos. Se retorna un objeto **Connection** que se puede utilizar para consultar y manipular la base de datos. Una vez que la conexión es hecha simplemente se puede empajar el SQL a la base de datos, en este caso se envía a través del

arreglo **CIDSQL**. Sin embargo, la primer vez que este programa se ejecuta, el comando “drop table” puede fallar, causando una excepción que es capturada, reportada y luego ignorada. La razón por la que se colocó el comando “drop table” es para permitir una fácil experimentación: se puede modificar el SQL que define las tablas y entonces ejecutar nuevamente el programa, causando que las viejas tablas sean remplazadas por las nuevas.

En este ejemplo, tiene sentido dejar que las excepciones sean lanzadas a la consola:

```
//: c15:jdbc:CIDCreateTables.java
// Crea las tablas de base de datos para la
// base de datos de interés comunitario.
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();
    }
} ///:~
```

Debe notarse que todos los cambios en la base de datos pueden ser controlados cambiando las cadenas en la tabla **CIDSQL**, sin modificar **CIDCreateTables**.

**executeUpdate()** retornará usualmente el número de filas que fueron afectadas por la instrucción SQL. **execceuteUpdate()** es mas comúnmente utilizada para ejecutar instrucciones INSERT, UPDATE, o DELETE que modifican una o mas filas. Para instrucciones como CREATE TABLE, DROP TABLE, y CREATE INDEX, **executeUpdate()** siempre retorna cero.

Para probar la base de datos, hay que cargarla con algún dato. Esto requiere una series de INSERTs seguidos de un SELECT para producir un grupo de resultados. Para agregar y cambiar los datos de prueba fácilmente, los datos de prueba son configurados como un arreglo en dos dimensiones de **Objects**,

y el método **executeInsert()** puede entonces utilizar la información en una fila de la tabla para crear el comando SQL apropiado.

```
//: c15:jdbc:LoadDB.java
// Carga y prueba la base de datos.
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" },
        // Corresponde algunas personas con eventos
        { "EVTMEMS",
            new Integer(1), // Dave esta yendo a
            new Integer(1), // el evento de Software.
            new Integer(0) },
        { "EVTMEMS",
            new Integer(2), // Bruce esta yendo a el
            new Integer(2), // evento arqueológico.
            new Integer(0) },
        { "EVTMEMS",
```

```

        new Integer(3), // Robert esta yendo a
        new Integer(1), // el evento de Software.
        new Integer(1) },
    { "EVTMEMS",
        new Integer(3), // ... y
        new Integer(2), // el evento arqueológico.
        new Integer(1) },
    };
// Utilice el grupo de datos por defecto:
public TestSet() {}
// Utilice un grupo de datos diferente:
public TestSet(Object[][] dat) { data = dat; }
}
public class LoadDB {
    Statement statement;
    Connection connection;
    TestSet tset;
    public LoadDB(TestSet t) throws SQLException {
        tset = t;
        try {
            // Carga el manejador (registrandose solo)
            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]);
    }
    // Lanza las excepciones a la consola:
    public static void main(String[] args)
        throws SQLException {
        LoadDB db = new LoadDB(new TestSet());
        db.load();
        try {
            // Obtiene un resultado de la base de datos cargada:
            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 **TestSet** contiene un grupo de datos por defecto que es producido si se utiliza el constructor por defecto; sin embargo, se puede también crear un objeto **TestSet** utilizando datos alternativos configurado con el segundo constructor. El grupo de datos es guardado en un arreglo de dos dimensiones de **Object** porque puede ser de cualquier tipo, incluyendo **String** o valores numéricos. El método **executeInsert()** utiliza RTTI para distinguir entre datos **String** (que deben estar entre comillas) y datos que no son **String** cuando genera el comando SQL de los datos. Luego de imprimir este comando a la consola, **executeUpdate()** es utilizado para enviarse a la base de datos.

El constructor para **LoadDB** hace la conexión, y **load()** se mueve a través de los datos y llama a **executeInsert()** para cada registro. **cleanup()** cierra la instrucción y la conexión; para garantizar que esta es llamada, se coloca dentro de la cláusula **finally**.

Una vez que la base de datos es cargada, una instrucción **executeQuery()** produce un grupo de resultados ejemplo. Dado que la consulta combina muchas tableas, este es un ejemplo de unión.

Hay mas información JDBC disponible en los documentos que vienen como parte de la distribución de Java de Sun. Además, se puede encontrar mas en

el libro *JDBC Database Access with Java* (Hamilton, Cattel, and Fisher, Addison-Wesley, 1997). Otros libros JDBC aparecen regularmente.

## Servlets

El acceso a clientes desde Internet o intranet corporativas es una forma segura de permitir que muchos usuarios accedan a los datos y recursos fácilmente<sup>3</sup>. Este tipo de acceso esta basado en clientes que utilicen los estándares para la World Wide Web de Hypertext Markup Language HTML y el protocolo de transferencia de texto (HTTP). El grupo de la API de Servlet resume una solución común en un marco de trabajo para responder a las peticiones HTTP.

Tradicionalmente, la forma de manejar un problema como el permitir a un cliente de Internet actualizar una base de datos es crear una página HTML con campos de texto y un botón “submit”. El usuario escribe la información apropiada en los campos de texto y presiona el botón “submit”. Los datos son enviados junto con una URL que le indica al servidor que hacer con los datos por especificación de la localización del programa Common Gateway Interface (CGI) que el servidor ejecuta, proporcionando el programa con los datos como son invocados. El programa CGI esta escrito normalmente en Perl, Python, C, C++ o cualquier lenguaje que puede leer de la entrada estándar y escribir a la salida estándar. Esto es todo lo que es proporcionado por el servidor Web: el programa CGI es invocado, y flujos estándares (o, opcionalmente por entrada una variable de ambiente) son utilizados para la entrada y la salida. El programa CGI es responsable por todo lo demás. Primero analiza los datos y decide cuando el formato es correcto. Si no lo es, el programa CGI debe producir HTML para describir el problema; esta página es manejada por el servidor Web (mediante la salida estándar del programa CGI), que lo envía de regreso al usuario. El usuario debe usualmente actualizar la página y tratar nuevamente. Si los datos son correctos, el programa CGI procesa los datos en una forma apropiada, tal vez agregándolo a la base de datos. Esto debe producir una página HTML apropiada para que sea retornada al usuario.

Sería ideal apostar por una solución totalmente basada en Java para este problema -Un applet en el lado del cliente para validar y enviar los datos, y un servlet en el lado del servidor para recibir y procesar los datos.

Desafortunadamente, a pesar de que los applets son una tecnología probada con abundancia de soporte, han sido problemáticos para utilizar en la Web porque no se puede confiar en que una versión particular de Java se encuentre en el navegador del cliente; de hecho no se puede confiar en que un navegador soporte Java del todo! En una intranet, se puede requerir que

---

<sup>3</sup> Dave Bartlett fue decisivo en el desarrollo de este material, y también la sección SJP.

un cierto soporte esté disponible, lo que permite un montón mas de flexibilidad en lo que se puede hacer, pero en la Web la estrategia mas segura es manejar todos los procesos en el lado del servidor y distribuir texto plano HTML a el cliente. De esta forma, ningún cliente le será negado el uso de su sitio porque no tienen el software apropiado instalado.

Dado que los servlets proporcionan una excelente solución para soporte de programación del lado del servidor, son una de las razones mas populares para migrar a Java. No solo proporcionan un marco de trabajo que remplaza a la programación CGI (y elimina un montón de problemas espinosos de CGI), también el código gana la portabilidad de utilizar Java, y se tiene acceso a toda la API de Java (excepto, claro, la que produce GUIs, como Swing).

## El servlet básico

La arquitectura de la API de un servlet es aquella del clásico proveedor de servicios con un método **service()** a través del cual todas las peticiones de los clientes serán enviadas por el software que contiene al servlet, y los métodos de ciclo de vida **init()** y **destroy()**, que son llamados solo cuando el servlet es cargado y descargado (esto sucede raramente).

```
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 retornar un objeto **ServletConfig** que contiene los parámetros de inicialización y arranque para el servlets. **getServletInfo()** retorna una cadena contenido información acerca del servlet, como el autor, la versión y los derechos de copia.

La clase **GenericServlet** es una implementación armazón de esta interfase y no es utilizada típicamente. La clase **HttpServlet** es una extensión de **GenericServlet** y es diseñada específicamente para manejar el protocolo HTTP -**HttpServlet** es el único que se utilizará la mayor parte del tiempo.

El atributo mas conveniente para la API de servlet son los objetos auxiliares que vienen junto con la clase **HttpServlet** para soportarla. Si se observa en el método **service()** en la interfase **Servlet**, se verá que tiene dos parámetros: **ServletRequest** y **ServletResponse**. Con la clase **HttpServlet** hay dos objetos que son extendidos 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();
    }
} //:~

```

**ServletsRule** es lo mas simple que se puede obtener con un servlet. El servlet es iniciado solo una vez llamando a su método **init()**, cuando se carga el servlet luego de que el contenedor es iniciado por primera vez. Cuando un cliente hace una petición a una URL que esta representada por un servlet, el contenedor del servlet intercepta esta petición y hace una llamada a el método **service()**, luego de configurar los objetos **HttpServletRequest** y **HttpServletResponse**.

La principal responsabilidad del método **service()** es interactuar con la petición HTTP que el cliente ha enviado, y crear una respuesta HTTP basada en los atributos contenidos dentro de la petición. **ServletRule** solo manipula el objeto respuesta sin observar que ha enviado el cliente.

Luego de configurar el tipo de contenido de la respuesta (que debe siempre ser hecho antes de que **Writer** o **OutputStream** sean procurados), el método **getWriter()** del objeto respuesta produce un objeto **PrintWriter**, que es utilizado para escribir los datos de respuesta basados en caracteres (alternativamente, **getOutputStream()** produce un **OutputStream**, utilizado para respuestas binarias, que es solo inicializado en soluciones mas específicas).

El resto del programa simplemente envía de regreso al cliente HTML (se asume que se entiende HTML, así es que esa parte no será explicada) como una secuencia de **String**. Sin embargo, debe notarse la inclusión del “contador de acceso” representado por la variable **i**. Esto es automáticamente convertido a un **String** en la instrucción **print()**.

Cuando se ejecuta el programa, se notará que el valor de **i** es retenido entre las peticiones del servlet. ¡Esto es una propiedad esencial de los servlets: dado que solo un servlet de una clase particular es cargado dentro del contenedor, y nunca es descargado (a no ser que el contenedor del servlet sea terminado, lo que a veces solo sucede normalmente si se reinicia el

servidor), y los campos de la clase servlets efectivamente se convierten en objetos persistentes! Esto significa que se puede fácilmente mantener valores entre peticiones a servlets, mientras que con CGI se tienen que escribir valores al disco para preservarlos, lo que requiere una considerable cantidad de engaños alrededor para hacerlo correctamente, y resulta en una solución que no es de plataforma cruzada.

Claro, a veces el servidor Web, y de esta forma el contenedor del servlet, debe ser reiniciado como parte del mantenimiento o durante una falla en la energía. Para evitar la perdida de información persistente, los métodos **init()** y **destroy()** son automáticamente llamados dondequiera que el servlet sea cargado o descargado, dando la oportunidad de guardar datos durante la baje, y restaurarlos después de reiniciar. El contenedor del servlet llama el método **destroy()** como si terminara solo, así es que siempre se obtiene una oportunidad de guardar datos importantes en la medida de que el servidor este configurado de una forma inteligente.

Hay otro tema cuando se utiliza **HttpServlet**. Esta clase proporciona los métodos **doGet()** y **doPost()** que diferencia entre un envío “GET” de CGI del cliente, y un “POST” CGI. GET y POST varían solo en los detalles de la forma que son enviados los datos, que es algo que personalmente prefiero ignorar. Sin embargo, mucha de la información publicada que he visto parece a favor de la creación de métodos **doGet()** y **doPost()** separados en lugar de un solo método genérico **service()**, que maneja ambos casos. Este favoritismo parece bastante común, pero nunca he visto explicado en una forma que me explique que es mas que una inercia de los programadores CGI que lo utilizan para prestar atención de donde son utilizados los métodos GET o POST. Así es que en el espíritu de “hacer las cosas lo mas simple que puedan trabajar”<sup>4</sup>, simplemente utilizaré el método **service()** en estos ejemplos, y dejemos que se preocupen acerca de los GET versus POST. Sin embargo, se debe tener en mente que puedo haberme perdido algo y de esta forma puede de hecho ser una buena razón para utilizar **doGet()** y **doPost()** en lugar de **service()**.

Donde quiera que un formulario sea enviado a un servlet, el **HttpServletRequest** llega cargado previamente con todos los datos del formulario, almacenado como pares de valores. Si se conoce los nombres de los campos, simplemente se puede utilizarlos directamente con el método **getParameter()** para buscar los valores. Se puede también obtener una **Enumeration** (la vieja forma del **Iterator**) para los campos de nombres, como es mostrada en el siguiente ejemplo. Este ejemplo también demuestra como un simple servidor puede ser utilizado para producir la página que contiene el formulario, y para responder a la página (una mejor solución será vista mas adelante, con JSPs. Si la **Enumeration** esta vacía, no hay

---

<sup>4</sup> Una afirmación primaria de la Programación Extrema (XP). Vea [www.xprogramming.com](http://www.xprogramming.com).

campos; esto significa que no ha sido enviado ningún formulario. Es este caso el formulario es producido, y el botón para enviarlo llamará nuevamente el mismo servlet. Si los campos existen, sin embargo, son desplegados.

```
//: 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 se envía un formulario -- se crea uno:
            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();
    }
} ///:~
```

Un inconveniente que se notará aquí es que Java no parece ser diseñado pensando en procesado de cadenas -el formateo de la página de retorno es penoso a causa de los quiebres de línea, marcas de escape, y los signos “+” necesarios para producir objetos **String**. Con una página HTML grande comienza a hacer irrazonable para codificar directamente en Java. Una solución es mantener la página en un fichero de texto separado, luego abrirla y manejarla en el servidor Web. Si se tiene que realizar algún tipo de sustitución a el contenido de la página no es mejor dado que Java tiene un tratamiento de cadenas muy pobre. En estos casos sería mejor utilizar una solución mas apropiada (Python puede ser mi elección; hay una versión que se incrusta solo en Java llamada JPython) para generar la página de respuesta.

## Servlets e hilado múltiple

El servlet contiene un montón de hilos que son enviados para manejar las peticiones de los clientes. Es bastante probable que dos clientes lleguen al mismo tiempo puedan procesar a través de **service()** en el mismo momento. Sin embargo el método **service()** debe escribirse de una forma segura para trabajar con hilos. Para cualquier acceso a recursos en común (ficheros, bases de datos) se necesitará ser protegido utilizando la palabra clave **synchronized**.

El siguiente ejemplo coloca una cláusula **synchronized** alrededor del método **sleep()** para hilos. Esto bloqueará todos los otros hilos hasta que el tiempo asignado (cinco segundos) sea completado. Cuando se pruebe esto se deberán iniciar varias instancias de navegadores y acceder al servidor tan rápido como sea posible en cada uno -se verá que cada uno tiene que esperar para aparecer.

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

Es también posible sincronizar el servlet entero colocando la palabra clave **synchronized** al inicio del método **service()**. De hecho, la única razón para utilizar la cláusula **synchronized** en otro lugar es si la sección crítica está en una ruta que pueda no ser ejecutada. En ese caso, se puede evitar de la misma forma la sobrecarga de la sincronización. De otra forma, todos los hilos tendrán que esperar de todas formas así es que se puede de la misma forma sincronizar el método entero.

# Manejando sesiones con servlets

HTTP es un protocolo sin sesiones, así es que no puede indicarse de un hit a otro si es la misma persona que esta consultando repetidamente su sitio, o se es una persona completamente diferente. Un esfuerzo grandioso se ha hecho en los mecanismos que permiten a los desarrolladores Web trazar sesiones. Las empresas no pueden hacer comercio electrónico sin mantener la pista de un cliente y los artículos que colocan en su carrito de compras, por ejemplo.

Hay muchos métodos para realizar trazado de sesiones, pero el mas común es con “cookies”, persistentes, que son una parte integral de los estándar de la Internet. El grupo de trabajo HTTP de Internet Engineering Task Force ha escrito cookies dentro del estándar oficial en el RFC 2109 ([ds.internic.net/rfc/rfc2109.txt](http://ds.internic.net/rfc/rfc2109.txt) busque en [www.cookiecentral.com](http://www.cookiecentral.com)).

Una cookie no es nada mas que un pequeño pedazo de información enviado por un servidor Web a un navegador. El navegador almacena la cookie en el disco local, y cuando otra llamada es echa a la URL esa cookie es asociada, la cookie es silenciosamente enviada junto con la llamada, de esta forma se proporciona la información deseada de regreso al servidor (generalmente, proporcionando alguna forma para que el servidor pueda indicar que es el mismo navegador). Los clientes pueden, sin embargo, apagar la habilidad del navegador de aceptar cookies, entonces otro método de trazado de sesiones (reescritura de URL o campos de formularios ocultos) debe ser incorporado a mano, dado que las capacidades de trazado de sesiones armadas en la API de servlet son diseñadas alrededor de las cookies.

## La clase **Cookie**

La APIU de servlet (versión 2.0 y mas alta) proporciona la clase **Cookie**. Esta clase incorpora todos los detalles de los cabezales HTTP y permite configurar varios atributos de cookie. Utilizando la cookie es implemente un tema de agregarla a el objeto de respuesta. el constructor toma un nombre de cookie como el primer argumento y un valor como el segundo. Las Cookies son agregadas a el objeto de respuesta antes de que se envíe cualquier comentario.

```
| Cookie oreo = new Cookie("TIJava", "2000");
| res.addCookie(cookie);
```

Las cookies son recuperadas llamando a el método **getCookies()** del objeto **HttpServletRequest**, que retorna un arreglo de objetos cookie.

```
| Cookie[] cookies = req.getCookies();
```

Se puede llamar a **getValue()** para cada cookie, para producir un **String** contenido el contenido de la cookie. En el ejemplo mas arriba, **getValue("TIJava")** producirá un **String** contenido “2000”.

## La clase Sesión

Una sesión es uno o mas peticiones de páginas por un cliente a un sitio Web durante un período de tiempo definido. Si se compran abarrotes en línea, por ejemplo, se quiere que una sesión sea confinada por el período de tiempo desde cuando se agrega primero un ítem a el “carrito de compras” hasta el punto cuando se paga. Cada ítem que se agrega a el carrito de compras resultará en una nueva conexión HTTP, que no tiene conocimiento de las conexiones anteriores o ítems en el carrito de compras. Para compensar este agujero de información, los mecanismos proporcionados por la especificación de la cookie permiten a el servlet realizar trazado de sesiones.

Un objeto **Session** vive del lado del servidor en el canal de comunicaciones; su meta es capturar datos útiles acerca de este cliente a medida que el cliente se mueve e interactúa con el sitio Web. Estos datos pueden ser pertinentes para la presente sesión, como los ítems en el carrito de compras, o pueden ser datos como la información de autentificación que es entrada cuando el cliente entra por primera vez a el sitio Web, y donde puede no ser reentrado durante un grupo de transacciones en particular.

La clase **Session** de la API de servlet utiliza la clase **Cookie** para hacer este trabajo. Sin embargo, todos los objetos necesarios para **Session** necesitan algún tipo único de identificador almacenado en el cliente que es pasado al servidor. Los sitios Web pueden también utilizar otros tipos de trazado de sesiones pero estos mecanismos serán mas difíciles de implementar dado que no son encapsulados en la API de servlet (esto es, se deben escribir a mano para tratar con la situación donde el cliente ha desabilitad las cookies).

He aquí un ejemplo que implementa el trazado de sesión con la API de 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 {
        // Recupera el objeto Session antes de que
        // cualquier cosa sea enviada al cliente.
        HttpSession session = req.getSession();
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HEAD><TITLE> SessionPeek </TITLE></HEAD><BODY>");
        out.println(" </BODY>");
```

```

        out.println("<h1> SessionPeek </h1>");
        // Un simple contador de hits para esta sesión.
        Integer ival = (Integer)
            session.getAttribute("sesspeak.cntr");
        if(ival==null)
            ival = new Integer(1);
        else
            ival = new Integer(ival.intValue() + 1);
        session.setAttribute("sesspeak.cntr", ival);
        out.println("You have hit this page <b>" +
            + ival + "</b> times.<p>");
        out.println("<h2>"); 
        out.println("Saved Session Data </h2>"); 
        // Da vueltas a través de todos los datos de la sesión:
        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> Estadísticas de la sesión </h3>"); 
        out.println("ID de sesión: " +
            + session.getId() + "<br>"); 
        out.println("Nueva sesión: " + session.isNew() +
            + "<br>"); 
        out.println("Tiempo de creado: " +
            + session.getCreationTime()); 
        out.println("<I>(" +
            new Date(session.getCreationTime())
            + ")</I><br>"); 
        out.println("Último acceso: " +
            session.getLastAccessedTime()); 
        out.println("<I>(" +
            new Date(session.getLastAccessedTime())
            + ")</I><br>"); 
        out.println("Intervalo inactivo de sesión: " +
            + session.getMaxInactiveInterval()); 
        out.println("ID de sesión ID en consulta: " +
            + req.getRequestedSessionId() + "<br>"); 
        out.println("Es un ID de sesión de una Cookie: " +
            + req.isRequestedSessionIdFromCookie()
            + "<br>"); 
        out.println("Es el ID de sesión de la URL: " +
            + req.isRequestedSessionIdFromURL()
            + "<br>"); 
        out.println("Es un ID de sesión válido: " +
            + req.isRequestedSessionIdValid()
            + "<br>"); 
        out.println("</BODY>"); 
        out.close(); 
    } 
    public String getServletInfo() { 
        return "Un servlet para trazado de sesión "; 
    } 
}

```

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

Dentro del método **service()**, **getSession()** es llamado para el objeto consultado, que retorna el objeto de **Session** asociado con esta consulta. El objeto **Session** no viaja a través de la red, este se encuentra en el servidor y es asociado con un cliente y sus consultas.

**getSession()** viene en dos versiones: sin parámetros, como es utilizada aquí, y **getSession(boolean)**. **getSession(true)** es un equivalente a **getSession()**. La única razón para el **boolean** es establecer cuando se quiere que el objeto de la sesión sea creado si no se encuentra. **getSession(true)** es la llamada mas probable, por lo tanto **getSession()**.

El objeto **Session**, si no es nuevo, nos dará detalles acerca de las visitas previas del cliente. Si el objeto **Session** es nuevo entonces el programa será iniciado para recoger información acerca de las actividades del cliente en esta visita. La captura de esta información de cliente es realizada mediante los métodos **setAttribute()** y **getAttribute()** del objeto sesión.

```
| java.lang.Object getAttribute(java.lang.String)
| void setAttribute(java.lang.String name,
|                   java.lang.Object value)
```

El objeto **Session** utiliza pares de valores simples para cargar información. El nombre es un **String** y el valor puede ser cualquier objeto derivado de **java.lang.Object**. **SessionPeck** mantiene la pista de la cantidad de veces que el cliente ha vuelto durante esta sesión. Esto es realizado con un objeto **Integer** colocado dentro del objeto **Session**. Si se utiliza la misma clave en una llamada **setAttribute()**, entonces el nuevo objeto sobrescribe la vieja. El contador de incremento es utilizado para desplegar el número de veces que el cliente nos ha visitado durante esta sesión.

**getAttributeNames()** esta relacionado con **getAttribute()** y **setAttribute()**; este retorna una enumeración de los nombres de los objetos que son ligados a el objeto **Session**. Un bucle **while** en **SessionPeck** muestra este método en acción.

Nos podríamos preguntar cuento tiempo un objeto **Session** se mantiene. La respuesta depende del contenedor del servlet que se esta utilizando; usualmente son 20 minutos por defecto (1800 segundos), que es lo que debería ver de la llamada a **getMaxInactiveInterval()** de **ServletPeek**. Las pruebas parecen producir resultados diversos entre contenedores de servlets. A veces el objeto **Session** puede quedarse toda la noche, pero nunca vi un caso donde el objeto **Session** desaparezca en menos que el tiempo especificado por el intervalo de inactividad. Se puede tratar con esto configurando el intervalo de inactividad con **setMaxInactiveInterval()** a 5 segundos y ver si el objeto **Session** permanece o es limpiado cuando llega el tiempo apropiado. Esto puede ser un atributo que se puede querer para investigar cuando elegir un contenedor de servlet.

## Ejecutando los ejemplos de servlet

Si no se ha trabajado todavía con un servidor de aplicación que maneje servlets de Sun y tecnologías JSP, de debe bajar la implementación Tomcat de servlets de Java y JSPs, que es una implementación libre, de código abierto para servlets, y es la implementación de referencia oficial sancionada por Sun. Esta puede encontrarse en [jakarta.apache.org](http://jakarta.apache.org)

Siga las instrucciones de instalación la implementación Tomcat, luego edite el fichero **server.xml** para apuntar a la localización de el árbol de directorio donde sus servlet serán colocados. Una vez que se inicia el programa Tomcat se puede probar sus programas servlet.

Esto es solo una breve introducción a servlets; hay libros enteros del tema. Sin embargo, esta introducción le dará idea suficiente para iniciarse. Además, muchas de las ideas de la siguiente sección son inversamente compatibles con servlets.

## Paginas de servidor Java (JSP Java Server Pages)

Las páginas de servidor Java es una extensión estándar de Java que es definida en la mayoría de las extensiones servlet. La meta de JSP es simplificar la creación y manejo de las paginas Web dinámicas.

La implementación referencia Tomcat, disponible libremente de [jakarta.apache.org](http://jakarta.apache.org) anteriormente mencionada automáticamente soporta JSPs.

JSPs permiten combinar el HTML de una página Web con partes de código Java en el mismo documento. El código Java es rodeado por etiquetas especiales que indican el contenedor JSP que debe utilizar el código para generar un servlet, o parte de uno. El beneficio de las JSPs es que se puede mantener un solo documento que represente ambos, la página y el código que lo habilita. El punto débil es que el mantenimiento de una página JSP debe ser hecho en HTML y Java (sin embargo, los ambientes GUI para JSPs deben estar por aparecer).

La primera ves que JSP es cargado por un contenedor (que es típicamente asociado con, o incluso parte de, un servidor Web), el código del servlet necesario para cumplir con las etiquetas JSP es automáticamente generado, compilado y cargado en el contendor del servlet. Las partes estáticas de la página HTML son producidas enviando objetos **String** estáticos a **write()**. Las porciones dinámicas son incluidas directamente en el servlet.

Ahora, de la misma forma que el fuente JSP para la página no sea modificado, se comporta como si fuera una página HTML estática con servlets asociados (todo el código HTML es actualmente generado por el servlet, sin embargo). Si se modifica el código para el JSP, este es automáticamente compilado nuevamente y recargado la siguiente vez que la página es consultada. Claro, dado todo este dinamismo se verá una respuesta lenta para la primer vez que se accede a una JSP. Sin embargo, dado que un JSP es usualmente utilizado mucho mas de lo que es modificado, normalmente no se verá afectado por esta demora.

La estructura de una página JSP es una cruza entre un servlet y una página HTML. La etiqueta JSP comienza y termina con signos de mayor y menor, exactamente igual que una etiqueta HTML, pero las etiquetas deben incluir signos de porcentaje también, así es que todas las etiquetas JSP son indicadas por

| <% código JSP aquí %>

El primer signo de porcentaje debe estar seguido por otros caracteres que determinen el preciso tipo de código JSP en la etiqueta.

He aquí un ejemplo JSP extremadamente simple que utiliza una librería Java estándar para obtener el tiempo en milisegundos, que es entonces dividido por 1000 para producir el tiempo en segundos. Dado que una *expresión JSP* (el `<%=`) es utilizada, el resultado del cálculo es convertido en **String** y colocado en la página Web generada:

```
//:! c15:jsp>ShowSeconds.jsp
<html><body>
<H1>El tiempo den segundos es:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

En los ejemplos JSP de este libro, la primera y última línea no son incluidas den el código actual que es extraído y colocado en el árbol de código fuente del libro.

Cuando los clientes crean una petición para la página JSP, el servidor Web debe tener configurado para pasar la petición al contenedor JSP, que entonces invoca la página. Como ha sido mencionado mas arriba, el primer momento que la página es invocada, los componentes específicos para la página son generados y compilados por el contenedor JSP así como uno mas servlets. En el ejemplo mas arriba, el servlet puede contener código para configurar el objeto **HttpServletResponse**, producir un objeto **PrintWriter** (que es siempre llamado `out`), y entonces convertir el cálculo del tiempo en un **String** que es enviado **out**. Como se puede ver, todo esto se logra con una instrucción muy corta, pero el programador/diseñador Web HTML no tiene las habilidades para escribir tal código.

## Objetos implícitos

Servlets incluye clases que proporcionan utilidades convenientes, como **HttpServletRequest**, **HttpServletResponse**, **Session**, etc. Los objetos de estas clases son armados dentro de la especificación JSP y automáticamente disponibles para utilizar en su JSP sin escribir líneas extras de código. Los objetos implícitos en una JSP son detallados en la tabla que sigue.

Variable implícita	OfType ( <code>javax.servlet</code> )	Descripción	Ámbito
<b>request</b>	Subtipo de <b>HttpServletRequest</b> dependiente del protocolo	La consulta que acciona la invocación del servicio	consulta
<b>response</b>	Subtipo de <b>HttpServletResponse</b> dependiente del protocolo	La respuesta a la consulta	página
<b>pageContext</b>	<b>jsp.PageContext</b>	El contexto de la página encapsula las características dependientes de la implementación y proporciona métodos convenientes y acceso a el espacio de nombres para este JSP	página
<b>session</b>	Subtipo de <b>http.HttpSession</b> dependiente del protocolo	El objeto de sesión creado por el cliente consultado. Vea objeto de sesión de servlet	sesión
<b>application</b>	<b>ServletContext</b>	El contexto servlet obtenido de la configuración del servlet (e.g., <b>getServletConfig()</b> , <b>getContext()</b> )	aplicación
<b>out</b>	<b>jsp.JspWriter</b>	El objeto donde se escribe el flujo de salida	página
<b>config</b>	<b>ServletConfig</b>	El <b>ServletConfig</b> para <small>auto ICD</small>	página

		esta JSP	
<b>page</b>	<b>java.lang.Object</b>	La instancia para esta clase de implementación de página procesada por la consulta actual	página

El ámbito de cada objeto puede variar significativamente. Por ejemplo, el objeto sesión tiene el alcance que excede la página, cuanto mas se extiende a lo largo de muchas consultas de clientes y páginas. El objeto **application** puede proporcionar servicios para un grupo de páginas JSP que juntas representan una aplicación Web.

## Directivas JSP

Las directivas son mensajes a el contenedor JSP y son indicadas por el “@”:

```
| <%@ directive {attr="value"}* %>
| Las directivas no envían nada a el flujo de salida, pero son importantes
| configurando los atributos de página JSP y las dependencias con el
| contenedor JSP. Por ejemplo, la línea:
| <%@ page language="java" %>
indica que el lenguaje de guiones que será utilizado dentro de la página es
Java. De hecho, la especificación JSP solo describe la semántica de los
guiones para el atributo de lenguaje igual a “Java”. La ambición de esta
directiva es crear flexibilidad en la tecnología JSP. En el futuro, si se quiere
elegir otro lenguaje, digamos Python (una elección muy buena para
guiones), entonces ese lenguaje será soportado por el ambiente en tiempo de
ejecución de Java exponiendo el modelo de objetos de tecnología Java a el
ambiente de guiones, especialmente las variables implícitas definidas mas
arriba, propiedades de JavaBeans, y métodos públicos.
```

La directiva mas importante es la directiva de página. Esta define un montón de atributos dependientes de la página y comunica estos atributos a el contenedor JSP. Estos atributos incluyen: **language**, **extends**, **import**, **session**, **buffer**, **autoFlush**, **isThreadSafe**, **info** y **errorPage**. Por ejemplo:

```
| <%@ page session="true" import="java.util.*" %>
Esta línea primer indica que la página requiere participación en una sesión
HTTP. Dado que no tenemos configurada la directiva de lenguaje el
contenedor JSP por defecto utiliza Java y la variable implícita de lenguaje de
guiones llamada session es del tipojavax.servlet.http.HttpSession. Si la
directiva es falsa entonces la variable implícita session puede no estar
disponible. Si la variable session no es especificada, entonces es por defecto
verdadera.
```

El atributo **import** describe los tipos que son disponibles para el ambiente de guiones. Este atributo es utilizado igual a como sería en el lenguaje de programación Java, i.e., una lista separada con comas de expresiones **import** comunes. Esta lista es importada por la implementación traducida de la página JSP y esta disponible para el ambiente de guiones. Nuevamente, esto es actualmente definido solo cuando el valor de la directiva del lenguaje es “java”.

## Elementos para hacer guiones de JSP

Una vez que las directivas han sido utilizadas para configurar el ambiente de guiones se puede utilizar los elementos del lenguaje de guiones. JSP 1.1 Tiene tres elementos de lenguaje para crear guiones -*declaraciones*, *scriptlets*, y *expresiones*. Una declaración declarará elementos, un scriptlet es un fragmento de instrucción, y una expresión es una expresión completa del lenguaje. En JSP cada elemento de guión comienza con un “<%”. La sintaxis para cada uno es:

```
<%! declaration %>
<% scriptlet %>
<%= expression %>
```

Los espacios en blanco son opcionales luego de “<%!” , “<%”, “<%=” y antes de “%>”.

Todas estas etiquetas son basadas en XML; se podría decir incluso que una página JSP puede ser proyectado en un documento XML. La sintaxis equivalente SML para los elementos de guiones mas arriba puede ser:

```
<jsp:declaración> declaración </jsp:declaración>
<jsp:scriptlet> scriptlet </jsp:scriptlet>
<jsp:exprésion> expresión </jsp:exprésion>
```

Además, hay dos tipos de comentarios:

```
<%-- comentario jsp --%>
<!-- comentario html-->
```

La primer forma permite agregar comentarios a las páginas fuentes JSP que no aparecerán de ninguna manera en el HTML que será enviado a el cliente. Claro, la segunda forma de comentario no es específica de JSRs -es solo un comentario HTML común. Lo que es interesante es que se puede insertar código JSP dentro de un comentario HTML y el comentario aparecerá en la página resultante, incluyendo el resultado del código JSP.

Las declaraciones son utilizadas para declarar variables y métodos en el lenguaje de guiones (actualmente solo Java) utilizado en una página JSP. La declaración debe ser una instrucción Java completa y no puede producir ninguna salida en el flujo **out**. En el ejemplo **Hello.jsp** que sigue, las

declaraciones para las variables **loadTime** **loadDate** y **hitCount** son todas instrucciones de Java enteras que declaran y inician 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 %>
<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("Adiós");
    out.println("Nos vemos");
%>
</body></html>
///:~
```

Cuando se ejecuta este programa se verá que las variables **loadTime**, **loadDate** y **hitCount** almacenan sus valores entre los uno y otro hit de la página, así es que claramente son campos y no variables locales.

Al final de este ejemplo hay un scriptlet que escribe “Adiós” en la consola del servidor Web y “Nos vemos” en el objeto **out** implícito de **JspWriter**. Los scriptlets pueden contener cualquier fragmento de código que sean instrucciones válidas de Java. Scriptlets son ejecutados en tiempo de procesamiento de consulta. Cuando todos los fragmentos scriptlet en una JSP son combinados en el orden en que aparecen en la página, suelen producir una instrucción como fue definida en el lenguaje de programación Java. Donde sea que produzca o no cualquier salida en el flujo de salida **out** depende del código en el scriptlet. Se debe ser cuidadoso porque el scriptlets puede producir efectos secundarios modificando los objetos que son visibles para el.

Las expresiones JSP pueden entreverarse con el HTML en el medio de la sección de **Hello.jsp**. Las expresiones debe ser instrucciones completas de

Java, que son evaluados, convertidas a **String** y enviadas a **out**. Si los resultados de la expresión no pueden ser forzadas a **String** entonces una excepción **ClassCastException** es lanzada.

## Extrayendo campos y valores

El siguiente ejemplo es similar a uno mostrado antes en esta sección de servlet. La primera vez que se entra en la página detecta que no tiene campos y retorna a la página que contiene un formulario, utilizando el mismo código como en el ejemplo del servlet, pero en formato JSP. Cuando se envía el formulario con los campos llenos a la misma URL de JSP, detecta los campos y los despliega. Esta es una bonita técnica porque permite tener la página conteniendo el formulario para el usuario lleno y el código de respuesta para esta página en un solo fichero, de esta forma se hace fácil crear y mantener.

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Ir a buscar los datos de un formulario HTML. --%>
<%-- Esta JSP también genera el formulario. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
Enumeration flds = request.getParameterNames();
if(!flds.hasMoreElements()) { // No hay campos %}
    <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>
//:~
```

La característica más interesante de este ejemplo es que demuestra como el código de scriptlet puede ser mezclado con HTML, aún en el punto de generar HTML dentro de un bucle **for** de Java. Esto es especialmente conveniente para crear cualquier tipo de formulario con HTML repetitivo que de otra forma sería requerido.

## Atributos de una pagina JSP y alcance

Dando vueltas en la documentación HTML para servlets y JSPs, se encontrarán características que reportan información acerca del servidor o de JSP en el que se está ejecutando actualmente. El siguiente ejemplo despliega algunos de estos datos.

```
//:! c15:jsp:PageContext.jsp
<%-- Viendo los atributos en el pageContext--%>
<%-- Debe notarse que se puede incluir cualquier cantidad de código
dentro de estas etiquetas scriptlet --%>
<%@ 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 de HTML incrustado y escritura a **out** para la salida de la página HTML resultante.

La primer pieza de información producida es el nombre del servlet, que probablemente será “JSP” pero que depende de cada implementación. Se puede también descubrir la versión actual del contenedor utilizando el objeto aplicación. Finalmente, luego de configurar un atributo de sesión, el “nombre de atributo” en un ámbito en particular son desplegadas. No se utilizan los ámbitos mucho en la mayoría de la programación JSP; simplemente son mostrados aquí para agregar interés al ejemplo. Hay cuatro atributos de ámbito, como sigue; el *ámbito de página* (ámbito 1), el *alcance de consulta* (ámbito 4), basado en un objeto **ServletContext**. Hay un **ServletContext** por “aplicación Web” por Maquina Virtual de Java (Una “aplicación Web” es una colección de servlets y contenido instalado sobre un subgrupo específico de espacios de nombres URL de servidor como lo es /catálogo). Esto es generalmente configurado utilizando un fichero de

configuración). En el ámbito de aplicación se pueden ver objetos que representan rutas para el directorio de trabajo y el directorio temporal.

## Manipulando sesiones en JSP

Las sesiones fueron introducidas en secciones previas de servlets, y también están disponibles dentro de JSPs. El siguiente ejemplo ejercita el objeto **session** y permite manipular la cantidad de tiempo antes de que la sesión sea inválida.

```
//:! 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="Invalidar"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Mantenerse alrededor"></FORM>
</body></html>
//:~
```

El objeto **session** es proporcionado por defecto así es que esta disponible sin ningún código extra. La llamada a **getID()**, **getCreationTime()** y **getMaxInactiveInterval()** son utilizados para desplegar información acerca de este objeto de sesión.

Cuando se inicia por primera vez esta sesión se verá un **maxInactiveInterval** de, por ejemplo, 1800 segundos (30 minutos). Esto depende de la forma en que su contenedor JSP/servlet sea configurado. El **MaxInactiveInterval** es acortado a 5 segundos para hacer las cosas interesantes. Si se refresca la página antes de los 5 segundos de expiración, entonces se verá:

```
| Session value for "My dog" = Ralph  
Pero si se espera mas que eso, "Ralph" se convertirá en null.
```

Para ver como la información de sesión puede ser transportada a otras páginas, y también para ver el efecto de un objeto de sesión invalidado contra simplemente dejarlo expirar, otros dos objetos JSP son creados. El primero (alcanzado presionando el botón “invalidar” en **SessionObject.jsp**) lee la información de sesión y luego explícitamente invalida la sesión:

```
//:! c15:jsp:SessionObject2.jsp  
<%--El objeto session transportado --%>  
<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, refrescamos **SessionObject.jsp**, inmediatamente después hacemos clic en el botón “invalidar” para traer **SessionObject2.jsp**. En este punto todavía se verá “Ralph”, y al rato (luego de los 5 segundos del intervalo de expiración), refrescamos **SessionObject2.jsp** para ver que la sesión ha sido invalidada a la fuerza y “Ralph” ha desaparecido.

Si se va para atrás a **SessionObject.jsp**, refrescamos la página así se puede tener un nuevo intervalo de 5 segundos, luego presionamos el botón “Mantenerse alrededor”, tomara la siguiente página, **SessionObject3.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="Retornar">  
</FORM>  
</body></html>  
///:~
```

Dado que esta página no invalida la sesión, “Ralph” se colgará tanto como se mantenga la página refrescándose antes de que los 5 segundos de intervalo expiren. Esto es como una mascota “Tomagotchi” - tanto como se juegue con “Ralph” seguirá aquí, de otra forma expira.

## Creando y modificando cookies

Cookies fueron introducidas en una sección anterior de servlets. Una vez mas, lo conciso de JSPs hace que el jugar con cookies mucho mas simple

aquí que cuando se utilizan servlets. El siguiente ejemplo muestra esto trayendo las cookies que vienen con la consulta, leyéndolas y modificando las edades máximas (fechas de expiración) y enganchando una nueva cookie a la respuesta de salida:

```
//:! 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>
//:~
```

Puesto que cada navegador almacena cookies a su forma, se pueden ver diferentes comportamientos con diferentes navegadores (no da confianza, pero puede haber algún tipo de error que pueda ser corregido en el momento en que se lea esto). También se pueden experimentar diferentes resultados si se apaga el navegador y se arranca nuevamente, o mejor dicho visitar una página diferente y luego retornar a **Cookies.jsp**. De puede notar que la utilización de objetos de sesión parece ser mas robustos que utilizar cookies directamente.

Luego de desplegar el identificador de sesión, cada cookie en el arreglo de cookies que comienza con el objeto **request** es desplegado, junto con su edad máxima. La edad máxima es cambiada y desplegada nuevamente para verificar el nuevo valor, entonces una nueva cookie es agregada a la respuesta. Sin embargo, el navegador puede parecer que ignora la edad máxima; vale la pena jugar con este programa y modificar el valor máximo de edad para ver el comportamiento bajo diferentes navegadores.

## Resumen de JSP

Esta sección solo ha sido una breve cobertura de JSPs, y aún incluso con lo que ha sido cubierto aquí (junto con lo que se ha aprendido de Java en el resto del libro, y con el conocimiento de HTML propio) se puede comenzar a escribir páginas web sofisticadas mediante JSPs. La sintaxis JSP no es

particularmente profunda o complicada, así es que si se entiende lo que se ha presentado en esta sección se está listo a ser productivo con JSPs. Se puede encontrar información suplementaria en libros actualizados acerca de servlets, o en [java.sun.com](http://java.sun.com).

Es especialmente bonito tener JSPs disponibles, incluso si su meta es solo producir servlets. Se descubrirá que si se tiene alguna pregunta acerca del comportamiento de un rasgo de servlet, es mucho más fácil y rápido escribir un programa JSP de prueba para contestar esa pregunta que escribir un servlet. Parte de los beneficios vienen de tener que escribir menos código y ser capaz de mezclar el HTML desplegado con el código Java, pero el aplacamiento comienza especialmente obvio cuando se ve que el Contenedor JSP maneja todo el proceso de compilar nuevamente y la recarga del JSP por nosotros dondequiera que el fuente sea modificado.

Así como JSPs son estupendo, sin embargo, vale la pena tener en mente que la creación de JSP requiere un nivel alto de habilidades más que solo programar en Java o crear páginas Web. Además, depurar una página JSP rota no es tan fácil como depurar un programa en Java, dado que (corrientemente) los mensajes de error son más confusos. Esto cambiaría cuando los sistemas de desarrollo mejoren, pero se puede también ver otras tecnologías creadas sobre Java y la Web que son mejor adaptadas a las habilidades de un diseñador de sitios web.

## RMI (Invocación de método remoto)

Las estrategias tradicionales para ejecutar código en otras máquinas a través de la red han sido confusas así como tediosas y propensas al error al implementarlas. La forma más bonita de pensar en este problema es que algún objeto sucede que vive en otra máquina, y que se puede enviar un mensaje a ese objeto remoto y obtener un resultado como si el objeto estuviera en la máquina local. Esta simplificación es exactamente lo que la *Invocación de método remoto de Java*(RMI Remote Method Invocation) permite que se haga. Esta sección nos guía a través de los pasos necesarios para crear sus propios objetos RMI.

### Interfases remotas

RMI hace pesado el uso de interfaces. Cuando se quiera crear una objeto remoto, se enmascara la implementación de capas bajas pasando una interfase. De esta forma, cuando el cliente obtiene una referencia a un

objeto remoto, lo que se obtiene realmente es una referencia a una interfase, que *sucede* que conecta a un código cabó que conversa a través de la red. Pero no se tiene que pensar acerca de esto, solo se envían los mensajes mediante la interfase referencia.

Cuando se crea una interfase remota, se debe seguir estas directivas:

1. La interfase remota debe ser pública (no puede tener un “acceso de paquete”, esto es, no puede ser “amigable”). De otra forma, un cliente obtendrá un error cuando se intente cargar un objeto remoto que implementa una interfase remota.
2. La interfase remota debe extender la interfase **java.rmi.Remote**.
3. Cada método en la interfase remota debe declarar **java.rmi.RemoteException** en su cláusula **throws** además de cualquier excepción específica de aplicación.
4. Un objeto remoto pasado como argumento o retorna un valor (directamente o incrustado en un objeto local) debe ser declarada como la interfase remota, no la clase de implementación.

He aquí una simple interfase remota que representa un exacto servicio de tiempo:

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;
interface PerfectTimeI extends Remote {
    long getPerfectTime() throws RemoteException;
} ///:~
```

Se ve como cualquier otra interfase excepto que extiende **Remote** y todos sus métodos lanzan **RemoteException**. Recuerde que una **interface** y todos sus métodos son automáticamente **public**.

## Implementando la interfase remota

El servidor debe contener una clase que extiende **UnicastRemoteObject** e implementar la interfase remota. Esta clase puede también tener métodos adicionales, pero solo los métodos en la interfase remota están disponibles para el cliente, claro, dado que el cliente obtendrá solo una referencia a la interfase, no la clase que la implementa.

Se debe definir explícitamente el constructor para el objeto remoto aún si solo se está definiendo un constructor por defecto que llame a el constructor de la clase base. Debe escribir dado que debe lanzar la **RemoteException**.

He aquí la implementación de la interfase remota **PerfectTimeI**:

```
//: c15:rmi:PerfectTime.java
```

```

// La implementación de
// el objeto remoto PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
public class PerfectTime
extends UnicastRemoteObject
implements PerfectTimeI {
    // Implementación de la interfase:
    public long getPerfectTime()
    throws RemoteException {
        return System.currentTimeMillis();
    }
    // Se debe implementar un constructor
    // para lanzar la RemoteException:
    public PerfectTime() throws RemoteException {
        // super(); // Llamado automáticamente
    }
    // Registración para servicio RMI. Lanzando
    // excepciones a la consola.
    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 la configuración del servidor. Cuando se están sirviendo objetos RMI, en algún punto del programa se debe.

1. Crear e instalar un manejador de seguridad que soporte RMI. El único disponible para RMS como parte de la distribución Java es **RMISecurityManager**.
2. Crear una o mas instancias de un objeto remoto. Aquí, se puede ver la creación del objeto **PerfectTime**.
3. Registre al menos una o mas instancias de un objeto remoto con el registro de objetos remotos RMI para propósitos de bootstrapping. Un objeto remoto puede tener métodos que producen referencias a otros objetos remotos. Esto permite configurarlos así el cliente puede ir al registro solo una vez, para tener el primer objeto remoto.

## Configurando el registro

Aquí, se vera una llamada a el método estático **Naming.bind()**. Sin embargo, esta llamada requiere que el registro sea ejecutado como un proceso separado en la computadora. El nombre del servidor de registro es **rmiregistry**, y bajo Windows de 32 bits se escribiría:

```
| start rmiregistry  
para ejecutarlo como tarea de fondo. En UNIX, este comando es:
```

```
| rmiregistry &  
Como muchos programas de redes, el rmiregistry está localizado en la dirección IP de cualquier máquina encendida, pero debe estar escuchando un puerto. Si se invoca el rmiregistry como mas arriba, sin argumentos, el puerto por defecto será 1099. Si se quiere algún otro puerto, se puede agregar un argumento en la línea de comandos para especificar el puerto. Para este ejemplo, el puerto el localizado en el 2005, así es que el rmiregistry debe ser iniciado de la siguiente forma en Windows 32 bits:
```

```
| start rmiregistry 2005  
y para UNIX:
```

```
| rmiregistry 2005 &  
La información acerca del puerto debe ser dada a el comando bind(), así como la dirección IP de la máquina donde el registro está localizado. Pero esto trae lo que puede ser un problema frustrante si se está esperando probar programas RMI de forma local de la forma en que los programas de red han sido probados antes en este capítulo. En la versión de JDK 1.1.1, hay algunos problemas5:
```

1. **localhost** no trabaja con RMI. De esta forma, para experimentar con RMI en una máquina solitaria, se debe proporcionar el nombre de la máquina. Para encontrar el nombre de la máquina bajo Windows de 32 bits, hay que ir a el Panel de control y seleccionar "Red". La lengüeta de "identificación", y se verá el nombre de la computadora. En mi caso, He llamado mi computadora "Peppy". Las mayúsculas son ignoradas.
2. RMI no trabaja a no ser que su computadora tenga activa una conexión TCP/IP, aún si todos los componentes simplemente están hablando con máquinas locales. Esto significa que se debe conectar a su proveedor de servicios antes de tratar de ejecutar el programa o se obtendrán algunos mensajes de excepción confusos.

Con todo esto en mente, el comando **bind()** se convierte:

```
| Naming.bind("//peppy:2005/PerfectTime", pt);
```

---

<sup>5</sup> Muchas neuronas murieron en agonía al descubrir esta información

Si se esta utilizando el puerto por defecto 1099, no necesita especificar un puerto así es que se puede decir:

```
| Naming.bind("//peppy/PerfectTime", pt);
```

Se debería ser capas de realizar pruebas locales dejando la dirección IP y utilizando solo el identificador:

```
| Naming.bind("PerfectTime", pt);
```

El nombre del servicio es arbitrario; sucede ser PerfectTime aquí, simplemente como el nombre de la clase, pero se podría solo llamarlo como se quiera. Lo importante es que es un único nombre en el registro que el cliente conoce para buscar procurar el objeto remoto. Si el nombre ya está en el registro, se debe obtener una **AlreadyBoundException**. Para prevenir esto, se puede siempre utilizar **rebind()** en lugar de **bind()**, dado que **rebind()** siempre agrega una nueva entrada o reemplaza la que ya está.

Aún cuando **main()** salga, el objeto puede ser creado y registrado así es que se mantiene vivo en el registro, esperando por un cliente que venga y consulte. Siembre y cuando el **rmiregistry** este ejecutándose y no se llame a **Naming.unbind()** en su nombre, el objeto estará ahí. Por esta razón, cuando se este desarrollando su código se necesita bajar el **rmiregistry** y reiniciarlo cuando se compile una nueva versión de su objeto remoto.

No se esta forzado a iniciar **rmiregistry** como un proceso externo. Si se conoce que la aplicación es la única que esta utilizando el registro se puede iniciar dentro del programa con esta linea:

```
| LocateRegistry.createRegistry(2005);
```

Como antes, 2005 es el número de puerto que será utilizado en este ejemplo. Este es el equivalente de ejecutar **rmiregistry 2005** en una línea de comando, pero se puede a menudo ser mas convincente cuando se esta desarrollando código RMI dado que elimina los pasos extras de ejecutar y parar el registro. Una vez que se ha ejecutado este código, se puede **bind()** utilizando **Naming** como antes.

## Creando cabos y esqueletos

Si se compila y ejecuta **PerfectTime.java**. No funcionara incluso si se tiene el **tmiregistry** ejecutándose correctamente. Esto es porque el marco de trabajo para RMI no esta ahí todavía. Se debe crear primero los cabos y el esqueleto que proporcionan las operaciones de conexiones de red y permiten pretender que el objeto remoto es simplemente otro objeto local en su máquina.

Lo que sucede detrás de la escena es complejo. Cualquier objeto que se pase dentro o retorne de un objeto remoto debe **implement Serializable** (si se quiere pasar referencias remotas en lugar de los objetos enteros, el argumento de los objetos pueden**implement Remote**) , así es que se puede

imaginar que los cabos y esqueleto realizan serialización automática y su inversa cuando “pongan en orden” todos los argumentos a través de la red y retornen el resultado. Afortunadamente, no se tiene que conocer nada de esto, pero se *tiene* que crear los cabos y esqueleto. Esto es un proceso simple: se invoca la herramienta **rmic** en el código compilado, y este crea los ficheros necesarios. Así es que el único requerimiento es que otro paso sea agregado a su proceso de compilación.

La herramienta **rmic** es particular acerca de los paquetes y los caminos de cases, sin embargo. **PerfectTime.java** esta en el **package c15.rmi**, e incluso si se invoca **rmic** en el mismo directorio en donde **PerfectTime.class** está localizado, **rmic** no encontrará el fichero, dado que busca en la ruta de clases. Así es que se debe especificar la localización de la ruta de la clase, de esta forma;

```
| rmic c15.rmi.PerfectTime
```

No se tiene que estar en el directorio que contiene **PerfectTime.class** cuando se ejecute este comando, pero los resultados serán colocados en el directorio actual.

Cuando **rmic** ejecute exitosamente, se tendrán dos nuevas clases en el directorio:

```
| PerfectTime_Stub.class
| PerfectTime_Skel.class
```

correspondientes a el cabo y a el esqueleto. Ahora se esta listo para que el cliente y el servidor conversen unos con otros.

## Utilizando el objeto remoto

El punto de RMI es hacer el uso de los objetos remotos simple. La cosa extra que se debe hacer en el programa cliente es buscar y traer la interfase remota del servidor. Desde ahora, simplemente es programación Java normal: enviar mensajes a objetos. Aquí esta el programa que utiliza **PerfecTime**:

```
//: c15:rmi:DisplayPerfectTime.java
// Utiliza el objeto remoto 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 ID es lo mismo que la utilizada para registrar el objeto con **Naming**, y la primer parte representa la URL y el número de puerto. Dado que se está utilizando una URL, se puede también especificar una máquina en Internet.

Lo que regresa desde **Naming.lookup()** debe ser convertido a la interfase remota, *no* a la clase. Si se utiliza la clase en lugar de esto, se obtendrá una excepción.

Se puede ver en la llamada a método

```
| t.getPerfectTime()
| una vez que se tenga la referencia al objeto remoto, la programación con
| esta es indistinguible de la programación con un objeto local (con una
| diferencia: los métodos remotos lanzan RemoteException).
```

## CORBA

En grandes, aplicaciones distribuidas, se necesita pueden no ser satisfechas por las estrategias anteriores. Por ejemplo, se puede querer realizar una conexión con datos heredados almacenados, o se pueden necesitar servicios de un objeto servidor independientemente de su localización física. Estas situaciones requieren algún tipo de llamada a procedimiento remoto (RCP Remote Procedure Call), y posiblemente independencia del lenguaje. Esto es donde CORBA puede ayudarnos.

CORBA no es una característica del lenguaje; es una tecnología de integración. Es una especificación que los vendedores pueden seguir para implementar productos integrables compatibles con CORBA. CORBA es parte de un esfuerzo de OMG para definir un marco de trabajo estándar para interoperabilidad de objetos distribuidos independientes del lenguaje.

CORBA proporciona la habilidad de hacer llamadas a procedimientos remotos en objetos Java y no Java, e interconecta con sistemas heredados en una forma transparente de la localización. Java agrega soporte de redes y un bonito lenguaje orientado a objetos para crear aplicaciones gráficas y no gráficas. Java y el modelo de objetos OMG se pueden corresponder muy bien unos con otros; por ejemplo, Java y CORBA implementan el concepto de interfase y modelo de referencias a objetos.

### Principios básicos de CORBA

La especificación de interoperabilidad de objetos desarrollada por la OMG es referida comúnmente como Arquitectura de manejo de objetos (OAM

Object Management Architecture) - La OMA define dos componentes: el modelo de corazón de objeto y la arquitectura de referencia OMA. El modelo de corazón de objeto declara los conceptos básicos de objeto, interfase, operación y mas (CORBA es un refinamiento del modelo de corazón de objeto). La arquitectura de referencia OMA define una infraestructura de capas bajas de servicios y mecanismos que permiten a los objetos operar entre ellos. La arquitectura de referencia OMA incluye el intermediario de consulta de objeto (ORB Object Request Broker), Servicios de objetos (También conocido como servicios CORBA), y facilidades en común.

El ORB es el canal de comunicación por el cual los objetos pueden consultar servicios de otros objetos, independientemente de la localización física. Esto significa que lo que se ve como una llamada a un método en el código del cliente es actualmente una operación compleja. Primero, una conexión con el objeto servidor debe existir, y para crear una conexión el ORB debe conocer donde reside el código de la implementación del servidor. Una vez que la conexión es establecida, el argumento del método debe ser colocado en orden, i.e. convertido en un flujo binario para ser enviado a través de la red. Otra información que debe ser enviada es el nombre de la máquina servidor, el proceso servidor y la identidad del objeto servidor dentro de ese proceso. Finalmente, esta información es enviada a través de un protocolo de cable de bajo nivel, la información es codificada en el lado del servidor, y la llamada es ejecutada. El ORB oculta toda esta complejidad del programador y hace la operación al menos tan simple como llamar a un objeto local.

No hay una especificación de como un corazón ORB podría ser implementado, pero para proporcionar una compatibilidad básica entre diferentes vendedores de ORB, la OMG define un grupo de servicios que son accesibles a través de interfaces estándares.

## Lenguaje de definición de interfase de CORBA (IDL Interfase Definition Language)

CORBA esta diseñando para lograr una transparencia de lenguajes: un objeto cliente puede llamar a métodos en un objeto servidor de diferentes clases, sin importar el lenguaje con el que está implementado. Claro, el objeto cliente debe conocer los nombres y firmas de los métodos que el servidor de objetos expone. Esto es donde IDL entra. El CORBA IDL es una forma neutral de lenguaje para especificar tipos de datos, atributos, operaciones, interfaces y mas. La sintaxis de IDL es similar a la sintaxis de C++, o la de Java. La siguiente tabla muestra la correspondencia entre algunos de los conceptos comunes a res lenguajes que pueden ser especificados a través de IDL de CORBA:

<b>CORBA IDL</b>	<b>Java</b>	<b>C++</b>
Módulo	Paquete	Espacio de nombres
Interfase	Interfase	clase abstracta pura
Método	Método	Función miembro

El concepto de herencia también es soportado, utilizando el operador dos puntos como en C++. El programador escribe una descripción IDL de los atributos, métodos e interfaces que son implementadas y utilizadas por el servidor y los clientes. El IDL es entonces compilado por un compilador IDL/Java proporcionado por el vendedor, que lee el fuente y genera el código Java.

El compilador IDL es una herramienta extremadamente útil: no solo genera un fuente equivalente de Java para el IDL, también genera el código que será utilizado para ordenar los argumentos de métodos y hacer las llamadas remotas. Este código, llamado el código cabó y esqueleto, es organizado en muchos fichero fuente Java y es usualmente parte del mismo paquete Java.

## El servicio de nombres

El servicio de nombres es uno de los servicios fundamentales de CORBA. Un objeto CORBA es accedido a través de una referencia, un pedazo de información que no es significativo para el lector humano. Pero las referencias pueden ser cadenas de nombres definidas por el programador asignadas. Esta operación es conocida como *stringifying la referencia* y uno de los componentes OMA, el servicio de nombres, es devoto de realizar conversiones de cadenas a objetos y de objetos a cadenas y su correspondencia. Dado que el servicio de nombres actúa como un directorio telefónico el cliente y el servidor pueden consultar y manipular, este se ejecuta como un proceso separado. La creación de una correspondencia de objeto a cadena es llamada *vincular(binding)* un objeto, y quitar la correspondencia es llamado *desvincular(unbinding)*. Tomar una referencia a un objeto pasando una cadena es llamado *resolver el nombre*.

Por ejemplo, al inicio, una aplicación de servidor puede crear un objeto servidor, ligar el objeto en el nombre de servicio, y entonces esperar por clientes para hacer consultas. Un cliente primero obtiene una referencia a un objeto servidor, resolviendo el nombre de cadena, y luego puede hacer llamadas en el servidor utilizando la referencia.

Nuevamente, la especificación de Servicio de Nombres es parte de CORBA, pero la aplicación que la implementa es proporcionada por el vendedor ORB. La forma a la que se tiene acceso a la funcionalidad del Servicio de Nombres puede variar de vendedor a vendedor.

# Un ejemplo

El código mostrado aquí no será elaborado a causa de que los distintos ORBs tienen diferentes formas de acceder a los servicios CORBA, así es que los ejemplos son específicos de los vendedores. (El ejemplo mas abajo utiliza JavaIDL, un producto libre de Sun que viene con un ORB liviano, un servicio de nombres, y un compilador IDL-a-Java). Además, dado que Java es joven y sigue evolucionando, no todas las características CORBA están presente en los distintos productos Java/CORBA.

Queremos implementar un servidor, que se ejecute en alguna máquina, que pueda ser consultado por la hora exacta. También queremos implementar un cliente que pregunte por la hora exacta. En este caso implementaremos ambos programas en Java, pero podemos utilizar también dos lenguajes diferentes (lo que a menudo sucede en situaciones reales).

## Escribiendo el fuente IDL

El primer paso es escribir la descripción IDL de los servicios proporcionados. Esto es hecho usualmente por el programador del servidor, que es entonces libre de implementar el servidor en cualquier lenguaje para el que exista un compilador IDL CORBA. El fichero IDL es distribuido a el lado del programador y comienza el puente entre lenguajes.

El siguiente ejemplo muestra la descripción IDL para nuestro servidor **ExactTime**:

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

Esto es una declaración de la interfase **ExactTime** dentro de el espacio de nombres **remotetime**. La interfase es construida con un solo método que retorna la hora actual en formato **string**.

## Creando cabos y esqueletos

El segundo paso es compilar el IDL para crear el cabo y el esqueleto del código Java que será utilizado para implementar el cliente y el servidor. La herramienta que viene con el producto JavaIDK es **idltojava**:

```
| idltojava remotetime.idl
```

Esto automáticamente generará código para ambos, el cabo y el esqueleto. **Idltojava** genera un **package** Java nombrado luego del módulo IDL, **remotetime**, y los ficheros generados son colocados en el subdirectorio **remotetime**. **\_ExactTimeImplBase.java** es el esqueleto que usaremos para implementar el objeto servidor, y **\_ExactTimeStub.java** será utilizado por el cliente. Hay representaciones Java de la interfase IDL en **ExactTime.java** y en un par de otros ficheros soportados utilizados, por ejemplo, para facilitar el acceso a las operaciones de servicio de nombres.

## Implementando el servidor y el cliente

Mas adelante se podrá ver el código para el lado del servidor. La implementación de el objeto servidor es en la clase **ExactTimeServer**. El **RemoteTimeServer** es la aplicación que crea un objeto servidor, registra este con el ORB, dando un nombre a la referencia al objeto, y entonces se sienta tranquilo a esperar por las consultas de los clientes.

```
//: 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()));
    }
}
// Implementación de la apliación remota
public class RemoteTimeServer {
    // Lanza excepciones a la consola:
    public static void main(String[] args)
        throws Exception {
        // Creación e inicialización ORB:
        ORB orb = ORB.init(args, null);
        // Crea el objeto servidor y lo registra:
        ExactTimeServer timeServerObjRef =
            new ExactTimeServer();
        orb.connect(timeServerObjRef);
        // Obtiene el nombre de contexto raíz:
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references(
                "NameService");
        NamingContext ncRef =
            NamingContextHelper.narrow(objRef);
        // Asigna una cadena de nombre a la
        // referencia a objeto (binding):
```

```

        NameComponent nc =
            new NameComponent("ExactTime", "");
        NameComponent[] path = { nc };
        ncRef.rebind(path, timeServerObjRef);
        // Espera por las consultas del cliente:
        java.lang.Object sync =
            new java.lang.Object();
        synchronized(sync){
            sync.wait();
        }
    }
} //:~
}

```

Como se puede ver, implementar el objeto servidor es simple; es una clase Java común que hereda del código del esqueleto generado por el compilador IDL. Las cosas se ponen un poco mas complicadas cuando llegamos a interactuar con el ORB y otros servicios CORBA.

## Algunos servicios CORBA

Esta es una descripción corta de que esta haciendo el código relacionado a JavaIDK (primeramente ignorando la parte del código CORBA que es dependiente del vendedor). La primera linea en el **main()** inicia el ORB, y por su puesto, esto es porque nuestro objeto servidor necesitará interactuar con el. Justo antes de la inicialización ORB, un objeto servidor es creado. Actualmente, el término correcto puede ser un *objeto servidor pasajero* (*transient servant object*), un objeto que recibe consultas de clientes, y cuyo tiempo de vida es el mismo que el proceso que lo crea. Una ves que el objeto servidor pasajero es creado, este es registrado con el ORB, que significa que el ORB conoce de su existencia y puede ahora reenviar consultas a el.

Llegado a este punto, todo lo que tenemos es **timeServerObjRef**, una referencia a objeto que es conocida solo dentro del proceso de servidor actual. El próximo paso será asignar un campo de cadena de nombre a este objeto servidor; los clientes utilizarán este nombre para localizar el objeto servidor. Logramos esta operación utilizando el Servicio de Nombres. Primero, necesitamos una referencia a objeto para el Servicio de Nombres; la llamada a **resolve\_initial\_references()** toma el campo de nombre de la referencia a objeto del Servicio de Nombres que es "NameService", en JavaIDL, y retorna una referencia a objeto. Esto es convertido a una referencia **NamingContext** específica utilizando el método **narrow()**. Podemos utilizar ahora el servicio de nombres.

Para ligar el objeto servidor con una referencia a objeto de cadena de campo, primero creamos un objeto **NameComponent**, lo inicializamos con "ExactTime", la cadena de nombres que queremos ligar a el objeto servidor. Entonces, utilizando el método **rebind()**, la referencia a la cadena de campo es ligada a la referencia a objeto. Utilizamos **rebind()** para asignar una referencia que ya existe. Un nombre es creado en CORBA por una secuencia

de Contextos de Nombre -esto es por lo cual utilizamos un arreglo para ligar el nombre a la referencia a objeto.

El objeto servidor esta listo finalmente para ser utilizado por los clientes. En este punto, el proceso servidor entra en un estado de espera. Nuevamente, esto es porque es un servidor pasajero, así es que su tiempo de vida es confinado a el proceso del servidor. JavaIDL no soporta actualmente objetos persistentes -objetos que sobreviven la ejecución del proceso que los ha creado.

Ahora tenemos una idea de que esta haciendo el código del servidor, veamos el código del cliente:

```
//: c15:corba:RemoteTimeClient.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class RemoteTimeClient {
    // Lanza excepciones a la consola:
    public static void main(String[] args)
        throws Exception {
        // creación ORB e inicialización:
        ORB orb = ORB.init(args, null);
        // Obtiene el contexto de nombres raiz:
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references(
                "NameService");
        NamingContext ncRef =
            NamingContextHelper.narrow(objRef);
        // Obtiene (resuelve) la referencia a el objeto
        // campo cadena para el servidor de hora.
        NameComponent nc =
            new NameComponent("ExactTime", "");
        NameComponent[] path = { nc };
        ExactTime timeObjRef =
            ExactTimeHelper.narrow(
            ncRef.resolve(path));
        // Realiza consultas a el objeto servidor:
        String exactTime = timeObjRef.getTime();
        System.out.println(exactTime);
    }
} ///:~
```

Las primeras líneas hacen lo mismo que hacen el en proceso servidor: el ORB es inicializado y una referencia a el servicio de nombres es resuelta. Luego, necesitamos una referencia a objeto para el objeto servidor, así es que pasamos la referencia a el objeto campo cadena a el método **resolve()**, y convertimos el resultado en una referencia a la interfase **ExactRime** utilizando el método **narrow()**. Finalmente, llamamos a **getTime()**.

## Activando el proceso servicio de nombres

Finalmente tenemos una aplicación servidor y una aplicación cliente lista para inter operar. Se ha visto que ambos necesitan el servicio de nombres para ligar y resolver las referencias a el objeto campo cadena. Se debe iniciar el proceso de servicio de nombres antes de ejecutar 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 diferente con otros productos. El servicio de nombres de JavaIDL se ejecuta dentro de una instancia de la JVM y escucha por defecto en el puerto de red 900.

## Activando el servidor y el cliente

Ahora esta listo para arrancar las aplicaciones servidor y cliente (en ese orden, dado que el servidor es pasajero). Si todo esta configurado correctamente, lo que se obtendrá es una sola línea de salida en la ventana de la consola del cliente, dando la hora actual. Claro, esto puede no ser muy excitante por si solo, pero se debería tener en cuenta una cosa : incluso si están en la misma máquina físicamente, las aplicaciones cliente y servidor se están ejecutando en máquinas virtuales diferentes y pueden comunicarse mediante capas de integración bajas, la ORB y el Servicio de Nombres.

Este es un simple ejemplo, diseñado para trabajar sin una red, pero un ORB es configurado usualmente para ser transparente con la localización. Cuando el servidor y el cliente están en diferentes máquina, la ORB puede resolver referencias a campos de nombres remotas utilizando un componente conocido como la *Implementation repository*. A pesar de que la Implementation Repository es parte de CORBA, no hay casi especificación, así es que difiere de vendedor a vendedor.

Como se puede ver, hay mucho mas de CORBA que lo que ha sido cubierto aquí, pero tendremos una idea básica. Si se quiere mas información acerca de CORBA, el lugar para comenzar es el sitio Web OMG, en [www.omg.org](http://www.omg.org). Ahí encontraremos documentación, documentos cortos, procedimientos y referencias a otras fuentes y productos CORBA.

## Applets de Java y CORBA

Los applets de Java pueden actuar como clientes CORBA. De esta forma, un applet puede acceder a información y servicios remotos expuestos como objetos CORBA. Pero un applet puede conectarse solo con el servidor de donde fue descargado, así es que todos los objetos CORBA con los que el applet interactúa deben estar en ese servidor. Esto es lo opuesto a lo que CORBA trata de hacer: darla transparencia total de la localización.

Esto es un tema de seguridad de redes. Si esta en una intranet, una solución es perder las restricciones de seguridad en el navegador. O, configurar políticas de seguridad en un firewall para conectarse con servidores externos.

Algunos productos ORB de Java ofrecen soluciones propietarias para este problema. Por ejemplo, algunas implementación que son llamadas Tunneling HTTP, mientras que otras tienen características especiales de firewall.

Este es un tema muy complejo para ser cubierto en un apéndice, pero es definitivamente algo de lo cual se tiene que tener cuidado.

## CORBA versus RMI

Se ve que una de las características principales de CORBA es el soporte RPC, que permite a los objetos locales llamar a métodos en objetos remotos. Claro, ya hay una característica nativa de Java que hace exactamente lo mismo: RMI (vea el capítulo 15). Mientras que RMI hace posible la RPC entre objetos Java, CORBA hace posible RPC entre objetos implementados en cualquier lenguaje. Esto es una gran diferencia.

Sin embargo, RMI puede ser utilizada para llamar servicios en una máquina remota, con código que no sea Java. Todo lo que se necesita es algún tipo de objeto Java envoltorio alrededor del código que no es Java en el lado del servidor. El objeto envoltorio se conecta externamente a clientes Java mediante RMI, e internamente se conecta a el código que no es Java utilizando uno de las técnicas mostradas arriba, como JNI o J/Direct.

Esta estrategia requiere que se escriba una capa de integración, que es exactamente lo que CORBA hace por nosotros, pero entonces no se necesitará una ORB de terceros.

## JavaBeans corporativos (Enterprise JavaBeans)

Supongamos<sup>6</sup> que necesitamos desarrollar una aplicación con hileras múltiples para ver y actualizar registros en una base de datos a través de una interfase Web. Se puede escribir un aplicación de bases de datos utilizando JDBC, una interfase Web utilizando JSP/servlets, y un sistema distribuido utilizando CORBA/RMI. ¿Pero que consideraciones extra debe hacer

---

<sup>6</sup> Esta sección fue contribución de Rober Castaneda, con ayuda de Dave Bartlett

cuando se desarrolla un sistema de objeto distribuidos en lugar de solo conocer las APIs? Aquí están los temas:

**Rendimiento:** Los objetos distribuidos que se crean deben tener buen rendimiento, como pueden potencialmente servir a muchos clientes a la vez. Se necesita utilizar técnicas de optimización como capturar recursos cooperativos como conexiones a bases de datos. Se deberán manejar también el tiempo de vida de sus objetos distribuidos.

**Escalabilidad:** Los objetos distribuidos deben ser también escalables. La escalabilidad en una aplicación distribuida significa que el número de instancias de sus objetos distribuidos pueden ser incrementados y movido a máquinas adicionales sin modificar ningún código.

**Seguridad:** Un objeto distribuido debe a menudo manejar la autorización de los clientes que acceden al el. De forma ideal, se puede agregar nuevos usuarios y roles a esta sin volver a compilar.

**Transacciones distribuidas:** Un objeto distribuido puede ser capaz de referenciar transacciones distribuidas transparentemente. Por ejemplo, si se está trabajando con dos bases de datos separadas, se debe ser capas de actualizarlas simultáneamente sin dentro de la misma transacción y recuperarla si no se reúne un cierto criterio.

**Reusabilidad:** El objeto distribuido ideal puede ser forzosamente movido a servidores de aplicación de otros vendedores. Esto puede ser bonito si se puede revender un componente objeto distribuido sin hacer modificaciones especiales, o comprar algún otro componente y utilizarlo sin tener que compilarlo o escribirlo nuevamente.

**Disponibilidad:** Si una de las máquinas en el sistema se cae, los clientes fallarán automáticamente sobre las copias de seguridad de los objetos que se ejecutan en otras máquinas.

Estas consideraciones, además de los problemas de negocios que se deben de resolver, pueden hacer un proyecto de desarrollo desalentador. Sin embargo, todo los temas *exceptuando* los problemas de negocios son redundantes -las soluciones deben ser reinventadas para cada aplicación de negocios distribuida.

Sun, junto con otros vendedores líderes de objetos distribuidos, se dieron cuenta que tarde o temprano cada equipo de desarrollo pretende reinventar estas soluciones en particular, así es que ellos han creado la especificación para los JavaBeans corporativos (EJB). EJB describe un modelo de componente del lado del servidor que aborda todas las consideraciones mencionadas arriba utilizando una estrategia estándar que permite a los desarrolladores crear componentes de negocios llamados EJBs que son ahilados del código de “plomería” de bajo nivel y se enfocan exclusivamente

en proporcionar lógica de negocios. Dado que EJBs son definidos en una forma estándar, son independientes de los vendedores.

## JavaBeans versus EJBs

Dada la similitud de los nombres, hay mucha confusión acerca de la relación entre el modelo de componentes de JavaBeans y la especificación de los JavaBeans corporativos. Mientras que ambas especificaciones comparten los mismos objetivos promoviendo la reutilización y la portabilidad del código Java entre desarrollo y herramientas de desarrollo con la utilización de patrones de diseño estándar, los motivos detrás de cada especificación son engranados a solucionar diferentes problemas.

Los estándares definidos en el modelo de los componentes JavaBeans están diseñados para crear componentes reutilizables que son típicamente utilizados en herramientas de desarrollos IDE y son comúnmente, a pesar de que no exclusivamente, componentes visuales.

La especificación de los JavaBean corporativos define un modelo de componentes para desarrollar código Java del lado del servidor. Dado que los EJBs pueden potencialmente ejecutar en muchas plataformas diferentes del lado del servidor -incluyendo servidores centrales que no tienen despliegues visuales- un EJB no puede hacer uso de librerías gráficas como AWT o Swing.

## La especificación EJB

La especificación para JavaBeans corporativos describe un modelo de componentes del lado del servidor. Este define seis roles que son utilizado para realizar las tareas en desarrollo y despliegue así como definiendo los componentes del sistema. Estos roles son utilizados en el desarrollo, despliegue y ejecución de un sistema distribuido. Vendedores, administradores y desarrolladores juegan los distintos roles, para permitir el la división del conocimiento técnico y del tipo de organización. El vendedor proporciona un marco de trabajo acertado técnicamente y el desarrollador crea los componentes específicos del tipo de organización; por ejemplo, un componente de "cuenta". El mismo grupo puede realizar uno o mas roles. Los roles definidos en la especificación EJB son resumidos en la siguiente tabla:

Rol	Responsabilidad
Proveedor de Bean corporativo	El desarrollador responsable por crear componentes EJB reutilizables. Estos componentes son empaquetados en un fichero jar

	especial (ejb-jar file)
Ensamblador de aplicación	Crea y ensambla aplicaciones de una colección de fichero ejb-jar. Esto incluye la escritura de aplicaciones que utilizan las colecciones de EJBs (e.g. servlets, JSP, Swing, ect, ect.).
Deployer	Toma una colección de fichero ejb-jar del ensamblador y/o proveedor de Bean y los despliega en un ambiente de tiempo de ejecución: uno o mas contenedores EJB .
Proveedor de Contenedores/Servidores EJB	Proporciona un ambiente en tiempo de ejecución y herramientas que son utilizadas para desplegar, administrar y ejecutar componentes EJB.
Administrador de sistemas	Maneja los diferentes componentes y servicios de tal forma que ellos pueden ser configurados y puedan interactuar correctamente, de la misma forma que asegurar que el sistema este levantado y funcionando.

## Componentes EJB

Los componentes EJB son elementos de lógica de negocios reutilizables que se adhieren a estrictos estándares y patrones de diseño como se define en la especificación EJB. Esto permite que los componentes sean portátiles. Esto también permite a otros servicios -como seguridad, ocultación y transacciones distribuidas- realizar en representación de los componentes. Un proveedor de Bean es responsable por el desarrollo de componentes EJB.

### Contenedor y servidor EJB

El *contenedor EJB* es un ambiente en tiempo de ejecución que contiene y ejecuta componentes EJB y proporciona un grupo de servicios estándares para aquellos componentes. Las responsabilidades de un contenedor EJB son estrechamente definidas en la especificación para permitir la neutralidad del vendedor. El contenedor EJB proporciona la “plomería” de bajo nivel de EJB, incluyendo las transacciones distribuidas, seguridad,

manejo del ciclo de vida de los Beans, ocultamiento, hilado y manejo de sesión. El proveedor de contenedores EJB es responsable de proporcionar un contenedor EJB.

Un *servidor EJB* es definido como un servidor de aplicación que contiene y ejecuta uno o más contenedores EJB. El Proveedor de servidor EJB es responsable por proporcionar un servidor EJB. Se puede generalmente asumir que el contenedor EJB y el Servidor EJB son lo mismo.

## Java Naming e Interfase de Directorio (JNDI) Java Naming and Directory Interface)

Java Naming e Interfase de Directorio (JNDI) es utilizado en JavaBean corporativos como el servicio de nombres para componentes EJB en la red y otros servicios de contenedores como las transacciones. Los mapas JNDI muy cerrado para otros estándares de nombres y directorios como CORBA CosNaming y pueden ser actualmente implementados como envoltura encima de ellos.

## API de Transacciones Java/Servicio de Transacciones JAvA (JTA/JTS Java Transaction API/Java Transaction Service)

JTA/JTS es utilizado en JavaBean corporativos como la API transaccional. Un proveedor de Bean corporativos puede utilizar el JTS para crear código transaccional, a pesar de que el contenedor EJB comúnmente implementa transacciones en EJB, en representación de los componentes EJB. El que despliega puede definir los atributos transaccionales de un componente EJB en tiempo de despliegue. El Contenedor EJB es responsable por manejar la transacción cuando es local o distribuida. La especificación JTS es la correspondencia Java a el CORBA OTS (Object Transaction Service)

## CORBA y RMI/IIOP

La especificación EJB define interoperabilidad con CORBA a través de la compatibilidad con protocolos CORBA. Esto se logra realizando una correspondencia entre servicios EJB como lo es JTS y JNDI y los servicios CORBA y la implementación de RMI encima del protocolo CORBA IIOP.

El uso de CORBA y RMI/IIOP en JavaBeans corporativos es implementados en el contenedor EJB y es el responsable de proporcionar el Contenedor EJB. El uso de CORBA y RMI/IIOP en el Contenedor EJB esta oculto del Componente EJB por si solo. Esto significa que el Proveedor de Bean Corporativo puede escribir sus componentes EJB y desplegarlo en cualquier

Contenedor EJB sin ninguna consideración de que protocolo de comunicaciones se está utilizando.

## Las partes de un componente EJB

Un EJB consiste en partes, incluyendo el Bean por si mismo, la implementación de algunas interfaces, y un fichero de información. Todo está empaquetado junto en un fichero jar especial.

### Bean Corporativo

Un Bean Corporativo es una clase Java que desarrolla el Proveedor de Beans Corporativos. Este implementa y proporciona la implementación de los métodos de negocios que el componente realiza. Esta clase no implementa ninguna autorización, autenticación, hilado múltiple o código transaccional.

### Interfase Home

Cada Bean Corporativo que es creado debe tener una interfase Home asociada. La interfase Home es utilizada como una fábrica de sus EJB. Los clientes utilizan la interfase Home para encontrar una instancia de la EJB o crear una nueva instancia de la EJB.

### Interfase Remota

La interfase remota es una Interfase Java que refleja los métodos del Bean Corporativo que se desea exponer al mundo exterior. La interfase remota juega un rol similar a una interfase IDL de CORBA.

### Descriptor de despliegue

El descriptor de despliegue es un fichero XML que contiene información acerca de su EJB. La utilización de XML permite desplegarse para cambiar atributos fácilmente del EJB. Los atributos configurables definidos en el descriptor de despliegue incluyen:

Los nombres de la interfase Home y Remota que son requeridos por el EJB.

El nombre para publicar dentro de JNDI para sus interfaces Home de EJBs

Los atributos transaccionales para cada método en su EJB

La lista de control de acceso por autentificación

## EJB-Jar file

El fichero jar de EJB es un fichero jar de java normal que contiene el EJB, las interfaces Home y Remota, así como el descriptor de despliegue.

## EJB operación

Una vez que se tiene un fichero EJB-Jar conteniendo el Bean, la interfaces Home y Remote, y el descriptor de despliegue, se puede colocar todas las partes juntas y en el proceso de entender por que las interfaces Home y Remota se necesitan y como el Contenedor EJB las utiliza.

El contenedor EJB implementa las interfaces Home y Remote que están en el fichero EJB-Jar. Como se ha mencionado antes, la interfase Home proporciona métodos para crear y encontrar sus EJB. Esto significa que el contenedor es responsable por el manejo del ciclo de vida del EJB. Este nivel de oblicuidad permiten que se sucedan optimizaciones. Por ejemplo, 5 clientes pueden simultáneamente pedir por la creación de un EJB a través de la Interfase Home, y el Contenedor EJB puede responder creando solo un EJB y compartiéndolo entre 5 clientes. Esto se logra a través de la Interfase Remota, que es también implementada por el Contenedor EJB. El objeto Remoto implementado juega el rol de un objeto apoderado para el EJB.

Todas las llamadas al EJB son “apoderadas” a través del Contendor EJB mediante la interfase Remota. Esta oblicuidad es la razón de por que el contenedor EJB puede controlar la seguridad y el comportamiento transaccional.

## Tipos de EJBs

La especificación de JavaBean Corporativos define diferentes tipos de EJBs que tienen diferentes características y comportamientos. Dos categorías de EJBs son definidas en la especificación: *Beans de sesión* y *Beans de entidad* y cada categoría tiene variaciones.

### Beans de sesión

Los Beans de Sesión son utilizados para representar casos de uso o cursos de trabajo en representación de un cliente. Estos representan operaciones en datos persistentes, pero no los datos persistentes por si mismos. Hay dos tipos de Beans de Sesión, *Stateless* y *Stateful*. Todos los Bean de sesión deben implementar la interfase **javax.ejb.SessionBean**. El contenedor EJB gobierna la vida de un Bean de Sesión.

Los **Bean de sesión Stateless** son el tipo mas simple de componente EJB para implementar. Ellos no mantienen ningún estado de conversación con

los clientes entre invocaciones de métodos así es que son fácilmente reutilizables en el lado del servidor y dado que pueden ser colocados en una caché, levantan bien en demanda. Cuando se utilizan Bean de Sesión Stateless, todos los estados deben ser almacenados fuera del EJB.

Los **Beans de Sesión Stateful** mantienen el estado entre las invocaciones. tienen una correspondencia lógica uno a uno con el cliente y pueden mantener el estado dentro de ellos mismos. El Contenedor EJB es responsable por la utilización en grupos y el cacheo de los Beans de Sesión Statful, que es alcanzado a través de la *Passivation* y la *Activation*. Si el Contenedor EJB se cae, los datos para todos los Beans de Sesión Stateful pueden ser perdidos. Algunos contenedores EJB de alta capacidad proporcionan recuperación de Bean se Sesión Stateful.

## Beans de Entidad

Los Beans de Entidad son componentes que representan datos persistentes y comportamientos de estos datos. Los Beans de Entidad pueden ser compartidos a lo largo de múltiples clientes, de la misma forma que los datos en una base de datos pueden ser compartidos. El contenedor EJB es responsable por el cacheo de los Beans de Entidad y por el mantenimiento de la integridad de los Beans de Entidad. Se espera que la vida de un Bean de Entidad esté disponible cuando el Contenedor EJB vuelva a estar disponible.

Hay dos tipos de Beans de Entidad: aquellos con persistencia Administrada con Contenedores y aquellos con persistencia de Bean Administrada.

**Persistencia de contenedor Administrada (CMP Container Managed Persistence)**. Una Bean de Entidad CMP tiene la persistencia implementada en su representación por el Contenedor EJB. A través de los atributos especificados en el descriptor de desarrollo, el Contenedor EJB corresponderá los atributos del Beans de entidad a algún almacenaje persistente (usualmente -pero no siempre- una base de datos). CMP reduce el tiempo de desarrollo para el EJB, así como reduce dramáticamente la cantidad de código requerido.

**Persistencia de Bean Administrada (BMP Bean Managed Persistence)**. Una Bean de Entidad BMP tiene la persistencia implementada por el proveedor de Bean Corporativos. El Proveedor de Bean Corporativos es responsable por la implementación de la lógica requerida para crear un nuevo EJB, actualizar algunos atributos de los EJBS, borrar un EJB y encontrar un EJB de un almacenamiento persistente. Esto involucra usualmente la escritura de código JDBC para interactuar con una base de datos u otro tipo de almacenamiento persistente. Con BMP, el desarrollador tiene el control total de como la persistencia del Bean de Entidad es manejado.

BMP también da la flexibilidad donde una implementación CMP puede no estar disponible. Por ejemplo, si se quiere crear un EJB que envuelva algún código de un sistema de computador central existente, se puede escribir la persistencia utilizando CORBA.

## Desarrollando un EJB

Como ejemplo, el ejemplo “Perfect Time” de la sección anterior de RMI será implementada como un componente EJB. El ejemplo será un simple Bean de Sesión Stateless.

Como se ha mencionado antes, los componentes EJB consisten en al menos una clase (el EJB) y dos interfaces: La interfase Home y la interfase Remote. Cuando se crea una interfase Remota para un EJB, se debe seguir esta líneas:

1. La interfase remota debe ser pública.
2. La interfase Remota debe extender la interfase **javax.ejb.EJBObject**.
3. Cada método en la interfase remota debe declarar **java.rmi.RemoteException** es su cláusula **throws** además de cualquier excepción específica de la aplicación.
4. Cualquier objeto pasado como argumento o valor de retorno (directamente o insertado dentro de un objeto local) debe ser un tipo de dato RMI-IIOP válido (Esto incluye otros objetos EJB).

He aquí una simple interfase remota para el EJB PerfectTime:

```
//: c15:ejb:PerfectTime.java
//# Se debe instalar el J2EE Java Enterprise
//# Edition de java.sun.com y agregar j2ee.jar
//# a su CLASSPATH para poder compilar este
//# fichero. Vea los detalles en java.sun.com.
// Interfase Remota para PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;
public interface PerfectTime extends EJBObject {
    public long getPerfectTime()
        throws RemoteException;
} ///:~
```

La interfase Home es la fábrica donde el componente será creado. Se puede definir métodos *create*, para crear instancias de EJBs, o métodos *finders* que localizan EJBs existentes y son utilizados solo por Beans de Entidad.

Cuando se crea una interfase Home para un EJB, de deben seguir estas líneas:

1. La interfase Home debe ser pública.
2. La interfase Home debe extender la interfase **javax.ejb.EJBHome**.

3. Cada método *create* en la interfase Home debe declarar **java.rmi.RemoteException** en su cláusula **throws** así como **javax.ejb.CreateException**.
4. El valor de retorno de un método *create* debe ser una interfase Remota.
5. El valor de retorno de un método *finder* (Beans de Entidad solamente) debe ser una Interfase Remota o una **java.util.Enumeration** o una **java.util.Collection**.
6. Un objeto pasado como argumento (directamente o empotrado dentro de un objeto local) debe ser un tipo de dato RMI-IIOP válido (esto incluye otros objetos EJB):

Las convenciones de nombres estándares para las interfases Home es tomar el nombre de la interfase Remota y agregar “Home” en el final. He aquí la interfase Home para el EJB PerfectTime:

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

Se puede ahora implementar la lógica de negocios. Cuando se crea la clase implementación EJB, se debe seguir estas líneas (note que se debería consultar la especificación EJB por una lista completa de líneas cuando se desarrolle JavaBean Corporativos):

1. La clase debe ser pública.
2. La clase debe implementar una interfase EJB (**javax.ejb.SessionBean** o **javax.ejb.EntityBean**).
3. La clase debería definir métodos que correspondan directamente con los métodos de la interfase Remota. Debe notarse que la clase no implementa la interfase Remota; refleja los métodos en la interfase Remota pero *no lanza* **java.rmi.RemoteException**.
4. Define uno o mas métodos **ejbCreate()** para inicializar el EJB.
5. El valor de retorno y los argumentos de todos los métodos deben ser un tipo de datos RMI-IIOP válido.

```
//: c15:ejb:PerfectTimeBean.java
// Simple Bean de sesión Stateless
// que retorna la hora del sistema.
import java.rmi.*;
import javax.ejb.*;
public class PerfectTimeBean
```

```

    implements SessionBean {
    private SessionContext sessionContext;
    //retorna la hora actual
    public long getPerfectTime() {
        return System.currentTimeMillis();
    }
    // Métodos EJB
    public void ejbCreate()
    throws CreateException {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void
        setSessionContext(SessionContext ctx) {
        sessionContext = ctx;
    }
}///:~

```

Dado que este es un simple ejemplo , los métodos EJB (**ejbCreate()**, **ejbRemove()**, **ejbActivate()**, **ejbPassivate()**) están todos vacíos. Estos métodos son invocados por el contenedor EJB y son utilizados para controlar el estado del componente. El método **setSessioncontext()** pasa un objeto **javax.ejb.SessionContext** que contiene información acerca del contexto del componente, como la actual transacción y la información de seguridad.

Luego de que hemos creado el JavaBean Corporativo, necesitamos luego crear un descriptor de desarrollo. El descriptor de desarrollo es un fichero XML que describe el componente EJB. El descriptor de desarrollo debe ser almacenado en un fichero llamado **ejb-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>Ejemplo para el capítulo 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>

```

| ///:~

Se puede ver el componente, la interfase Remota y la interfase Home definida dentro de la etiquete <**session**> de este descriptor de desarrollo. Los descriptores de desarrollo deben ser generados automáticamente utilizando herramientas de desarrollo EJB.

Junto con el descriptor de desarrollo estándar **ejb-jar.xml**, la estado de la especificación EJB que cualquier etiqueta específica de vendedor pueda ser almacenado en un fichero separado. Esto es para alcanzar niveles altos de portabilidad entre componentes y contenedores EJB de diferentes marcas.

Los ficheros deben ser archivados dentro de un fichero estándar Java Archive (JAR). El descriptor de desarrollo debe ser colocado dentro del subdirectorio /**META-INF** en el fichero Jar.

Una vez que el componente EJB esta definido en el descriptor de desarrollo, el que despliega debe entonces desplegar el componente EJB dentro de un Contenedor EJB. En el momento en que esto fue escrito, el proceso de despliegue era bastante “GUI intensivo” y específico para cada Contenedor EJB individual, así es que esta visión general no documenta ese proceso. Cada Contenedor EJB, sin embargo tendrá un proceso documentado para desplegar un EJB.

Dado que un componente EJB es un objeto distribuido, el proceso de despliegue debe también crear algún cabo de cliente para llamar el componente EJB. Estas clases deben ser colocadas en la ruta de clases de la aplicación del cliente. Dado que los componentes EJB pueden ser implementados sobre RMI-IIOP (CORBA) o RMI-JRMP, los cabos generalmente pueden variar entre Contenedores EJB; a pesar de ello son clases generadas.

Cuando un programa cliente quiere invocar un EJB, debe buscar el componente EJB dentro de JNDI y obtener la referencia a la interfase Home de el componente EJB. La interfase Home es utilizada para crear una instancia del EJB.

En este ejemplo el programa cliente es un programa Java simple, pero se debería recordar que fácilmente puede ser 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 {
    // Obtiene un contexto JNDI utilizando
    // El servicio de Nombres JNDI:
    javax.naming.Context context =
        new javax.naming.InitialContext();
    // Busca la interfase home en el
    // servicio de Nombres JNDI:
```

```

Object ref = context.lookup("perfectTime");
// Convierte el objeto remoto a la interfase home:
PerfectTimeHome home = (PerfectTimeHome)
javax.rmi.PortableRemoteObject.narrow(
    ref, PerfectTimeHome.class);
// Crea un objeto remoto de la interfase home:
PerfectTime pt = home.create();
// InvocagetPerfectTime()
System.out.println(
    "Perfect Time EJB invocado, la hora es: " +
    pt.getPerfectTime() );
}
} //:~

```

La secuencia de este ejemplo es explicada en los comentarios. Debe notarse que el uso del método **narrow()** para realizar un tipo de conversión del objeto antes de que una conversión Java sea realizada. Esto es muy similar a lo que sucede en CORBA. Debe notarse también que el objeto Home se convierte en una fábrica para objetos **PerfectTime**.

## Resumen de EJB

La especificación de JavaBean Corporativos es un dramático paso adelante en la estandarización y simplificación de computación de objetos distribuidos. Es una gran parte de la plataforma de la edición de Java 2 Corporativo (J2EE Java 2 Enterprise Edition) y recibe mucho soporte de la comunidad de objetos distribuidos. Muchas herramientas están disponibles actualmente o estarán disponibles en un futuro cercano para ayudar a acelerar el desarrollo de componentes EJB.

Esta visión general ha sido solo un breve tour de EJBs. Por mas información acerca de la especificación EJB se debería ver la página inicial de JavaBean Corporativos en [java.sun.com/products/ejb/](http://java.sun.com/products/ejb/), donde se puede bajar la última especificación y la implementación de referencia de J2EE. Esto puede ser utilizado para desarrollar y desplegar sus propios componentes EJB.

## Jini: servicios distribuidos

Esta sección<sup>7</sup> da un vistazo general de la tecnología Jini de Sun Microsystems. Esta describe algunos datos básicos y muestra como la arquitectura de Jini ayuda a levantar el nivel de abstracción en programación de sistemas distribuidos efectivamente convirtiendo la programación de redes en programación orientada a objetos.

---

<sup>7</sup> Esta sección fue contribución de Billy Venners ([www.artima.com](http://www.artima.com)).

## Jini en contexto

Tradicionalmente, los sistemas operativos están diseñados asumiendo que la computadora va a tener un procesador, algo de memoria, y un disco. Cuando se arranca una computadora, lo primero que hace es buscar un disco. Si no encuentra un disco, no puede funcionar como computadora. Cada vez más, sin embargo, las computadoras están apareciendo en una forma diferente: como dispositivos insertados que tienen un procesador, algo de memoria, y una conexión a la red -pero no disco. Lo primero que un teléfono celular hace cuando arranca, por ejemplo, es buscar la red de telefonía. Si no encuentra la red, no puede funcionar como teléfono celular. Esta tendencia en los ambientes hardware, de centrados en disco a centrados en redes, afectará en la forma en que organizamos el software -y aquí es donde Jini entra.

Jini es un intento de repensar la arquitectura de las computadoras, dándole la importancia creciente de las redes y la proliferación de procesadores en dispositivos que no tienen disco duro. Estos dispositivos, que vienen de muchos vendedores diferentes, necesitarán interactuar a través de las redes. La red por sí misma es muy dinámica -los dispositivos y servicios serán agregados y quitados regularmente. Jini proporciona mecanismos para habilitar suavemente agregar, quitar y encontrar dispositivos y servicios en la red. Además, Jini proporciona un modelo de programación que hace fácil para los programadores configurar sus dispositivos para que fácilmente hablen con otros. Creado sobre Java, serialización de objetos, y RMI (que juntos habilitan el mover objetos a través de la red entre máquinas virtuales) Jini intenta extender los beneficios de la programación orientada a objetos a la red. En lugar de requerir vendedores de dispositivos para agregar en el protocolo de red a través de los cuales los dispositivos puedan interactuar. Jini habilita a los dispositivos a hablar con cada uno de los otros a través de interfaces a objetos.

## ¿Qué es Jini?

Jini es un grupo de APIs y protocolos de red que pueden ayudar a crear y desplegar sistemas distribuidos que son organizados como *federaciones de servicios*. Un servicio puede ser cualquier cosa que se coloque en la red y esté listo para realizar una función útil. Dispositivos de Hardware, software, canales de comunicación -incluso los mismos usuarios humanos- pueden ser servicios. Una unidad de disco habilitada para Jini, por ejemplo, puede ofrecer un servicio de "almacenamiento". Una impresora habilitada para Jini puede ofrecer un servicio de " impresión ". Una federación de servicios, entonces, es un grupo de servicios, disponibles actualmente en la red, que

un cliente (sea un programa, servicio o usuario) puede traer junto a el para ayudar a lograr alguna meta.

Para realizar una tarea, un cliente hace un listado de los servicios de ayuda. Por ejemplo, un programa cliente puede levantar imágenes de un servicio de almacenamiento de una cámara digital, bajar las imágenes a un servicio de almacenaje persistente ofrecido por una unidad de disco y enviar una página de versiones pequeñas de las imágenes a el servicio de una impresora color. En este ejemplo, el programa cliente crea un sistema distribuido consistente en si mismo, el servicio de almacenamiento de imágenes, el servicio de almacenaje persistente, y el servicio de impresión color. El cliente y los servicios de este sistema distribuido trabajo juntos para realizar la tarea: para descargar y almacenera imágenes de una cámara digital e imprimir una página de imágenes pequeñas.

La idea detrás del mundo de la *federaciones* que la visión de Jini de la red no involucra una autoridad central de control. Dado que ningún servicio está a cargo, el grupo de todos los servicios disponibles en la red de una federación -un grupo compuesto por pares iguales. En lugar de una autoridad central, la infraestructura en tiempo de ejecución de Jini proporciona una forma para clientes y servicios para encontrar cada uno de los otros (mediante un servicio de búsqueda, que almacena un directorio de servicios disponibles actualmente). Después que se los servicios localizan a los otros, tienen el suyo propio. El cliente y sus servicios enlistados realizan sus tareas independientemente de la infraestructura en tiempo de ejecución de Jini. Si el servicio de búsqueda de Jini se cae, cualquier sistema distribuido traído junto mediante el servicio de búsqueda antes de que se cayera puede continuar su trabajo. Jini incluso incluye un protocolo de redes que los clientes pueden utilizar para encontrar servicios en la ausencia de un servicio de búsqueda.

## Como trabaja Jini

Jini define una *infraestructura en tiempo de ejecución* que reside en la red y proporciona mecanismos que habilitan a agregar, quitar, localizar y acceder a servicios. La infraestructura en tiempo de ejecución reside en tres lugares, en los servicios de búsqueda que están en la red, en los proveedores de servicios (como lo son los dispositivos habilitados de Jini), y en los clientes. Los *servicios de búsqueda* son el mecanismo de organización central de Jini para sistemas basados en Jini. Cuando un nuevo servicio comienza a estar disponible en la red, estos se registran a si mismos mediante un servicio de búsqueda. Cuando los clientes quieren encontrar un servicio para asistir con alguna tarea, estos consultan a un servicio de búsqueda.

La infraestructura en tiempo de ejecución que utiliza un protocolo a nivel de red, llamado *descubrimiento*, y dos protocolos a nivel de objetos, llamados *asociación* y *búsqueda*.

El protocolo de descubrimiento habilita a los clientes y servicios a localizar los servicios de búsqueda. La asociación habilita a el servicio a registrarse a si mismo en un servicio de búsqueda. La búsqueda habilita a el cliente a preguntar por servicios que puedan ayudar a cumplir sus metas.

## El proceso de descubrimiento

El descubrimiento trabaja de esta forma: Imaginemos que tenemos una unidad de disco habilitada para Jini que ofrece un servicio de almacenamiento persistente. Tan rápido como se conecte la unidad de disco a la red, enviará un *anuncio de presencia* tirando un paquete de emisión simultánea en un puerto bien conocido. Incluida en el anuncio de presencia esta una dirección IP y un número de puerto donde la unidad de disco pueda ser contactada mediante un servicio de búsqueda.

Los servicios de búsqueda realizan un monitoreo del puerto esperando paquetes de anuncio de presencia. Cuando un servicio de búsqueda recibe un anuncio de presencia, abre e inspecciona el paquete. El paquete contiene información que habilita el servicio de búsqueda a determinar que puede y que no puede contactar la parte que realiza el envío del paquete. Si puede, contacta el que envía directamente haciendo una conexión TCP a la dirección IP y el número de puerto extraído del paquete. Utilizando RMI, el servicio de búsqueda envía un objeto, llamado un *registrar de servicio* a través de la red a quien originó el paquete. El propósito del objeto registrado de paquete es facilitar las siguientes comunicaciones entre el servicio de búsqueda. Mediante la invocación de métodos en este objeto, el que envía el paquete de anunciamiento puede realizar la asociación y búsqueda en el servicio de búsqueda. En el caso de la unidad de disco puede entonces registrar su servicio de almacenamiento persistente mediante el proceso de asociación.

## El proceso de asociación

Una vez que el proveedor de servicios tiene un objeto registrado de servicios, el producto final del proceso de descubrimiento, está listo para hacer una asociación -para convertirse en parte de la federación de servicios que están registrados en el servicio de búsqueda. Para hacer una asociación, el proveedor del servicio invoca el método **register()** en un objeto servicio registrado, pasando como parámetro un objeto llamado ítem de servicio, un fajo de objetos que describen el servicio. El método **register()** envía una

copia del ítem de servicio a el servicio de búsqueda, donde el ítem de servicio es almacenado.

Una vez que esto es completado, el proveedor de servicios ha finalizado el proceso de asociación: el servicio ha sido registrado en el servicio de búsqueda.

El ítem de servicio es un contenedor para muchos objetos, incluyendo un objeto llamado *objeto de servicio*, que el cliente puede utilizar para interactuar con el servicio. El ítem de servicio puede también incluir cualquier cantidad de *atributos* que puedan ser cualquier objetos. Algunos atributos potenciales son íconos, clases que proporcionan GUIs para el servicios y objetos que dan mas información acerca del servicio.

Los objetos de servicios usualmente implementan una o mas interfaces a través de las cuales los clientes interactúan con el servicio. Por ejemplo, un servicio de búsqueda es un servicio Jini, y su objeto de servicio es el registrador de servicios. El método **register()** invocado por el proveedor de servicios durante la asociación es declarado en la interfase **ServiceRegistrar** (un miembro de el paquete **net.jini.core.lookup**), que todos los objetos registradores de servicios implementan. Los clientes y los proveedores de servicios hablan con el servicio de búsqueda a través del objeto registrador de servicio invocando métodos declarados en la interfase **ServiceRegistrar**. Asimismo, una unidad de disco puede proporcionar un objeto servicio que implementa alguna interfase de servicio de almacenaje muy conocida. Los clientes pueden buscar y interactuar con la unidad de disco a través de esta interfase de servicio de almacenaje.

## El proceso de búsqueda

Una ves que un servicio ha sido registrado con un servicio de búsqueda mediante el proceso de asociación, este servicio esta disponible para la utilización por clientes que consultan ese servicio de búsqueda. Para construir un sistema distribuido de servicios que puedan trabajar juntos para realizar una tarea, un cliente debe localizar y enlistar la ayuda para realizar alguna tarea, un cliente debe localizar y enlistar la ayuda para los servicios individuales. Para encontrar un servicio, las consultas de los servicios de los clientes mediante un proceso llamado *búsqueda*.

Para realizar una búsqueda, un cliente invoca el método **lookup()** en un objeto registrador de servicio (Un cliente, como un proveedor de servicio, obtiene un registrador de servicio a través del proceso anteriormente descrito de descubrimiento). El cliente pasa como argumento a **lookup()** una *plantilla de servicio* un objeto que sirve como criterio de búsqueda para la consulta. La plantilla de servicio puede incluir una referencia a un arreglo de objetos **Class**. Estos objetos **Class** indican a el servicio de búsqueda el tipo

Java (o tipos) del objeto servicio deseado por el cliente. La plantilla de servicio puede también incluir un *identificador de servicio* que identifica excepcionalmente un servicio, y atributos, que deben coincidir exactamente el atributo levantado por el proveedor de servicios en el ítem de servicios. La plantilla de servicios puede también contener comodines para cualquiera de esos campos. Un comodín en el campo identificador de servicio, por ejemplo, coincidirá con cualquier identificador de servicio. El método **lookup()** envía la plantilla de servicio a el servicio de búsqueda, que realiza la consulta y retorna cero a cualquier objeto de servicio coincidente. El cliente obtiene una referencia a el objeto de servicio correspondiente como valor de retorno del método **lookup()**.

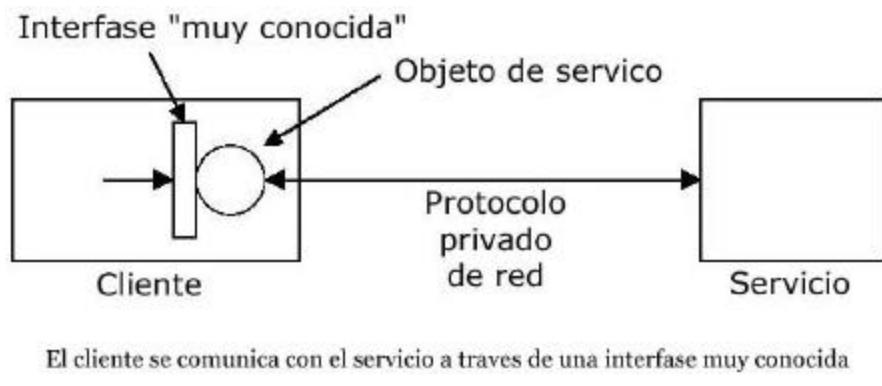
En el caso general, un cliente busca un servicio por tipo de Java, usualmente una interfase. Por ejemplo, si un cliente necesita utilizar una impresora, debe componer una plantilla de servicio que incluya un objeto **Class** para una interfase muy conocida de servicios de impresión. Todos los servicios de impresión implementarán esta interfase conocida. El servicio de búsqueda retornará un objeto de servicio (u objetos) que implementan esta interfase. Los atributos pueden ser incluidos en la plantilla de servicio para limitar el número coincidencias para esta búsqueda basada en el tipo. El cliente utilizará el servicio de impresión invocando métodos de esta interfase conocida de servicio de impresora en el objeto de servicio.

## Separación de interfase e implementación

La arquitectura Jini trae la programación orientada a objetos a las redes habilitando a los servicios de redes a tomar ventaja de algo fundamental de los objetos: la separación de la interfase y la implementación. Por ejemplo, un objeto de servicio puede dar acceso a clientes a el servicio de muchas formas. El objeto puede actualmente representar el servicio entero, que se puede bajar a el cliente durante la búsqueda y ejecución localmente. Alternativamente, el objeto de servicios puede representar actualmente el servicio entero, que es bajado a el cliente durante la búsqueda y la ejecución local. Alternativamente, el objeto servicio puede servir simplemente como un apoderado para un servidor remoto. Entonces cuando el cliente invoca métodos en el objeto de servicios, este envía las peticiones a través de la red al servidor, que hace el trabajo real. Una tercer opción es que el objeto de servicio local y un servidor remoto haga cada uno parte del trabajo.

Una consecuencia importante de la arquitectura de Jini es que el protocolo de red utilizado para comunicarse entre un objeto de servicio apoderado y un servidor remoto no necesita ser conocido por el cliente. Como se ilustra en la figura mas abajo, el protocolo de redes es parte del la implementación

del servicio. Este protocolo es una decisión privada del desarrollador del servicio. El cliente no puede comunicarse con el servicio mediante este protocolo privado porque el servicio inyecta algo de su propio código (el objeto de servicio) en el espacio de direcciones del cliente. El objeto de servicio inyectado puede comunicarse con el servicio mediante RMI, CORBA, DCOM, algún protocolo confeccionado en casa creado sobre los socket y flujos, o cualquier otro. El cliente simplemente no necesita preocuparse acerca de los protocolos de red, porque estos pueden hablar con la interfase conocida que el objeto de servicio implementa. El objeto de servicio tiene cuidado de cualquier comunicación necesaria en la red.



Diferentes implementaciones de la misma interfase de servicio pueden utilizar estrategias y protocolos de red completamente diferentes. Un servicio puede utilizar hardware especializado para satisfacer consultas de clientes, o pueden hacer todo el trabajo en software. De hecho la estrategia de implementación tomado por un solo servicio puede evolucionar a través del tiempo. El cliente puede estar seguro de que el objeto de servicio que entiende la implementación actual del servicio, dado que el cliente recibe el objeto de servicio (mediante el servicio de búsqueda) del mismo proveedor de servicios. Para el cliente, un servicio se ve como una interfase muy conocida, independientemente de como el servicio es implementado.

## Sistemas distribuidos abstractos

Jini intenta levantar el nivel de abstracción para programación de sistemas distribuidos, del nivel de protocolo de red a el nivel de interfase de objeto. En la proliferación emergente de dispositivos insertados conectados a las redes, muchas partes de un sistema distribuido puede venir de diferentes vendedores. Jini hace innecesario para los vendedores de dispositivos el estar de acuerdo con los protocolos a nivel de red lo que permite a sus dispositivos interactuar. En lugar de eso, los vendedores deben estar de acuerdo de las interfaces Java a través de los cuales sus dispositivos pueden

interactuar. El proceso de descubrimiento, asociación y búsqueda proporcionado por la infraestructura en tiempo de ejecución de Jini, habilita a los dispositivos a localizar cada uno de los otros en la red. Una vez que ellos han localizada cada uno de los otros, los dispositivos pueden comunicarse con cada uno a través de las interfaces Java.

## Resumen

Junto con Jini para dispositivos de redes locales, este capítulo ha introducido algo, pero no todos, de los componentes que Sun refiere como J2EE: La *Edición de Java 2 Corporativa*. La meta de J2EE es crear un grupo de herramientas que permitan al desarrollador Java crear aplicaciones basadas en servidor mas rápidamente, y de una forma independiente de la plataforma. Crear estas aplicaciones no es difícil y lleva mucho tiempo, es especialmente duro crearlas de tal forma que sean fácilmente trasladadas a otras plataformas, y también mantener la lógica de negocios separadas de los detalles de las capas mas bajas de la implementación. J2EE proporciona un marco de trabajo para asistir a la creación de aplicaciones basadas en servidor; estas aplicaciones están en demanda ahora, y esta demanda parece incrementarse.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

1. Compile y ejecute los programas **JavverServer** y **JavverClient** de este capítulo. Ahora edite estos fichero para quitar los buffers para la entrada y la salida, luego compile y ejecútelo nuevamente para observar los resultados.
2. Cree un servidor que pregunte por una palabra clave, luego abra un fichero y envíe el fichero a través de la conexión de red. Cree un cliente que se conecte a este servidor, le de la palabra clave apropiada, luego capture y guarde el fichero. Pruebe el par de programas en su máquina utilizando **localhost** (la dirección IP local 128.0.0.1 producida por la llamada a **InetAddress.getByName(null)**).
3. Modifique el servidor en el Ejercicio 2 de tal forma que utilice hilado múltiple para manejar múltiples clientes.
4. Modifique **JabberClient.java** de tal forma que la limpieza de la salida no se suceda y observe el efecto.
5. Modifique **MultiJabberServer** de tal forma que utilice *fondo común de hilos*. En lugar de lanzar un hilo cada vez que un cliente se desconecta,

el hilo debe colocarse a si mismo en una “*fondo común disponible*” de hilos. Cuando un nuevo cliente quiera conectarse, el servidor buscará en el fondo común disponible por un hilo para manejar la consulta, y si no hay ninguno disponible, se cree uno nuevo. De esta forma el número de hilos necesario crecerá naturalmente a la cantidad requerida. El valor de el fondo común de hilos es tal que no requiere la sobrecarga de crear o destruir un nuevo hilo para cada nuevo cliente.

6. Partiendo de **ShowHTML.java**, cree un applet que sea una puerta de enlace protegida mediante palabra clave a una porción particular de su sitio Web.
7. Modifique **CIDCreateTables.java** de tal forma que lea las cadenas SQL de un fichero de texto en lugar de CIDSQl.
8. Configure el sistema de tal forma que puede ejecutar exitosamente **CIDCreateTables.java** y **LoadDB.java**.
9. Modifique **ServletsRule.java** sobrecargando el método **destroy()** para guardar el valor de **i** en un fichero, y el método **init()** para restaurar el valor. Demuestre que esto trabaja reiniciando el contenedor del servlet. Si no tiene un contenedor de servlet existente, necesitará bajar, instalar y ejecutar Tomcat de [jakarta.apache.org](http://jakarta.apache.org) para ejecutar servlets.
10. Cree un servlet que agregue una cookie a el objeto respuesta, consecuentemente almacenada en el sitio del cliente. Agregue código al servlet que recupere y despliegue la cookie. Si no se tiene un contenedor de servlet existente, necesitará bajar, instalar y ejecutar Tomcat de [jakarta.apache.org](http://jakarta.apache.org) para poder ejecutar servlets.
11. Cree un servlet que utilice un objeto **Session** para almacenar la información de la sesión de su elección. En el mismo servlet, recupere y despliegue la información de esa sesión. Si no tiene un contenedor se servlet, necesitará bajar, instalar y ejecutar Tomcat de [jakarta.apache.org](http://jakarta.apache.org) para ejecutar servlets.
12. Cree un servlet que cambie el intervalo inactivo de una sesión a 5 segundos llamando a **getMaxInactiveInterval()**. Pruebe para ver que la sesión ciertamente expira luego de los 5 segundos. Si no tiene un contenedor se servlet, necesitará bajar, instalar y ejecutar Tomcat de [jakarta.apache.org](http://jakarta.apache.org) para ejecutar servlets.
13. Cree una página JSP que imprima una línea de texto utilizando la etiqueta **<H1>**. Configure el color de este texto de forma aleatoria, utilizando código Java insertado en la página JSP. Si no tiene un contenedor se servlet, necesitará bajar, instalar y ejecutar Tomcat de [jakarta.apache.org](http://jakarta.apache.org) para ejecutar servlets.

14. Modifique el valor de edad máxima en **Cookies.jsp** y observe el comportamiento bajo dos navegadores diferentes. También note la diferencia entre visitar nuevamente la página, y iniciar nuevamente el navegador. Si no tiene un contenedor se servlet, necesitará bajar, instalar y ejecutar Tomcat de *jacarta.apache.org* para ejecutar servlets.
15. Cree un JSO con un campo que permita a el usuario entrar el tiempo de expiración y un segundo campo que almacene datos que son almacenados en la sesión. El botón submit refresca la página y trae el tiempo actual de expiración y los datos de sesión y los coloca como valores por defecto en los campos antes mencionados. Si no tiene un contenedor se servlet, necesitará bajar, instalar y ejecutar Tomcat de *jacarta.apache.org* para ejecutar servlets.
16. (Mas retos) Tome el programa **VLookup.java** y modifíquelo de tal forma que cuando se haga clic en el nombre resultante automáticamente tome ese nombre y lo copie en el portapapeles (así es que simplemente se puede pegar en un correo electrónico). Se necesitará mirar atrás en el Capítulo 13 para recordar como utilizar el portapapeles en JFC.

# A: Pasando & retornando objetos

Por ahora se puede estar razonablemente confortable con la idea de que cuando se están “pasando” objetos, se está pasando una referencia.

En muchos lenguajes de programación se puede utilizar la forma “común” del lenguaje de pasar objetos, y la mayor parte del tiempo todo funciona bien. Pero parece que siempre se llega a un punto en donde se debe hacer algo irregular y de pronto las cosas se ponen un poquito mas complicadas (o en el caso de C++, bastante complicadas). Java no es la excepción, y es importante que se entienda exactamente que sucede si se pasan y manipulan objetos. Este apéndice proporcionara ese entendimiento.

Otra forma de hacer la pregunta de este apéndice, si se viene de un lenguaje mas equipado, es “¿Tiene Java punteros?”. Algunos han afirmado que los punteros son rígidos y peligrosos y por entonces malos, y dado que Java es siempre celestial e iluminado y nos elevará de la carga terrenal de la programación, no es posible que tenga ese tipo de cosas. Sin embargo, es mas certero decir que Java tiene punteros; ciertamente, todos los objetos identificados en Java (excepto las primitivas) es uno de esos punteros, pero su uso esta restringido y protegido por el compilador sino que también por el sistema en tiempo de ejecución. O para colocarlo de otra forma, Java tiene punteros, pero no punteros aritméticos. Estos son los que llamo “referencias”, y se puede pensar en ellas como “punteros seguros”, no distinto a las tijeras seguras de la escuela -no tienen filo, así es que uno no se puede herir sin realizar un gran esfuerzo, pero pueden a veces ser lentas y tediosas.

## Pasando referencias

Cuando se pasa una referencia en un método, se sigue apuntando a el mismo objeto. Un simple experimento demuestra esto:

```

//: appendixa:PassReferences.java
// Pasando referencias.
public class PassReferences {
    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);
    }
} //:~

```

El método **toString()** es automáticamente invocado en la instrucción para imprimir, y **PassReferences** hereda directamente de **Object** sin redefinir **toString()**. De esta forma, la versión **Object** de **toString()** es utilizada, lo que imprime la clase del objeto seguido por la dirección donde ese objeto es localizado (no la referencia, el actual almacenamiento del objeto). La salida se ve de la siguiente forma:

```

p inside main(): PassReferences@1653748
h inside f(): PassReferences@1653748

```

Se puede ver que **p** y **h** se refieren a el mismo objeto. Esto es mucho mas eficiente que duplicar exactamente un nuevo objeto **PassReferences** así es que se puede enviar un argumento a un método. Pero esto nos trae un tema muy importante.

## Aliasing

Aliasing significa que mas de una referencia esta ligada a el mismo objeto, como en el ejemplo anterior. El problema con el aliasing se sucede cuando alguien *escribe* en ese objeto. Si los dueños de las otras referencias no esperan que el objeto cambie, estarán sorprendidos. Esto puede ser demostrado con un simple ejemplo:

```

//: appendixa:Alias1.java
// Aliasing dor referencias a un objeto.
public class Alias1 {
    int i;
    Alias1(int ii) { i = ii; }
    public static void main(String[] args) {
        Alias1 x = new Alias1(7);
        Alias1 y = x; // Asignando 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);
    }
} //:~

```

En la línea:

```
| Alias1 y = x; // Assign the reference  
una nueva referencia Alias1 es creada, pero en lugar de ser asignada a un  
nuevo objeto creado con new, esta es asignada a una referencia existente.  
Así es que los contenidos de las referencias x, que es la dirección del objeto x  
es apuntar a, es asignada a y, y de esta forma x y y son enganchados a el  
mismo objeto. Así es que cuando el i de x es incrementado en la instrucción:
```

```
| x.i++;  
i de y será afectado de la misma forma. Esto puede verse en la salida:
```

```
| x: 7  
y: 7  
Incrementing x  
x: 8  
y: 8
```

Una buena solución en este caso es simplemente no hacerlo: no realizar un alias concientemente de mas de una referencia a un objeto en el mismo alcance. El código será mucho mas fácil de entender y de depurar. Sin embargo, cuando se está pasando una referencia como un argumento -que es la forma en que Java se supone que trabaja -automáticamente se crea un alias porque la referencia local que es creada puede modificar el “objeto externo” (el objeto que fue creado fuera del alcance del método). He aquí un ejemplo:

```
//: appendix:Alias2.java  
// Las llamadas a los métodos implicitamente  
// crea un alias de sus argumentos.  
public class Alias2 {  
    int i;  
    Alias2(int ii) { i = ii; }  
    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);  
    }  
} //:~
```

La salida es:

```
| x: 7  
Calling f(x)  
x: 8
```

El método cambia su argumento, el objeto externo. Cuando se dan este tipo de situaciones, de debe decidir si tiene sentido, si el usuario lo espera, y si nos causará problemas.

En general, se llama a un método para producir un valor de retorno y/o un cambio en el estado del objeto *al que el método llama* (Un método es como

se “envía un mensaje” a ese objeto). Es mucho menos común llamar a un método para manipular sus argumentos; esto es referido como “llamar un método por sus *efectos secundarios*”. Sin embargo, cuando se crea un método que modifica sus argumentos el usuario debe ser instruido claramente y advertido acerca del uso de ese método y sus potenciales sorpresas. Dadas las confusiones y trampas, es mucho mejor evitar cambiar el argumento.

Si se necesita modificar un argumento durante una llamada a método y no pretende modificar el argumento externo, entonces se debe proteger ese argumento haciendo una copia dentro del método. Esto es el tema de la mayoría de este apéndice.

## Haciendo copias locales

Como repaso: Todos los argumentos pasados en Java son realizado pasando referencias. Esto es, cuando se pasa “un objeto”, se esta pasando realmente una referencia a un objetos que vive fuera del método, así es que si se realiza cualquier modificación con esa referencia, se modificará el objeto exterior. Además:

- El aliasing se sucede automáticamente durante el pasaje del argumento.
- No hay objetos locales, solo referencias locales.
- Las referencias tienen alcance, los objetos no.
- El tiempo de vida de un objeto nunca es un tema en Java.
- No hay soporte del lenguaje (e.g. “const”) para prevenir que los objetos sean modificados (esto es, para prevenir los efectos negativos del aliasing).

Si solo se esta leyendo información de un objeto y no se esta modificando, pasar una referencia es la forma mas eficiente de hacer las cosas. Sin embargo, a veces es necesario ser capas de tratar el objeto como si fuera “local” así es que esos cambios que se hacen afecten solo una copia local y no modifiquen el objeto externo. Muchos lenguajes de programación soportan la habilidad de hacer copias locales automáticamente del objeto externo, dentro del método<sup>1</sup>. Java no lo hace, pero permite que se produzca este efecto.

---

<sup>1</sup> En C, que generalmente maneja pequeñas cantidades de datos, por defecto se pasa por valor. C++ sigue esta forma, pero con objetos el pasaje por valor no es usualmente la forma mas eficiente. Además, codificar las clases para soportar pasaje por valor en C++ es un gran dolor de cabeza.

## Pasar por valor

Esto nos trae un tema de terminología, que siempre parece bueno para un argumento. El término es “pasar por valor”, y el significado depende de como se perciba la operación del programa. El significado general es que se obtenga una copia local de lo que se esta pasando, pero la pregunta real es como se piensa acerca de lo que se esta pasando. Cuando llegamos a el significado de “pasar por valor”, hay dos campos medianamente distintos:

1. Java pasa todo por valor. Cuando se están pasando primitivas en un método, se esta obteniendo una copia de la primitiva. Cuando se esta pasando una referencia a un método, se obtiene una copia de la referencia. Por lo tanto, todo es pasado por valor. Claro, se asume que se esta pensando (e importando) que las referencias son pasadas, pero parece como que los diseñadores de Java han hecho un largo camino para permitir que se ignore (la mayoría del tiempo) que se esta trabajando con una referencia. Esto es, parece que permitir que se piense en la referencia como “el objeto”, dado que implícitamente no la referencia dondequiera que se realice una llamada a un método.
2. Java pasa primitivas por valor (no hay argumentos aquí), pero los objetos son pasados por referencia. Esto es la visión del mundo que la referencia es un alias para el objeto, así es que *no* se piensa acerca de pasar referencias, en lugar de eso “Se esta pasando un objeto”. Dado que no se tiene una copia local del objeto cuando de pasa dentro de un método, los objetos claramente no son pasados por valor. Esto parece ser un soporte de esta visión dentro de Sun, dado que uno de las palabras claves “reservadas pero no implementadas” es **byvalue** (No hay conocimiento, sin embargo, de si esa palabra clave verá la luz del día algún día).

Habiendo dado a ambos campos una buena ventilada, y luego de decir “depende de como piense una referencia”. Intentaremos soslayar el tema. Al final, *eso* no es lo importante -lo que es importante es que se entienda que pasar una referencia permite al objeto llamado ser cambiado inesperadamente.

## Clonando objetos

La razón mas probable para hacer una copia local de un objeto es si se va a modificar ese objeto y no se quiere modificar el objeto llamado. Si se decide que se quiere hacer una copia local, simplemente utilice el método **clone()** para realizar la operación. Esto es un método que esta definido como **protected** en la clase **Objeto** base, y que debería sobrescribir como pública en todas las clases derivadas que se quieran clonar. Por ejemplo, la clase de

la librería estándar **ArrayList** sobrescribe **clone()**, así es que podemos llamar a **clone()** para un **ArrayList**:

```
//: appendixa:Cloning.java
// La operación clone() trabaja para solo algunos
// pocos ítems en la librería estandar de Java.
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 {
    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();
        // Incrementa todos los elementos de v2:
        for(Iterator e = v2.iterator();
            e.hasNext();)
            ((Int)e.next()).increment();
        // Observa si se cambian los elemenots de v:
        System.out.println("v: " + v);
    }
} ///:~
```

El método **clone()** produce un **Object**, que debe ser convertido a el tipo adecuado. Este ejemplo muestra como el método **clone()** de **ArrayList** *no* trata de clonar automáticamente cada uno de los objetos que el **ArrayList** contiene -el viejo **ArrayList** y el clonado son aliased a los mismos objetos. Esto es a menudo llamado una *copia superficial* dado que se copian solo la parte “superficial” de un objeto. El objeto actual consiste en esta “superficie”, mas todos los objetos a los que sus referencias apuntan, además de todos *aquellos* objetos que los apuntan, etc. Esto es a menudo referido como la “telaraña de objetos”. Copiar todo el lío es llamado una *copia profunda*

Se puede ver el efecto de una copia superficial en la salida, donde las acciones realizadas en **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]
```

El no tratar de clonar los objetos contenidos en el **ArrayList** es una suposición justa porque no hay garantía de que estos objetos puedan ser clonables<sup>2</sup>.

## Agregando la capacidad de clonarse a una clase

A pesar de que el método `clone` esta definido en **Object** la clase base de todas las clases, la clonación *no* es automáticamente disponible en cada clase<sup>3</sup>. Esto parecerá 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 va contra esta idea; si se quiere que exista para una clase, se debe específicamente agregar código para hacer que la clonación funcione.

### Utilizando un truco con `protected`

Para prevenir la capacidad de clonarse por defecto en cada clase que se cree, el método **clone()** es **protected** en la clase base **Object**. Este término medio no solo hace que no esté disponible por defecto al cliente programador que simplemente utiliza la clase (no creando una subclase), también significa que no se puede llamar a **clone()** mediante una referencia a la clase base (A pesar de que puede parecer que sea útil en algunas situaciones, como lo es clonar de forma polimórfica un montón de **Objects**). Esto es en efecto una forma de darnos en tiempo de ejecución, la información que el objeto no se puede clonar -y raramente suficiente en la mayoría de las clases en la librería estándar de Java no tienen la capacidad de clonarse. De esta forma, si se dice:

```
Integer x = new Integer(1);
x = x.clone();
```

---

<sup>2</sup> El termino utilizado en inglés es ‘cloneable’. No es un término que este tampoco en el diccionario inglés, pero es lo que utilizan en la librería de Java, así es que es utilizado aquí, también, con alguna esperanza de reducir la confusión.

<sup>3</sup> Se puede aparentemente crear un ejemplo simple de contador para esa instrucción, como esta:

```
public class Cloneit implements Cloneable {
    public static void main (String[] args)
        throws CloneNotSupportedException {
        Cloneit a = new Cloneit();
        Cloneit b = (Cloneit)a.clone();
    }
}
```

Sin embargo, eso solo trabajará porque **main()**es un método de **Cloneit** y de esta forma tiene permiso para llamar el método protegido la clase base **clone()**. Si se llama a este de una clase diferente, no compilará.

Se obtendrá, en tiempo de compilación, un mensaje de error que dice que **clone()** no es accesible (dado que **Integer** no la sobrescribe y a su versión **protected** por defecto).

Si, sin embargo, se esta en una clase derivada de **Object** (como todas las clases son), entonces se tiene permiso de llamar a **Object.clone()** porque es **protected** y se es un heredero. La clase base **clone()** tiene una funcionalidad completa -esta realiza la duplicación a nivel de bit actual *del objeto de la clase derivada* de esta forma actual como la operación común de clonación. Sin embargo, se necesitará entonces hacer *la operación de clonación public* para que sea accesible. Así es que, cuando se clone, hay dos temas clave:

- Virtualmente se llama siempre a **super.clone()**
- Hay que hacer su **clone() public**

Probablemente se quiera sobrescribir **clone()** en una clase derivada mas adelante, de otra forma el (no **public**) **clone()** será utilizado, y esto significa hacer las cosas incorrectamente (a pesar de que, dado que **Object.clone()** realiza una copia del objeto actual, este puede). El truco de **protected** funciona solo una vez -la primera vez que se hereda de una clase que no tiene capacidad para clonarse y se quiere hacer que esa clase se pueda clonar. En cualquier clase heredada de su clase el método **clone()** esta disponible dado que no es posible reducir el acceso a un método durante la derivación. Esto es, una vez que una clase tiene la capacidad para clonarse, todas las clases derivadas de esta tendrán la capacidad para clonarse a no ser que se utilicen los mecanismos proporcionados (descritos mas tarde) para “apagar” la clonación.

## Implementación de la interfase Cloneable

Solo hay una cosa mas que se necesita hacer para completar la capacidad de clonación de un objeto: implementar la interfase **Cloneable**. ¡Esta interfase es un poco extraña, porque esta vacía!

```
| interface Cloneable {}
```

La razón para la implementar esta interfase vacía obviamente no es porque se esta haciendo una conversión ascendente a **Cloneable** y llamar a uno de sus métodos. El uso de **interface** aquí es considerado por algunos como un “hackeo” porque esta utilizando una característica con un objetivo distinto que para lo cual fue pensado. Implementar la interfase **Cloneable** actúa como un tipo de bandera, engarzado dentro del tipo de la clase.

Hay dos razones para la existencia de la interfase **Cloneable**. Primero, se debe tener una referencia a una conversión ascendente a el tipo base y no saber cuando es posible clonar ese objeto. En este caso, se puede utilizar la palabra clave **instanceof** (descripta en el capítulo 12) para encontrar cuando la referencia esta conectada a un objeto que puede ser clonado:

```
| if(myReference instanceof Cloneable) // ...
```

La segunda razón para esta mezcla en el diseño para darle la capacidad de clonarse fue pensando que tal vez no se quiere que todos los tipos de objetos tengan la capacidad de clonarse. Así es que **Object.clone()** verifica que una clase implemente la interfase **Cloneable**. Si no, lanza una excepción **CloneNotSupportedException**. Así es que en general, se está forzado a implementar **Cloneable** como parte del soporte de clonación.

## Clonación exitosa

Una vez que se han entendido los detalles de implementar el método **clone()**, se es capaz de crear clases que puedan ser fácilmente duplicadas para proporcionar una copia local:

```
//: appendixa:LocalCopy.java
// Creando copias locales con clone().
import java.util.*;
class MyObject implements Cloneable {
    int i;
    MyObject(int ii) { i = ii; }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("MyObject can't clone");
        }
        return o;
    }
    public String toString() {
        return Integer.toString(i);
    }
}
public class LocalCopy {
    static MyObject g(MyObject v) {
        // Pasando una referencia, modificando el objeto externo:
        v.i++;
        return v;
    }
    static MyObject f(MyObject v) {
        v = (MyObject)v.clone(); // Copia local
        v.i++;
        return v;
    }
    public static void main(String[] args) {
        MyObject a = new MyObject(11);
        MyObject b = g(a);
        // Probando la equivalencia de la referencia,
        // no hay equivalencia de objeto:
        if(a == b)
            System.out.println("a == b");
        else
```

```

        System.out.println("a != b");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        MyObject c = new MyObject(47);
        MyObject d = f(c);
        if(c == d)
            System.out.println("c == d");
        else
            System.out.println("c != d");
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
} //:~

```

Antes que nada, **clone()** debe ser accesible así es que se debe hacerla pública. Segundo, para la parte inicial de la operación **clone()** se debería llamar a la versión de la clase base de **clone()**. La **clone()** que es llamada aquí es la que esta predefinida dentro de **Object**, y se puede llamar porque esta protegida y por esta razón accesible en las clases derivadas.

**Object.clone()** resuelve que tan grande el objeto es, creando suficiente memoria para uno nuevo, y copia todos los bits de la vieja a la nueva. Esto es llamado una *copia a nivel de bits* y es típicamente lo que se espera que haga un método **clone()**. Pero antes **Object.clone()** realiza las operaciones, primero verifica para ver si una clase es **Cloneable**- esto es, cuando implementa la interfase **Cloneable**. Si no lo hace, **Object.clone()** lanza una **CloneNotSupportedException** para indicar que no se puede clonar. De esta forma, se tiene que rodear la llamada a **super.clone()** con un bloque try-catch, para capturar una excepción que debería no suceder nunca (dado que se esta implementando la interfase **Cloneable**).

En **LocalCopy**, los dos métodos **g()** y **f()** demuestran la diferencia entre dos estrategias para el pasaje de argumentos. **g()** muestra el pasaje por referencia en donde se modifica el objeto externo y se retorna una referencia a este objeto externo, donde **f()** clona el argumento, de esta forma lo desacopla y deja el objeto original solo. Este puede entonces proceder a hacer lo que quiera, y nunca retornar una referencia a este nuevo objeto sin ningún efecto indeseable al original. Note la instrucción algo curiosa:

```
| v = (MyObject)v.clone();
```

Esto es donde el objeto local es creado. Para evitar confusiones por este tipo de instrucción, recuerde que es mas bien un extraño idioma de codificación es perfectamente factible en Java porque cada identificador de objeto es actualmente una referencia. Así es que la referencia **v** es utilizada para clonar una copia de a lo que hace referencia y esto retorna una referencia a el tipo base **Object** (dado que esta definido de esta forma en **Object.clone()**) que puede entonces ser convertido a el tipo apropiado.

En el **main()**, la diferencia entre los efectos de las dos estrategias de pasaje de argumentos en los dos métodos diferentes es probado. La salida es:

```
a == b  
a = 12  
b = 12  
c != d  
c = 47  
d = 48
```

Es importante notar que las pruebas de equivalencia en Java no observan dentro de los objetos que son comparados para ver si sus valores son los mismos. Las operaciones `==` y `!=` son simplemente comparaciones de referencias. Si la dirección dentro de las referencias son la misma, las referencias apuntan a el mismo objeto y son por lo tanto "iguales". ¡Así es que lo que las operaciones están probando realmente es cuando las referencias son alias del mismo objeto!

## El efecto de `Object.clone()`

¿Que sucede actualmente cuando un `Object.clone()` es llamado que hace que sea tan esencial a llamar a `super.clone()` cuando se sobrecarga `clone()` en una clase? El método `clone()` en la clase raíz es responsable por la creación de la cantidad correcta de almacenamiento y hace la copia a nivel de bits del objeto original en un nuevo almacenaje del objeto. Esto es, no solo crea espacio y copia un `Object`-resuelve el tamaño del objeto preciso que será copiado y lo duplica. Dado todo esto está sucediendo desde el código en el método `clone` definido en la clase raíz (que no tiene ni idea de que es lo que esta heredando de esta), se puede adivinar que el proceso que involucra RTTI para determinar el objeto que actualmente se esta clonando. De esta forma, el método `clone()` puede crear la cantidad adecuada de almacenamiento y hacer la copia a nivel de bits correcta para ese tipo.

Cualquier cosa que se haga, la primer parte del proceso de clonado suele normalmente ser una llamada a `super.clone()`. Esto establece un trabajo de base para la operación de clonado haciendo un duplicado exacto. En este punto se puede realizar otras operaciones necesarias para completar el clonado.

Para saber con seguridad cual de estas otras operaciones es, se necesita entender exactamente que adquiere `Object.clone()`. ¿En particular, automáticamente clona el destino para todas las referencias? El siguiente ejemplo prueba esto:

```
//: appendix:Snake.java  
// Prueba la clonación para ver si el destino de  
// las referencias son también clonados.  
public class Snake implements Cloneable {  
    private Snake next;  
    private char c;  
    // Valor de i == al número de segmentos  
    Snake(int i, char x) {  
        c = x;
```

```

        if(--i > 0)
            next = new Snake(i, (char)(x + 1));
    }
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.out.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);
}
} //:~

```

Una **Snake** es creada de un montón de segmentos, cada uno del tipo **Snake**. De esta forma, es una lista enlazada por separado. Los segmentos son creados de forma recursiva, quitándole valor al primer argumento del constructor para cada segmento hasta que cero es alcanzado. Para darle a un segmento una etiqueta única, el segundo argumento, un **char**, es incrementado para cada llamada recursiva a el constructor.

El método **increment()** de forma recursiva incrementa cada etiqueta así es que se puede ver el cambio, y **toString()** imprime de forma recursiva cada etiqueta. La salida es:

```

s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f

```

Esto significa que solo el primer segmento es duplicado por **Object.clone()**, por consiguiente no hace una copia superficial. Si se quiere que la totalidad de la serpiente (Snake) sea duplicada -una copia profunda- se debe realizar operaciones adicionales dentro del método **clone()** sobrescrito.

Típicamente se llamará a **super.clone()** en cualquier clase derivada de una clase que se pueda clonar para asegurarse de que todas las operaciones en la clase base (incluyendo **Object.clone()**) se realicen. Esto es seguido de una llamada explícita a **clone()** para cada referencia en su objeto; de otra forma estas referencias serán realizadas mediante alias a aquellos objetos originales. Esto es análogo a la forma en que los constructores son llamados -el constructor de la clase base primero, luego el siguiente constructor derivado, y así hasta el constructor mas derivado. La diferencia es que **clone()** no es un constructor, así es que no hay nada que suceda automáticamente. Hay que asegurarse de hacerlo uno mismo.

## Clonando un objeto compuesto

Es un problema que se encontrará cuando se trate de copiar en profundidad un objeto compuesto. Se debe asumir que el método **clone()** en los objetos miembro realizará una copia profunda de *sus* referencias, y así sucesivamente. Esto es un compromiso considerable. En efectivo significa que para que una copia profunda trabaje se debe controlar todo el código en todas las clases, o al menos suficiente conocimiento acerca de todas las clases involucradas en una copia profunda para saber que se está realizando su propia copia profunda correctamente.

Este ejemplo muestra que debe hacer para lograr una copia profunda cuando se trata con un objeto compuesto:

```
//: appendix:DeepCopy.java
// Clonando un objeto compuesto.
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(System.err);
        }
        return o;
    }
}
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(System.err);
        }
        return o;
    }
}

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(System.err);
        }
        // Must clone references:
        o.depth = (DepthReading)o.depth.clone();
        o.temperature =
            (TemperatureReading)o.temperature.clone();
        return o; // Upcasts back to Object
    }
}
public class DeepCopy {
    public static void main(String[] args) {
        OceanReading reading =
            new OceanReading(33.9, 100.5);
        // Now clone it:
        OceanReading r =
            (OceanReading)reading.clone();
    }
} //:~

```

**DepthReading** y **TemperatureReading** son bastante similares; ambos contienen solo primitivas. Sin embargo, el método **clone()** puede ser bastante simple: llama a **super.clone()** y retorna el resultado. Debe notarse que el código de **clone()** para ambas clases es idéntico.

**OceanReading** esta compuesto por los objetos **DepthReading** y **TemperatureReading** y de esta forma, para producir una copia profunda, sus **clone()** deben clonar las referencias dentro de **OceanReading**. Para lograr esto, a el resultado de **super.clone()** se le debe realizar una conversión a un objeto **OceanReading** (de esta forma se puede acceder a las referencias **depth** y **temperature**).

## Una copia profunda con **ArrayList**

Volvemos a visitar el ejemplo de **ArrayList** de temprano en este apéndice. Esta vez la clase **Int2** se puede clonar, así es que el **ArrayList** puede ser copiado en profundidad:

```
//: appendix:AddingClone.java
// Se debe ir a través de unos pocos giros para
// agregar capacidad de clonación a clases propias.
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;
    }
}
// Una vez que es clonable, la herencia
// no quita esta capacidad:
class Int3 extends Int2 {
    private int j; // Duplicado automáticamente
    public Int3(int i) { super(i); }
}
public class AddingClone {
    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);
        // Todo lo heredable se puede también clonar:
        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 se clona cada elemento:
        for(int i = 0; i < v.size(); i++)
            v2.set(i, ((Int2)v2.get(i)).clone());
        // Se incrementan todos los elementos de v2:
        for(Iterator e = v2.iterator();
            e.hasNext(); )
```

```

        ((Int2)e.next()).increment();
        // Vemos si cambian elementos de v:
        System.out.println("v: " + v);
        System.out.println("v2: " + v2);
    }
} //:~

```

**Int3** es heredado de **Int2** y un nuevo miembro primitivo **int j** es agregado. Se puede pensar que se necesitará sobrescribir **clone()** nuevamente para asegurarse que **j** es copiado, pero este no es el caso. Cuando el **clone()** de **Int2** es llamado como **clone()** de **Int3**, este llama a **Object.clone()**, que determina que se está trabajando con un **Int3** y duplica todos los bits en el **Int3**. Siempre y cuando no se agreguen referencias que necesiten ser clonadas, la única llamada a **Object.clone()** realiza todas las duplicaciones necesarias, sin importar que tan abajo en la jerarquía este definido **clone()**.

Se puede ver lo que se necesita para realizar una copia profunda de un **ArrayList**: luego de que **ArrayList** es clonado, se tiene que recorrer y clonar cada uno de los objetos apuntados por el **ArrayList**. Se tiene que hacer algo similar a esto para realizar una copia de un **HashMap**.

El resto del ejemplo muestra que la clonación se sucede llegando a esto, una vez que un objeto es clonado, se puede cambiar y el objeto original no es tocado.

## Copia profunda mediante serialización

Cuando se considere una serialización de un objeto Java (introducido en el capítulo 11), se debe observar que un objeto que es serializado y luego deserializado es, en efecto clonado.

¿Así es que, por qué no utilizar serialización para realizar una copia profunda? He aquí un ejemplo que compara las dos estrategias midiendo el tiempo:

```

//: appendixA: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 {
    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");
        }
        // Clonar el campo, también:
        o.o3 = (Thing3)o3.clone();
        return o;
    }
}
public class Compete {
    static final int SIZE = 5000;
    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 se obtienen las 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 se trata de clonar:
        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 así es que hay alguna copia profunda realizándose. Es interesante notar que mientras que las clases **Serializable** son fáciles de configurar, hay mucho mas trabajo para duplicarlas. La clonación involucra un montón de trabajo para configurar las clases, pero la duplicación actual de objetos es relativamente simple. El resultado realmente cuenta la historia. He aquí la salida de diferentes corridas:

```
| Duplication via serialization: 940 Milliseconds
| Duplication via cloning: 50 Milliseconds
| Duplication via serialization: 710 Milliseconds
| Duplication via cloning: 60 Milliseconds
| Duplication via serialization: 770 Milliseconds
| Duplication via cloning: 50 Milliseconds
```

A pesar de la diferencia significante de tiempo entre la serialización y el clonado, se notará también que la técnica de serialización parece variar mas en su duración, por lo que el clonado tiende a ser mas estable.

## Agregando capacidad de clonación a descendiendo una jerarquía

Si se crean nuevas clases, su clase base por defecto es **Object**, que por defecto no se puede clonar (como se verá en la siguiente sección). Siempre y cuando no se agregue explícitamente la capacidad de clonarse, esta no se obtendrá. Pero se puede agregar esta en cualquier capa y entonces tendrá capacidad de clonarse desde esa capa para abajo, de esta forma:

```
//: appendixa:HorrorFlick.java
// You can insert Cloneability
// at any level of inheritance.
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 no debería de pasar nunca:
            // ¡Ya es clonable!
            throw new InternalError();
        }
    }
}
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 que la capacidad de clonarse fuera agregada, el compilador lo detiene de tratar de clonar cosas. Cuando la capacidad de clonarse es agregada en **Scientist**, entonces **Scientist** y todos sus descendientes serán capaces de clonarse.

## ¿Por que este extraño diseño?

Si todo esto parece ser un esquema extraño, es porque lo es. Se puede preguntarse por que trabaja de esta forma. ¿Cual es el significado detrás de este diseño?

Originalmente, Java fue diseñado como lenguaje para controlar cajas de hardware y en definitiva sin la Internet en mente. En un lenguaje de propósito general como este, tiene sentido que el programador sea capas de clonar cualquier objeto. De esta forma, **clone()** fue colocado en la clase raíz **Object**, pero era un método público así es que siempre se podía clonar un objeto. Esto parecía ser la estrategia mas sensible, y después de todo. ¿Que podía dañar?

Bueno, cuando Java se distinguió como lenguaje de programación en la Internet, las cosas cambiaron. De pronto, hay temas de seguridad, y claro, estos temas tratan con la utilización de los objetos, y no necesariamente se quiere que cualquiera sea capas de clonar objetos de seguridad. Así es que lo que se esta viendo es un montón de parches aplicados en el simple, original y directo esquema: **clone()** no es protegido en **Object**. Debe sobrescribir e implementar **Cloneable** y tratar con las excepciones.

Tiene valor notar que se debe utilizar la interfase **Cloneable** *solo* si se va a llamar a el método **clone()** de **Object**, dado que este método verifica en tiempo de ejecución para asegurarse que su clase implementa **Cloneable**. Pero por consistencia (y dado que **Cloneable** esta vacío de todas formas) se debería implementar.

# Controlando la capacidad de clonación

Podría sugerir que, para remover la capacidad de clonación, el método **clone()** simplemente se hace privado, pero esto no funcionará dado que no se puede tomar un método de una clase base y hacerlo menos accesible en una clase derivada. Así es que no es tan simple. Y aún, es necesario ser capaz de controlar cuando un objeto puede ser clonado. Hay actualmente un montón de actitudes que se pueden tomar para esto en una clase que se diseña:

1. Indiferencia. No se hace nada con la clonación, lo que significa que su clase no puede ser clonada pero una clase que herede de esta puede agregar clonación si quiere. Esto trabaja solo en el **Object.clone()** por defecto que no hará nada razonable con todos los campos en su clase.
2. Dar soporte a **clone()**. Seguir la práctica estándar de implementar **Cloneable** y sobrescribir **clone()**. En el **clone()** sobrescrito, se llama a **super.clone()** y se capturan todas las excepciones (así es que el sobreescribir **clone()** no lanza ninguna excepción).
3. Dar soporte a la clonación opcionalmente. Si su clase guarda referencias a otros objetos que pueden o no ser clonados (una clase contenedor, por ejemplo), el **clone()** puede tratar de clonar todos los objetos para los cuales se tienen referencias, y si estos lanzan excepciones solo se pasan estas excepciones al programador. Por ejemplo, considere un tipo especial de **ArrayList** que trata de clonar todos los objetos que almacena. Cuando se escribe tal **ArrayList**, no se conoce qué tipo de objetos el cliente programador puede poner en su **ArrayList**, así es que no se sabe si se pueden ser clonados.
4. No implementar **Cloneable** pero sobreescritir **clone()** como protegida, produciendo el comportamiento de la correcta copia para todos los campos. De esta forma, cualquiera que herede de esta clase puede sobreescritir **clone()** y llamar a **super.clone()** para producir el correcto comportamiento de copia. Debe notarse que su implementación puede y debe invocar a **super.clone()** aún cuando el método espera un objeto **Cloneable** (este lanzará una excepción de otra manera), dado que no la invocará directamente en un objeto de su tipo. Esta será invocada solo a través de su clase derivada, que, si trabaja exitosamente, implementa **Cloneable**.
5. Se puede tratar de prevenir la clonación no implementando **Cloneable** y sobreescribiendo **clone()** para lanzar una excepción. Esto es exitoso

solo si cualquier clase derivada de esta llama a **super.clone()** en su redefinición de **clone()**. De otra forma, un programador puede ser capaz de pasarlo por alto.

6. Prevenir la clonación haciendo su clase **final**. Si **clone()** no ha sido sobrescrita por ninguna de sus clases antepasadas, entonces no puede ser. Si fue, entonces sobrescribirla nuevamente y lanzar **CloneNotSupportedException**. Haciendo la clase **final** es la única forma de garantizar que se previene la clonación. Además, cuando se trata con objetos de seguridad u otras situaciones en donde se quiere controlar el número de objetos creados se debe hacer todos los constructores privados y proporcionar uno o mas métodos especiales para crear objetos. De esta forma, estos métodos puede restringir el número de objetos creados y las condiciones en que ellos son creados (Un caso particular de esto es el patrón *singleton* mostrado en *Thinking in Patterns with Java* que se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com)).

He aquí un ejemplo que muestra las distintas formas que la clonación puede ser implementada y entonces, mas adelante en la jerarquía, “apagada”.

```
//: appendix:CheckCloneable.java
// Prueba para ver si una referencia puede ser clonada.
// No se puede clonar esta porque no
// sobrescribe clone():
class Ordinary {}
// Sobrescribe clone, pero no implementa
// Cloneable:
class WrongClone extends Ordinary {
    public Object clone()
        throws CloneNotSupportedException {
            return super.clone(); // Throws exception
    }
}
// Hace todo las cosas correctas para clonar:
class IsCloneable extends Ordinary
    implements Cloneable {
    public Object clone()
        throws CloneNotSupportedException {
            return super.clone();
    }
}
// Apaga la clonación lanzando una excepción:
class NoMore extends IsCloneable {
    public Object clone()
        throws CloneNotSupportedException {
            throw new CloneNotSupportedException();
    }
}
class TryMore extends NoMore {
    public Object clone()
        throws CloneNotSupportedException {
```

```

        // Llama NoMore.clone(), lanza una excepcion:
        return super.clone();
    }
}

class BackOn extends NoMore {
    private BackOn duplicate(BackOn b) {
        // De alguna forma hace una copia de b
        // y retorna esa copia. Esto es una copia
        // ficticia, solo para solo para verlo:
        return new BackOn();
    }
    public Object clone() {
        // No llama a NoMore.clone():
        return duplicate(this);
    }
}
// No se puede heredar de esto, así es que no se
// puede sobrescribir el método clone como en BackOn:
final class ReallyNoMore extends NoMore {}
public class CheckCloneable {
    static Ordinary tryToClone(Ordinary ord) {
        String id = ord.getClass().getName();
        Ordinary x = null;
        if(ord instanceof Cloneable) {
            try {
                System.out.println("Attempting " + id);
                x = (Ordinary)((IsCloneable)ord).clone();
                System.out.println("Cloned " + id);
            } catch(CloneNotSupportedException e) {
                System.err.println("Could not clone "+id);
            }
        }
        return x;
    }
    public static void main(String[] args) {
        // Conversión ascendente:
        Ordinary[] ord = {
            new IsCloneable(),
            new WrongClone(),
            new NoMore(),
            new TryMore(),
            new BackOn(),
            new ReallyNoMore(),
        };
        Ordinary x = new Ordinary();
        // Esto no comila, dado que clone() es
        // protegido en Object:
        // x = (Ordinary)x.clone();
        // tryToClone() prueba primero para ver si
        // una clase implementa Cloneable:
        for(int i = 0; i < ord.length; i++)
            tryToClone(ord[i]);
    }
} ///:~

```

La primer clase, **Ordinary**, representa el tipo de clases que han sido vistas a través de este libro: no tienen soporte para clonación, pero de todos modos, no hay prevenciones para evitarlo. Pero si se tiene una referencia a un objeto **Ordinary** al que se le puede realizar una conversión ascendente de una clase mas derivada, no se puede decir si se puede ser clonada o no.

La clase **WrongClone** muestra una forma incorrecta de implementar clonación. Esta no sobrecarga **Object.clone()** y hace ese método público, pero no implementa **Cloneable**, así es que cuando **super.clone()** es llamado (lo que resulta en una llamada a **Object.clone()**), **CloneNotSupportedException** es lanzada así es que la clonación no trabaja.

En **IsCloneable** se puede ver todas las acciones correctas realizadas para clonación: **clone()** es sobreescrita y **Cloneable** es implementada. Sin embargo, este método **clone()** y muchos otros que siguen en este ejemplo *no capturan* **CloneNotSupportedException**, pero en lugar de eso se lo pasan a el que llama, que debe entonces colocar un bloque try-catch alrededor de este. En el método **clone()** propio se debe típicamente capturar **CloneNotSupportedException** dentro de **clone()** en lugar de pasarlo. Así como se verá, en este ejemplo es mas informativo pasar las excepciones.

La clase **NoMore** intenta “apagar” la clonación en la forma en que los diseñadores de Java pretenden: en la clase derivada **clone()**, de lanza **CloneNotSupportedException**. ¿El método **clone()** en la clase **TryMore** de forma adecuada llama a **super.clone()**, y esto resuelve a **NoMore.clone()** dentro del método **clone()** sobreescrito? En **BackOn**, se puede ver como esto puede suceder. Esta clase utiliza un método separado **duplicate()** para hacer una copia del objeto actual y llama este método dentro de **clone()** en lugar de llamar **super.clone()**. La excepción nunca es lanzada y la nueva clase puede clonarse. No se puede confiar en lanzar una excepción para prevenir que una clase se le agregue la capacidad de clonarse. La única solución segura es mostrada en **ReallyNoMore**, que es **final** y de esta forma no puede ser heredada. Esto significa que si **clone()** lanza una excepción en la clase **final**, no puede ser modificada con herencia y la prevención de clonación está asegurada (No se puede explícitamente llamar a **Object.clone()** de una clase que tiene un nivel arbitrario de herencia; se está limitado llamar a **super.clone()**, que tiene el acceso a solo la clase base directa). De esta forma, si se hace cualquier objeto que involucre temas de seguridad, querremos hacer estas clase finales.

El primer método que se ve en la clase **CheckCloneable** es **tryToClone()**, que toma un objeto **Ordinary** y verifica si se puede clonar con **instanceof**. Si se puede, convierte el objeto en un **IsCloneable**, llama a **clone()** y convierte el resultado nuevamente en **Ordinary**, capturando cualquier excepción que sea lanzada. Note que el uso de identificación en tiempo de ejecución (capítulo 12) para imprimir el nombre de la clase así se puede ver que esta sucediendo.

En **main()**, diferentes tipos de objetos **Ordinary** son creados y convertidos ascendenteamente a **Ordinary** en la definición del arreglo. Las primeras dos líneas de código luego de crear un objeto plano **Ordinary** y tratar de clonarlo. Sin embargo, este código puede no compilar porque **clone()** es un método protegido en **Object**. El resto de el código se mueve a través del arreglo y trata de clonar cada objeto, reportando el éxito o la falla de cada uno. La salida es:

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

Así es que para resumir, si se quiere que una clase tenga la capacidad de clonarse:

1. Se implementa la interfase **Cloneable**.
2. Se sobrescribe **clone()**.
3. Se llama a **super.clone()** dentro de **clone()**.
4. Se capturan las excepciones dentro de su **clone()**.

Esto producirá los efectos mas convenientes.

## El constructor copia

La clonación puede parecer un proceso complicado de poner a punto. Parece ser como que debería haber una alternativa. Una estrategia que puede sucedernos (especialmente si es un programador C++) es hacer un constructor especial cuyo trabajo sea duplicar un objeto. En C++, esto es llamado el *constructor copia*. Al principio, esto parece como una solución obvia, pero de hecho esto no trabaja. He aquí un ejemplo:

```
//: appendix:CopyConstructor.java
// Un constructor para copiar un objeto del mismo
// tipo, como un intento de crear una copia local.
class FruitQualities {
    private int weight;
    private int color;
    private int firmness;
    private int ripeness;
    private int smell;
    // etc.
    FruitQualities() { // Constructor por defecto
        // hacer algo con significado...
```

```

    }
    // Otros constructores:
    // ...
    // Constructor copia:
    FruitQualities(FruitQualities f) {
        weight = f.weight;
        color = f.color;
        firmness = f.firmness;
        ripeness = f.ripeness;
        smell = f.smell;
        // etc.
    }
}
class Seed {
    // Miembros...
    Seed() { /* Default constructor */ }
    Seed(Seed s) { /* Copy constructor */ }
}
class Fruit {
    private FruitQualities fq;
    private int seeds;
    private Seed[] s;
    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:
    Fruit(Fruit f) {
        fq = new FruitQualities(f.fq);
        seeds = f.seeds;
        // Llama a todos los constructores copia de Seed:
        for(int i = 0; i < seeds; i++)
            s[i] = new Seed(f.s[i]);
        // Otras actividades de constructores copia...
    }
    // Para permitir que los constructores derivados (u otros
    // métodos) colocar en diferentes calidades:
    protected void addQualities(FruitQualities q) {
        fq = q;
    }
    protected FruitQualities getQualities() {
        return fq;
    }
}
class Tomato extends Fruit {
    Tomato() {
        super(new FruitQualities(), 100);
    }
    Tomato(Tomato t) { // Constructor copia
        super(t); // Conversión ascendente para constructores
    }
}

```

```

        // copia base
        // Otras actividades de constructores copias...
    }
}

class ZebraQualities extends FruitQualities {
    private int stripedness;
    ZebraQualities() { // Constructor por defecto
        // Has algo significativo...
    }
    ZebraQualities(ZebraQualities z) {
        super(z);
        stripedness = z.stripedness;
    }
}
class GreenZebra extends Tomato {
    GreenZebra() {
        addQualities(new ZebraQualities());
    }
    GreenZebra(GreenZebra g) {
        super(g); // Calls Tomato(Tomato)
        // Restaura las calidades correctas:
        addQualities(new ZebraQualities());
    }
    void evaluate() {
        ZebraQualities zq =
            (ZebraQualities)getQualities();
        // Hace algo con las calidades
        // ...
    }
}
public class CopyConstructor {
    public static void ripen(Tomato t) {
        // Usa 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); // Hmm... ¿Esto trabajará?
        System.out.println("In slice, f is a " +
            f.getClass().getName());
    }
    public static void main(String[] args) {
        Tomato tomato = new Tomato();
        ripen(tomato); // OK
        slice(tomato); // OOPS!
        GreenZebra g = new GreenZebra();
        ripen(g); // OOPS!
        slice(g); // OOPS!
        g.evaluate();
    }
} ///:~

```

Parece ser un poco extraño al principio. Claro, frutas como calidades. ¿Pero por qué no simplemente colocamos los miembros datos representando

aquellas calidades directamente in la clase **Fruit**? Hay dos razones potenciales. La primera es que se puede querer fácilmente insertar o cambiar las calidades. Debe notarse que **Fruit** tiene un método **addQualities()** protegido para permitir a las clases derivadas hacer esto (Se puede pensar en hacer lo que parece mas lógico para hacer que es tener un constructor protegido en **Fruit** que tome un argumento **FruitQualities**, pero los constructores no se heredan así es que no estará disponible en una clase de segundo o mayor nivel). Por hacer que las calidades de las frutas dentro de una clase separada, se tiene una gran flexibilidad, incluyendo la habilidad de cambiar las calidades en el medio del tiempo de vida de un objeto **Fruit** en particular.

La segunda razón para hacer **FruitQualities** un objeto separado es en el caso que se quiera agregar nuevas calidades o cambiar el comportamiento mediante herencia y polimorfismo. Debe notarse que para **GreenZebra** (que *realmente es* un tipo de tomate -He cultivado de estos y son fabulosos), el constructor llama a **addQualities()** y lo pasa a el objeto **ZebraQualities**, que es derivado de **FruitQualities** así es que puede ser enganchado a la referencia **FruitQualities** en una clase base. Claro, cuando **GreenZebra** utiliza **FruitQualities** debe realizarse una conversión descendente a el tipo correcto (como se ha visto en **evaluate()**), pero siempre se sabe que tipo es **ZebraQualities**.

También se ha visto que hay una clase **Seed**, y esta **Fruit** (que por definición contiene sus propias semillas<sup>4</sup>) conteniendo un arreglo de **Seeds**.

Finalmente, debe notarse que cada clase tiene un constructor copia, y que cada constructor copia debe tener cuidado de llamar los constructores copia para la clase base y los objetos miembros para producir una copia profunda. El constructor copia es probado dentro de la clase **CopyConstructor**. El método **ripen()** toma un argumento **Tomato** y realiza una construcción copia en el para duplicar el objeto:

```
| t = new Tomato(t);  
| mientras que slice() toma un objeto mas genérico Fruit y también lo duplica:  
| f = new Fruit(f);  
| Estos son probados con diferentes tipos de Fruit en main(). Aquí está la  
salida:
```

```
In ripen, t is a Tomato  
In slice, f is a Fruit  
In ripen, t is a Tomato  
In slice, f is a Fruit
```

Aquí es donde el problema se muestra. Luego de que el constructor copia que se le sucede a **Tomato** dentro de **slice()**, el resultado no es mas un objeto **Tomato**, simplemente es un **Fruit**. Se ha perdido toda su calidad de tomate.

---

<sup>4</sup> Excepto por el pobre aguacate, que ha sido reclasificado a simplemente “gordo”.

Mas adelante, cuando se tome un **GreenZebra**, ambos, **ripen()** y **slice()** lo convertirán en **Tomato** y **Fruit**, respectivamente. De esta forma, desafortunadamente, el esquema del constructor copia no es bueno para nosotros en Java cuando intentamos hacer una copia local de un objeto.

### ¿Por que funciona en C++ y no en Java?

El constructor copia es una parte fundamental de C++, dado que automáticamente hace una copia local de un objeto. Aún cuando el ejemplo arriba prueba que no trabaja para Java. ¿Por que? En Java todo lo que manipulamos es una referencia, mientras que en C++ se puede tener entidades como referencias y se puede *también* pasar los objetos directamente. Esto es para lo que un constructor copia de C++ es: cuando se quiere tomar un objeto y pasarlo por valor, se duplica el objeto. Así es que trabaja bien en C++, pero se debe tener en mente que este esquema falla en Java, así es que no lo utilice.

## Clases de solo lectura

Mientras la copia local producida por **clone()** nos da los efectos deseados en la clase apropiada, es un ejemplo de forzar al programador (el autor del método) a ser responsable de prevenir los padecimientos del aliasing. ¿Que si se esta haciendo una librería de propósito general y comúnmente utilizada en la que no se puede asumir de que será siempre clonada en los lugares adecuados? ¿O mejor, que si se *quiere* permitir aliasing por un tema de eficiencia -para prevenir la duplicación innecesaria de objetos- pero no se quiere los efectos negativos del aliasing?

Una solución es crear *objetos inmutables* que pertenecen a las clases de solo lectura. Se puede definir una clase de tal forma que no tenga métodos que causen cambios en el estado interno del objeto. En este tipo de clases, el aliasing no tiene impacto dado que se puede leer solo el estado interno, así es que si muchas partes de código están leyendo el mismo objeto no hay problema.

Como un simple ejemplo de objetos inmutables, la librería estándar de Java contiene las clases “envoltura” para todos los tipos primitivos. Ya debemos haber descubierto que, si se quiere almacenar un **int** dentro de un contenedor como lo es un **ArrayList** (que toma solo referencias a **Objects**), se puede envolver el **int** dentro de la clase **Integer** de la librería estándar:

```
//: appendix:ImmutableInteger.java
// La clase Integer no puede ser cambiada.
import java.util.*;
public class ImmutableInteger {
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
```

```

        for(int i = 0; i < 10; i++)
            v.add(new Integer(i));
        // Pero como se puede cambiar el entero
        // dentro de Integer?
    }
} //:~

```

La clase **Integer** (así como todas las clases “envoltura” de primitivas) implementan la inmutabilidad en una forma muy simple: no tienen métodos que permitan cambiar el objeto.

Si no necesita un objeto que almacene un tipo primitivo que pueda ser modificado, lo debe crear uno mismo. Afortunadamente, esto es trivial:

```

//: appendixa:MutableInteger.java
// A changeable wrapper class.
import java.util.*;
class IntValue {
    int n;
    IntValue(int x) { n = x; }
    public String toString() {
        return Integer.toString(n);
    }
}
public class MutableInteger {
    public static void main(String[] args) {
        ArrayList 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)).n++;
        System.out.println(v);
    }
} //:~

```

Debe notarse que **n** es amigable para simplificar el código.

**IntValue** puede ser incluso simplificado si la inicialización por defecto a cero es adecuada (entonces no se necesita el constructor) y no hay que preocuparse acerca de imprimirla (entonces no se necesita el **toString()**):

```

class IntValue { int n; }

```

Traer el elemento y convertirlo es un poco complicado, pero esto es una característica de **ArrayList**, no de **IntValue**.

## Creando clases de solo lectura

Es posible crear su propia clase de solo lectura. He aquí un ejemplo:

```

//: appendixa:Immutable1.java
// Los objetos no pueden ser modificados
// son inmunes a el aliasing.
public class Immutable1 {
    private int data;

```

```

public Immutable1(int initVal) {
    data = initVal;
}
public int read() { return data; }
public boolean nonzero() { return data != 0; }
public Immutable1 quadruple() {
    return new Immutable1(data * 4);
}
static void f(Immutable1 il) {
    Immutable1 quad = il.quadruple();
    System.out.println("il = " + il.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());
}
} //:~

```

Todos los datos son privados, y se puede ver que ninguno de los métodos públicos modifican los datos. Efectivamente, el método que parece modificar un objeto es **quadruple()**, pero este crea un nuevo objeto **Immutable1** y deja el original sin tocar.

El método **f()** toma un objeto **Immutable1** y realiza varias operaciones en él, y la salida de **main()** demuestra que no hay cambios en **x**. De esta forma, los objetos de **x** pueden ser aliased muchas veces sin daño porque la clase **Immutable1** está diseñada para garantizar que los objetos no sean cambiados.

## El inconveniente de la inmutabilidad

Crear una clase inmutable parece al principio proporcionar una solución elegante. Sin embargo, cuandoquiera que necesite modificar un objeto de ese nuevo tipo se debe sufrir la sobrecarga de la creación de un nuevo objeto, así como la potencial causa de recolección de basura más frecuente. Para algunas clases esto no es un problema, pero para otras (como lo es la clase **String**) es prohibitivamente costoso.

La solución es crear una clase compañera que *pueda* ser modificada. Entonces, cuando se está haciendo un montón de modificaciones, se puede cambiar a utilizar la clase compañera modificable y regresar a la clase inmutable cuando se haya terminado.

El ejemplo más arriba puede ser modificado para mostrar esto:

```

//: appendixA:Immutable2.java
// Una clase compañera para hacer
// cambios a objetos inmutables.
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 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());
    }
}

```

**Immutable2** contiene métodos que, como antes, preservan la inmutabilidad de los objetos produciendo nuevos objetos cuandoquiera que una modificación es deseada. Estos son los métodos **add()** y **multiply()**. La clase compañera es llamada **Mutable**, y también tiene los métodos **add()** y **multiply()**, pero estos modifican el método **Mutable** en lugar de hacer uno nuevo. Además, **Mutable** tiene un método para utilizar sus datos 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 es realizado con la clase **immutable2** y se puede ver que cuatro nuevos objetos **immutable2** son creados en el proceso (Y cada vez **val** es reasignado, el objeto anterior se convierte en basura).

En el método **modify2()**, se puede ver que la primera acción es tomar **Immutable2** y producir un **Mutable** de él (Esto es como llamar a **clone()** como se vio temprano, pero en este momento un tipo diferente de objeto es creado). Entonces el objeto **Mutable** es utilizado para realizar un montón de operaciones de cambio *sin* requerir la creación de muchos objetos nuevos. Finalmente, retrocede para convertirse en un **Immutable2**. Aquí, dos nuevos objetos son creados (el **Mutable** y el **Immutable2** resultante) en lugar de cuatro.

Esta estrategia tiene sentido, entonces, cuando:

1. Se necesitan objetos inmutables y
2. A menudo necesitamos hacer un montón de modificaciones o
3. Es costoso crear un nuevo objeto inmutable

## Strings inmutables

Considere el siguiente código:

```
//: appendix:Stringer.java
public class Stringer {
    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
    }
}
```

Cuando **q** es pasado a **upcase()** es una copia de la referencia a **q**. El objeto de esta referencia esta conectado para estar colocado en una única posición física. Las referencias son copiadas como son pasadas.

Mirando la definición para **upcase()**, se puede ver que la referencia que es pasada tiene el nombre **s**, y existe solo mientras el cuerpo de **upcase()** está siendo ejecutado. Cuando **upcase()** se completa, la referencia local a **s** se desvanece. **upcase()** retorna el resultado, que es la cadena original con todos los caracteres llevados a mayúsculas. Claro, actualmente retorna una referencia a el resultado. Pero esto deja afuera que la referencia que retorna es para un nuevo objeto, y la **q** original es dejada sola. ¿Como sucede esto?

## Constantes implícitas

Si se dice:

```
| String s = "asdf";  
| String x = Stringer.upcase(s);
```

¿Realmente se quiere que el método **upcase()** cambie el argumento? En general, no se quiere, porque un argumento usualmente es visto por el lector del código como una parte de información proporcionada al método, no algo a ser modificado. Esto es una garantía importante, dado que hace el código mas fácil de escribir y de entender.

En C++, la disponibilidad de esta garantía es suficientemente importante para colocar una palabra clave especial, **const**, para permitir a el programador asegurarse que una referencia (puntero o referencia en C++) no pueda ser utilizada para modificar el objeto original. Pero entonces era requerido que el programador C++ fuera diligente y recordara utilizar **const** en todos lados. Puede ser confuso y fácil de olvidar.

## Sobrecargando '+' y el **StringBuffer**

Los objetos de la clase **String** son diseñados para ser inmutables, utilizando la técnica mostrada previamente. Si se examina la documentación en línea para la clase **String** (que es resumida brevemente en este apéndice), se verá que todos los métodos en la clase que aparece modificar a **String** realmente crea y retorna un objeto **String** nuevo conteniendo la modificación. El **String** original queda sin tocar. De esta forma, no hay característica en Java como **const** en C++ para hacer que el compilador soporte la inmutabilidad de sus objetos. Si se quiere, lo debe armar uno mismo como lo hace **String**.

Dado que los objetos **String** son inmutables, se puede crear un alias de un **String** cuantas veces quiera. Dado que es solo lectura no hay posibilidad de que una referencia cambie algo que afecte las otras referencias. Así es que un objeto solo lectura soluciona el problema de aliasing de forma bonita.

También es posible manejar todos los casos en donde se necesita modificar un objeto creando una versión completamente nueva del objeto con las modificaciones, como lo hace **String**. Sin embargo, para algunas operaciones esto no es eficiente. Un caso puntual es el operador '+' que ha sido sobrecargado para los objetos **String**. La sobrecarga significa que se le ha dado un significado extra cuando se utiliza con una clase particular ('+' y '+=' para **String** son los únicos operadores que son sobrecargados en Java, y Java no le permite al programador sobrecargar ningún otro<sup>5</sup>).

Cuando se utiliza con objetos **String**, el '+' permite concatenar **String**:

```
| String s = "abc" + foo + "def" + Integer.toString(47);
```

Se puede imaginar como esto *puede* funcionar: El **String** "abc" puede tener un método **append()** que crea un nuevo objeto **String** contenido "abc" concatenado con el contenido de **foo**. El nuevo objeto **String** puede entonces crear otro **String** nuevo al que se le agregue "def", y así sucesivamente.

Esto puede ciertamente trabajar, pero requiere la creación de una gran cantidad de objetos **String** solo para juntar este **String** nuevo, y luego se tiene un montón de objetos intermediarios que necesitan ser recolectados por el recolector de basura. Sospecho que los diseñadores de Java examinaron esta estrategia primero (que es una lección de diseño de software -no se sabe realmente nada acerca de un sistema hasta que se pone a prueba y se tiene algo trabajando). Sospecho también que descubrieron que entrega un rendimiento inaceptable.

La solución es una clase compañera mutable similar a la mostrada anteriormente. Para **String**, esta clase compañera es llamada **StringBuffer**, y el compilador automáticamente crea una **StringBuffer** para evaluar ciertas expresiones, en particular cuando los operadores sobrecargados + y += son utilizado con objetos **String**. Este ejemplo muestra que sucede:

```
//: appendix:ImmutableStrings.java
// Demostrando StringBuffer.
public class ImmutableStrings {
    public static void main(String[] args) {
        String foo = "foo";
        String s = "abc" + foo +
            "def" + Integer.toString(47);
        System.out.println(s);
        // El "equivalente" utilizando StringBuffer:
        StringBuffer sb =
```

---

<sup>5</sup> C++ permite a el programador sobrecargar operadores. Dado que esto a menudo es un proceso complicado (vea el Capítulo 10 de Pensando en C++, 2da edición, Prentice-Hall, 2000), los diseñadores de Java la tienen conceptualizada como una característica 'mala' que no debe ser incluida en Java. No es tan mala dado que terminaron haciéndolo ellos mismos, e irónicamente suficiente, la sobrecarga de operadores puede ser mucho mas fácil de utilizar en Java que en C++. esto puede ser visto en Python (vea www.Python.org) que tiene recolección de basura y sobrecarga directa de operadores.

```

    new StringBuffer("abc"); // Crea el String!
    sb.append(foo);
    sb.append("def"); // Crea el String!
    sb.append(Integer.toString(47));
    System.out.println(sb);
}
} //:~
```

En la creación del **String s**, el compilador está haciendo el equivalente grosero de un código subsiguiente que utiliza **sb**: un **StringBuffer** es creado mediante **append()** se le agregan nuevos caracteres directamente en el objeto **StringBuffer** (en lugar de eso se hacen nuevas copias cada vez). Mientras esto es mas eficiente, tiene valor notar que cada vez que se crea una cadena de caracteres entre comillas como lo es “**abc**” y “**def**”, el compilador lo convierte en un objeto **String**. Así es que pueden haber mas objetos creados de los que se esperan, a pesar de que la eficiencia otorgada por **StringBuffer**.

## Las clases **String** y **StringBuffer**

He aquí un vistazo general de los métodos disponibles para **String** y **StringBuffer** así es que se puede obtener una idea de que forma pueden interactuar. Estas tablas no contienen ningún método individual, en lugar de eso solo los que son importantes a esta discusión. Los métodos que son sobrecargados están resumidos en una sola fila.

Primero, la clase **String**:

Método	Sobrecarga de argumentos	Uso
<b>Constructor</b>	Sobrecargado: Defecto, <b>String</b> , <b>StringBuffer</b> , arreglos <b>char</b> , arreglos <b>bytes</b>	Creación de objetos <b>String</b>
<b>length()</b>		Número de caracteres en el <b>String</b> .
<b>charAt()</b>	<b>int</b> Index	El carácter en una posición en el <b>String</b> .
<b>getChars()</b> , <b>getBytes()</b>	El comienzo y el final de donde copiar, el arreglo en donde copiar, un índice dentro del arreglo destino.	Copia <b>chars</b> o <b>bytes</b> en un arreglo externo.
<b>toCharArray()</b>		Produce un <b>char[]</b> contenido los

		caracteres en el <b>String</b> .
<b>equals()</b> , <b>equalsIgnoreCase()</b>	Un <b>String</b> con que comparar.	Una prueba de igualdad del contenido de dos <b>Strings</b> .
<b>compareTo()</b>	Un <b>String</b> con que comparar.	El resultado es negativo, cero, o positivo dependiendo de el orden lexicográfico del <b>String</b> y el argumento. ¡Las mayúsculas y las minúsculas no son iguales!
<b>regionMatches()</b>	El desplazamiento dentro de este <b>String</b> , el otro <b>String</b> y su desplazamiento y largo para comparar. La sobrecarga agrega "ignore case".	Resultado <b>boolean</b> que indica cuando la región coincide.
<b>startsWith()</b>	El <b>String</b> con que debe comenzar. La sobrecarga agrega desplazamiento dentro del argumento.	Resultado <b>boolean</b> indicando cuando el <b>String</b> comienza con el argumento.
<b>endsWith()</b>	Un <b>String</b> que puede ser sufijo de este <b>String</b>	Resultado del tipo <b>boolean</b> indicando cuando el argumento es un sufijo.
<b>indexOf()</b> , <b>lastIndexOf()</b>	Sobrecargado: <b>char</b> , <b>char</b> e índice de comienzo, <b>String</b> , <b>String</b> , y índice de comienzo.	Retorna -1 si el argumento no es encontrado dentro de este <b>String</b> , de otra forma retorna el índice donde el argumento comienza. <b>lastIndexOf()</b> busca desde el final.
<b>substring()</b>	Sobrecargado: posición de comienzo, posición de comienzo y posición	Retorna un nuevo objeto <b>String</b> que contiene el grupo de

	de finalización.	caracteres especificado.
<b>concat()</b>	El <b>String</b> a concatenar.	Retorna un nuevo objeto <b>String</b> conteniendo los caracteres del <b>String</b> original seguido de los caracteres en el argumento.
<b>replace()</b>	El carácter a buscar que será remplazado, el nuevo carácter que lo remplazará.	Retorna un nuevo objeto <b>String</b> con los reemplazos hechos. Usa el viejo <b>String</b> si ninguna coincidencia es encontrada.
<b>toLowerCase()</b> <b>toUpperCase()</b>		Retorna un nuevo objeto <b>String</b> con los caracteres cambiados a mayúsculas o minúsculas. Usa el viejo <b>String</b> si no se necesita realizar cambios.
<b>trim()</b>		Retorna un nuevo objeto <b>String</b> sin los espacios en blanco de cada final. Utiliza el viejo <b>String</b> si no se necesita hacer cambios.
<b>valueOf()</b>	Sobrecargado: <b>Object</b> , <b>char[]</b> , <b>char[]</b> y el desplazamiento y el contador, <b>boolean</b> , <b>char</b> , <b>int</b> , <b>long</b> , <b>float</b> , <b>double</b> .	Retorna un <b>String</b> contenido una representación del argumento.
<b>intern()</b>		Produce uno y solo una referencia a <b>String</b> por cada secuencia única de caracteres.

Se puede ver que cada método **String** cuidadosamente retorna un objeto **String** nuevo cuando es necesario cambiar el contenido. También debe notarse que si el contenido no necesita ser cambiado el método solo

retornará una referencia a el **String** original. Esto ahorra espacio de almacenamiento y sobrecarga.

He aquí la clase **StringBuffer**:

Método	Argumento, sobrecarga	Uso
<b>Constructor</b>	Sobrecargado: defecto, largo del buffer a crear, <b>String</b> de donde crearlo.	Crea un nuevo objeto <b>StringBuffer</b> .
<b>toString()</b>		Crea un <b>String</b> de este <b>StringBuffer</b> .
<b>capacity()</b>		Retorna el número de espacios actualmente ubicados.
<b>ensureCapacity()</b>	Entero indicando la capacidad deseada.	Hace que el <b>StringBuffer</b> almacene al menos el número de espacios deseado.
<b>setLength()</b>	Entero indicando el nuevo largo de la cadena de caracteres en el buffer.	Trunca o expande la cadena de caracteres previa. Si se expande, se agregan nulos.
<b>charAt()</b>	Entero indicando la localización del elemento deseado.	Retorna el <b>char</b> en la posición en el buffer.
<b>setCharAt()</b>	Entero indicando la localización del elemento deseado y el nuevo valor <b>char</b> para el elemento.	Modifica el valor en la posición.
<b>getChars()</b>	El comienzo y el final de donde copiar, el arreglo que copiar y un índice dentro del arreglo destino.	Copia <b>chars</b> dentro de un arreglo externo. No hay <b>getBytes()</b> como en <b>String</b> .
<b>append()</b>	Sobrecargado: <b>Object</b> , <b>String</b> , <b>char[]</b> , <b>char[]</b> con desplazamiento y largo, <b>boolean</b> , <b>char</b> , <b>int</b> , <b>long</b> , <b>float</b> , <b>double</b> .	El argumento es convertido a una cadena y agregado a el final del buffer actual, incrementando el buffer si es necesario.
<b>insert()</b>	Sobrecargado, cada uno con <del>..... del</del>	El segundo argumento <del>..... del</del>

	un primer argumento del desplazamiento en donde comenzar a insertar: <b>Object</b> , <b>String</b> , <b>char[]</b> , <b>boolean</b> , <b>char</b> , <b>int</b> , <b>long</b> , <b>float</b> , <b>double</b> .	es convertido a cadena e insertado dentro del buffer actual comenzando en el desplazamiento. El buffer es incrementado si es necesario.
<b>reverse()</b>		El orden de los caracteres en el buffer es invertido

El método mas comúnmente utilizado es **append()**, que es utilizado por el compilador cuando se evalúan expresiones **String** que contienen los operadores '+' y '+=''. El método **insert()** tiene una forma similar, y ambos método realizan manipulaciones significantes a el buffer en lugar de crear nuevos objetos.

## Los **Strings** son especiales

Por ahora se ha visto que la clase **String** no es solo otra clase en Java. Hay un montón de casos especiales en **String**, no es el menor que sea una clase incluida y fundamental para Java. Entonces el echo de que una cadena de caracteres entre comillas sea convertido a **String** por el compilador y los operadores sobrecargados especiales + y +=. En este apéndice se ha visto el caso especial restante: la inmutabilidad cuidadosamente creadas utilizando el compañero **StringBuffer** y algo de magia extra en el compilador.

## Resumen

Dado que todo es una referencia en Java, y dado que todo objeto es creado en el heap y se recolecta la basura solo cuando no es mas utilizado, el buen gusto de la manipulación de objetos cambia, especialmente cuando se pasan y retornan objetos. Por ejemplo, en C o C++, si se quiere inicializar alguna parte almacenada en un método, probablemente se pida que el usuario pase la dirección de ese pedazo de almacenamiento a el método. De otra forma hay que preocuparse acerca de que quien es responsable por destruir el almacenaje. De esta forma, la interfase y comprensión de estos métodos es mas complicada. Pero en Java, nunca hay que preocuparse de la responsabilidad o cuando un objeto seguirá existiendo cuando es necesario, dado que siempre se tiene cuidado por nosotros. Se puede crear un objeto en el punto que se necesita, y no en breve, y nunca preocuparse acerca de los mecanismos de pasar la responsabilidad para ese objeto: simplemente se

pasa la referencia. A veces la simplificación que esto proporciona es inadvertida, otras veces es asombrosa.

Las desventajas de toda esta magia de capas bajas es doble:

1. Siempre recibe el golpe de eficiencia por el manejo extra de memoria (a pesar de que este puede ser pequeño), y siempre hay una pequeña cantidad de incertidumbre acerca del tiempo que algo va a tomar en ejecutarse (dado que el recolector de basura puede ser forzado a la acción cuando comencemos a quedarnos sin memoria). Para la mayoría de las aplicaciones, los beneficios pesan mas que los inconvenientes, y particularmente las secciones críticas en lo que respecta al tiempo pueden ser escritas utilizando métodos **nativos** (Vea el Apéndice B).
2. Aliasing: A veces se puede accidentalmente terminar con dos referencias a el mismo objeto, que es un problema solo si se asume que ambas referencias apuntan a objetos *distintos*. Aquí es cuando se necesita pagar con un poco de atención y, cuando sea necesario, clonar mediante **clone()** un objeto para prevenir que la otra referencia sea sorprendida por un cambio inesperado. Alternativamente, se puede soportar aliasing por un tema de eficiencia creando objetos inmutables cuyas operaciones pueden retornar objetos nuevos del mismo tipo o de algún tipo diferente, pero no cambiar nunca el objeto original así es que cualquier objeto alias de ese objeto no tiene modificaciones.

Algunas personas dicen que la clonación en Java es un cuello de botella en el diseño, y lo mandan al demonio, así es que ellos implementan su propia versión se clonación<sup>6</sup> y nunca llaman a el método **Object.clone()**, de esta forma eliminan la necesidad de implementar **Cloneable** y capturar la **CloneNotSupportedException**. Esto es ciertamente una estrategia razonable y dado que **clone()** es soportada con muy poca frecuencia dentro de la librería estándar de Java, aparentemente es igualmente segura. Pero siempre y cuando no se llame a **Object.clone()** no se necesita implementar **Cloneable** o capturar la excepción, así es que puede parecer aceptable de la misma forma.

## Ejercicios

La solución de los ejercicios seleccionados pueden encontrarse en el documento electrónico *The Thinking in Java Annotated Solution Guide* disponible por una pequeña propina en [www.BruceEckel.com](http://www.BruceEckel.com).

---

<sup>6</sup> Doug Lea, que favorablemente ha solucionado este tema, me ha sugerido esto, diciendo que simplemente se crea una función **duplicate()** para cada clase.

1. Demuestre un segundo nivel de aliasing. Cree un método que tome una referencia a un objeto pero no modifique el objeto de esa referencia. Sin embargo, la llamada a el método, pasarle la referencia, y este segundo método no modifique el objeto.
2. Cree una clase **myString** contenido un objeto **String** que se inicialice en el constructor utilizando el argumento del constructor. Agregue un método **toString()** y un método **concatenate()** que agregue un objeto **String** en su cadena interna. Implemente **clone()** en **myString**. Cree dos métodos **static** que tomen una referencia a **x** a **myString** como argumento y llamen a **x.concatenate("test")**, pero en el segundo método se llame a **clone()** primero. Pruebe los dos métodos y muestre los diferentes efectos.
3. Cree una clase llamada **Battery** contenido un **int** que sea un número de batería (un identificador único). Haga que tenga capacidad para clonarse y dele 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 método **clone()** para **Toy** que automáticamente clone todos los objetos **Battery**. Pruebe esto clonando **Toy** e imprimiendo el resultado.
4. Cambie **CheckCloneable.java** de tal forma que todos los métodos **clone()** capturen la **CloneNotSupportedException** en lugar de pasarl a el que llama.
5. Utilice la técnica de la clase compañera mutable, haga una clase inmutable contenido un **int**, un **double** y un arreglo de **char**.
6. Modifique **Complete.java** para agregar mas objetos miembro a las clases **Thing2** y **Thing4** y vea si se puede determinar como los tiempos varían con la complejidad -tanto cuando es una relación lineal simple o si parece mas complicada.
7. Partiendo de **Snake.java**, cree una versión de copia profunda de la serpiente.
8. Herede un **ArrayList** y haga que su **clone()** realice una copia profunda.

# B: La interfase nativa de Java (JNI)

*El material en este apéndice fue contribuido y utilizado con el permiso de Andrea Provaglio ([www.AndreaProvaglio.com](http://www.AndreaProvaglio.com)).*

El lenguaje Java y su API estándar es suficientemente completa para escribir aplicaciones totalmente maduras. Pero en algunos casos se debe llamar a código que no es nativo de Java; por ejemplo, si se quiere acceder a características propias de un sistema operativo, realizar una interfase con dispositivos especiales de hardware, reutilizar un código preexistente que no es Java, o implementar secciones críticas de código.

Realizar una interfase con código que no es Java requiere soporte dedicado en el compilador y en la máquina virtual, y herramientas adicionales para corresponder el código Java con el código que no lo es. La solución estándar para llamar a código que no es Java es proporcionada por JavaSoft y es llamada la *Interfase Nativa de Java*(Java Native Interface) que será introducida en este apéndice. Este no es un tratamiento profundo, y en algunos casos se asume que se tiene un conocimiento parcial de los conceptos relacionados y las técnicas.

JNI es una interfase medianamente rica que permite llamar a métodos nativos desde una aplicación Java. Esto fue agregado en Java 1.1, manteniendo cierto grado de compatibilidad con su equivalente en Java 1.0: la interfase de métodos nativos (NMI). NMI tiene características de diseño que la hacen inadecuado para adoptarlo a través de todas las máquinas virtuales. Por esta razón, las versiones futuras del lenguaje no van a soportar mas NMI, y no será cubierto aquí.

Actualmente, JNI esta diseñada para realizar una interfase con métodos nativos escritos solo en C o C++. Utilizando JNI, sus métodos nativos pueden:

- Crear, inspeccionar y actualizar objetos Java (incluyendo arreglos y **Strings**)
- Llamar a métodos Java
- Capturar y lanzar excepciones
- Cargar clases y obtener información de la clase
- Realizar verificaciones de tipo en tiempo de ejecución

De esta forma, virtualmente todo lo que se puede hacer con clases y objetos en un Java común lo puede hacer en métodos nativos.

## Llamando a un método nativo

Comenzaremos con un simple ejemplo: un programa Java que llame un método nativo, que de hecho llama a la función de la librería estándar de C **printf()**.

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");
        // Hackeo en Linux, si no se puede obtener la ruta de la
        // librería configure su ambiente:
        // System.load(
        //     "/home/bruce/tij2/appendixb/MsgImpl.so");
    }
    public static void main(String[] args) {
        ShowMessage app = new ShowMessage();
        app.ShowMessage("Generado con JNI");
    }
} ///:~
```

La declaración de los métodos nativos es seguido por un bloque **static** que llama a **System.loadLibrary()** (que puede llamar en cualquier momento, pero este estilo es mas apropiado). **System.loadLibrary()** carga una DLL en memoria y la enlaza con ella. La DLL debe estar en la ruta de la librería de sistema. La extensión del nombre es agregada automáticamente por la JVM dependiendo de la plataforma.

En el código mas arriba se puede ver una llamada a el método **System.load()**, que esta comentado. La ruta específica aquí es absoluta, en lugar de confiarse en una variable de ambiente. Utilizando una variable de

ambiente es naturalmente una mejor y mas portátil solución, pero si no se puede imaginar esa salida se puede comentar la llamada a **loadLibrary()** y quitar el comentario de esta, ajustando la ruta a su propio directorio.

## El generador de ficheros de cabecera: javah

Ahora compile su fichero fuente de Java y ejecuta **javah** en el fichero **.class** resultante, especificando el modificador **-jni** (esto es realizado automáticamente para nosotros en el makefile de la distribución del código fuente para este libro):

```
| javah -jni ShowMessage  
javah lee el fichero de clase Java y para cada declaración de método nativo genera un prototipo de función en un fichero de cabecera C o C++. He aquí la salida: el fichero fuente ShowMessage.h (editada ligeramente para 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 se puede ver por la directiva de preprocesador **#ifdef \_\_cplusplus**, este fichero puede ser compilado por un compilador C o C++. La primer directiva **#include** incluye **jni.h**, un fichero cabecera que, además de otras cosas, define los tipos que se pueden ver utilizando el resto del fichero.

**JNIEXPORT** y **JNICALL** son macros que se expanden para corresponder con directivas específicas de la plataforma. **JNIEnv**, **jobject** y **jstring** son definiciones de tipos de datos, que serán explicados brevemente.

# Convención de nombres y firmas de funciones

JNI impone una convención de nombres (llamada *name mangling* en los métodos nativos. Esto es importante, puesto que es parte del mecanismo por el cual la máquina virtual enlaza las llamadas Java a los métodos nativos. Básicamente, todos los métodos nativos comienzan con la palabra “Java”, seguida por el nombre de la clase en donde la declaración nativa Java aparece, seguida por el nombre del método Java. El infraguión es utilizado como separador. Si el método nativo de Java es sobrecargado, entonces la firma de la función es agregada a el nombre igualmente; se puede ver la firma nativa en los comentarios que preceden al prototipo. Por mas información acerca de las convenciones de nombres y las firmas de los métodos nativos, por favor refiérase a la documentación JNI.

## Implementando su DLL

En este punto, todo lo que se tiene que hacer es escribir el fichero de código C o C++ que incluye el fichero cabecera generado por **javad** e implementar los métodos nativos, luego compilar y generar la librería de enlace dinámico. Esta parte es dependiente de la plataforma. El siguiente código es compilado y enlazado in un fichero llamado **MSGImpl.dll** para Windows o **MsgImpl.so** para Unix/Linux (el paquete makefile con el listado de código contiene los comandos para hacer esto -esta disponible también en el CD ROM que viene con este libro, o libremente en [www.BruceEckel.com](http://www.BruceEckel.com)):

```
//: appendixb:MsgImpl.cpp
//# Probado con VC++ & BC++. La ruta a Include
//# debe ser ajustada para encontrar los ficheros
//# cabecera JNI headers. el makefile para este
for this chapter (in the
//# capitulo (en el codido fuente que se puede
//# bajar) por un ejemplo.
#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 argumentos que son pasados dentro del método nativo son la puerta de enlace para regresar dentro de Java. El primero, del tipo **JNIEnv**, contiene todos los ganchos que permiten regresar a la JVM (veremos esto en la siguiente sección). El segundo argumento tiene un significado diferente

dependiendo del tipo de método. Para métodos no estáticos como el ejemplo mas arriba, el segundo argumento es el equivalente del puntero “this” en C++ y similar a **this** en Java: es una referencia a el objeto que llama el método nativo. Para métodos estáticos, es una referencia a el objeto **Class** donde el método es implementado.

El argumento restante representa el objeto Java pasado dentro de la llamada al método nativo. Las primitivas también son pasadas de esta forma, pero ellas vienen por valor.

En las siguientes secciones explicaremos este código mirando las formas en las que se accede y controla la JVM desde adentro de un método nativo.

## Accediendo a funciones JNI: el argumento **JNIEnv**

Las funciones JNI son aquellas que el programador utiliza para interactuar con la JVM desde adentro de un método nativo. Como se puede ver en el ejemplo mas arriba, cada método nativo JNI recibe un argumento especial como primer parámetro: el argumento **JNIEnv**, que es un puntero a una estructura de datos JNI especial del tipo **JNIEnv\_**. Un elemento de la estructura de datos JNI es un puntero a una función JNI. Las funciones JNI pueden ser llamadas del método nativo por eliminación de referencias a estos punteros (es tan simple como suena). Cada JVM proporciona su propia implementación de funciones JNI, pero sus direcciones siempre serán desplazamientos predefinidos.

A través del argumento **JNIEnv**, el programador tiene acceso a un gran grupo de funciones. Estas funciones pueden ser agrupadas en las siguientes categorías:

- \* Obtención de información de versión
- \* Realización de operaciones de clases y objetos.
- \* Manejo de referencias locales o globales a objetos Java
- \* Acceso a campos de instancia y estáticos
- \* Llamada a métodos de instancia y estáticos
- \* Realización de operaciones con cadenas y arreglos.
- \* Generar y manejar excepciones Java

El número de funciones JNI es bastante largo y no será cubierto aquí. En lugar de eso, mostraremos los fundamentos detrás del uso de estas

funciones. Por información mas detallada, se debe consultar la documentación del compilador JNI.

Si se observa el fichero cabecera **jni.h**, se puede ver dentro del condicional de procesador **#ifdef \_\_cplusplus**, la estructura **JNIEnv**: es definida como una clase cuando es compilada por un compilador C++. Esta clase contiene una gran cantidad de funciones en línea que nos dejan acceder a las funciones JNI con una sintaxis fácil y familiar. Por ejemplo, la línea de C++ en el ejemplo anterior:

```
| env->ReleaseStringUTFChars(jMsg, msg);  
puede también ser llamada desde C de esta forma:  
| (*env)->ReleaseStringUTFChars(env, jMsg, msg);  
Se notará que el estilo C es (naturalmente) mas compilado -se necesita una  
eliminación de referencia del puntero env, y se debe también pasar el mismo  
puntero como el primer parámetro a la llamada de función JNI. El ejemplo  
en el apéndice utiliza el estilo de C++.
```

## Accediendo a cadenas Java

Como un ejemplo de acceso a una función JNI, considere el código en **MsgImpl.cpp**. Ahí, el argumento **JNIEnv env** es utilizado para acceder a un **String** de Java. Los **Strings** de Java están en formato Unicode, así es que si se recibe uno y se quiere pasarlo a una función que no es Unicode (**printf()**, por ejemplo), se debe primero convertir a caracteres ASCII con la función JNI **GetStringUTFChars()**. Esta función toma un **String** Java y lo convierte a caracteres UTF-8 (Estos son 8 bits para almacenar valores ASCII o 16 bits para almacenar Unicode. Si el contexto de la cadena original fue compuesto solo de ASCII, la cadena resultante será ASCII también).

**GetStringUTFChars()** es uno de las funciones miembro en **JNIEnv**. Para acceder a la función JNI, utilizamos la sintaxis típica de C++ para llamar a funciones miembro a través de un puntero. Se utiliza la forma mas arriba para acceder a la totalidad de las funciones JNI.

## Pasando y utilizando objetos Java

En el ejemplo anterior pasamos un **String** a un método nativo. Se puede también pasar objetos Java de nuestra propia creación a un método nativo. Dentro del método nativo, se puede acceder a los campos y métodos del objeto que fue recibido.

Para pasar objetos, se utiliza la sintaxis común de Java cuando se declara un método nativo. En el siguiente ejemplo, **MyJavaClass** tiene un campo público y un método **publico**. La clase **UseObjects** declara un método nativo que toma un objeto de la clase **MyJavaClass**. Para ver si el método nativo manipula su argumento, el campo **public** del argumento es configurado, el método nativo es llamado, y el valor del campo **public** es impreso.

```

class MyJavaClass {
    public int aValue;
    public void divByTwo() { aValue /= 2; }
}
public class UseObjects {
    private native void
        changeObject(MyJavaClass obj);
    static {
        System.loadLibrary("UseObjImpl");
        // Hackeo de Linux, si no se pudeo obtener la ruta
        // de la librería configurada en su ambiente:
        // 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);
    }
} //:~

```

Luego de compilar el código y ejecutar **javah**, se puede implementar el método nativo. En el ejemplo siguiente, una vez que los ID de campo y método son obtenidos, son accedidos a través de las funciones JNI.

```

//: appendixb:UseObjImpl.cpp
/// Probado con VC++ & BC++. La ruta e el include
/// debe ser ajustada para encontrar los cabezales JNI. Vea
/// el makefile para este capítulo (en el código que se
/// pude bajar) por un ejemplo.
#include <jni.h>
extern "C" 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 **object**, que es el lado nativo de la referencia a el objeto Java que pasamos del código Java. Simplemente leemos **aValue**, lo imprimimos, cambiamos el valor, llamamos a el método de objeto **divByTwo()**, e imprimimos el valor nuevamente.

Para acceder a un campo o método de Java, se debe primero obtener su identificador utilizando **GetFieldID()** para campos y **GetMethodID()** para métodos. Estas funciones toman el objeto clase, y la cadena conteniendo el nombre del elemento, y una cadena que le da la información de tipo: el tipo de dato del campo, o la información de firma para un método (detalles se pueden encontrar en la documentación JNI). Estas funciones retornan un identificador que se utiliza para acceder al elemento. Esta estrategia puede parecer compleja, pero su método nativo no tienen conocimiento de la capa interna del objeto Java. En lugar de eso, se debe acceder a los campos y métodos a través de índices retornados por la JVM. Esto permite diferentes JVMs implementar diferentes capas de objetos sin impactar en los métodos nativos.

Si se ejecuta un programa Java, se verá que el objeto que es pasado del lado de Java es manipulado por su método nativo. ¿Pero que exactamente es pasado? ¿Un puntero o una referencia Java? ¿Y que es lo que el recolector de basura hace durante una llamada a un método nativo?

El recolector de basura continua operando durante la ejecución de un método nativo, pero es garantido que sus objetos no serán recolectados durante la llamada a el método nativo. Para asegurarnos de esto, las *referencias locales* son creadas antes, y destruidas exactamente después de la llamada a el método nativo. Dado que su tiempo de vida envuelve la llamada, se debe conocer que estos objetos serán válidos a través de la llamada a los métodos nativos.

Dada que estas referencias son creadas y subsecuentemente destruidas cada vez que la función es llamada, no se pueden hacer copias locales en el los métodos nativos, en variables **statics**. Si se quiere una referencia que perdure a través de las invocaciones, se necesita una referencia global. Las referencias globales no son creadas por la JVM, pero el programador puede hacer una referencia global fuera de la local llamando a funciones específicas JNI. Cuando se crea una referencia global, nos hacemos responsables por el tiempo de vida del objeto referenciado. La referencia local (y el objeto al que se refiere) estará en memoria hasta que el programador explícitamente libere la referencia con la función JNI apropiada. Es similar a **malloc()** y **free()** en C.

# JNI y excepciones Java

Con JNI, las excepciones Java pueden ser lanzadas, capturadas, impresas y lanzadas nuevamente como se hace dentro de un programa Java. Pero le toca al programador llamar las funciones JNI dedicadas para tratar con excepciones. Aquí están las funciones JNI para manejo de excepciones:

- \* **Throw()**  
Lanza un objeto de excepción existente. Utilizado en métodos nativos para lanzar nuevamente una excepción.
- \* **ThrowNew()**  
Genera un nuevo objeto de excepción y lo lanza.
- \* **ExceptionOccurred()**  
Determina si una excepción fue lanzada y no ha sido limpiada.
- \* **ExceptionDescribe()**  
Imprime una excepción y la traza de la pila.
- \* **ExceptionClear()**  
Limpia una excepción pendiente.
- \* **FatalError()**  
Se alcanzó un error fatal. No retorna.

En medio de estas, no se puede ignorar **ExceptionOccurred()** y **ExceptionClear()**. La mayoría de las funciones JNI pueden generar excepciones, y no hay ninguna característica del lenguaje que pueda utilizar en lugar de un bloque try de Java, así es que se debe llamar a **ExceptionOccurred()** luego de cada llamada a función JNI para ver si una excepción fue lanzada. Si se detecta una excepción, se debe elegir si manejárla (y posiblemente relanzarla). Hay que asegurarse, sin embargo, de que la excepción es eventualmente limpia. Esto se puede realizar en la función utilizando **ExceptionClear()** o alguna otra función si la excepción es lanzada nuevamente, pero debe ser realizado.

Hay que asegurarse que la excepción es limpia, porque de otra forma los resultados serán impredecibles si se llama a una función JNI cuando una excepción está pendiente. Hay unas pocas funciones JNI que son seguras a llamar durante una excepción; en medio de esto, claro, son todas las funciones de manejo de funciones.

# JNI e hilado

Dado que Java es un lenguaje de hilado múltiple, muchos hilos pueden llamar a un método nativo concurrentemente (los métodos nativos pueden ser suspendidos en el medio de sus operaciones cuando un segundo hilo lo llama). Esto es enteramente responsabilidad del programador garantizar que la llamada nativa es segura en el hilado; i.e., no modifica datos compartidos sin realizar una monitorización. Básicamente, se tienen dos opciones: declarar el método nativo como **synchronized**, o implementar alguna otra estrategia dentro del método nativo para asegurar correctamente, la manipulación de datos concurrente.

También, no se debería nunca pasar un puntero **JNIEnv** a través de los hilos, dado que la estructura interna apunta a su localización a través de un hilo y contienen información que tiene sentido solo en un hilo particular.

## Utilizando un código preexistente base

La forma mas fácil para implementar métodos nativos JNI es comenzar escribiendo prototipos de métodos nativos en clases Java, compilando esa clase, y ejecutando el fichero **.class** a través de **javah**. ¿Pero que si se tiene un largo, preexistente código base que quiere llamar desde Java? Volver a nombrar todas las funciones en sus DLLs para que correspondan con las convenciones de nombres no es una solución viable. La mejor estrategia es escribir una DLL envoltura “fuera” de su código base. El código Java llama a funciones en esta nueva DLL, que convierte las llamadas a las funciones DLL originales. Esta solución no es simplemente un trabajo a causa de esto, en muchos casos se debe hacer esto de todas formas porque se debe llamar funciones JNI por la referencia al objeto antes de que pueda utilizarlo.

## Información adicional

Se pueden encontrar material introductorio suplementario, incluyendo un ejemplo C (en lugar de C++) y una discusión de temas Microsoft, en el Apéndice A de la primera edición de este libro, que puede ser encontrada en el CD ROM que viene con este libro, o bajarlo libremente de [www.BruceEckel.com](http://www.BruceEckel.com) Información mas extensa está disponible en [java.sun.com](http://java.sun.com) (en el motor de búsqueda, seleccione “training & tutorial” por palabras claves “native methods”). El capítulo 11 de Core Java 2, Volumen

II, de Horstmann & Cornell (Prentice-Hall, 2000) entrega una cobertura excelente de métodos nativos.

## C: Líneas guía para programar en Java

Este apéndice contiene sugerencias que ayuda para guiarnos a realizar un diseño de programa de bajo nivel, y escribiendo código.

Naturalmente, estas son líneas guía y no reglas. La idea es utilizarlos como inspiración, y para recordar que hay situaciones ocasionales donde se necesita girar o quebrar una regla.

### Diseño

1. **La elegancia siempre paga.** A corto plazo puede parecer como que toma mucho mas tiempo comenzar con una solución a un problema que tenga verdadera gracia, pero cuando trabaja la primera vez y fácilmente se adapta a situaciones en lugar de requerir horas, días o meses de lucha, se verá la recompensa (aún si no se puede cuantificar). No solo entrega un programa que es fácil de armar y depurar, también será fácil de entender y mantener, y esto es donde los valores financieros se encuentran. Este punto puede tomar algo de experiencia para entender, dado que puede parecer que no se está siendo productivo mientras se está haciendo parte de código elegante. Hay que resistir el impulso de apurarse; este solo hará que nos atrasemos.
2. **Primero haga el trabajo, luego hágalo rápido.** Esto es verdad aún si se está seguro de que una pieza de código es realmente importante y será un cuello de botella principal en su diseño. No lo haga. Obtengamos el sistema funcionando primero con un diseño tan simple como se pueda. Entonces si no es lo suficientemente rápido, profilelo. Al menos siempre descubrirá que “su” cuello de botella no es el problema. Ahorre el tiempo para las cosas realmente importante.
3. **Recuerde el principio “divida y conquistará”.** Si el problema que está trabajando es muy confuso, trate de imaginar cuales serán las operaciones básicas del programa, dando la existencia de una pieza “mágica” que maneja las partes complicadas. Esta “pieza” es un objeto -escriba el código que utiliza el objeto, luego observe en el objeto y encapsule *su* parte complicada en otros objetos, etc.

4. **Separé la clase creadora de la clase usuario (*cliente programador*)**. La clase usuario es el “cliente” y no se necesita o se quiere saber que esta sucediendo detrás de las escenas de la clase. La clase creador debe ser la experta en el diseño de clases y escribir la clase de tal forma que pueda ser utilizado por la mayor parte de los programadores principiantes posible, manteniendo robusta la aplicación. El uso de la librería será fácil solo si es transparente.
5. **Cuando se crea una clase, intente hacer sus nombres tan claros que los comentarios son innecesarios**. Su meta debe ser hacer la interfase de cliente programador conceptualmente simple. Para llegar a esto, use sobrecarga de métodos cuando sea apropiado para crear una interfase intuitiva y fácil de usar.
6. **El análisis y diseño debe producir, al menos, las clases en el sistema, sus interfaces públicas, y las relaciones con otras clases, especialmente la clase base**. Si su diseño metodológicamente mas que esto, preguntémonos si todas las partes producidas por esa metodología tienen valor en la vida del programa. Si no lo hacen, mantenerlas nos costará. Los miembros del equipo de desarrollo tienden a no mantener nada que no contribuya a su productividad; esto es hecho de la vida que muchos diseñadores de métodos no se explican.
7. **Automaticice todo**. Escriba el código de prueba primero (antes de escribir la clase), y manténgalo con la clase. Automaticice la ejecución de sus pruebas a través de una herramienta makefile o similar. De esta forma, cualquier cambio puede ser automáticamente verificado ejecutando el código de prueba, y se descubrirán inmediatamente los errores. Dado que se conoces lo que se tiene la red de seguridad de su marco de trabajo de prueba, se será atrevido acerca de hacer rápidos cambios cuando se descubran las necesidades. Recuerde que las mejoras grandes en lenguaje vienen con las pruebas proporcionadas por el chequeo de tipo, manejo de excepciones, etc., pero esas características solo uno las puede aprovechar. Debe ir el resto del camino en la creación de un sistema robusto cumpliendo con las pruebas que verifican las características que son especificadas por su clase o programa.
8. **Escriba el código de prueba primero (antes de escribir la clase) para verificar que el diseño de su clase esta completo**. Si no puede escribir un código de prueba, no se sabe como se ve su clase. Además, el acto de escribir el código de prueba será a menudo limpiará las características adicionales o limitaciones que se necesitan en la clases -estas características o limitaciones no siempre aparecen durante el análisis y diseño. La prueba también proporciona un código de ejemplo que muestra como su clase puede ser utilizada.

9. **Todos los problemas de diseño pueden ser simplificados introduciendo un nivel extra de oblicuidad conceptual.** Esto regla fundamental de ingeniería de software<sup>1</sup> es básica para la abstracción, la característica primaria de la programación orientada a objetos.
10. **Una oblicuidad debería tener un significado** (junto con la guía 9). Este significado puede ser algo tan simple como “colocar código utilizado comúnmente en un solo método”. Si se agregan niveles de oblicuidad (abstracción, encapsulación, etc.) que no tienen significado, puede ser tan malo como no tener una oblicuidad adecuada.
11. **Hacer las clases tan atómicas como sea posible.** Darle a cada clase un único propósito claro. Si las clases en el diseño de un sistema crecen muy complicadas, hay que dividirlas en otras más simples. El indicador más obvio de esto es un tamaño escarpado: si una clase es grande, la posibilidad es que esté haciendo demasiado y deberá ser dividida.

Las pistas para sugerir un rediseño de una clase son:

  - 1) Una instrucción switch complicada: considere utilizar polimorfismo.
  - 2) Una gran cantidad de métodos que cubren ampliamente diferentes tipos de operaciones: considere utilizar varias clases.
  - 3) Una gran cantidad de variables miembro que ataúnen ampliamente diferentes características; considerando utilizar muchas clases.
12. **Vigile las largas listas de argumentos.** Las llamadas a los métodos entonces se vuelven difíciles de escribir, leer y mantener. En lugar de esto, trate de mover los métodos a una clase donde sea (mas) apropiado, y/o pasar objetos como argumentos.
13. **No hay que ser repetitivo.** Si una parte de código es recurrente en muchos métodos en clases derivadas, hay que colocar ese código en un solo método en la clase base y llamarlo desde los métodos de la clase derivada. No solo está ahorrando espacio de código, se proporciona una fácil propagación de cambios. A veces descubrir este código común agregara una funcionalidad muy de mucho valor en su interfase.
14. **Tenga cuidado de las instrucciones switch o cláusulas if-else encadenados.** Esto es típicamente un indicador de codificación de *validación de tipo* lo que significa que se está eligiendo que código ejecutar basado en algún tipo de información (el tipo exacto puede no ser obvio al inicio). Se puede usualmente reemplazar este tipo de

---

<sup>1</sup> Me lo explico Andrew Joenig

código con herencia y polimorfismo; una llamada a un método polimórfico puede realizar la validación de tipo por nosotros, y permitir mas confiable y fácil extensibilidad.

15. **De un punto de diseño del diseño, hay que encontrar y separar cosas que cambian de las cosas que se quedan iguales.** Esto es, buscar elementos en un sistema que puede querer cambiar sin forzar la creación de un nuevo diseño, luego se encapsulan estos elementos en clases. Se puede aprender significativamente mas acerca de este concepto en *Pensando en Patrones con Java* que se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com)
16. **No extienda funcionalidades fundamentales con una subclase.** Si un elemento de la interfase es esencial para una clase debería estar en la clase base, no agregada mediante derivación. Si se esta agregando métodos por herencia, tal vez se debería pensar nuevamente el diseño.
17. **Menos es mas.** Comience con una interfase mínima para una clase, tan pequeña y simple como se necesite para solucionar el problema a mano, pero no hay que tratar de anticipar todas las formas en que la clase *puede* ser utilizada. Cuando la clase sea utilizada, se descubrirán formas que se debe expandir la interfase.

Sin embargo, una vez que la clase está en uso no se puede reducir el tamaño de la interfase sin trastornar el código del cliente, lo que fuerza una nueva compilación. Pero aún si los nuevos métodos reemplazan la funcionalidad de los viejos, se deja la interfase quieta (se puede combinar la funcionalidad en la implementación de las capas mas bajas si se quiere). Si se necesita expandir la interfase de un método existente agregando mas argumentos, cree un método sobrecargado con los nuevos argumentos; de esta forma no se necesita trastornar ninguna llamada a el método existente.
18. **Lea sus clases en vos alta para asegurarse de que son lógicas.** Refiriéndose a la relación entre una clase y la clase derivada como “es una” y a los objetos miembro como “tiene una”.
19. **Cuando estás decidiendo entre herencia y composición, pregúntese si necesita realizar una conversión ascendente a el tipo base.** Si no, prefiera la composición (objetos miembro) a la herencia. Esto puede eliminar la necesidad percibida de múltiples tipos base. Si se hereda, los usuarios pueden pensar que supuestamente tienen que realizar una conversión ascendente.
20. **Use datos miembros para variaciones en valores y sobrecarga de métodos para variaciones en el comportamiento.** Esto es, si encontramos una clase que utiliza variables de estado junto con métodos que cambian el comportamiento basados en estas variables,

probablemente deberíamos volver a diseñarlo para expresar las diferencias en comportamiento dentro de subclases y métodos sobrecargados.

21. **Tenga cuidado de las sobrecargas.** Un método puede no ejecutar código condicionalmente basado en el valor de un argumento. En este caso, debería crear dos o mas métodos sobrecargados en su lugar.
22. **Use jerarquías de excepciones** -preferiblemente derivadas de clases específicas apropiadas en la jerarquía de excepciones estándar de Java. La persona que captura las excepciones puede entonces capturar los tipos específicos de excepciones, seguidos por el tipo base. Si se agrega una nueva excepción derivada, el código existente del cliente seguirá capturando la excepción del tipo base.
23. **A veces la agregación simple hace el trabajo.** un “sistema de pasajero confortable” en una aerolínea consiste en elementos desconectados: asientos, aire acondicionado, video, etc., y aún se necesita crear muchos de estos en un avión. ¿Se hacen miembros privados y se crea una interfase totalmente nueva? No -en este caso, los componentes también son parte de la interfase pública, así es que se deben crear objetos miembros públicos. Aquellos objetos que tienen sus propias implementaciones privadas, que siguen siendo seguros. Hay que ser cuidados de que la agregación simple no es una solución para ser utilizada a menudos, pero sucede.
24. **Considere la perspectiva del cliente programador y de la persona que mantiene el código.** Se debe diseñar la clase para que sea tan obvia para usar como sea posible. Hay que anticiparse a los tipos de cambios que serán hechos, y diseñar la clase para que esos cambios sean fáciles.
25. **Tenga cuidado del “síndrome del objeto gigante”.** Esto es a menudo una afición de los programadores procesales que son nuevos en la POO y terminan escribiendo programas procesales y se afirman dentro de uno o dos objetos gigantes. Con la excepción de los marcos de trabajo de aplicación, los objetos representan conceptos en su aplicación, no la aplicación.
26. **Si se debe hacer algo poco agraciado, al menos localice la fealdad dentro de una clase.**
27. **Si se debe hacer algo que no es portátil, se realiza una abstracción para ese servicio y se localiza dentro de una clase.** Este nivel extra de oblicuidad previene que un programa no sea portátil al ser distribuido (este idioma es personificado en el patrón *Bridge*).
28. **Los objetos no deben simplemente almacenar algún dato.** También deben tener comportamientos bien definidos (Ocasionalmente,

“objetos datos” son apropiados, pero solo cuando son utilizadas para empaquetar expresamente un grupo de ítems cuando un contenedor generalizado es inapropiado).

29. **Se debe elegir la composición primero cuando creamos nuevas clases de clases existentes.** Se debe utilizar herencia solo si es requerido por su diseño. Si se usa herencia donde la composición puede trabajar, sus diseños se convertirán en innecesariamente complicados.
30. **Use herencia y sobrecarga de métodos para expresar diferencias en comportamiento, y campos para expresar variaciones en estados.** Un ejemplo extremo de que no hacer es heredar diferentes clases para representar colores en lugar de usar un campo “color”.
31. **Tenga cuidado de los *variación*.** Dos objetos semánticamente diferentes pueden tener acciones o responsabilidades idénticas, y es una tentación natural tratar de crear una subclase de la otra solo para beneficiarse de la herencia. Esto es llamado variación, pero no es una justificación real forzar una relación entre una clase super y una subclase donde no existe. Una mejor solución es crear una clase base general que produzca una interfase para ambas clases derivadas -esto requiere un poco mas de espacio, pero nos seguimos beneficiando de la herencia, y probablemente haremos un descubrimiento importante en el diseño.
32. **Tenga cuidado de las *limitaciones* durante la herencia.** Los diseños claros agregan nuevas habilidades a las clases heredadas. Un diseño sospechoso quita viejas habilidades durante la herencia sin agregar nuevas. Pero las reglas son hechas para ser quebradas, y si se esta trabajando con una librería de clases vieja, puede ser mas eficiente restringir una clase existente es su subclase que reestructurar toda la jerarquía para que su clase encaje en donde debería, encima de la vieja clase.
33. **Se debe utilizar patrones de diseño para eliminar la “funcionalidad abierta”.** Esto es, si solo un objeto de su clase debe ser creado, no se cierre a la aplicación y escriba un comentario “Hacer solo uno de estos”. Envuelva en un singleton. Si se tiene un montón de código desordenado en su programa principal que crea sus objetos, encuentre un patrón de creación como una método de fábrica en donde se pueda encapsular esa creación. Eliminando la “funcionalidad abierta” no solo hará su código mas fácil de entender y mantener, también lo hará mas antibalas contra los bien intencionados encargados de mantenimiento que vengan después de usted.
34. **Tenga cuidado de la “parálisis de análisis”.** Recuerde que se debe usualmente adelantar en un proyecto antes de conocer todo, y que a menudo la mejor y mas rápida manera de aprender acerca de alguno

de sus factores desconocido es ir al siguiente paso en lugar de tratar de imaginarlo en la cabeza. No se puede conocer la solución hasta que se *tenga* la solución. Java tiene incorporados firewalls; deje que ellos trabajen para nosotros. Los errores en una clase o grupo de clases no destruyen la integridad del sistema completo.

35. **Cuando se piense que tiene un buen análisis, diseño o implementación, haga un ensayo final.** Traer a alguien de afuera de su grupo -este no tiene que ser un consultor, puede ser alguien de otro grupo dentro de su empresa. Viendo nuevamente su trabajo nuevamente con un par de ojos frescos puede revelar problemas en una etapa en que sea mucho mas fácil de corregir, y además paga por el tiempo y el dinero “perdido” en el proceso de revisión.

## Implementación

36. **En general, seguir las convención de codificación de Sun.** Estas están disponibles en [java.sun.com/doc/codeconv/index.html](http://java.sun.com/doc/codeconv/index.html)(el código en este libro sigue estas convenciones y todo lo que he podido ser capaz). Estas son utilizadas por lo que constituye dudosamente el cuerpo del código mas largo por el mayor número de programadores Java al que será expuesto. Si se adhiere tenazmente a el estilo de codificación que siempre utilizó, lo hará mas difícil para la persona que lee. Cualquiera sea el estilo de codificación que uno decida utilizar, hay que asegurarse que es consistente a lo largo de todo el proyecto. Hay una herramienta para formatear el código automáticamente en [home.wtal.de/software/solutions/jindent](http://home.wtal.de/software/solutions/jindent).
37. **Cualquiera sea el estilo de codificación que se utilice, realmente hace la diferencia en su equipo (y aún mejor, su empresa) estandarizarlo.** Esto significa el punto en que cada uno lo considera un juego justo el corregir la codificación de alguien mas si no conforma. El valor de la estandarización es que se toma menos ciclos de cerebro para analizar el código, y de esta forma enfocarse mas en lo que el código significa.
38. **Seguir las reglas de estandarización de las mayúsculas y minúsculas.** Se deben de colocar la primer letra de los nombres de clases en mayúsculas. La primer letra de los campos, métodos, y objetos (referencias) deben estar en minúsculas. Todos los identificadores deben tener las palabras juntas, y llevar mayúscula la primer letra de todas las palabras en el medio. Por ejemplo:

**ThisIsAClassName**

**thisIsAMethodOrFieldName**

Se debe de llevar *todas* las letras de identificadores de primitivas **static final** que tienen inicializadores constantes en sus definiciones a mayúsculas. Esto indica que son constantes en tiempo de compilación.

**Los paquetes son un caso especial**-estos tienen letras minúsculas, aún para las palabras intermediarias. La extensión de dominio (com, ort, net, edu, etc.) deben estar en minúsculas (Esto fue un cambio entre Java 1.1 y Java 2.).

39. **No cree sus propias decoraciones de nombres de miembros de datos privados.** Esto es usualmente visto en la forma de infraguiones y caracteres agregados. La notación húngara es el peor ejemplo de esto, donde se agregan caracteres extra para indicar el tipo de dato, uso, localización, etc., como si estuviera escribiendo lenguaje ensamblador y el compilador no proporciona asistencia extra después de todo. Estas notaciones son confusas, difíciles de leer, y desagradables al implementar y mantener. Se debe dejar las clases y paquetes hacer el alcance de nombres por nosotros.
40. **Se debe seguir una “forma canónica”** cuando se cree una clase de propósito general. Incluyendo las definiciones para **equals()**, **hashCode()**, **toString()**, **clone()** (implementar **Cloneable**), e implementar **Comparable** y **Serializable**.
41. **Se debe utilizar las convenciones de nombres de JavaBeans “get”, “set”, y “is”** para métodos que lean y cambien campos **private**, aún si no se tiene pensado hacer un JavaBean en el momento. No solo hace mas fácil de utilizar la clase como un Bean, pero es una forma estándar de nombrar este tipo de métodos y será mas fácil de entender por el lector.
42. **Por cada clase que se cree, considere incluir un **static public test()** que contenga el código para probar esa clase.** No se necesita quitar el código de prueba para utilizar la clase en un proyecto, y si se quiere hacer algunos cambios se puede fácilmente ejecutar las pruebas. Este código también proporciona ejemplos de como utilizar la clase.
43. **A veces se necesita heredar para acceder a miembros protegidos de la clase base.** Esto puede liderar la percepción de una necesidad de múltiples tipos base. Si no necesita una conversión ascendente, primero derive la clase nueva para realizar el acceso protegido. Luego haga que es nueva clase se objeto miembro dentro de cualquier clase que se necesite utilizar, en lugar de heredar.
44. **Evite el uso de métodos finales con propósitos de eficiencia.** Utilice **final** solo cuando el programa se esta ejecutando, pero no con la suficiente velocidad, y su análisis muestre que una invocación a un método es el cuello de botella.

45. **Si dos clases están asociadas entre ellas en alguna tipo de forma funcional (como lo son contenedores e iteradores), trate de hacer que una de ellas sea interna a la otra.** Esto no solo enfatiza la asociación entre las clases, también permite que el nombre de clase sea reutilizado dentro de un solo paquete anidándola dentro de otra clase. La librería de contenedores de Java hace esto definiendo un **Iterator** interno dentro de cada clase contenedora, consecuentemente proporcionan los contenedores con una interfase común. La otra razón se quiere utilizar una clase interna es parte de una implementación oculta en lugar de que la asociación de clase y la prevención de contaminación de espacios de nombres notada mas arriba.
46. **En cualquier momento se pueden notar clases que parecen tener un alto acoplamiento entre ellas, considere que la codificación y las mejoras de rendimiento que puede obtener utilizando clases internas.** El uso de clases internas no desacoplará las clases, pero en lugar de eso hará el acoplamiento mas explícito y mas conveniente.
47. **No se debe caer presa de la optimización prematura.** Esta forma nos sitúa en enajenación. En particular no se preocupe acerca de escribir (o evitar) métodos nativos, haciendo algunos métodos **final**, o código retorcido por un tema de eficiencia cuando se esta construyendo en un comienzo el sistema. La meta primaria debe ser probar el diseño, a no ser que el diseño requiera cierta eficiencia.
48. **Se debe mantener los alcances lo mas pequeño que sea posible así la visibilidad y el tiempo de vida de los objetos son lo mas pequeños posible.** Esto reduce la posibilidad de utilizar un objeto en el contexto equivocado y oculta la dificultad de encontrar errores. Por ejemplo, supongamos que se tiene un contenedor y una pieza de código que interactúan a través de esta. Si se copia el código para utilizar con un nuevo contenedor, se puede accidentalmente terminar utilizando el tamaño del viejo contenedor como el límite superior del nuevo. Si, sin embargo, el viejo contenedor esta fuera de alcance, el error será capturado en tiempo de compilación.
49. **Se deben utilizar los contenedores de la librería estándar de Java.** Convertirse en competente con su uso y veremos incrementar nuestra productividad enormemente. Prefiera **ArrayList** por secuencias, **HashSet** para grupos, **HashMap** para arreglos asociativos, y **LinkedList** para pilas (en lugar de **Stack**) y colas.
50. **Para que un programa sea robusto, cada componente debe ser robusto.** Use todas las herramientas proporcionadas por Java: control de acceso, excepciones, verificación de tipo, y demás, en cada clase que

se cree. De esta forma se puede mover de forma segura a el siguiente nivel de abstracción cuando se este armando un sistema.

51. **Se deben preferir los errores en tiempo de compilación que los errores en tiempo de ejecución.** Se debe tratar de manejar un error tan cerca del punto donde se sucede como sea posible. Se debe preferir tratar con el error en ese punto donde lanza una excepción. Capturar alguna excepción en el manejador mas cercano que tiene suficiente información para tratar con ella. Se debe hacer lo que se pueda con la excepción en el nivel actual; si no soluciona el problema, lance nuevamente la excepción.
52. **Se debe tener cuidado con las definiciones de métodos largas.** Los métodos debes ser breves, unidades funcionales que describen e implementan un parte discreta de la interfase de una clase. Un método que es largo y complicado es difícil y caro de mantener, y probablemente trate de hacer mucho por si solo. Si vemos un método así, este indica que, al menos este debe ser quebrado en múltiples métodos. Puede también sugerir la creación de una nueva clase. Pequeños métodos también fomentarán la reutilización dentro de la clase (A veces los métodos deben ser largos, pero estos seguirán haciendo solo una cosa).
53. **Se deben mantener las cosas tan “privadas como sea posible”.** Una vez que se publique un aspecto de una librería (un método una clase, un campo), no se puede sacar. Si se hace, arruinará el código existente de alguien mas, forzándolo a escribirlo y diseñarlo nuevamente. Si se publicita solo lo que debe, se puede cambiar todo lo demás sin impunidad, y dado que los diseños tienden a evolucionar esto es una libertad importante. De esta forma, los cambios en la implementación tendrán un mínimo impacto en las clases derivadas. La privacidad es especialmente importante cuando se trata con el hilado múltiple -solo los campos **private** pueden ser protegidos contra el uso no **synchronized**.
54. **Use los comentarios liberalmente, y use la sintaxis de documentación de javadoc para producir la documentación de su programa.** Sin embargo, los comentarios pueden agregar significado genuino a el código; los comentarios que solo reiteran lo que el código claramente expresa molestos. Note que el detalle típicamente verbal de las clases Java y los nombres de métodos reducen la necesidad de mas comentarios.
55. **Se debe evitar la utilización de “números mágicos”**-que son números colocados ampliamente en el código. Esos son una pesadilla si se necesita cambiarlos, dado que nunca se sabe si “100” significa “el tamaño del arreglo” o “algo en su totalidad”. En lugar de eso, se debe

crear una constante con un nombre descriptivo y usar los identificadores de constantes a través del programa. Esto hace el programa mas fácil de entender y mucho mas fácil de mantener.

56. **Cuando se creen constructores, se deben de considerar las excepciones.** En el mejor de los casos, el constructor no hará nada que lance una excepción. En el siguiente escenario, la clase debe estar compuesta y heredada de clases robustas solamente, así es que estas no necesitan limpieza si una excepción es lanzada. De otra forma, se debe limpiar clases compuestas dentro de una cláusula **final**. Si un constructor debe fallar, la acción apropiada es lanzar una excepción, así es que el que llama no continúa a ciegas, pensando que el objeto fue creado correctamente.
57. **Si una clase requiere alguna limpieza cuando el cliente programador termina con un objeto, coloque el código de limpieza en un solo método bien definido** -con un nombre como **cleanup()** que claramente sugiera su propósito. Además, se debe colocar una bandera **boolean** en la clase para indicar cuando un objeto ha sido limpiado de tal forma que **finalize()** pueda verificar por la “condición de muerto” (Vea el capítulo 4).
58. **La responsabilidad de un finalize() puede solos ser verificar la “condición de muerto” de un objeto con motivos de depuración** (Vea el capítulo 4). En casos especiales, puede ser necesario liberar memoria que de otra forma no sería liberada por el recolector de basura. Dado que el recolector de basura puede no ser llamado por un objeto, no se puede utilizar **finalize()** para realizar la limpieza necesaria. En el método **finalize()** para la clase, se debe verificar para estar seguro que el objeto ha sido limpiado y lanzar una clase derivada de **RuntimeException** si no lo ha hecho, para indicar un error de programación. Antes de confiar en este esquema, hay que asegurarse de que **finalize()** trabaja en el sistema (Puede que se necesite llamar a **System.gc()** para asegurar ese comportamiento).
59. **Si un objeto debe ser limpiado (por otro que el recolector de basura) dentro de un alcance en particular, se debe utilizar la siguiente estrategia:** Se debe inicializar el objeto y, si es exitoso, inmediatamente entrar a un bloque **try** con una cláusula **finally** que realice la limpieza.
60. **Cuando se sobrecarga finalize() durante la herencia, se debe recordar llamar a super.finalize().** (Esto no es necesario si **Object** es la superclase inmediata). Se debería llamar a **super.finalize()** como acto final de su **finalize()** sobrecargado en lugar que lo primero, para asegurarse que los componentes de la clase base siguen siendo válidos cuando se necesitan.

61. **Cuando se está creando un contenedor de tamaño fijo de objetos, se deberían de transferir a un arreglo**-especialmente si se está retornando este contenedor desde un método. De esta forma se obtiene el beneficio del chequeo de tipo del arreglo en tiempo de compilación, y el recipiente del arreglo puede no necesitar una conversión de objetos en el arreglo para utilizarlos. Debe notarse que la clase base de la librería de contenedores, **java.util.Collection**, tiene dos métodos **toArray()** para lograr esto.
62. **Se deben elegir interfaces sobre clases abstractas.** Si se sabe que algo va a ser una clase base, la primera elección debe ser hacerla una **interface**, y solo si se está forzado a tener una definición de métodos o variables miembro se debe cambiar a una clase **abstract**. Una **interface** habla acerca de lo que el cliente quiere hacer, mientras que una clase tiende a enfocarse en (o permitir) detalles de implementación.
63. **Dentro de los constructores, solo se debe hacer lo que es necesario para configurar el objeto en un estado adecuado.** Activamente se debe elegir llamar a otros métodos (excepto para métodos **final**) dado que estos métodos pueden ser sobrescritos por algún otro para producir resultados inesperados durante su construcción (Vea el capítulo 7 por detalles). Pequeños, simples constructores son menos propensos a lanzar excepciones o causar problemas.
64. **Para evitar experiencias altamente frustrantes, se debe asegurar que hay solo una clase desempaquetada por cada nombre en cualquier parte de su ruta de clases.** De otra forma, el compilador puede encontrar primero otra clase idénticamente nombrada, y reportar mensajes de error que no tengan sentido. Si se sospecha de que se está teniendo un problema de ruta de clases, se debe tratar de localizar los ficheros **.class** con los mismos nombres en cada uno de los puntos de partida de la ruta de clases. Idealmente, coloque todas las clases dentro de los paquetes.
65. **Se debe tener cuidado de sobrecargas accidentales.** Si se intenta sobrescribir un método de una clase base y no se tiene considerablemente bien su ortografía, se terminará agregando un nuevo método en lugar de sobrescribir un método existente. Sin embargo, esto es perfectamente legal, así es que no se obtendrá ningún mensaje de error del compilador o del sistema en tiempo de ejecución -el código simplemente no funcionará correctamente.
66. **Se debe tener cuidado de la optimización prematura.** Primero se debe hacer el trabajo, luego se debe hacerlo rápido -pero solo si se debe, y solo si se prueba que es un cuello de botella en una sección particular del código. A no ser que se tenga un análisis para descubrir un cuello de botella, probablemente se estará perdiendo el tiempo. El costo

escondido de los trucos para obtener rendimiento es que el código se convierte en menos entendible y mas difícil de mantener.

67. **Recuerde que el código es leer mucho mas que escribir.** Diseños limpios hace programas fáciles de entender, pero los comentarios, las explicaciones detalladas y los ejemplos son invalorables. Estos nos ayudaran a nosotros y a cualquiera que venga luego de usted. Sin nada mas, la frustración de tratar de lograr información útil de la documentación en línea de Java debería convencernos.

# D: Recursos

## Software

**La JDK en [java.sun.com](http://java.sun.com)** Incluso si se ha elegido utilizar un ambiente de desarrollo de terceros, es siempre una buena idea tener la JDK a mano en caso de que aparezca lo que puede ser un error del compilador. La JDK es el la piedra de toque, y si ahí hay un error, las posibilidades es que sea bien conocido.

**La documentación HTML de Java de [java.sun.com](http://java.sun.com)** Nunca he encontrado un libro de referencia de la librería estándar de Java que no esté fuera de fecha o falte información. A pesar de que la documentación HTML de Sun esté plagada de pequeños errores y a veces es bruscamente inutilizable, toda la información de las clases y métodos está al menos *ahí*. Las personas a veces no se sienten confortables al comienzo utilizando los recursos en línea en lugar de un libro impreso, pero tiene valor ir allí y abrir la documentación HTML en un principio, así es que se puede al menos una imagen general. Si no se puede imaginar nada en este punto, se debería ir a los libros impresos.

## Libros

**Pensando en Java, 1<sup>ra</sup> Edición.** Disponible indexado completamente, con sintaxis HTML resaltada en el CD-ROM que viene con este libro, o libremente en [www.BruceEckel.com](http://www.BruceEckel.com) Incluye viejo material y material que no es considerado suficientemente interesante de llevar en la segunda edición.

**Core Java 2**, de Horstmann & Cornell, Volume I - Fundamentales (Prentice-Hall, 1999). Volumen II - Características avanzadas, 2000. Grande, de gran amplitud, y el primer lugar a donde voy cuando estoy buscando respuestas. El libro que recomiendo cuando termine *Pensando en Java* se necesita ampliar.

**Java in a Nutshell: A Desktop Quick Reference, 2nd Edition**, por David Flanagan (O'Reilly, 1997). Un resumen compacto de la documentación en línea de Java. Personalmente, prefiero buscar en la documentación en línea en [java.sun.com](http://java.sun.com), especialmente porque cambia muy a menudo. Sin embargo, mucha gente sigue prefiriendo la documentación impresa y esta encaja en las cuentas; también proporciona mas debate que los documentos en línea.

**The Java Class Libraries: An Annotated Reference**, de Patrick Chan y Rosanna Lee (Addison-Wesley, 1997). Lo que la documentación en línea *deberían* haber sido: suficientes descripciones para hacerla utilizable. Uno de las personas que realizaron la revisión técnica de *Thinking in Javadice*, “Si contara solo con un libro, este sería este (bueno, además de los suyos, claro)”. No estoy fascinado con lo que es. Es grande, es caro, y la calidad de los ejemplos no me satisfacen. *Pero* es un lugar donde buscar cuando se este estancado y parece tener mas profundidad (y mucho mayor tamaño) que *Java in a Nutshell*

**Java Network Programming**, por Elliotte Rusty Harold (O'Reilly, 1997). No se comienza a entender las redes de trabajo en Java hasta que se encuentra este libro. También encontré su sitio Web, Café au Lait, estimulante, testarudo, y con una perspectiva en fecha con los desarrolladores de Java, sin trabas de lealtad a ningún vendedor. Regularmente en fecha se mantiene con velocidad -cambiando noticias acerca de Java. Vea [metalab.unc.edu/javafaq/](http://metalab.unc.edu/javafaq/)

**JDBC Database Access with Java**, por Hamilton, Cattell & Fisher (Addison-Wesley, 1997). Si no se conoce nada acerca de SQL y bases de datos, esto es una bonita, gentil introducción. También contiene detalles así como “referencias comentadas” de la API (nuevamente, lo que la referencia en línea debería haber sido). El inconveniente, al igual que todos los libros de la Serie de Java (“Los UNICOS libros autorizados por JavaSoft”) es que han sido blanqueados así es que solo dice cosas maravillosas acerca de Java-no se encontrarán esquinas oscuras en esta serie.

**Java Programming with CORBA**, por Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997). Un tratamiento serio del tema con los ejemplos de código para tres Java ORBs (Visibroker, Orbix, Joe).

**Design Patterns**, por Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995). El libro seminal que comienza con el movimiento de patrones en la programación.

**Practical Algorithms for Programmers**, por Binstock & Rex (Addison-Wesley, 1995). Los algoritmos están en C, así es que fácilmente se pueden trasladar a Java. Cada algoritmo es explicado a fondo.

## Análisis & diseño

**Extreme Programming Explained**, por Kent Beck (Addison-Wesley, 2000). Amo este libro. Si, tengo tendencia a tomar una estrategia radical por las cosas. Pero siempre siento que puede haber muy diferentes, y muchos procesos de desarrollo mejores, y pienso que XP comienza a cerrar muy bien. El único libro que ha tenido un impacto similar en mi es *PeopleWare* (descrito mas adelante), que toma primariamente el ambiente y trata con la

cultura corporativa. *Extreme Programming Explained* habla acerca de la programación y da vueltas muchas cosas, aún los “hallazgos” recientes, en sus oídos. Entonces aún yendo mas lejos para decir que esas imágenes están OK con tal de que no pierda mucho tiempo con ellas y esta deseoso de lanzarlas lejos (Notará que este libro no tienen el “sello de aprobación de UML” en su cubierta). Puedo discernir acera de trabajar en una empresa basada únicamente en si utilizan XP. Pequeño libro, capítulos pequeños, fácil de leer, emocionante pensar acerca de el. Comience a imaginarse a usted mismo trabajando en una atmósfera como esa y vendrán visiones de un mundo totalmente nuevo.

**UML Distilled, 2<sup>nd</sup> Edition**, por Martin Fowler (Addison-Wesley, 2000). Encontrar UML por primera vez, es desalentador porque hay muchos diagramas y detalles. De acuerdo con Fowler, muchas de estas cosas son innecesarias así es que ha dejado solo lo esencial. Para la mayoría de los proyectos, solo se necesitan saber algunas herramientas de diagramación, y la meta de Fowler es comenzar con un buen diseño en lugar de preocuparse por todos los artefactos que se consiguen ahí. Un libro bonito, pequeño, que se puede leer; lo primero que se debería obtener si se necesita entender UML.

**UML Toolkit**, por Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997). Explica UML y como utilizarlo, y tiene un estudio de casos en Java. Un CD ROM que lo acompaña contiene el código Java y una versión recortada de Rational Rose. Una excelente introducción a UML y como utilizarlo para crear un sistema real.

**The Unified Software Development Process**, por Ivar Jacobson, Grady Booch, y James Rumbaugh (Addison-Wesley, 1999). Iba preparado a para que me desagrada este libro. Parecía tener todas las características de un texto de colegio aburrido. Fui agradablemente sorprendido-solo algunos huecos en el libro contiene explicaciones que parecen como si aquellos conceptos no fueran claros para los autores. La mayoría del libro no es solo claro, también divertido. Y lo mejor de todo, el proceso crea un montón de sentido práctico. No es Extreme Programming (y no tiene su claridad acerca de la prueba) pero es también parte de la fuerza de UML-incluso si no se puede adoptar XP, muchas de las personas se suben abordo de la camioneta “UML es bueno” (independientemente del nivel de experiencia con ella) y de esta forma se puede probablemente adoptarla. Pienso que este libro puede ser el navío almirante de UML, y que el único que se puede leer después de Fowler's UML Distilled cuando se busquen mas detalles.

Antes de que se elija cualquier método, es útil obtener una perspectiva de alguien que no trate de venderle uno. Es fácil adoptar un método sin realmente entender que es lo que se quiere dejar afuera o que hará por nosotros. Otros lo están utilizando, lo que parece una razón convincente. Sin embargo, los humanos tienen una pequeña excentricidad: si quieren creer

que algo solucionará sus problemas, intentarán con eso (Esto es experimentación, lo que es bueno). Pero si no soluciona sus problemas, pueden redoblar los esfuerzos y comenzar a hacer público las cosas maravillosas que han descubierto, una gran cosa (Esto es negación, lo cual no es bueno). La suposición aquí puede ser que si se puede tener otras personas en el mismo bote, no estará solo, incluso si esta yendo para ningún lado (o hundiéndose).

Esto no sugiere que todas las metodologías van a ningún lado, y que se deba ir armado hasta los dientes con herramientas de metal que nos ayuden a estar en el modo de experimentación (“Esto es sin trabajar, intentando alguna otra cosa”) y fuera del modo de negación (“No, esto no es realmente un problema. Todo está maravilloso, no necesitamos cambiar”). Pienso que los siguientes libros, leídos antes de elegir un método, proporcionarán estas herramientas.

**Software Creativity**, por Robert Glass (Prentice-Hall, 1995). Este es el mejor libro que he visto que plantea la *perspectiva* de la totalidad del tema metodológico. Es una colección de ensayos y trabajos cortos que ha escrito Glass y algunas veces adquirido (P.J. Plauger es un contribuyente), reflexionando muchos años y estudiando el tema. Entretienen y solo lo suficiente para decir lo que es necesario; no nos pasean ni aburren. No solo es una bola de humo, tampoco; hay cientos de referencias a otros estudios y trabajos. Todos los programadores y directores deberían leer este libro antes de esquivar la mira metodológica.

**Software Runaways: Monumental Software Disasters**, por Robert Glass (Prentice-Hall, 1997). Lo mejor acerca de este libro es que nos trae a el primer plan de lo que no queremos hablar: cuantos proyectos no solo fallan, también fallan espectacularmente. Encuentro que la mayoría de nosotros sigue pensando “Eso no me puede suceder a mi” (o “Esto no puede suceder *nuevamente*”), y pienso que esto nos coloca en una desventaja. Teniendo en mente que cosa puede siempre llevarnos mal, se estará en mucho mejor posición para hacer las cosas correctamente.

**Peopleware, 2<sup>nd</sup> Edition**, por Tom DeMarco and Timothy Lister (Dorset House, 1999). A pesar de que tiene trasfondos en el desarrollo de software, este libro es acerca de proyectos y equipos en general. Pero el foco es en las personas y sus necesidades, en lugar de la tecnología y sus necesidades. Hablan acerca de cierto habiente donde las personas serán felices y productivos, en lugar de decidir que reglas deben seguir estas personas para ser componentes adecuados de una máquina. Esta actitud reciente, pienso, es el mayor contribuyente para hacer sonreír y asentir a los programadores cuando un método XYZ es adoptado y entonces con tranquilidad hacer lo que sea que siempre hacen.

**Complexity**, by M. Mitchell Waldrop (Simon & Schuster, 1992). Este cuenta la llegada en conjunto de un grupo de científicos de diferentes disciplinas en Santa Fe, New México, para discutir problemas reales que sus disciplinas individuales no pudieron resolver (El inventario de mercado en economía, la formación inicial de la vida en biología, por que las personas hacen lo que hacen en sociología, etc.). Cruzando física, economía, química, matemática, ciencias de la computación, sociología y otras, una aproximación multidisciplinaria a estos problemas se está desarrollando. Pero mas importante, una forma diferente de pensar acerca de estos problemas ultra complejos esta emergiendo: fuera del determinismo matemático acerca y la ilusión se que se puede escribir una ecuación que prediga todos los comportamientos, y ir primero observando y buscando por un patrón y tratando de emular ese patrón de cualquier forma posible (El libro cuenta, por ejemplo, el surgimiento de los algoritmos genéticos). Este tipo de cosas, creo, es útil para observar las diferentes formas de manejar mas y mas complejos proyectos de software.

## Python

**Learning Python**, by Mark Lutz and David Ascher (O'Reilly, 1999). Un a muy buena introducción de lo que rápidamente se ha vuelto mi lenguaje favorito, un excelente compañero de Java. El libro incluye una introducción a JPython, que permite combinar Java y Python en un solo programa (el intérprete de JPython es compilado a Java bytecodes puros de Java, así es que no se necesita nada especial que agregar para lograr esto). La unión de estos lenguajes prometen grandes posibilidades.

## Mi propia lista de libros

Listados en orden de publicación. No todos estos están actualmente disponibles.

**Computer Interfacing with Pascal & C**, (Publicación independiente mediante el Eisys imprint, 1988. Solo disponible a través de [www.BruceEckel.com](http://www.BruceEckel.com)). Una introducción a la electrónica desde cuando CP/M todavía era el rey y DOS se iniciaba. He utilizado lenguaje de alto nivel y a menudo el puerto paralelo de la computadora para manejar varios proyectos electrónicos. Adaptado de mis columnas en la primera y mejor revista en la que he escrito, Micro Cornucopia. (Para parafrasear Larry O'Brien, editor por largo tiempo de Software Development Magazine: la mejor revista de computadoras que se haya publicado -ellos incluso tenían planes para crear un robot en una maceta!) Alas, Micro C se perdieron antes de que apareciera la Internet. La creación de este libro fue una experiencia de publicación extremadamente satisfactoria.

**Using C++**, (Osborne/McGraw-Hill, 1989). Uno de los primeros libros publicados de C++. Este no se imprime mas y es reemplazado por la segunda edición, el renombrado *C++ Inside & Out*.

**C++ Inside & Out**, (Osborne/McGraw-Hill, 1993). Como he notado, actualmente, la 2<sup>da</sup> edición de **Using C++**. El C++ en este libro es razonablemente preciso , pero es de alrededor de 1992 y *Thinking in C++* es un intento de reemplazarlo. Se puede encontrar mas acerca de este libro y bajar los fuentes en [www.BruceEckel.com](http://www.BruceEckel.com)

**Thinking in C++, 1<sup>ra</sup> Edition**, (Prentice-Hall, 1995).

**Thinking in C++, 2<sup>nd</sup> Edition**, Volumen 1, (Prentice-Hall, 2000). Que se puede bajar libremente de [www.BruceEckel.com](http://www.BruceEckel.com)

**Black Belt C++, the Master's Collection**, Bruce Eckel, editor (M&T Books, 1994). Ya no se edita mas. Una colección de capítulos para iluminar basados en las presentaciones de C++ siguiendo la pista de las conferencias acerca de desarrollo de software, que he llevado en hombros. La cubierta de este libro me ha estimulado a ganar control sobre las siguientes diseños de cubiertas.

**Thinking in Java, 1<sup>ra</sup> Edition**, (Prentice-Hall, 1998). La primera edición de este libro ganó el premio Software Development Magazine Productivity, de la Java Developer's Journal Editor's Choice Award, y el JavaWorld Reader's Choice Award for best book. Se puede bajar de [www.BruceEckel.com](http://www.BruceEckel.com).