# C: Guías de programación Java

Este apéndice contiene sugerencias para ayudar en el diseño de programas de bajo nivel, y en la escritura de código.

Naturalmente, esto son guías, pero no reglas. La idea es usarlas como inspiración, y recordar que hay situaciones ocasionales en las que es necesario vulnerar o romper una regla.

## Diseño

1. **La elegancia siempre es rentable**. A corto plazo, podría parecer que lleva mucho tiempo alcanzar una solución totalmente correcta a un problema, pero cuando funciona a la primera y se adapta sencillamente a situaciones nuevas en vez de requerir horas, días o meses de devanarse los sesos, se ve la recompensa (incluso aunque no pueda medirse). Así no sólo se logra un programa que es fácil de construir y depurar, sino que es además fácil de entender y mantener, y es aquí donde radica el valor económico del esfuerzo. Puede ser necesario tener experiencia para llegar a entender este punto, porque puede parecer, en determinados momentos, que no se está siendo productivo al intentar convertir en elegante un fragmento de código. Hay que resistirse a la tendencia a apresurarse; al final, sólo sirve para ralentizar.

2. **Primero hay que hacer que funcione, y después, que sea rápido**. Esto es cierto incluso cuando se está seguro de que determinado fragmento de código es importante y que puede convertirse en un cuello de botella en el sistema. No lo hagas. Primero, intenta que funcione con el diseño más simple posible. Después, si no es lo suficientemente rápido, optimízalo. Casi siempre se descubre que el problema del cuello de botella no era tal. Hay que ahorrar tiempo para lo que es verdaderamente importante.

3. **Recuerde el principio "divide y vencerás"**. Si el problema al que uno se enfrenta es demasiado complicado, puede intentar pensarse en cuál sería el funcionamiento básico del programa, si se dispusiera de un "fragmento mágico" que se encargara de las partes complicadas. Ese "fragmento" es un objeto —escribe el código que usa el objeto, después mira el objeto y encapsula sus partes *complicadas* en otros objetos, etc.

4. **Separe el creador de la clase del usuario de la misma *(programador cliente)*.** El usuario de la clase es el "cliente" y no necesita saber qué es lo que está ocurriendo tras el comportamiento de la clase. El creador de la clase debe ser experto en diseño de clases y escribir la clase de forma que pueda ser usada por los programadores más novatos posible, eso sí, funcionando de forma robusta dentro de la aplicación. El uso de bibliotecas sólo es sencillo si es transparente.

5. **Al crear una clase, intente hacer que los nombres sean tan sencillos que hagan innecesarios los comentarios.** La meta debería ser hacer la interfaz con el programador cliente conceptualmente simple. Para lograr este fin, puede usarse la sobrecarga de métodos cuando sea apropiado para crear una interfaz intuitiva, fácil de usar.

6. **El análisis y diseño deben producir, al menos, las clases del sistema, sus interfaces públicas y sus relaciones con otras clases, especialmente las clases base.** Si tu metodología de diseño produce más que esto, pregúntate a ti mismo si los fragmentos que produce esa *metodología* tienen valor a lo largo de la vida del programa. Si no lo tienen, mantenerlos supondrá un coste elevado. Los miembros de los equipos de desarrollo tienden a no mantener nada que no contribuya a su productividad; es algo más que probado que hay muchos métodos de diseño que es mejor no utilizar.

7. **Automatice todo.** Escribe primero código de prueba (antes de escribir la clase) y mantenlo con la clase. Automatiza la ejecución de las pruebas a través de un *makefile* o una herramienta semejante. De esta forma, cualquier cambio podrá verificarse automáticamente ejecutando el código de prueba, y se descubrirán errores inmediatamente. Dado que se sabe que se tiene la red de seguridad del marco de trabajo de prueba, será más fácil hacer cambios inmediatamente tras descubrirlos. Recuérdese que las mayores mejoras en lo que a lenguajes se refiere provienen de pruebas preconstruidas, proporcionadas en la forma de comprobación de tipos, manejo de excepciones, etc., pero que estas facetas no llegan más allá. Hay que ir más allá y crear un sistema robusto creando todas las pruebas que verifican facetas específicas de cada clase o programa.

8. **Escriba el código de prueba (antes de escribir la clase) para verificar que el diseño de la clase sea completo.** Si no se puede escribir código de prueba, entonces uno no está seguro de qué apariencia tiene la clase. Además, el escribir código de prueba a menudo ayuda a que afloren facetas y limitaciones adicionales necesarias para la clase —estas facetas y limitaciones no siempre aparecen durante el análisis y diseño. Las pruebas también proporcionan código de ejemplo que muestra cómo usar esa clase.

9. **Todos los problemas del diseño de software pueden simplificarse introduciendo un nivel adicional de indirección.** Esta regla fundamental de ingeniería del software[1], es la base de la abstracción, la faceta principal de la programación orientada a objetos.

10. **Toda indirección debería tener un significado** (hace referencia a la regla 9). Este significado puede ser tan simple como "poner código comúnmente utilizado en un método simple". Si se añaden niveles de indirección (abstracción, encapsulación, etc.) que no tienen significado, su efecto puede ser tan pernicioso como no tener la indirección adecuada.

11. **Haga las clases tan atómicas como sea posible.** Da a cada clase un propósito simple y claro. Si las clases o el diseño del sistema se vuelven demasiado complicados, divide las clases en otras más simples. El indicador más obvio de este aspecto está relacionado con el ta-

[1] Que me explicó a mí Andrew Koeing.

maño: si una clase es grande, seguro que está haciendo demasiadas cosas, y por tanto, tiene más posibilidades de fallar.

Algunas pistas que sugieren el rediseño de una clase son:

1) Una sentencia switch complicada: considérese el uso del polimorfismo.

2) Un conjunto grande de métodos que cubren distintos tipos de operaciones: la solución pasaría por utilizar varias clases.

3) Un conjunto grande de variables miembro relativos a características bastante heterogéneas: la solución pasaría por utilizar varias clases.

12. **Vigile las listas de parámetros largas.** En ocasiones es difícil escribir, leer y mantener llamadas a métodos. En su lugar, debería intentar moverse el método a una clase en la que sea (más) adecuado, y/o pasar objetos como parámetros.

13. **No se repita a sí mismo.** Si un fragmento de código recurre a muchos métodos de clases derivadas, puede ponerse ese código en un único método de una clase base e invocarlo desde los métodos de las clases derivadas. Así no sólo se ahorra espacio de código, sino que se proporciona una propagación más sencilla de los cambios. En ocasiones, descubrir este código común añade funcionalidad de gran valor a la interfaz.

14. **Vigile las sentencias *switch* y las cláusulas *if-else* encadenadas.** Esto suele ser un indicador de *codificación de comprobación de tipos*, lo que significa que se está eligiendo qué código ejecutar a partir de alguna información de tipos (puede que el tipo exacto no sea obvio a la primera). Generalmente este tipo de código puede reemplazarse con herencia y polimorfismo; una llamada a un método polimórfico puede encargarse de la comprobación de tipos, y permitir una extensibilidad más sencilla y de confianza.

15. **Desde el punto de vista del diseño, busque y separe las cosas que varían de los elementos que permanecen invariantes.** Es decir, busca los elementos del sistema que uno podría desear que cambien sin que esto conllevara un rediseño, y encapsula esos elementos en clases. Puede aprenderse mucho más de este concepto en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*.

16. **No extienda la funcionalidad fundamental mediante subclases.** Si un elemento de una interfaz resulta esencial para una clase, debería estar en la clase base y no añadirse durante la derivación. Si se añaden métodos por herencia, probablemente haya que replantearse el diseño.

17. **Menos es más.** Empieza con una interfaz mínima para una clase, tan pequeña y simple como se necesite para solucionar sencillamente el problema, pero no intentes anticiparte a todas las formas en las que *podría* llegar a usarse la clase. Al usar la clase, se descubrirán formas de expandir la interfaz.

    Sin embargo, una vez que una clase esté en uso, no se podrá manipular su interfaz sin molestar al código cliente. Si hay que añadir más métodos, bien; esto no afectará al código más allá de forzar su recompilación. Pero incluso si los métodos nuevos reemplazan la funcionalidad de

los viejos, hay que dejar la interfaz existente a un lado (puede combinarse, si se desea, con la funcionalidad de la implementación subyacente). Si se desea expandir la interfaz de un método existente añadiendo más parámetros, crea un método sobrecargado con estos nuevos parámetros; de esta forma las llamadas existentes al método existente no se verán afectadas.

18. **Lea sus clases en alto para asegurarse de que tienen sentido.** Hay que hacer referencia a la relación entre una clase base y una clase derivada como "es-un" y con los miembros objeto como "tiene-un".

19. **Al decidir entre herencia y composición, pregúntese si necesita hacer conversiones hacia arriba al tipo base**. Si no, es mejor la composición (objetos miembro) antes que la herencia. Esto puede eliminar la necesidad de múltiples tipos de clase base. Si se hereda, los usuarios pensarán que se supone que tendrán que hacer conversiones hacia arriba.

20. **Use miembros de datos para variaciones en valor y superposición de métodos para variaciones de comportamiento**. Es decir, si se encuentra una clase que usa variables de estado junto con métodos que cambian de estado en función de esas variables, probablemente habría que rediseñarla para que exprese las diferencias de comportamiento entre subclases y métodos superpuestos.

21. **Vigile la sobrecarga**. Un método no debería ejecutar condicionalmente código basado en el valor de un parámetro. En este caso, deberían crearse en su lugar dos o más métodos sobrecargados.

22. **Use jerarquías de excepciones** —preferiblemente derivadas de clases específicas apropiadas en la jerarquía de excepciones Java estándares. La persona que capture las excepciones puede así capturar los tipos específicos de excepción seguidos del tipo base. Si se añaden excepciones derivadas nuevas, el código cliente existente seguirá capturando la excepción a través del tipo base.

23. **En ocasiones la agregación simple se encarga del trabajo**. Un "sistema de confort para viajeros" de una línea aérea consiste en elementos inconexos: asiento, aire acondicionado, vídeo, etc., y sigue siendo necesario crear muchos de éstos en un avión. ¿Se hacen miembros privados y se construye una interfaz nueva entera? No —en este caso, los componentes también son parte de la interfaz pública, por lo que habría que crear objetos miembro públicos. Estos objetos tienen sus propias implementaciones privadas, que siguen siendo seguras. Hay que ser conscientes de que la simple agregación no es una solución a utilizar a menudo, pero en ocasiones, se da.

24. **Considere la perspectiva del programador cliente y de la persona que mantiene el código**. Diseñe su clase para que su uso sea tan obvio como sea posible. Anticípese al tipo de cambios que se harán, y diseñe su clase de forma que estos cambios sean sencillos.

25. **Vigile "el síndrome del objeto gigante"**. Éste suele ser una aflicción de los programadores procedurales que son nuevos en POO y que a menudo acaban escribiendo un programa procedimental y pegándolo en uno o dos objetos gigantes. A excepción de los *marcos de trabajo* de aplicación, los objetos representan conceptos de la aplicación, y no la aplicación en sí.

26. **Si hay que hacer algo feo, al menos, concentre la fealdad dentro de una clase.**

27. **Si hay que hacer algo no portable, haga una abstracción de ese servicio y ubíquela dentro de una clase.** Este nivel extra de indirección evita que se distribuya la falta de portabilidad por todo el programa. (Todo queda dentro del Patrón Bridge.)

28. **Los objetos no deberían simplemente guardar datos.** También deberían tener comportamientos bien definidos. (En ocasiones, los "objetos de datos" son apropiados, pero sólo cuando se usan para empaquetar y transportar un grupo de elementos en casos en los que un contenedor generalizado se vuelve inadecuado.)

29. **Elija primero la composición, cuando se trate de crear nuevas clases a partir de las ya existentes.** Sólo se debería usar la herencia si el diseño lo requiere. Si se usa la herencia donde la composición sería suficiente, los diseños se vuelven innecesariamente complicados.

30. **Use la herencia y la superposición de métodos para expresar diferencias en comportamiento, y campos para expresar variaciones de estado.** Un ejemplo extremo de lo que no se debe hacer es heredar de distintas clases para representar los colores en vez de usar un campo "color".

31. **Vigile la *varianza*.** Dos objetos semánticamente diferentes pueden tener acciones o responsabilidades idénticas, y hay una tentación natural que intenta convertir a uno en subclase del otro simplemente para beneficiarse de la herencia. A esto se le llama *varianza*, pero no hay una justificación real para forzar una relación superclase/subclase donde no existe. Una solución mejor es crear una clase base general que produzca una interfaz para ambas como clases derivadas —lo cual requiere algo más de espacio, pero sigue ofreciendo los beneficios de la herencia, y probablemente aportará un descubrimiento importante para el diseño.

32. **Vigile la *limitación* durante la herencia.** Los diseños más simples añaden a los heredados nuevas capacidades. Un diseño sospechoso elimina las viejas capacidades durante la herencia sin añadir nuevas. Pero las reglas se hacen para romperlas, y si se está trabajando a partir de una biblioteca de clases vieja habría que reestructurar la jerarquía de forma que la clase nueva encaje donde debería, sobre la clase vieja.

33. **Use patrones de diseño para eliminar "funcionalidades desnudas".** Es decir, si sólo debería crearse un objeto de una clase, no hay por qué escribir un comentario "Hacer sólo uno de éstos". Envuélvalo en un *singleton*. Si se tiene mucho código entremezclado en el programa principal que crea los objetos, busque a un patrón creativo como un método fábrica en el que pueda encapsularse esa creación. Eliminar "funcionalidad desnuda" no sólo hará que el código sea más fácil de entender y mantener, sino que también lo hará más seguro frente a mantenedores "bien-intencionados" que vengan tras de ti.

34. **Vigile la "parálisis del análisis".** Recuerde que hay que avanzar dentro de un proyecto antes de poder saber todo, y que a menudo la forma mejor y más rápida de aprender alguno de los factores desconocidos pasa por avanzar a la siguiente etapa en vez de tratar de adivinarla mentalmente. No se puede saber la solución hasta que se *tenga*. Java tiene cortafuegos preconstruidos; deje que trabajen para ti. Tus fallos en una clase o en un conjunto de clases no destruirán la integridad de todo el sistema.

35. **Cuando se piensa que se tiene un buen análisis, diseño o implementacion, recórralos.** Traiga a alguien a su grupo —no tiene por qué ser un consultor, sino que puede ser cualquiera de otro grupo de la compañía. Repasar el trabajo con un par de ojos frescos puede revelar problemas en una etapa en la que es aún fácil repararlos, cundiendo finalmente este tiempo y coste invertido que el proceso de recorrido.

# Implementación

36. **En general, siga las convenciones de codificación de Sun.** Éstas están disponibles en *http://java.sun.com/docs/codeconv/indesc.html* (el código de este libro ha seguido estas convenciones en la medida en que me ha sido posible). Éstas se usan para lo que constituye probablemente el cuerpo de código más grande al que se expondrán la gran mayoría de programadores Java. Si uno se sale de un estilo de codificación de uso general, el resultado final será código más difícil de leer. Sea cual sea la convención de codificación que se decida usar, hay que asegurarse hacerlo de forma consistente en todo el proyecto. Hay una herramienta que reformatea código Java automáticamente en: *http://home.wtal.de/software-solutions/jindent.*

37. **Sea cual sea el estilo de codificación que se use, verdaderamente hay diferencia si el equipo (y aún mejor, si toda la compañía) lo estandariza.** Esto significa que todo el mundo considere que debe ajustarse al estilo —que puede no ser de uno mismo— aunque no esté conforme. El valor de la estandarización es que lleva menos quebraderos de cabeza ajustar el código, por lo que es más fácil centrarse en el significado del código.

38. **Siga las reglas estándares de mayúsculas y minúsculas.** Ponga en mayúscula la primera letra de un nombre de clase. La primera letra de los campos, métodos y objetos (referencias) debería ser minúscula. Todos los identificadores deberían llevar sus palabras juntas, y poner en mayúscula la primera letra de todas las palabras intermedias. Por ejemplo:

    ```
    EstoEsUnNombredeClase

    estoesUnMetodoOCampo
    ```

    Hay que poner en mayúscula *todas* las palabras de los identificadores primitivos **static final** que tengan inicializadores constantes en sus definiciones. Esto indica que son constantes de tiempo de compilación.

    **Los paquetes son un caso especial** —sus nombres están formados sólo por letras minúsculas, incluso para palabras intermedias. La extensión de dominio (com, org, net, edu, etc.) también deberían ir en minúsculas. (Éste es uno de los cambios introducidos por Java 2 sobre Java 1.1).

39. **No cree sus propios nombres de miembros de datos private "decorados".** Esto se suele ver en forma de caracteres y guiones bajos encadenados. La notación húngara es el peor ejemplo de esto. En ella, se adjuntan caracteres extra que indican el tipo de datos, uso, localización, etc., como si se estuviera escribiendo en lenguaje ensamblador y el compilador no proporcionara asistencia extra de ningún tipo. Estas notaciones son confusas, difíciles de leer y

desagradables de forzar y mantener. Hay que dejar que las clases y los paquetes se encarguen del ámbito de los nombres.

40. **Siga una "forma canónica"** al crear una clase para uso de propósito general. Incluya definiciones de **equals( ), hashCode( ), toString( ), clone( )** (implementa **Cloneable**), e implemente **Comparable** y **Serializable**.

41. **Use las convenciones de nombres "get", "set" e "is" de los JavaBeans**, para los métodos que lean y cambien campos **private**, incluso si no se piensa que en ese momento se esté construyendo un JavaBean. Esto no sólo facilita el uso de la clase como un Bean, sino que es una forma estándar de nombrar a estos tipos de métodos y de esa forma será más fácil de entender por parte del lector.

42. **Por cada clase que crees, considere incluir un static public test( ) que contenga código para probar esa clase.** No es necesario eliminar el código de prueba para usar la clase en un proyecto, y si se hacen cambios se pueden volver a ejecutar las pruebas de forma sencilla. Este código también proporciona ejemplos de cómo usar la clase.

43. **En ocasiones es necesario heredar para poder acceder a miembros protected de la clase base.** Esto puede llevar a la necesidad de múltiples tipos base. Si no se necesita hacer una conversión hacia arriba, derive primero una clase nueva para que lleve a cabo el acceso protegido. Después, convierta esta clase nueva en un objeto miembro de cualquier clase que necesite usarla, en vez de heredarla.

44. **Evite el uso de métodos final sólo por propósitos de eficiencia.** Use **final** sólo cuando se esté ejecutando el programa, no siendo lo suficientemente rápido, y se verá claro que el cuello de botella radica en un método.

45. **Si dos clases están asociadas mutuamente de alguna forma funcional (como los contenedores y los iteradores), intente convertir a una en clase interna de la otra.** Esto no sólo enfatiza la asociación entre clases, sino que permite que se reutilice el nombre de la clase dentro de un único paquete anidándola dentro de otra clase. La biblioteca de contenedores de Java hace esto definiendo una clase interna **Iterator** dentro de cada clase contenedor, proporcionando así a los contenedores una interfaz común. La otra razón por la que usar una clase interna es parte de la implementacion **private**. Aquí, es beneficioso el ocultamiento de la implementación de la clase interna, frente a la asociación y para evitar la contaminación del espacio de nombres descrita líneas más arriba.

46. **Cada vez que se encuentren clases que parezcan estar fuertemente acopladas entre sí, hay que considerarlas mejoras de codificación y mantenimiento que podrían lograrse usando clases internas.** El uso de clases internas no desacoplará las clases, pero hará el acoplamiento más explícito y conveniente.

47. **No caiga en optimización prematura.** Ésta lleva a la locura. En concreto, no se preocupe por la escritura (o la evitación) de métodos nativos, haciendo que algunos métodos sean **final**, o tratando de lograr que el código sea eficiente desde el mismo momento en que se

construye el sistema por primera vez. La meta principal debe ser probar el diseño, a menos que el diseño exija de por sí cierta eficiencia.

48. **Mantenga los ámbitos de las variables tan pequeños como sea posible de forma que la visibilidad y el tiempo de vida de los objetos sea también lo menor posible.** Esto reduce la opción de usar un objeto en el contexto erróneo y de ocultar un error difícil de encontrar. Por ejemplo, suponga que se tiene un contenedor y un fragmento de código que itera por él. Si se copia ese código para ser utilizado en otro contenedor, se podría acabar de forma accidental usando el contenedor viejo como límite superior del nuevo. Si, sin embargo, el contenedor viejo está fuera de ámbito, el error se detectará en tiempo de compilación.

49. **Use los contenedores de la biblioteca estándar de Java.** Sea sensato y habitúase a su uso, y así incrementará enormemente su productividad. Seleccione **ArrayList** para las secuencias, **HashSet** para los conjuntos, **HashMap** para arrays asociativos, y **LinkedList** para pilas (en vez de **Stack**) y colas.

50. **Para que un programa sea robusto, cada componente debe ser robusto.** Utilice las herramientas que le proporciona Java: control de accesos, excepciones, comprobación de tipos, etc., en cada clase que cree. De esa forma se puede pasar al siguiente nivel de abstracción con seguridad, durante la construcción del sistema.

51. **Prefiera los errores en tiempo de compilación a los errores en tiempo de ejecución.** Intente manejar los errores tan cerca del punto en el que ocurren como sea posible. Prefiera manipular el error en ese momento frente a lanzar una excepción. Capture las excepciones en el gestor más cercano con información suficiente para manipularlo. Haga lo que pueda con la excepción en el nivel actual; si eso no soluciona el problema, vuelva a lanzar la excepción.

52. **Vigile las definiciones de métodos largas.** Los métodos deberían ser unidades breves, funcionales que describen e implementan una parte discreta de una interfaz de clase. Un método que sea largo y complicado es difícil y caro de mantener, y está intentando hacer probablemente demasiado él solo. Si se ve un método así, indica que, al menos, debería dividirse en varios métodos. También puede sugerir la creación de una nueva clase. Los métodos pequeños también pueden potenciar la reutilización dentro de la clase. (En ocasiones, los métodos deben ser grandes, si bien siguen haciendo sólo una cosa.)

53. **Mantenga las cosas "tan *private* como sea posible".** Una vez que se hace público cierto aspecto de una biblioteca (un método, una clase, un campo), no se puede quitar. Si se hace, se puede estropear el código existente de alguna persona, forzándole a rediseñarlo y rescribirlo. Si se hace público sólo lo que se debe, se puede cambiar todo el resto con impunidad, y puesto que los diseños tienden a evolucionar, ésta es una libertad considerable. Así, los cambios en la implementacion tendrán un impacto mínimo en las clases derivadas. La privacidad es especialmente importante cuando se trabaja con varios hilos —sólo los campos **private** pueden protegerse del uso no **syncrhonized**.

54. **Use comentarios de forma liberal, y haga uso de la sintaxis de comentarios-documentación javadoc para producir la documentación de sus programas.** Sin embargo, los comentarios deberían añadir significado genuino al código; los comentarios que sólo rei-

teran lo que el código claramente expresa, son molestos. Nótese que simplemente poner nombres de clases y métodos Java con algo de nivel de detalle puede hacer innecesarios muchos comentarios.

55. **Evite el uso de "números mágicos"** —que son números estrechamente vinculados al código. Éstos constituyen una pesadilla si se necesita variarlos, dado que nunca sabe si "100" es el "tamaño del array" o "cualquier otra cosa". En su lugar, pueden crearse constantes con un nombre descriptivo y usar el identificador de la constante por todo el programa. Esto hace que el programa sea más fácil de entender y mantener.

56. **Al crear constructores, considere las excepciones.** En el mejor caso, el constructor no hará nada que lance una excepción. En el escenario inmediatamente mejor, la clase estará compuesta y se habrá heredado sólo de clases robustas, por lo que no será necesaria ninguna eliminación si se lanza una excepción. Si no, habrá que eliminar las clases compuestas internas a una cláusula **finally**. Si un constructor tiene que fallar, la acción apropiada es que lance una excepción, de forma que el llamador no continúe a ciegas, pensando que el objeto se creó correctamente.

57. **Si su clase necesita cualquier eliminación, cuando el programador cliente haya acabado con el objeto, ubicará el código de eliminación en un método único bien definido** —de nombre parecido a **limpiar( )** que sugiere claramente su propósito. Además, ubique un indicador **boolean** en la clase para indicar si se ha eliminado el objeto de forma que **finalize( )** pueda comprobar "la condición de muerte" (ver Capítulo 4).

58. **La responsabilidad de *finalize()* sólo puede ser verificar "la condición de muerte" de un objeto de cara a la depuración.** (Ver Capítulo 4). En casos especiales, podría ser necesario liberar memoria que de otra forma no sería liberada por el recolector de basura. Puesto que puede que no se invoque a éste para su objeto, no se puede usar **finalize( )** para llevar a cabo la limpieza necesaria. Para ello, hay que crear su propio método de "limpieza". En el método **finalize( )** de una clase, compruebe si el objeto se ha eliminado y lance una clase derivada de **RuntimeException**, si no lo ha hecho, para indicar un error de programación. Antes de confiar en un esquema así, hay que asegurarse de que **finalize( )** funcione en tu sistema. (Podría ser necesario invocar a **System.gc( )** para asegurar este comportamiento.)

59. **Si hay que eliminar un objeto (de otra forma que no sea el recolector de basura) dentro de determinado ámbito, use el enfoque siguiente:** inicialice el objeto y, caso de tener éxito, entre inmediatamente en un bloque **try** con una cláusula **finally** que se encargue de la limpieza.

60. **Al superponer *finalize()* durante la herencia, recuerde llamar a *super.finalize( )*.** (Esto no es necesario si la superclase inmediata es **Object**.) Debería llamarse a **super.finalize( )** como última acción del **finalize( )** superpuesto en vez de lo primero, para asegurar que los componentes de la clase base sigan siendo válidos si son necesarios.)

61. **Cuando se está creando un contenedor de objetos de tamaño fijo, transfiéralos a un array** —especialmente si se está devolviendo este contenedor desde un método. Así, se logra el beneficio de la comprobación de tipos en tiempo de compilación de los arrays, y el reci-

piente del array podría no necesitar convertir los objetos del mismo para poder usarlos. Nótese que la clase base de la biblioteca de contenedores, **java.util.Collection**, tiene dos métodos **toArray( )** para lograr esto.

62. **Eliga *interfaces* frente a clases abstractas.** Si se sabe que algo va a ser una clase base, la primera opción debería ser convertirlo en un **interface**, y sólo si uno se ve forzado a tener definiciones de métodos o variables miembros, modificarlo a clase **abstract**. Un **interface** habla de lo que el cliente desea hacer, mientras que una clase tiende a centrase en (o permitir) detalles de implementación.

63. **Dentro de constructores, haga sólo lo que es necesario para dejar el objeto en el estado adecuado.** Evite de forma activa llamar a otros métodos (excepto para los métodos **final**) puesto que estos métodos pueden ser superpuestos por alguien más para producir resultados inesperados durante la construcción. (Ver Capítulo 7 si se desean más detalles.) Los constructores pequeños y simples tienen menos probabilidades de lanzar excepciones o causar problemas.

64. **Para evitar una experiencia muy frustrante, asegúrese de que sólo hay una clase no empaquetada de cada nombre en cualquier parte del classpath.** Si no, el compilador puede encontrar primero la otra clase de nombre idéntico, y pasar mensajes de error que no tienen sentido. Si sospecha tener un problema de *classpath*, intente buscar archivos **.class** con los mismos nombres en cada punto de comienzo del *classpath*. De forma ideal, ponga todas tus clases en paquetes.

65. **Vigile la sobrecarga accidental.** Si se intenta superponer un método de clase base y no se deletrea bien, se puede acabar teniendo un método nuevo en vez de superponer el ya existente. Sin embargo, esto es perfectamente legal, por lo que no se obtendrá ningún mensaje de error por parte del compilador o del sistema de tiempo de ejecución —simplemente, el código no funcionará correctamente.

66. **Vigile la optimización prematura.** Primero haga que funcione, y después, que lo haga rápido —pero sólo si se debe, y sólo si se prueba que hay un cuello de botella de rendimiento en determinada sección del código. A menos que se haya usado un comprobador de eficiencia para descubrir un cuello de botella, probablemente se esté malgastando el tiempo. El coste oculto de problemas de rendimiento es que el código se vuelve menos entendible y mantenible.

67. **Recuerde que el código se lee más de lo que se escribe.** Los diseños limpios facilitan los programas fáciles de entender, pero los comentarios, explicaciones detalladas y ejemplos, tienen un valor incalculable. Todos ellos le ayudan tanto a usted como a cualquiera que venga por detrás. Si no, puede experimentarse a modo de ejemplo la frustración que le depara el intentar entender la documentación en línea de Java.