CAPITULO 12. MANEJO DE ERRORES

12.2 Programación defensiva

12.2.1 Interacción cliente-servidor

Los implementadores pueden adoptar como mínimo dos puntos de vista posibles al diseñar e implementar una clase servidor:

- Pueden asumir que los objetos cliente sabrán lo que están haciendo y requerirán servicios sólo de una manera sensata y bien definida.
- Pueden asumir que el servidor operará en un ambiente esencialmente hostil, en el que se deben tomar todas las medidas posibles para prevenir que los objetos cliente usen el servidor incorrectamente.

Estas visiones representan claramente extremos opuestos; en la práctica, la mayoría de las situaciones asumirán posiciones intermedias. Estos diferentes puntos de vista proporcionan una base muy útil para discutir asuntos del estilo:

- ¿Cuántas verificaciones de las solicitudes del cliente deben realizar los métodos del servidor?
- ¿Cómo debe informar el servidor, los errores a sus clientes?
- ¿Cómo puede un cliente anticipar un fallo en una solicitud al servidor?
- ¿Cómo puede tratar un cliente el fallo de una solicitud?

12.2.2 Validar argumentos

Un objeto servidor es más vulnerable cuando su constructor y sus métodos reciben los valores de los argumentos a través de sus parámetros. Los valores que se pasan a un constructor se utilizan para establecer el estado inicial de un objeto; los valores que se pasan a un método se usarán para influir sobre el efecto general de la llamada al método y quizás también sobre el resultado que produce. Por lo tanto, es vital que un servidor sepa si puede confiar en que los valores de los argumentos son válidos o si necesita verificar su validez por sí mismo.

12.3 Informar de errores del servidor

En lugar de simplemente programar alrededor del problema en el servidor y dejar el problema localizado allí, es una buena práctica hacer que el servidor realice algún esfuerzo para indicar que ha surgido un problema, ya sea del propio cliente o de un usuario humano o del programador. En este sentido, existe la posibilidad de que funcione bien un cliente escrito incorrectamente. ¿Cuál es la mejor manera de que un servidor informe de los problemas cuando éstos ocurren? No hay una sola respuesta a esta pregunta y generalmente, la respuesta más adecuada dependerá del contexto particular en el que se use el objeto servidor. En las siguientes secciones exploraremos un conjunto de opciones para informar errores mediante un servidor.

12.3.1 Notificar al usuario

La manera más obvia en que un objeto puede tratar de responder cuando detecta algo erróneo es intentar notificar al usuario de la aplicación de alguna forma. Las principales opciones son imprimir un mensaje de error usando System.out o mostrar una ventana de mensaje de error. Los dos problemas principales que tiene este abordaje son los siguientes:

- Asumen que la aplicación será usada por un usuario humano que verá el mensaje de error. Hay muchas aplicaciones que corren de manera completamente independiente de un usuario humano, en las que un mensaje de error o una ventana de error será completamente pasada por alto.
- Aun cuando exista un humano que pueda ver el mensaje de error, es raro que dicho usuario esté en posición de hacer algo con respecto al problema. Imagine a un usuario de un cajero automático enfrentado a un NullPointerException . Solamente en aquellos casos en los que la acción directa del usuario conduzca al problema puede estar capacitado para tomar alguna medida correctiva adecuada..

Los programas que imprimen mensajes de error inapropiados tienden más a confundir al usuario que a tener alguna utilidad para el mismo. Por lo tanto, excepto en un muy limitado conjunto de circunstancias, la notificación al usuario no es, en general, una solución al problema del informe de errores.

12.3.2 Notificar al objeto cliente

Un enfoque radicalmente diferente al que abordamos hasta ahora consiste en que el servidor ofrezca alguna indicación al objeto cliente de que algo anduvo mal. Hay dos maneras de hacer esto:

- Un servidor puede usar el valor de retorno de un método para devolver una bandera que indique si fue exitoso o si ocurrió un fallo en la llamada a dicho método.
- Un servidor puede *lanzar una excepción* desde el método servidor si algo anda mal. Esto introduce una nueva característica de Java que se encuentra también en otros lenguajes de programación.

Ambas técnicas tienen el beneficio de asegurar que el programador del cliente tenga en cuenta que puede fallar una llamada a un método sobre otro objeto. Sin embargo, sólo la decisión de lanzar una excepción evita activamente que el programador del cliente ignore las consecuencias del fallo del método.

Cuando un método servidor ya tenga un tipo de retorno distinto de void (para evitar efectivamente que se retorne un valor de diagnóstico boolean) todavía existe alguna forma de indicar que ha ocurrido un error mediante el tipo de retorno. Es común que los métodos que retoman referencias a objetos utilicen el valor null para indicar un fallo o un error. En los métodos que retornan valores de tipos primitivos, se suele devolver algún valor fuera de los límites válidos que cumple un rol similar: por ejemplo, el método indexOf de la clase String devuelve un valor negativo para indicar que falló en encontrar el carácter buscado.

Claramente, este enfoque no se puede usar en aquellos lugares en los que todos los valores del tipo de retorno ya tienen significados válidos para el cliente. En tales casos, generalmente será necesario pasar a la técnica alternativa de *lanzar una excepción* que, de hecho, ofrece importantes ventajas. Para ayudar a apreciar estas ventajas, es valioso considerar dos cuestiones asociadas al uso de los valores de retorno como indicadores de fracaso o de error:

- Desafortunadamente, no hay manera de requerir al cliente que controle el valor de retorno en relación a sus propiedades de diagnóstico. En consecuencia, un cliente podría fácilmente actuar como si nada hubiera ocurrido y luego terminar con un NullPointerException, o peor todavía, podría usar el valor de retorno de diagnóstico como si fuera un valor de retorno normal, creando un error lógico difícil de diagnosticar.
- En algunos casos, podríamos usar el valor de diagnóstico con dos propósitos muy diferentes. Un propósito es notificar al cliente si su petición fue exitosa o no. El otro es indicar que hubo algún error en su solicitud, como por ejemplo, que se pasó un valor incorrecto como argumento. En muchos casos, una solicitud no exitosa no representa un error lógico de programación sino que se hizo una solicitud incorrecta. Debemos esperar respuestas muy diferentes de un cliente en estos dos casos. No existe una manera satisfactoria y general de resolver este conflicto usando simplemente valores de retorno.

Otro problema es que pasa con los errores en los constructores.

12.4 Principios del lanzamiento de excepciones

El lanzamiento de una excepción es la manera más efectiva que tiene un objeto servidor para indicar que es incapaz de completar la solicitud del cliente. Una de las mayores ventajas que tiene esta técnica es que usa un valor especial de retorno que hace casi imposible que un cliente ignore el hecho de que se ha lanzado una excepción y continúe indiferente. El fracaso del cliente al manejar una excepción dará por resultado que la aplicación termine inmediatamente. Además, el mecanismo de la excepción es independiente del valor de retorno de un método y se puede usar en todos los métodos, más allá del tipo de valor que retornan.

Una **excepción** es un objeto que representa los detalles de un fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.

12.4.1 Lanzar una excepción

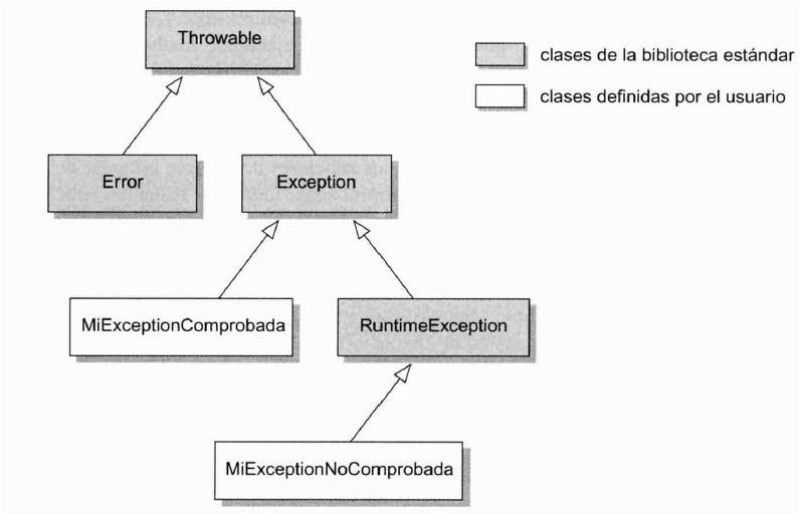
Se lanza una excepción usando una *sentencia throw* dentro de un método. El lanzamiento de una excepción tiene dos etapas: primero se crea un objeto Excepction (en este caso un objeto NullPointerException) y luego se lanza el objeto Exception usando la palabra clave throw. Estas dos etapas se combinan casi invariablemente en una única sentencia: throw new *TipoDeExcepcion* (" cadena opcional de diagnóstico");

Cuando se crea un objeto excepción, se puede pasar una cadena de diagnóstico a su constructor. Esta cadena estará disponible para el receptor de la excepción mediante el método de acceso getMessage del objeto Exceptio o de su método toString.

Se puede expandir la documentación de un método para que incluya los detalles de cualquier excepción que lance mediante la etiqueta @throws del documentador de java (*javadoc*).

12.4.2 Clases Exception

Un objeto excepción es siempre una instancia de una clase de una jerarquía de herencia especial. Podemos crear nuevos tipos de excepciones creando subclases en esta jerarquía. Hablando estrictamente, las clases de excepciones siempre son subclases de la clase Throwable que está definida en el paquete java.lang. Normalmente se definen y usan las clases de excepciones como subclases de la clase Exception, también definida en java.lang. El paquete java.lang define varias clases de excepciones que se ven comúnmente como por ejemplo: NullPointerException, IndexOutOfBoundsException y ClassCastException.



Java divide las clases de excepciones en dos categorías: *excepciones comprobadas (controladas) y no comprobadas (no controladas).* Todas las subclases de la clase estándar de Java RunTimeException son excepciones no comprobadas, todas las restantes subclases de Exception son excepciones comprobadas.

Sumamente simplificado, la diferencia es ésta: las excepciones comprobadas están pensadas para aquellos casos en los que el cliente debe esperar que una operación falle (por ejemplo: cuando grabamos en un disco, sabemos que el disco puede estar lleno). En estos casos, el cliente está obligado a comprobar si la operación fue exitosa. Las excepciones no comprobadas están pensadas para aquellos casos que no deben fallar en una operación normal; generalmente indican un error en el programa.

Saber qué categoría de excepción conviene lanzar en una circunstancia en particular no es una ciencia exacta pero podemos ofrecer las siguientes sugerencias:

- Una regla a priori que se puede aplicar es usar excepciones no comprobadas en las situaciones que podrían producir un fallo (total) en el programa, ya que se sospecha de la existencia de un error lógico en el programa que le impedirá continuar funcionando. Se desprende que las excepciones comprobadas deben usarse cuando ocurrió un problema pero existe alguna posibilidad de que el cliente efectúe alguna recuperación. Un problema con esta política es que asume que el servidor es suficientemente consciente del contexto en el que se está usando como para ser capaz de determinar si es probable que la recuperación del cliente sea posible.
- Otra regla a priori es usar excepciones no comprobadas en aquellas situaciones que pueden ser razonablemente evitadas. Por ejemplo, el uso de un índice no válido para acceder a un array es el resultado de un error lógico de programación que es completamente evitable y el hecho de que la excepción ArraylndexOutOfBoundsExcepction no es comprobada encaja con este modelo. Se desprende que las excepciones comprobadas deben usarse para situaciones de fallos que están bajo el control del programador como por ejemplo, que un disco se llene cuando se intenta grabar un archivo.

Las reglas formales de Java que gobiernan el uso de las excepciones son significativamente diferentes para las excepciones comprobadas y para las no comprobadas. En términos simples, las reglas aseguran que un objeto cliente que llama a un método que puede disparar una excepción comprobada puede contener tanto código para anticipar la posibilidad de un problema como código para intentar manejar el problema cuando éste ocurra.

12.4.3 El efecto de una excepción

¿Qué ocurre cuando se lanza una excepción? En realidad, hay dos efectos a considerar: el efecto en el método en que se lanzó la excepción y el efecto en el invocador. Cuando se lanza una excepción, la ejecución del método que la disparó termina inmediatamente (no continúa hasta el final del cuerpo). Una consecuencia particular de esto es que no se requiere un método con un tipo de retorno distinto de void para ejecutar una sentencia return en la ruta en que se lanza una excepción. Esto es razonable porque el lanzamiento de una excepción es una indicación de la incapacidad del método disparador para continuar con la ejecución normal, que incluye la imposibilidad de retornar un resultado válido.

La ausencia de una sentencia return en la ruta en que se dispara una excepción es aceptable. En su lugar, el compilador indicará un error si se han escrito sentencias a continuación de la sentencia *throw* porque podrían no ejecutarse nunca.

El efecto de una excepción en el sitio del programa que invocó al método es un poco más complejo. En particular, el efecto completo depende de si se ha escrito o no código para *capturar* la excepción.

Lo que realmente ocurre a continuación de una excepción depende de si se captura o no. Si no se captura la excepción, el programa simplemente terminará con la indicación de que se ha lanzado una Exception sin capturar.

12.4.4 Excepciones no comprobadas

Las **excepciones no comprobadas** son las más fáciles de usar desde el punto de vista del programador, porque el compilador impone muy pocas reglas para su uso. Este es el sentido de «no comprobadas»: el compilador no aplica ningún control especial sobre el método en el que se lanza una excepción no comprobada, ni tampoco en el lugar desde donde se invocó dicho método. Una clase Exception es no comprobada si es una subclase de la clase RuntimeException definida en el paquete java.lang. Hay muy poco para agregar sobre cómo lanzar una excepción no comprobada: simplemente usar una sentencia *throw*.

Si seguimos también la convención de que las excepciones no comprobadas deben usarse en aquellas situaciones en las que esperamos que el resultado sea la terminación del programa, (es decir, que no se va a capturar la excepción), entonces tampoco hay más para discutir sobre lo que debe hacer el método invocador puesto que no hará nada y dejará que el programa falle. Sin embargo, si existe la necesidad de capturar una excepción no comprobada, entonces se puede escribir un manejador de dicha excepción, exactamente de la misma manera que para una excepción comprobada.

Una excepción no comprobada, que se usa comúnmente es IlegalArgumentException, es lanzada por un constructor o un método para indicar que los valores de sus argumentos no son los adecuados.

Es valioso tener un método que conduzca una serie de controles de validez de sus argumentos antes de proceder con el propósito principal del método. Esto hace menos probable que un método ejecute parte de sus acciones antes de lanzar una excepción debida a valores incorrectos en sus argumentos. Una razón particular para evitar esta situación es que la modificación parcial de un objeto probablemente lo deje en un estado inconsistente para su futuro uso. Si una operación falla por alguna razón, idealmente, el objeto deberá quedar en el estado en que estaba antes de que se intentara realizar la operación.

12.4.5 Impedir la creación de un objeto

Un uso importante de las excepciones es impedir que se creen objetos cuando no se los puede preparar con un estado inicial válido. Generalmente, este será el resultado del pasaje al constructor de argumentos inadecuados. El proceso de lanzamiento de una excepción desde un constructor es exactamente el mismo que el lanzamiento desde un método.

Una excepción que se lanza desde un constructor tiene el mismo efecto sobre el cliente que una excepción que se lanza desde un método. En consecuencia, el intento de crear un objeto con parámetros no válidos que provoquen que se lance una excepción fallará completamente; no dará por resultado que se almacene un valor null en la variable a la que se trataba de asignar el objeto.

12.5 Manejo de excepciones

Los principios del lanzamiento de excepciones se aplican tanto para las excepciones comprobadas como para las no comprobadas, pero las reglas particulares de Java indican que el manejo de una excepción se convierte en un requerimiento sólo en el caso de excepciones comprobadas. Una clase de excepción comprobada es una subclase de Exception pero no de RuntimeException. Existen varias reglas que se deben seguir cuando se usan excepciones comprobadas porque el compilador obliga a tener controles tanto en los métodos que lanzan una excepción comprobada como en el invocador de dicho método.

Las **excepciones comprobadas** son un tipo de excepción cuyo uso requiere controles adicionales del compilador. En particular las excepciones comprobadas en *Java* requieren el uso de cláusulas *throws* y de sentencias *try*.

12.5.1 Excepciones comprobadas: la cláusula throws

El primer requerimiento del compilador es que un método que lanza una excepción comprobada debe declarar que lo hace mediante una *cláusula throws* que se agrega en su encabezado. Por ejemplo, un método que lanza una IOException comprobada del paquete java.io debe tener el siguiente encabezado:

```
public void grabarEnArchivo (String archivoDestino) throws IOException
```

Si bien se permite el uso de la cláusula *throws* para las excepciones no comprobadas, el compilador no lo requiere. Recomendamos que se use una cláusula *throws* solamente para enumerar las excepciones comprobadas que lanza un método.

Es importante distinguir entre la cláusula *throws* en el encabezado de un método y la etiqueta que se utiliza en el comentario que precede al método; la última es completamente opcional para ambos tipos de excepción.

12.5.2 Captura de excepciones: la sentencia try

El segundo requerimiento es que el invocador de un método que lanza una excepción comprobada debe proveer un tratamiento para dicha excepción. Esto generalmente implica escribir un *manejador de excepción* bajo la forma de una *sentencia try*.

```
try {
Aquí se protege una o más sentencias.
}
catch (Exception e) {
Aquí se informa y se recupera de la excepción
}
```

El código de un programa que protege sentencias que podrían lanzar una excepción se denomina **manejador de excepción**. El código proporciona información y/o código para recuperarse del error.

En vías de comprender cómo funciona un manejador de excepción es esencial apreciar que una excepción impide que el invocador continúe con el control normal del flujo. Una excepción interrumpe la ejecución de la sentencia del invocador que la causó y de aquí en adelante, tampoco se ejecutará cualquier sentencia que esté inmediatamente a continuación de la sentencia que produjo el problema. La pregunta que surge entonces es, «¿Dónde continúa la excepción en el invocador?». La sentencia *try* proporciona la respuesta: si se dispara una excepción desde una sentencia invocada dentro del bloque *try* la ejecución continúa en el correspondiente bloque *catch*.

Las sentencias ubicadas dentro de un bloque *try* se conocen como *sentencias protegidas*. Si no se dispara ninguna excepción durante la ejecución de las sentencias protegidas, entonces se salteará el bloque *catch* cuando se llegue al fina l del bloque *try*. La ejecución continuará con cualquier sentencia que esté a continuación de la sentencia *try* completa.

El bloque *catch* nombra el tipo de excepción que tiene designado tratar dentro de un par de paréntesis inmediatamente a continuación de la palabra *catch.* Así como el nombre del tipo de la excepción, también incluye un nombre de variable (tradicionalmente, simplemente «e») que se puede usar para hacer referencia al objeto Exception que fue lanzado. Una referencia a este objeto puede ser muy útil para proporcionar la información que se podrá usar para recuperarse del problema. Una vez que se completó el bloque *catch*, el control *no* retorna a la sentencia que causó la excepción.

12.5.3 Lanzar y capturar varias excepciones

Algunas veces, un método lanza más de un tipo de excepción para indicar diferentes tipos de problemas. Cuando se trate de excepciones comprobadas deben enumerarse todas en la cláusula *throws* del método, separadas por comas. Por ejemplo:

```
public void procesar ( ) throws EOException, FileNotFoundException
```

Un manejador de excepción debe capturar todas las excepciones comprobadas que se lanzan desde sus sentencias protegidas, de modo que una sentencia *try* puede contener varios bloques *catch*.

Cuando se lanza una excepción mediante una llamada a método dentro de un bloque *try*, los bloques *catch* se evalúan en el orden en que están escritos hasta que se encuentra una coincidencia en el tipo de excepción. Una vez que se llega al fin al de un único bloque *catch*, la ejecución continúa debajo del último bloque *catch*. Si se desea, se puede usar polimorfismo para evitar la escritura de varios bloques *catch.* Sin embargo, esto puede ser a expensas de ser capaz de tomar medidas de recuperación de un tipo específico. Del proceso natural de coincidencias se desprende que es importante el orden de los bloques c*atch* en una única sentencia *try* y que un bloque c*atch* para un tipo de excepción en particular no puede estar debajo de uno de sus supertipos; el bloque del supertipo anterior siempre encontrará coincidencia antes que el bloque del subtipo que se controla.

12.5.4 Propagar una excepción

Hasta ahora, hemos sugerido que una excepción debe ser capturada y manejada en la oportunidad más temprana posible. Es decir, una excepción lanzada en un método procesar debe ser capturada y manejada en el método que invocó a procesar. En la realidad, este no es estrictamente el caso ya que Java permite que una excepción se *propague* desde el método receptor hasta su invocador y posiblemente, más allá. Un método propaga una excepción simplemente al no incluir un manejador de excepción para proteger la sentencia que puede lanzarla. Sin embargo, para una excepción comprobada, el compilador requiere que el método propagador incluya una cláusula *throws* aun cuando no lance en sí mismo una excepción. Si la excepción es no comprobada, la cláusula *throws* es opcional y preferimos omitirla.

La propagación es común en los lugares en que el método invocador es incapaz de tomar una medida de recuperación o bien, no necesita ninguna, pero esto podría ser posible o necesario dentro de llamadas de nivel más alto.

12.5.5 La cláusula finally

Una sentencia *try* puede incluir un tercer componente que es opcional: la *cláusula final1y,* que se omite con frecuencia. La cláusula *finally* se proporciona para sentencias que se deben ejecutar tanto si se lanza una excepción para las sentencias protegidas como si no. Si el control alcanza el final del bloque *try* entonces se saltea el bloque *cacth* y se ejecuta la cláusula *finally.* Recíprocamente, si se lanza una excepción a partir del bloque *try*, entonces se ejecuta el bloque *catch* apropiado y luego se sigue con la ejecución de la cláusula *finally*.

- Se ejecuta una cláusula *finally* aunque se ejecute una sentencia *return* en los bloques *try* o *catch.*
- Si se lanza una excepción en el bloque *try* pero no se captura, entonces también se ejecuta la cláusula *finally* .

En el último caso, la excepción no capturada podría ser una excepción no comprobada que no requiere un bloque *catch*, por ejemplo. Sin embargo, también podría ser una excepción comprobada que no se maneja mediante un bloque *catch* pero que se propaga desde un método. En tal caso, la *cláusula finally* aún podría ser ejecutada. Como consecuencia, es posible que no se tenga ningún bloque *catch* en una sentencia *try* que tiene un bloque *try* y una cláusula *finally*.

12.6 Definir nuevas clases de excepción

Cuando las clases estándares de excepciones no describen satisfactoriamente la naturaleza del problema, se pueden definir nuevas clases más descriptivas usando herencia. Las nuevas clases de excepciones comprobadas pueden definirse como subclases de una clase de excepción comprobada existente (tal como Exception) y las nuevas excepciones no comprobadas debieran ser subclases de la jerarquía RuntimeException.

Todas las clases de excepción existentes soportan la inclusión de una cadena de diagnóstico que se pasa al constructor. Sin embargo, una de las principales razones para definir nuevas clases de excepción es la inclusión de más información dentro del objeto Exception para brindar el diagnóstico de error y de recuperación.

El principio de incluir información que podría colaborar en la recuperación del error debe tenerse en cuenta particularmente cuando se definen nuevas clases de excepción comprobadas. La definición de los parámetros formales del constructor de una excepción ayudará a asegurar que la información de diagnóstico esté disponible. Además, cuando la recuperación no sea posible o no se intente, asegura que se sobrescriba el método toString de la excepción de modo que incluya la información adecuada y de esta manera, ayudará a diagnosticar el motivo del error.

12.7 Usar aserciones

12.7.1 Controlar la consistencia interna

Cuando diseñamos o implementamos una clase frecuentemente tenemos un sentido intuitivo de las cosas que deben ser ciertas en un punto dado de la ejecución, pero raramente las establecemos formalmente. Típicamente estas son condiciones que deseamos establecer mientras desarrollamos una clase, antes de liberarla. En un sentido, los tipos de pruebas que discutimos en el Capítulo 6 son un intento de determinar si hemos implementado una representación correcta de lo que la clase o un método debe hacer. Las características de ese estilo de prueba es que las pruebas son *externas* a la clase que está siendo probada. La práctica de controlar los argumentos que hemos introducido en este capítulo cambia ligeramente el énfasis desde el control completamente externo hacia una combinación de control interno y externo.

Sin embargo, el control de argumentos se intenta primordialmente para proteger a un objeto servidor de ser usado incorrectamente por un objeto cliente. Esto deja aún de lado la cuestión de si debemos incluir algunos controles internos para asegurar que el objeto servidor se comporte como es debido. Una manera en que podríamos implementar el control interno durante el desarrollo sería a través del mecanismo normal de lanzamiento de excepciones. En la práctica debemos usar excepciones no comprobadas porque no podemos esperar que las clases cliente regulares incluyan manejadores de excepciones para aquellos casos que son esencialmente errores internos del servidor. Nos enfrentamos con la cuestión de eliminar estos controles internos una vez que se completó el proceso de desarrollo para evitar el potencialmente alto costo de estos controles en tiempo de ejecución que casi seguro van a pasar desapercibidos.

12.7.2 La sentencia assert

Para satisfacer la necesidad de llevar a cabo controles eficientes de la consistencia interna que puedan permanecer activos durante el desarrollo del código pero desactivados cuando se lo libera, se introdujo la *facilidad de aserción* en la versión l.4 del SDK de Java.

Una **aserción** es la afirmación de un hecho que debe ser verdadero en la ejecución normal del programa. Podemos usar aserciones para establecer explícitamente lo que asumimos y para detectar errores de programación más fácilmente.

La palabra clave assert va seguida de una expresión booleana. El propósito de la sentencia es afirmar algo que debe ser verdadero en este punto del método.

De esta manera, una sentencia assert cumple con dos propósitos. Por un lado, expresa explícitamente lo que asumimos como verdadero en un punto determinado de la ejecución y por lo tanto, aumenta la legibilidad tanto del desarrollador actual como la del futuro programador de mantenimiento y, por otro lado, realmente realiza el control de modo que nos notifica si el valor que asumirnos no fue el correcto. Esta sentencia puede ser de gran ayuda para encontrar errores temprana y fácilmente.

```
assert claveEnUso(clave);
assert tamanioConsistente() : "El tamaño de la libreta es inconsistente en eliminarContacto";
```

Si la expresión booleana en una sentencia *assert* se evalúa *true*, entonces la sentencia *assert* no tiene más efecto; si la sentencia se *evalúa false* se lanzará un AssertionError. Este último es una subclase de Error y forma parte de la jerarquía que representa errores irrecuperables: no se debe proveer ningún manejador a los clientes.

La segunda sentencia ilustra la forma alternativa de una sentencia *assert.* La cadena seguida de un punto y coma se pasará al constructor de AssertionError para ofrecer una cadena de diagnóstico. La segunda expresión no tiene por qué ser una cadena explícita, puede ser cualquier expresión con un valor determinado que se convertirá en un String antes de ser pasada al constructor.

La primera sentencia *assert* muestra que una aserción usará frecuentemente métodos que ya existen en la clase (claveEnUso). El segundo ejemplo ilustra que puede ser de utilidad proporcionar un método específico para los fines de llevar a cabo una prueba de aserción (en este ejemplo, tamanioConsistente). Esta modalidad podría usarse cuando el control involucra cálculos significativos.

12.7.3 Pautas para usar aserciones

Las aserciones están pensadas primordialmente para ofrecer una forma de realizar controles de consistencia durante las fases de desarrollo y de prueba de un proyecto. No están pensadas para ser usadas en el código liberado. Es por este motivo que el compilador de Java incluirá las sentencias *assert* en el código compilado sólo si se lo solicitamos. Se desprende que las sentencias *assert* nunca deben usarse para implementar la funcionalidad normal de un programa.

12.8 Recuperarse del error y anularlo

12.8.1 Recuperarse del error

El primer requerimiento de una recuperación exitosa del error es que los clientes tomen nota de cualquier notificación de error que reciban. Esto puede sonar obvio pero no es poco común que algunos programadores asuman que una llamada a método no fallará y por lo tanto, no se preocupen por controlar el valor que retorna. Aunque es difícil que se ignoren los errores cuando se usan excepciones, con frecuencia se captura y se informa la excepción pero no se toma ninguna medida respecto del hecho de que probablemente sea incorrecto continuar de manera indiferente.

La sentencia *try* de Java es la clave para proporcionar un mecanismo de recuperación del error cuando se lanza una excepción. La recuperación de un error implicará generalmente, tomar alguna medida correctiva dentro del bloque *catch* y luego, probar nuevamente.

- La anticipación de un error y la recuperación del mismo, generalmente requerirán un control de flujo más complejo que si el error no pudiera ocurrir.
- Las sentencias del bloque c*atch* son la clave para preparar el intento de recuperación.
- La recuperación frecuentemente implica probar nuevamente.
- No se puede garantizar el éxito de la recuperación.
- Debe haber algunas rutas de escape que eviten que el intento de recuperación se realice desesperada y eternamente.
- No siempre habrá un usuario humano al que se le pueda pedir un ingreso alternativo.
- Registrar el error debiera ser responsabilidad del cliente.

12.8.2 Anular el error

Debe quedar claro que llegar a una situación que lanza una excepción será, en el peor de los casos, fatal para la ejecución de un programa y, en el mejor caso, dificultoso de recuperar desde el cliente. En primer lugar, puede ser más simple tratar de evitar el error, pero esto generalmente requiere de la colaboración entre el servidor y el cliente.

Algunos principios generales importantes de esta filosofía son:

- Si el control de validación del servidor y los métodos de prueba del estado están visibles para un cliente, el cliente, generalmente, estará capacitado para evitar las causas que producen que el servidor lance una excepción.

• Si se puede evitar una excepción de esta manera, entonces la excepción que se lance realmente representa un error lógico de programación en el cliente. Esto sugiere el uso de excepciones no comprobadas para tales situaciones.
• Usar excepciones no comprobadas significa que el cliente no tiene que usar una sentencia *try* cuando ya se estableció que no se lanzará la excepción. Esta es una ganancia significativa porque tener que escribir sentencias *try* para situaciones que "no pueden ocurrir" es molesto para el programador y hace menos probable que se tenga seriamente en cuenta la provisión de una recuperación adecuada para las situaciones genuinas de error.

Los efectos no son todos positivos. Aquí hay algunas razones por las que este enfoque no siempre es práctico:

• Hacer que los controles de validación del servidor y los métodos de prueba de estado sean públicamente visibles para sus clientes, podría representar una pérdida significativa del encapsulamiento y dar por resultado un grado de acoplamiento más alto entre el servidor y el cliente que no es deseable.
• Probablemente no sea seguro que un servidor asuma que sus clientes *harán* los controles necesarios para evitar una excepción. Como resultado, esos controles frecuentemente estarán duplicados en ambos, cliente y servidor. Si los controles son computacional mente "caros" de hacer entonces la duplicación puede ser indeseable o prohibitiva. Sin embargo, desde nuestro punto de vista, es mejor sacrificar la supuesta eficiencia en función de programación más segura, cuando esta elección sea posible.

12.9 Estudio de caso: entrada/salida de texto

Un área importante de programación en la que no se puede ignorar la recuperación del error es la relacionada con las operaciones de entrada/salida (E/S), pues el programador de una aplicación puede tener menos control directo sobre el ambiente externo en el que se ejecuta. Por ejemplo, el archivo de datos que requiere cierta aplicación puede haber sido borrado accidentalmente o se puede haber corrompido de alguna manera, antes de que la aplicación se ejecute; o puede frustrarse un intento de guardar resultados en el sistema de archivos porque excede el límite de archivos posibles. Existen varias maneras en las que puede fa llar una operación de E/S.

El API de Java incluye el paquete java.io que contiene numerosas clases para implementar operaciones de E/S independientes de la plataforma en que se realicen. El paquete define la clase de excepción comprobada IOExcepction, como un indicador general de que algo anduvo mal en una operación de E/S. Otras clases de excepción proveen información de diagnóstico más detallada, como por ejemplo: EOFException y FileNotFoundException.

12.9.1 Lectores, escritores y flujos

Varias de las clases del paquete java.io se ubican dentro de dos categorías principales: aquellas que operan con archivos de texto y las que operan con archivos binarios. Las clases comprometidas con archivos de texto se conocen como *lectores* y *escritores* mientras que las comprometidas con los archivos binarios se conocen como manejadores de *flujo*(stream)

12.9.3 Salida de texto con FileWriter

Hay tres pasos involucrados en el almacenamiento de datos en un archivo:

l. Se abre el archivo.
2. Se escriben los datos.
3. Se cierra el archivo.

La naturaleza de la salida por archivo implica que cualquiera de estos pasos podría fallar por distintos motivos, muchos de ellos completamente ajenos al control del programador de la aplicación. En consecuencia, será necesario anticipar las excepciones que se lanzarán en cada paso.

Cuando se trata de escribir un archivo de texto, es habitual la creación de un objeto FileWriter cuyo constructor toma el nombre del archivo sobre el que se escribirá. El nombre del archivo puede ser una cadena o un objeto File. La creación de un FileWriter tiene el efecto de abrir el archivo externo y prepararlo para recibir alguna salida. Si el intento de abrir el archivo fracasa por algún motivo, entonces el constructor lanzará una IOException.

Una vez que el archivo se abrió satisfactoriamente, se puede usar el método write del escritor para guardar caracteres, generalmente en forma de cadenas, en el archivo. Podría fallar cualquier intento de escritura, aun cuando el archivo se haya abierto exitosamente; estos fallos son raros, pero no imposibles.

Una vez que se ha escrito toda la salida, es importante cerrar formalmente el archivo. Esto asegura que todos los datos hayan sido realmente escritos en el sistema externo de archivos y generalmente, tiene el efecto de liberar algunos recursos internos o externos. Nuevamente, aunque en raras ocasiones, podría fallar el intento de cerrar un archivo.

El modelo básico que surge de la discusión anterior podría ser:

```
try {
   FileWriter escritor = new FileWriter( " ... nombre del archivo”);
   while (hay más texto para escribir) {
        escritor.write(siguiente parte de texto);
        escritor.close();
   }
} catch (IOException e) {
   algo anduvo mal al acceder al archivo
}
```

La cuestión principal que surge es cómo tratar cualquier excepción que se lance durante los tres pasos. La excepción que se lanza cuando se intenta abrir un archivo es realmente la única en la que es posible hacer algo y sólo si existe alguna forma de generar un nombre de archivo alternativo para intentar nuevamente. Debido a que esta alternativa requerirá generalmente de la intervención de un usuario humano de la aplicación, las posibilidades de tratar la excepción exitosamente son obviamente específicas de la aplicación y del contexto. Si fracasa un intento de escribir en el archivo, es poco probable que la repetición del intento resulte exitosa. De manera similar, el fracaso al cerrar un archivo, generalmente, no merece la pena ningún esfuerzo de recuperación y la consecuencia será probablemente un archivo incompleto.

12.9.4 Entrada de texto con FileReader

El complemento de la salida de texto mediante un FileWriter es la entrada de texto mediante un FileReader. Tal como se podría esperar, para la entrada de texto se requiere un conjunto complementario de tres pasos: abrir el archivo, leerlo y cerrarlo. Mientras que las unidades naturales para la escritura de texto son los caracteres y las cadenas, las unidades naturales para la lectura de texto son los caracteres y las líneas. Sin embargo, pese a que la clase FileReader contiene un método para leer un solo carácter, no contiene ningún método para leer una línea. El problema con la lectura de líneas desde un archivo reside en que no hay un límite predefinido para la longitud de una línea. Esto quiere decir que cualquier método que devuelva la línea siguiente completa desde el archivo, debe ser capaz de leer un número arbitrario de caracteres. Por este motivo, generalmente se envuelve un objeto FileReader con un objeto BufferedReader que define un método readLine para leer una línea. Este método siempre elimina el carácter de terminación de línea en la cadena que retorna y se usa el valor null para indicar el fin de archivo.

```
try {
   BufferedReader lector = new BufferedReader(new FileReader(" ... nombre del archivo .. . "));
   String linea = lector. readLine () ;
   while (line != null) {
        hacer algo con la línea
        linea = lector. readLine () ;
   }
   lector.close();
} catch (FileNotFoundException e) {
   no se encontró el archivo especificado
} catch (IOException e) {
   algo anduvo mal al leer a al cerrar el archi va
}
```

Tal como en la salida, la cuestión que surge es qué hacer con cualquier excepción que se lanza durante todo el proceso. La clase File ofrece métodos que hacen posible reducir la probabilidad de que fracase la operación de apertura del archivo. Por ejemplo, define métodos de consulta tales como exists y canRead que permiten controlar el estado de un archivo antes de abrirlo. Tales controles no son aplicables generalmente cuando se trata de escribir en un archivo porque un archivo no debe existir antes de ser escrito.

12.9.5 Scanner Leer entradas desde el terminal

System.out es de tipo java.io.PrintStream se corresponde con lo que frecuentemente se denomina un destino de *salida estándar*. De manera similar, existe la correspondiente fuente de *entrada estándar* disponible en System.in, que es de tipo java.io.InputStream. Normalmente, no se usa directamente un InputStream cuando se necesitan leer entradas del usuario desde la terminal porque entrega la entrada de a un carácter por vez. En su lugar, generalmente se pasa un System.in al constructor de un Scanner, definida en el paquete java.Util.

```
Scanner lector = new Scanner (System. in);
String linea = lector. nextLine() ;
```

El método nextLine de Scanner retorna la siguiente línea completa desde la entrada estándar (sin la inclusión del carácter final newLine).

La c1ase Scanner no se limita a aportar entrada desde System.in; incluye un constructor que toma un parámetro File y suministra entradas leídas desde dicho archivo.

Probablemente, la característica más importante de la clase Scanner es su habilidad para "analizar" entradas; en otras palabras, para identificar si la entrada de texto tiene una estructura con algún sentido. Por ejemplo, una invocación al método nextInt de Scanner produce que se lea una secuencia de caracteres y se los convierta en su correspondiente valor entero. Esto nos evita leer una entrada como un texto y luego convertirla en números (o en otro tipo de datos) por nuestros propios medios.

12.9.6 Serialización de objetos

La **serialización** permite leer y escribir objetos completos y jerarquías de objetos en una única operación. Cada objeto involucrado debe ser de una clase que implemente la interfaz Serializable.

Esto funciona con estos dos objetos simples y con objetos de múltiples componentes como son las colecciones. Es una característica importante que evita, por ejemplo, tener que leer y escribir objetos campo por campo.

Para ser elegida para participar en la serialización, una clase debe implementar la interfaz Serializable que se define en el paquete java.*io.* Sin embargo, es valioso notar que esta interfaz no define ningún método. Quiere decir que el proceso de serialización es manejado automáticamente por el sistema en tiempo de ejecución y requiere que se escriba poco código definido por el usuario.