Manejo de excepciones: un análisis más detallado - Parte 2

11.5 Jerarquía de excepciones en Java

Todas las clases de excepciones heredan, ya sea en forma directa o indirecta, de la clase **Exception**, formando una *jerarquía de herencias*. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

La figura 11.4 muestra una pequeña porción de la jerarquía de herencia para la clase **`Throwable`** (una subclase de **`Object`**), que es la superclase de la clase **`Exception`**. Sólo pueden usarse objetos **`Throwable`** con el mecanismo para manejar excepciones. La clase **`Throwable`** tiene dos subclases: **`Exception`** y **`Error`**. La clase **`Exception`** y sus subclases, por ejemplo, **`RuntimeException`** (paquete **`java.lang`**) e **`IOException`** (paquete **`java.io`**), representan situaciones excepcionales que pueden ocurrir en un programa en Java, y que pueden ser atrapadas por la aplicación. La clase **`Error`** y sus subclases representan las situaciones anormales que ocurren en la JVM. La mayoría de los errores tipo **`Error`** ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones. Por lo general no es posible que las aplicaciones se recuperen de los errores tipo **`Error`**.

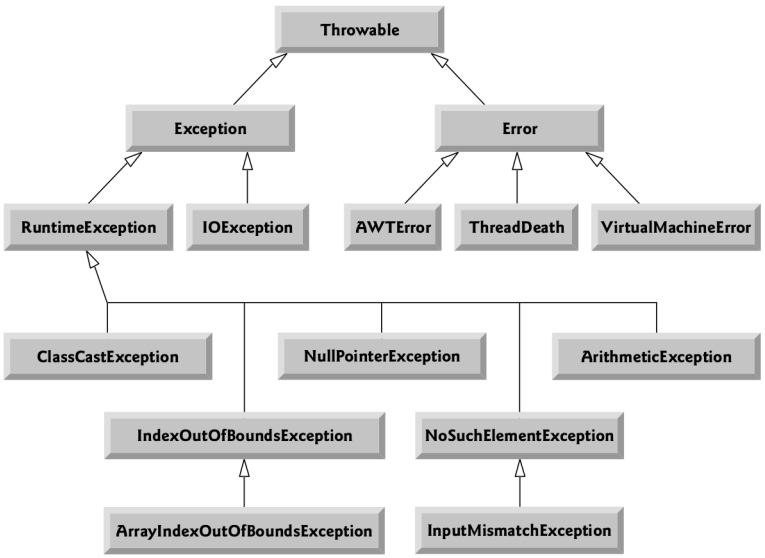

Figura 11.4 Porción de la jerarquía de herencia de la clase **`Throwable`**

La jerarquía de excepciones de Java contiene cientos de clases. En la API de Java puede encontrar información acerca de las clases de excepciones de Java. La documentación para la clase **`Throwable`** se encuentra en docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html. En este sitio puede buscar las subclases de esta clase para obtener más información acerca de los objetos **`Exception`** y **`Error`** de Java.

Comparación entre excepciones verificadas y no verificadas

Java clasifica a las excepciones en dos categorías: **excepciones verificadas** y **excepciones no verificadas**. Esta distinción es importante, ya que el compilador de Java implementa requerimientos especiales para las excepciones *verificadas* (que veremos en breve). El tipo de una excepción determina si es verificada o no verificada.

Las excepciones `RuntimeException` son excepciones no verificadas

Todos los tipos de excepciones que son subclases directas o indirectas de la clase **`RuntimeException`** (paquete **`java.lang`**) son excepciones no verificadas. Por lo general, se deben a los defectos en el código de nuestros programas. Algunos ejemplos de excepciones no verificadas son:

- Las excepciones **`ArrayIndexOutOfBoundsException`** (que vimos en el capítulo 7). Puede evitar estas excepciones asegurándose de que los índices de sus arreglos siempre sean mayores o iguales a 0 y menores que la longitud (**`length`**) del arreglo.
- Las excepciones **`ArithmeticException`** (que se muestran en la figura 11.3). Para evitar la excepción **`ArithmeticException`** que ocurre al dividir entre cero, revise el denominador para determinar si es 0 antes de realizar el cálculo.

Las clases que heredan de manera directa o indirecta de la clase **`Error`** (figura 11.4) son *no verificadas*, ya que los errores del tipo **`Error`** son problemas tan serios que su programa no debe ni siquiera intentar lidiar con ellos.

Excepciones verificadas

Todas las clases que heredan de la clase **`Exception`** pero *no* directa o indirectamente de la clase **`RuntimeException`** se consideran como excepciones verificadas. Por lo general, dichas excepciones son provocadas por condiciones que no están bajo el control del programa; por ejemplo, en el procesamiento de archivos, el programa no puede abrir un archivo si éste no existe.

El compilador y las excepciones verificadas

El compilador verifica cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza una excepción verificada. De ser así, el compilador asegura que la excepción verificada sea *atrapada* o *declarada* en una cláusula **`throws`**. A esto se le conoce como **requerimiento de atrapar o declarar**. En los siguientes ejemplos le mostraremos cómo atrapar o declarar excepciones verificadas. En la sección 11.3 vimos que la cláusula **`throws`** especifica las excepciones que lanza un método. Dichas excepciones no se atrapan en el cuerpo del método. Para satisfacer la parte relacionada con *atrapar* del *requerimiento de atrapar o declarar*, el código que genera la excepción debe envolverse en un bloque **`try`**, y debe proporcionar un manejador **`catch`** para el tipo de excepción verificada (o una de sus superclases). Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula **`throws`** que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo. Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error. Esto obliga a los programadores a pensar sobre los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza excepciones verificadas.

Tip para prevenir errores 11.2

Los programadores deben atender las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores simplemente ignoran las excepciones.

Error común de programación 11.2

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase mencionar más expresiones en su cláusula `throws` de las que tiene el método sobrescrito de la superclase. Sin embargo, la cláusula `throws` de una subclase puede contener un subconjunto de la lista throws de una superclase.

Observación de ingeniería de software 11.5

Si su método llama a otros métodos que lanzan excepciones verificadas, éstas deben atraparse o declararse. Si una excepción puede manejarse de manera significativa en un método, éste debe atrapar la excepción en vez de declararla.

El compilador y las excepciones no verificadas

A diferencia de las excepciones verificadas, el compilador de Java *no* examina el código para determinar si una excepción no verificada es atrapada o declarada. Por lo general, las excepciones no verificadas se pueden *evitar* mediante una codificación apropiada. Por ejemplo, la excepción **`ArithmeticEx-`**

ception no verificada que lanza el método cociente (líneas 9 a 13) en la figura 11.3 puede evitarse si el método se asegura de que el denominador no sea cero antess de tratar de realizar la división. *No* es obligatorio que se enumeren las excepciones no verificadas en la cláusula **throws** de un método; aun si se hace, *no* es obligatorio que una aplicación atrape dichas excepciones.

Observación de ingeniería de software 11.6

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, usted deberá proporcionar un código apropiado para el manejo de excepciones cuando sepa que podrían ocurrir. Por ejemplo, un programa debería procesar la excepción `NumberFormatException` del método `parseIntt` de la clase `Integer`, aun cuando `NumberFormatException` sea una subclase indirecta de `RuntimeException` (y, por ende, una excepción no verificada). Esto hará que sus programas sean más robustos.

Atrapar excepciones de subclases

Si se escribe un manejador **catch** para atrapar objetos de excepción de un tipo de *superclase*, también se pueden atrapar todos los objetos de las *subclases* de esa clase. Esto permite que un bloque **catch** maneje las excepciones relacionadas mediante el *polimorfismo*. Si estas excepciones requieren un procesamiento distinto, puede atrapar individualmente a cada subclase.

Sólo se ejecuta la primera cláusula `catch` que coincida

Si hay *varios* bloques **catch** que coinciden con un tipo específico de excepción, sólo se ejecuta el *primer* bloque **catch** que coincida cuando ocurra una excepción de ese tipo. Es un error de compilación atrapar el mismo tipo *exacto* en dos bloques **catch** distintos asociados con un bloque **try** específico. Sin embargo, puede haber varios bloques **catch** que coincidan con una excepción; es decir, varios bloques **catch** cuyos tipos sean los mismos que el tipo de excepción, o de una superclase de ese tipo. Por ejemplo, podríamos colocar después de un bloque **catch** para el tipo **ArithmeticException** un bloque **catch** para el tipo **Exception**; ambos coincidirían con las excepciones **ArithmeticException**, pero sólo se ejecutaría el primer bloque **catch** que coincidiera.

Error común de programación 11.3

*Al colocar un bloque **catch** para un tipo de excepción de la superclase antes de los demás bloques **catch** que atrapan los tipos de excepciones de las subclases, evitamos que esos bloques **catch** se ejecuten, por lo cual se produce un error de compilación.*

Tip para prevenir errores 11.3

Atrapar los tipos de las subclases en forma individual puede ocasionar errores si usted olvida evaluar uno o más de los tipos de subclase en forma explícita; al atrapar a la superclase se garantiza que se atraparán los objetos de todas las subclases. Al colocar un bloque `catch` para el tipo de la superclase después de los bloques `catch` de todas las otras subclases aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.

Observación de ingeniería de software 11.7

En la industria no se recomienda lanzar o atrapar el tipo `Exception`; aquí lo usamos sólo para demostrar la mecánica del manejo de excepciones. En capítulos subsiguientes, por lo general lanzaremos y atraparemos tipos de excepciones más específicos.

11.6 Bloque `finally`

Los programas que obtienen ciertos recursos deben devolverlos al sistema para evitar las denominadas **fugas de recursos**. En lenguajes de programación como C y C++, el tipo más común de fuga de recursos es la *fuga de memoria*. Java realiza la *recolección automática de basura* en la memoria que

ya no es utilizada por los programas, evitando así la mayoría de las fugas de memoria. Sin embargo, pueden ocurrir otros tipos de fugas de recursos en Java. Por ejemplo, los archivos, las conexiones de bases de datos y conexiones de red que no se cierran apropiadamente cuando ya no se necesitan, podrían no estar disponibles para su uso en otros programas.

Tip para prevenir errores 11.4

Hay una pequeña cuestión en Java: no elimina completamente las fugas de memoria. Java no hace recolección de basura en un objeto, sino hasta que no existen más referencias a ese objeto. Por lo tanto, si los programadores mantienen por error referencias a objetos no deseados, pueden ocurrir fugas de memoria.

El bloque **`finally`** (que consiste en la palabra clave **`finally`**, seguida de código encerrado entre llaves) es opcional, y algunas veces se le llama cláusula **`finally`**. Si está presente, se coloca después del último bloque **`catch`**. Si no hay bloques **`catch`**, el bloque **`finally`** sigue justo después del bloque **`try`**.

Cuándo se ejecuta el bloque `finally`

El bloque **`finally`** se ejecutará, independientemente de que se lance o no una excepción en el bloque **`try`** correspondiente. El bloque **`finally`** también se ejecutará si un bloque **`try`** se sale mediante el uso de una instrucción return, break o continue, o simplemente al llegar a la llave derecha de cierre del bloque **`try`**. El único caso en donde el bloque **`finally`** no se ejecutará es si la aplicación sale antes de tiempo de un bloque **`try`**, llamando al método **System.exit**. Este método, que demostraremos en el capítulo 15, termina de *inmediato* una aplicación.

Si una excepción que ocurre en un bloque **`try`** no puede ser atrapada por uno de los manejadores **`catch`** de ese bloque **`try`**, el programa omite el resto del bloque **`try`** y el control pasa al bloque **`finally`**. Luego, el programa pasa la excepción al siguiente bloque **`try`** exterior (por lo general en el método que hizo la llamada), en donde un bloque **`catch`** asociado podría atraparla. Este proceso puede ocurrir a través de muchos niveles de bloques **`try`**. Además, la excepción podría quedar sin atraparse (como vimos en la sección 11.3).

Si un bloque **`catch`** lanza una excepción, el bloque **`finally`** se sigue ejecutando. Luego la excepción se pasa al siguiente bloque **`try`** exterior, que de nuevo suele ser el método que hizo la llamada.

Liberar recursos en un bloque `finally`

Como un bloque **`finally`** casi siempre se ejecuta, por lo general contiene código para liberar recursos. Suponga que se asigna un recurso en un bloque **`try`**. Si no ocurre una excepción, se ignoran los bloques **`catch`** y el control pasa al bloque **`finally`**, que libera el recurso. Después, el control pasa a la primera instrucción después del bloque **`finally`**. Si ocurre una excepción en el bloque **`try`**, éste termina. Si el programa atrapa la excepción en uno de los bloques **`catch`** correspondientes, procesa la excepción, después el bloque **`finally`** libera el recursoo y el control pasa a la primera instrucción después del bloque **`finally`**. Si el programa no atrapa la excepción, el bloque **`finally`** de todas formass libera el recurso y se hace un intento por atrapar la excepción en uno de los métodos que hacen la llamada.

Tip para prevenir errores 11.5

El bloque `finally` es un lugar ideal para liberar los recursos adquiridos en un bloque `try` (como los archivos abiertos), lo cual ayuda a eliminar fugas de recursos.

Tip de rendimiento 11.1

Siempre debe liberar cada recurso de manera explícita y lo antes posible, una vez que ya no sea necesario. Esto hace que los recursos estén disponibles para que su programa los reutilice lo más pronto posible, con lo cual se mejora la utilización de recursos y el rendimiento de los programas.

Demostración del bloque `finally`

La figura 11.5 demuestra que el bloque **`finally`** se ejecuta, aun cuando *no* se lance una excepción en el bloque **`try`** correspondiente. El programa contiene los métodos **`static main`** (líneas 6 a 18), **`lanzaExcepcion`** (líneas 21 a 44) y **`noLanzaExcepcion`** (líneas 47 a 64). Los métodos **`lanzaExcepcion`** y **`noLanzaExcepcion`** se declaran como **`static`**, por lo que **`main`** puede llamarlos directamente sin instanciar un objeto **`UsoDeExcepciones`**.

```
 1. // Fig. 11.5: UsoDeExcepciones.java
 2. // El mecanismo de manejo de excepciones try...catch...finally.
 3.
 4. public class UsoDeExcepciones1
 5. {
 6.     public static void main(String[] args)
 7.     {
 8.         try
 9.         {
10.             lanzaExcepcion();
11.         }
12.         catch (Exception excepcion) // excepción lanzada por ThrowException
13.         {
14.             System.err.println("La excepcion se manejo en main");
15.         }
16.
17.         noLanzaExcepcion();
18.     }
19.
20.     // demuestra los bloques try...catch...finally
21.     public static void lanzaExcepcion() throws Exception
22.     {
23.         try // lanza una excepción y la atrapa de inmediato
24.         {
25.             System.out.println("Metodo lanzaExcepcion");
26.             throw new Exception(); // genera la excepción
27.         }
28.         catch (Exception excepcion) // atrapa la excepción lanzada en el bloque try
29.         {
30.             System.err.println(
31.                 "La excepcion se manejo en el metodo lanzaExcepcion" );
32.             throw excepcion; // vuelve a lanzar para procesarla más adelante
33.
34.             // no se llegaría al código que se coloque aquí; se producirían errores de compilación
35.
36.         }
37.         finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
38.         {
39.             System.err.println("Se ejecuto finally en lanzaExcepcion");
40.         }
41.
42.         // no se llegaría al código que se coloque aquí; se producirían errores de compilación
43.
44.     }
45.
46.     // demuestra el uso de finally cuando no ocurre una excepción
47.     public static void noLanzaExcepcion()
48.     {
49.         try // el bloque try no lanza una excepción
50.         {
51.             System.out.println("Metodo noLanzaExcepcion");
52.         } // fin de try
53.         catch (Exception excepcion) // no se ejecuta
54.         {
55.             System.err.println(excepcion);
56.         }
57.         finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
58.         {
59.             System.err.println(
60.                 "Se ejecuto finally en noLanzaExcepcion");
61.         }
62.
63.         System.out.println("Fin del metodo noLanzaExcepcion");
64.     }
65. } // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
La excepcion se manejo en el metodo lanzaExcepcion
Se ejecuto finally en lanzaExcepcion
La excepcion se manejo en main
Metodo noLanzaExcepcion
Se ejecuto finally en noLanzaExcepcion
Fin del metodo noLanzaExcepcion
```

Figura 11.5 Mecanismo de manejo de excepciones **`try`** ... **`catch`** ... **`finally`**

Tanto **`System.out`** como **`System.err`** son **flujos**; es decir, una secuencia de bytes. Mientras que **`System.out`** (conocido como **flujo de salida estándar**) muestra la salida de un programa, **`System.err`** (conocido como **flujo de error estándar**) muestra los errores de un programa. La salida de estos flujos se puede *redirigir* (es decir, enviar a otra parte que no sea el *símbolo del sistema*, como a un *archivo*). El uso de dos flujos distintos permite al programador *separar* fácilmente los mensajes de error de cualquier otra salida. Por ejemplo, los datos que se imprimen de **`System.err`** se podrían enviar a un archivo de registro, mientras que los que se imprimen de **`System.out`** se podrían mostrar en la pantalla. Para simplificar, en este capítulo no redirigiremos la salida de **`System.err`**, sino que mostraremos dichos mensajes en el *símbolo del sistema*. En el capítulo 15 aprenderá más acerca de los flujos.

Lanzamiento de excepciones mediante la instrucción throw

El método main (figura 11.5) empieza a ejecutarse, entra a su bloque **`try`** y de inmediato llama al método **`lanzaExcepcion`** (línea 10). El método **`lanzaExcepcion`** lanza una excepción tipo **`Exception`**. La instrucción en la línea 26 se conoce como instrucción **`throw`** y se ejecuta para indicar que ocurrió una excepción. Hasta ahora sólo hemos atrapado las excepciones que lanzan los métodos que son llamados. Los programadores pueden lanzar excepciones mediante el uso de la instrucción **`throw`**. Al igual que con las excepciones lanzadas por los métodos de la API de Java, esto indica a las aplicaciones cliente que ocurrió un error. Una instrucción **`throw`** especifica que un objeto se lanzará. El operando de **`throw`** puede ser de cualquier clase derivada de **`Throwable`**.

Observación de ingeniería de software 11.8

Cuando se invoca el método `toString` en cualquier objeto `Throwable`, su objeto `String` resultante incluye la cadena descriptiva que se suministró al constructor, o simplemente el nombre de la clase, si no se suministró una cadena.

Observación de ingeniería de software 11.9

Una excepción puede lanzarse sin contener información acerca del problema que ocurrió. En este caso, el simple conocimiento de que ocurrió una excepción de cierto tipo puede proporcionar suficiente información para que el manejador procese el problema en forma correcta.

Observación de ingeniería de software 11.10

Lance excepciones desde los constructores para indicar que los parámetros del constructor no son válidos. Esto evita que se cree un objeto en un estado inválido.

Relanzamiento de excepciones

La línea 32 de la figura 11.5 **vuelve a lanzar la excepción**. Las excepciones se vuelven a lanzar cuando un bloque **`catch`**, al momento de recibir una excepción, decide que no puede procesar la excepción o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción, se difiere el manejo de la misma (o tal vez una porción de ella) hacia otro bloque **`catch`** asociado con una instrucción **`try`** exterior. Para volver a lanzar una excepción se utiliza la **palabra clave `throw`**, seguida de una referencia al objeto excepción que se acaba de atrapar. Las excepciones no se pueden volver a lanzar desde un bloque **`finally`**, ya que el parámetro de la excepción (una variable local) del bloque **`catch`** ha dejado de existir.

Cuando se vuelve a lanzar una excepción, el *siguiente bloque* **`try`**y *circundante* la detecta, y la instrucción **`catch`** de ese bloque **`try`** trata de manejarla. En este caso, el siguiente bloque **`try`** circundante se encuentra en las líneas 8 a 11 en el método **`main`**. Sin embargo, antes de manejar la excepción que se volvió a lanzar, se ejecuta el bloque **`finally`** (líneas 37 a 40). Después, el método **`main`** detecta la excepción que se volvió a lanzar en el bloque **`try`**, y la maneja en el bloque **`catch`** (líneas 12 a 15).

A continuación, **`main`** llama al método **`noLanzaExcepcion`** (línea 17). Como no se lanza una excepción en el bloque **`try`** de **`noLanzaExcepcion`** (líneas 49 a 52), el programa ignora el bloque **`catch`** (líneas 53 a 56), pero el bloque **`finally`** (líneas 57 a 61) se ejecuta de todas formas. El control pasa a la instrucción que está después del bloque **`finally`** (línea 63). Después, el control regresa a **`main`** y el programa termina.

Error común de programación 11.4

*Si no se ha atrapado una excepción cuando el control entra a un bloque **`finally`**, y éste lanza una excepción que no se atrapa en el bloque **`finally`**y, se perderá la primera excepción y se devolverá la del bloque **`finally`** al método que hizo la llamada.*

Tip para prevenir errores 11.6

*Evite colocar código que pueda lanzar una excepción en un bloque **`finally`**. Si se requiere dicho código, enciérrelo en bloques **`try`**...**`catch`** dentro del bloque **`finally`**.*

Error común de programación 11.5

Suponer que una excepción lanzada desde un bloque **`catch`** *se procesará por ese bloque* **`catch`***, o por cualquier otro bloque* **`catch`** *asociado con la misma instrucción* **`try`***, puede provocar errores lógicos.*

Buena práctica de programación 11.1

El mecanismo de manejo de excepciones de Java está diseñado para eliminar el código de procesamiento de errores de la línea principal del código de un programa, para así mejorar su legibilidad. No coloque bloques **`try`***...* **`catch`***...* **`finally`** *alrededor de cada instrucción que pueda lanzar una excepción. Esto dificulta la legibilidad de los programas. En vez de ello, coloque un bloque* **`try`** *y alrededor de una porción considerable de su código, y después de ese bloque* **`try`** *y coloque bloques* **`catch`** *para manejar cada posible excepción, y después de esos bloques* **`catch`** *coloque un solo bloque* **`finally`** *y (si se requiere).*

11.7 Limpieza de la pila y obtención de información de un objeto excepción

Cuando se lanza una excepción, pero *no se atrapa* en un alcance específico, la pila de llamadas a métodos se "limpia" y se hace un intento de atrapar (**`catch`**) la excepción en el siguiente bloque **`try`** exterior. A este proceso se le conoce como **limpieza de la pila**. Limpiar la pila de llamadas a métodos significa que el método en el que no se atrapó la excepción *termina*, que todas las variables locales en ese método *quedan fuera de alcance* y que el control regresa a la instrucción que invocó originalmente a ese método. Si un bloque **`try`** encierra a esa instrucción, se hace un intento de atrapar esa excepción. Si un bloque **`try`** no encierra a esa instrucción o si no se atrapa la excepción, se lleva a cabo otra vez la limpieza de la pila. La figura 11.6 demuestra la limpieza de la pila, y el manejador de excepciones en main muestra cómo acceder a los datos en un objeto excepción.

Limpieza de la pila

En **`main`**, el bloque **`try`** (líneas 8 a 11) llama a **`metodo1`** (declarado en las líneas 35 a 38), el cual a su vez llama a **`metodo2`** (declarado en las líneas 41 a 44), que a su vez llama a **`metodo3`** (declarado en las líneas 47 a 50). La línea 49 de **`metodo3`** lanza un objeto **`Exception`**: éste es el *punto de lanzamiento*. Puesto que la instrucción **`throw`** en la línea 49 no está encerrada en un bloque **`try`**, se produce la *limpieza de la pila*; **`metodo3`** termina en la línea 49 y después devuelve el control a la instrucción en **`metodo2`** que invocó a **`metodo3`** (es decir, la línea 43). Debido a que *ningún* bloque **`try`** encierra la línea 43, se produce la *limpieza de la pila* otra vez; **`metodo2`** termina en la línea 43 y devuelve el control a la instrucción en **`metodo1`** que invocó a **`metodo2`** (es decir, la línea 37). Como ningún bloque **`try`** encierra la línea 37, se produce una vez más la *limpieza de la pila*; **`metodo1`** termina en la línea 37 y devuelve el control a la instrucción en **`main`** que invocó a **`metodo1`** (la línea 10). El bloque **`try`** de las líneas 8 a 11 encierra a esta instrucción. Puesto que no se manejó la excepción, el bloque **`try`** termina y el primer bloque **`catch`** concordante (líneas 12 a 31) atrapa y procesa la excepción. Si no hubiera bloques **`catch`** que coincidieran, y la excepción *no se declara* en cada método que la lanza, se produciría un error de compilación. Recuerde que éste no es siempre el caso; para las excepciones *no verificadas* la aplicación se compilará, pero se ejecutará con resultados inesperados.

```
 1. // Fig. 11.6: UsoDeExcepciones.java
 2. // Limpieza de la pila y obtención de datos de un objeto excepción.
 3.
 4. public class UsoDeExcepciones
 5. {
 6.     public static void main(String[] args)
 7.     {
 8.         try
 9.         {
10.            metodo1();
11.         }
12.         catch (Exception excepcion) // atrapa la excepción lanzada en metodo1
13.         {
14.             System.err.printf("%s\n\n", excepcion.getMessage());
15.             excepcion.printStackTrace();
16.
17.             // obtiene la información de rastreo de la pila
18.             StackTraceElement[] elementosRastreo = excepcion.getStackTrace();
```

```
19.
20.             System.out.println("\nRastreo de la pila de getStackTrace:\n");
21.             System.out.println( "Clase\t\t\tArchivo\t\t\tLinea\tMetodo");
22.             // itera a través de elementosRastreo para obtener la descripción de la
23.             // excepción
24.             for (StackTraceElement elemento : elementosRastreo)
25.             {
26.                 System.out.printf("%s\t", elemento.getClassName());
27.                 System.out.printf("%s\t", elemento.getFileName());
28.                 System.out.printf("%s\t", elemento.getLineNumber());
29.                 System.out.printf("%s\n", elemento.getMethodName());
30.             }
31.         }
32.     } // fin de main
33.
34.     // llama a metodo2; lanza las excepciones de vuelta a main
35.     public static void metodo1() throws Exception
36.     {
37.         metodo2();
38.     }
39.
40.     // llama a metodo3; lanza las excepciones de vuelta a metodo1
41.     public static void metodo2() throws Exception
42.     {
43.         metodo3();
44.     }
45.
46.     // lanza la excepción Exception de vuelta a metodo2
47.     public static void metodo3() throws Exception
48.     {
49.         throw new Exception("La excepcion se lanzo en metodo3");
50.     }
51. } // fin de la clase UsoDeExcepciones

La excepcion se lanzo en metodo3

java.lang.Exception: La excepcion se lanzo en metodo3
at UsoDeExcepciones.metodo3(UsoDeExcepciones.java:49)
at UsoDeExcepciones.metodo2(UsoDeExcepciones.java:43)
at UsoDeExcepciones.metodo1(UsoDeExcepciones.java:37)
at UsoDeExcepciones.main(UsoDeExcepciones.java:10)

Rastreo de la pila de getStackTrace:
Clase              Archivo               Linea  Metodo
UsoDeExcepciones   UsoDeExcepciones.java  49    metodo3
UsoDeExcepciones   UsoDeExcepciones.java  43    metodo2
UsoDeExcepciones   UsoDeExcepciones.java  37    metodo1
UsoDeExcepciones   UsoDeExcepciones.java  10    main
```

Figura 11.6 Limpieza de la pila y obtención de datos de un objeto excepción

Obtención de datos de un objeto excepción

Recuerde que las excepciones se derivan de la clase **`Throwable`**, la cual ofrece un método llamado **`printStackTrace`** que envía al flujo de error estándar el rastreo de la pila a (lo cual se describe en la sección 11.2). A menudo, esto ayuda en la prueba y en la depuración. La clase **`Throwable`** también proporciona un método llamado **`getStackTrace`**, que obtiene la información de rastreo de la pila que podría imprimir **`printStackTrace`**. El método **`getMessage`** de la clase **`Throwable`** devuelve la cadena descriptiva almacenada en una excepción.

Tip para prevenir errores 11.7

Una excepción que no sea atrapada en una aplicación hará que se ejecute el manejador de excepciones predeterminado de Java. Éste muestra el nombre de la excepción, un mensaje descriptivo que indica el problema que ocurrió y un rastreo completo de la pila de ejecución. En una aplicación con un solo hilo de ejecución, la aplicación termina. En una aplicación con varios hilos, termina el hilo que produjo la excepción. En el capítulo 23 hablaremos sobre la tecnología multihilos.

Tip para prevenir errores 11.8

El método toString de `Throwable` (heredado en todas las subclases de `Throwable`) devuelve un objeto `String` que contiene el nombre de la clase de la excepción y un mensaje descriptivo.

El manejador de **`catch`** en la figura 11.6 (líneas 12 a 31) demuestra el uso de **`getMessage`**, **`printStackTrace`** y **`getStackTrace`**. Si queremos mostrar la información de rastreo de la pila a flujos que no sean el flujo de error estándar, podemos utilizar la información devuelta por **`getStackTrace`** y enviar estos datos a otro flujo, o usar las versiones sobrecargadas del método **`printStackTrace`**. En el capítulo 15 veremos cómo enviar datos a otros flujos.

En la línea 14 se invoca al método **`getMessage`** de la excepción, para obtener la descripción de la misma. En la línea 15 se invoca al método **`printStackTrace`** de la excepción, para mostrar el rastreo de la pila, el cual indica en dónde ocurrió la excepción. En la línea 18 se invoca al método **`getStackTrace`** de la excepción, para obtener la información del rastreo de la pila como un arreglo de objetos **`StackTraceElement`**. En las líneas 24 a 30 se obtiene cada uno de los objetos **`StackTraceElement`** en el arreglo, y se invocan sus métodos **`getClassName`**, **`getFileName`**,

getLineNumber y **getMethodName** para obtener el nombre de la clase, el nombre del archivo, el número de línea y el nombre del método, respectivamente, para ese objeto **StackTraceElement**. Cada objeto **StackTraceElement** representa la llamada a un método en la pila de llamadas a métodos.

Los resultados del programa muestran que la información de rastreo de la pila que imprime **printStackTrace** sigue el patrón: *nombreClase.nombreMétodo(nombreArchivo:númeroLínea)*, en donde *nombreClase*, *nombreMétodo* y *nombreArchivo* indican los nombres de la clase, el método y el archivo en los que ocurrió la excepción, respectivamente, mientras que *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 11.2. El método **getStackTrace** permite un procesamiento personalizado de la información sobre la excepción. Compare la salida de **printStackTrace** con la salida creada a partir de los objetos **StackTraceElement**, y podrá ver que ambos contienen la misma información de rastreo de la pila.

Observación de ingeniería de software 11.11

A veces tal vez sea conveniente ignorar una excepción escribiendo un manejador `catch` *con un cuerpo vacío. Antes de hacerlo, asegúrese de que la excepción no indique una condición que un código más arriba en la jerarquía necesite conocer o de la cual deba recuperarse.*

11.8 Excepciones encadenadas

Algunas veces un método responde a una excepción lanzando un tipo distinto de excepción, específico para la aplicación actual. Si un bloque **catch** lanza una nueva excepción, se *pierden* tanto la información como el rastreo de la pila de la excepción original. En las primeras versiones de Java, no había mecanismo para envolver la información de la excepción original con la de la nueva excepción para proporcionar un rastreo completo de la pila e indicando en dónde ocurrió el problema original en el programa. Esto hacía que depurar dichos problemas fuera un proceso bastante difícil. Las **excepciones encadenadas** permiten que un objeto excepción mantenga la información completa sobre el rastreo de la pila de la excepción original. En la figura 11.7 se demuestran las excepciones encadenadas.

```
 1. // Fig. 11.7: UsoDeExcepcionesEncadenadas.java
 2. // Las excepciones encadenadas.
 3.
 4. public class UsoDeExcepcionesEncadenadas
 5. {
 6.     public static void main(String[] args)
 7.     {
 8.         try
 9.         {
10.             metodo1();
11.         }
12.         catch (Exception excepcion) // excepciones lanzadas desde metodo1
13.         {
14.             excepcion.printStackTrace();
15.         }
16.     }
17.
18.     // llama a metodo2; lanza las excepciones de vuelta a main
19.     public static void metodo1() throws Exception
20.     {
21.         try
22.         {
23.             metodo2();
24.         } // fin de try
25.         catch (Exception excepcion ) // excepción lanzada desde metodo2
26.         {
27.             throw new Exception("La excepcion se lanzo en metodo1", excepcion);
28.         }
29.     }
30.
31.     // llama a metodo3; lanza las excepciones de vuelta a metodo1
32.     public static void metodo2() throws Exception
33.     {
34.         try
35.         {
36.             metodo3();
37.         }
38.         catch (Exception excepcion) // excepción lanzada desde metodo3
39.         {
40.             throw new Exception("La excepcion se lanzo en metodo2", excepcion);
41.         }
42.     }
43.
44.     // lanza excepción Exception de vuelta a metodo2
45.     public static void metodo3() throws Exception
46.     {
47.         throw new Exception("La excepcion se lanzo en metodo3");
48.     }
49. } // fin de la clase UsoDeExcepcionesEncadenadas
```

```
java.lang.Exception: La excepcion se lanzo en metodo1
at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:27)
at UsoDeExcepcionesEncadenadas.main(UsoDeExcepcionesEncadenadas.java:10)
Caused by: java.lang.Exception: La excepcion se lanzo en metodo2
at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:40)
at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:23)
... 1 more
```

```
Caused by: java.lang.Exception: La excepcion se lanzo en metodo3
at UsoDeExcepcionesEncadenadas.metodo3(UsoDeExcepcionesEncadenadas.java:47)
at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:36)
... 2 more
```

Figura 11.7 Excepciones encadenadas

Flujo de control del programa

El programa consiste de cuatro métodos: **`main`** (líneas 6 a 16), **`metodo1`** (líneas 19 a 29), **`metodo2`** (líneas 32 a 42) y **`metodo3`** (líneas 45 a 48). La línea 10 en el bloque **`try`** de **`main`** llama a **`metodo1`**. La línea 23 en el bloque **`try`** de **`metodo1`** llama a **`metodo2`**. La línea 36 en el bloque **`try`** de **`metodo2`** llama a **`metodo3`**. En **`metodo3`**, la línea 47 lanza una nueva excepción **`Exception`**. Como esta instrucción no se encuentra dentro de un bloque **`try`**, el **`metodo3`** termina y la excepción se devuelve al método que hace la llamada (**`metodo2`**), en la línea 36. Esta instrucción se encuentra dentro de un bloque **`try`**; por lo tanto, el bloque **`try`** termina y la excepción es atrapada en las líneas 38 a 41. En la línea 40, en el bloque **`catch`**, se lanza una nueva excepción. En este caso, se hace una llamada al constructor **`Exception`** con *dos* argumentos). El segundo argumento representa a la excepción que era la causa original del problema. En este programa, la excepción ocurrió en la línea 47. Como se lanza una excepción desde el bloque **`catch`**, el **`metodo2`** termina y devuelve la nueva excepción al método que hace la llamada (**`metodo1`**), en la línea 23. Una vez más, esta instrucción se encuentra dentro de un bloque **`try`**, por lo tanto este bloque termina y la excepción es atrapada en las líneas 25 a 28. En la línea 27, en el bloque **`catch`** se lanza una nueva excepción y se utiliza la excepción que se atrapó como el segundo argumento para el constructor de **`Exception`**. Puesto que se lanza una excepción desde el bloque **`catch`**, el **`metodo1`** termina y devuelve la nueva excepción al método que hace la llamada (**`main`**), en la línea 10. El bloque **`try`** en main termina y la excepción es atrapada en las líneas 12 a 15. En la línea 14 se imprime un rastreo de la pila.

Salida del programa

Observe en la salida del programa que las primeras tres líneas muestran la excepción más reciente que fue lanzada (es decir, la del **`metodo1`** en la línea 27). Las siguientes cuatro líneas indican la excepción que se lanzó desde el **`metodo2`**, en la línea 40. Por último, las siguientes cuatro líneas representan la excepción que se lanzó desde el **`metodo3`**, en la línea 47. Además, observe que, si lee la salida en forma inversa, ésta muestra cuántas excepciones encadenadas más quedan pendientes.

11.9 Declaración de nuevos tipos de excepciones

Para crear aplicaciones de Java, la mayoría de los programadores de Java utilizan las clases *existentes* de la API de Java, de distribuidores independientes y de bibliotecas de clases gratuitas (que por lo general se pueden descargar de Internet). Los métodos de esas clases suelen declararse para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, de modo que sus programas sean más robustos.

Si usted crea clases que otros programadores utilizarán en sus programas, tal vez le sea conveniente declarar sus propias clases de excepciones que sean específicas para los problemas que pueden ocurrir cuando otro programador utilice sus clases reutilizables.

Un nuevo tipo de excepción debe extender a uno existente

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. Una clase de excepción es igual que cualquier otra clase; sin embargo, una nueva clase común de excepción contiene sólo cuatro constructores:

- Uno que no toma argumentos y pasa un objeto String como mensaje de error predeterminado al constructor de la superclase
- Uno que recibe un mensaje de error personalizado como un objeto String y lo pasa al constructor de la superclase

- Uno que recibe un mensaje de error personalizado como un objeto **`String`** y un objeto **`Throwable`** (para encadenar excepciones), y pasa ambos objetos al constructor de la superclase
- Uno que recibe un objeto **`Throwable`** (para encadenar excepciones) y pasa sólo este objeto al constructor de la superclase.

Buena práctica de programación 11.2

Asociar cada uno de los tipos de fallas graves en tiempo de ejecución con una clase `Exception` *con nombre apropiado, ayuda a mejorar la claridad del programa.*

Observación de ingeniería de software 11.12

Al definir su propio tipo de excepción, estudie las clases de excepción existentes en la API de Java y trate de extender una clase de excepción relacionada. Por ejemplo, si va a crear una nueva clase para representar cuando un método intenta realizar una división entre cero, podría extender la clase ***`ArithmeticException`****, ya que la división entre cero ocurre durante la aritmética. Si las clases existentes no son superclases apropiadas para su nueva clase de excepción, debe decidir si su nueva clase debe ser una clase de excepción verificada o no verificada. Si a los clientes se les pide manejar la excepción, la nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a* ***`Exception`*** *pero no a* ***`RuntimeException`****). La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. Si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada), la nueva clase de excepción debe extender a* ***`RuntimeExcepcion`****.*

Ejemplos de una clase de excepción personalizada

En el capítulo 21 en línea, Custom Generic Data Structures, proporcionaremos un ejemplo de una clase de excepción personalizada. Declararemos una clase reutilizable llamada **`Lista`**, la cual es capaz de almacenar una lista de referencias a objetos. Algunas operaciones que se realizan comúnmente en una Lista, como eliminar un elemento de la parte frontal o posterior de la lista, no se permitirán si la **`Lista`** está vacía. Por esta razón, algunos métodos de **`Lista`** lanzan excepciones de la clase de excepción **`ListaVaciaException`**.

Buena práctica de programación 11.3

Por convención, todos los nombres de las clases de excepciones deben terminar con la palabra `Exception`*.*

11.10 Precondiciones y poscondiciones

Los programadores invierten una gran parte de su tiempo en mantener y depurar código. Para facilitar estas tareas y mejorar el diseño en general, ellos comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

Precondiciones

Una **precondición** debe ser verdadera cuando se *invoca* a un método. Las precondiciones describen las restricciones en los parámetros de un método, y en cualquier otra expectativa que tenga el método en relación con el estado actual de un programa, *justo antes de empezar a ejecutarse*. Si no se cumplen las precondiciones, entonces el comportamiento del método es *indefinido*; puede *lanzar una excepción, continuar con un valor ilegal o tratar de recuperarse* del error. Nunca hay que esperar un comportamiento consistente si no se cumplen las precondiciones.

Poscondiciones

Una **poscondición** es verdadera *una vez que el método regresa con éxito*. Las poscondiciones describen las *restricciones en el valor de retorno*, así como cualquier otro *efecto secundario* que pueda tener el método. Al definir un método, usted debe documentar todas las poscondiciones, de manera que otros sepan qué pueden esperar al llamar a su método, y debe asegurarse que su método cumpla con todas sus poscondiciones, si en definitiva se cumplen sus precondiciones.

Lanzamiento de excepciones cuando no se cumplen las precondiciones o poscondiciones

Cuando no se cumplen sus precondiciones o poscondiciones, los métodos por lo general lanzan excepciones. Como ejemplo, examine el método **`charAt`** de **`String`**, que tiene un parámetro int: un índice en el objeto **`String`**. Para una precondición, el método **`charAt`** asume que indice es mayor o igual que cero, y menor que la longitud del objeto **`String`**. Si se cumple la precondición, ésta establece que el método devolverá el carácter en la posición en el objeto **`String`** especificada por el parámetro indice. En caso contrario, el método lanza una excepción **`IndexOutOfBoundsException`**. Confiamos en que el método **`charAt`** satisfaga su poscondición, siempre y cuando cumplamos con la precondición. No necesitamos preocuparnos por los detalles acerca de cómo el método en realidad obtiene el carácter en el índice.

Por lo general, las precondiciones y poscondiciones de un método se describen como parte de su especificación. Al diseñar sus propios métodos, debe indicar las precondiciones y poscondiciones en un comentario antes de la declaración del método.

11.11 Aserciones

Al implementar y depurar una clase, algunas veces es conveniente establecer condiciones que deban ser verdaderas en un punto específico de un método. Estas condiciones, conocidas como **aserciones**, ayudan a asegurar la validez de un programa al atrapar los errores potenciales e identificar los posibles errores lógicos durante el desarrollo. Las precondiciones y las poscondiciones son dos tipos de aserciones. Las precondiciones son aserciones sobre el estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones sobre el estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo del programa, Java incluye dos versiones de la instrucción **`assert`** para validar aserciones mediante la programación. La instrucción **`assert`** evalúa una expresión **`boolean`** y, si es **`false`**, lanza una excepción **`AssertionError`** (una subclase de **`Error`**). La primera forma de la instrucción **`assert`** es

`assert expresión;`

la cual lanza una excepción **`AssertionError`** si expresión es **`false`**. La segunda forma es

`assert expresión1 : expresión2;`

que evalúa *expresión1* y lanza una excepción **`AssertionError`** con *expresión2* como el mensaje de error, en caso de que *expresión1* sea **`false`**.

Puede utilizar aserciones para implementar las *precondiciones* y *poscondiciones* mediante la programación, o para verificar cualquier otro estado intermedio que le ayude a asegurar que su código esté funcionando en forma correcta. La figura 11.8 demuestra la instrucción assert. En la línea 11 se pide al usuario que introduzca un número entre 0 y 10, y después en la línea 12 se lee el número. La línea 15 determina si el usuario introdujo un número dentro del rango válido. Si el número está fuera de rango, la instrucción **`assert`** reporta un error; en caso contrario, el programa continúa en forma normal.

```
1. // Fig. 11.8: PruebaAssert.java
2. // Comprobar mediante assert que un valor esté dentro del rango.
3. import java.util.Scanner;
4.
5. public class PruebaAssert
6. {
7.     public static void main(String[] args)
8.     {
9.         Scanner entrada = new Scanner(System.in);
10.
11.        System.out.print("Escriba un numero entre 0 y 10: ");
12.        int numero = entrada.nextInt();
13.
14.        // asegura que el valor sea >= 0 y <= 10
15.        assert (numero >= 0 && numero <= 10) : "numero incorrecto: " + numero;
```

```
16.
17.         System.out.printf("Usted escribio %d\n", numero);
18.      }
19. } // fin de la clase PruebaAssert


Escriba un numero entre 0 y 10: 5
Usted escribio 5


Escriba un numero entre 0 y 10: 50
Exception in thread ``main'' java.lang.AssertionError: numero incorrecto: 50
at PruebaAssert.main(PruebaAssert.java:15)
```

Figura 11.8 Comprobar mediante **`assert`** que un valor esté dentro del rango.

El programador utiliza las aserciones principalmente para depurar e identificar errores lógicos en una aplicación. Hay que habilitar las aserciones de manera explícita al ejecutar un programa, ya que reducen el rendimiento y son innecesarias para el usuario del mismo. Para ello, use la opción de línea de comandos **`-ea`** del comando **`java`**, como en

```
java -ea PruebaAssert
```

Observación de ingeniería de software 11.13

Los usuarios no deben encontrar ningún error tipo `AssertionError`; éstos deben usarse sólo durante el desarrollo del programa. Por esta razón, nunca se debe atrapar una excepción tipo `AssertionError`. En vez de ello, debemos permitir que el programa termine para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. No debemos usar la instrucción assertt para indicar problemas en tiempo de ejecución en el código de producción (como lo hicimos en la figura 11.8 para fines demostrativos); debemos usar el mecanismo de las excepciones para este fin.

11.12 Cláusula `try` con recursos: desasignación automática de recursos

Por lo general, el *código para liberar recursos* debe colocarse en un bloque **`finally`**, para asegurar que se libere un recurso sin importar que se hayan lanzado excepciones cuando se utilizó ese recurso en el bloque **`try`** correspondiente. Hay una notación alternativa, la instrucción **`try` con recursos** (que se introdujo en Java SE 7), la cual simplifica la escritura de código en el que uno o más recursos se obtienen, se utilizan en un bloque **`try`** y se liberan en el correspondiente bloque **`finally`**. Por ejemplo, una aplicación de procesamiento de archivos podría procesar un archivo con una instrucción **`try`** con recursos, para asegurar que el archivo se cierre de manera apropiada cuando ya no se necesite; demostraremos esto en el capítulo 15. Cada recurso debe ser un objeto de una clase que implemente a la interfaz **`Autocloseable`**, y por ende proporciona un método llamado **`close`**. La forma general de una instrucción **`try`** con recursos es:

```
try (NombreClasee elObjeto = new NombreClase())
{
    // aquí se usa elObjeto
}
catch (Exception e)
{
    // atrapa las excepciones que ocurren al usar el recurso
}
```

en donde *NombreClase* es una clase que implementa a la interfaz **`Autocloseable`**. Este código crea un objeto de tipo *NombreClasee* y lo utiliza en el bloque **`try`**, después llama a su método **`close`** para liberar los recursos utilizados por el objeto. La instrucción **`try`** con recursos llama de manera *implícita* al método **`close`** de *elObjeto* al *final del bloque* **`try`**. Usted puede asignar varios recursos en los paréntesis que van después de **`try`**, separándolos con un signo de punto y coma (;). En los capítulos 15 y 24 veremos ejemplos de la instrucción **`try`** con recursos.

11.13 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores. Aprendió que el manejo de excepciones permite a los programadores eliminar el código para manejar errores

de la “línea principal” de ejecución del programa. Le mostramos cómo utilizar los bloques **`try`** para encerrar código que puede lanzar una excepción, y cómo utilizar los bloques **`catch`** para lidiar con las excepciones que puedan surgir.

Aprendió acerca del modelo de terminación del manejo de excepciones, el cual indica que una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento. Vimos la diferencia entre las excepciones verificadas y no verificadas, y cómo especificar mediante la cláusula throws las excepciones que podría lanzar un método.

Aprendió a utilizar el bloque **`finally`** para liberar recursos, ya sea que ocurra o no una excepción. También aprendió a lanzar y volver a lanzar excepciones. Después, aprendió a obtener información sobre una excepción, mediante el uso de los métodos **`printStackTrace`**, **`getStackTrace`** y **`getMessage`**. Luego le presentamos las excepciones encadenadas, que permiten a los programadores envolver la información de la excepción original con la información de la nueva excepción. Después, le enseñamos a crear sus propias clases de excepciones.

Presentamos las precondiciones y poscondiciones para ayudar a los programadores que utilizan sus métodos a comprender las condiciones que deben ser verdaderas cuando se hace la llamada al método y cuando éste regresa, respectivamente. Cuando no se cumplen las precondiciones y poscondiciones, los métodos por lo general lanzan excepciones. Hablamos sobre la instrucción **`assert`** y cómo puede utilizarse para ayudarnos a depurar los programas. En especial, esta instrucción se puede utilizar para asegurar que se cumplan las precondiciones y poscondiciones.

También le presentamos la cláusula **`catch`** múltiple para procesar varios tipos de excepciones en el mismo manejador **`catch`**, y la instrucción **`try`** con recursos para desasignar de manera automática un recurso después de usarlo en el bloque **`try`**. En el siguiente capítulo veremos un análisis más detallado de las interfaces gráficas de usuario (GUI).