13

Manejo de excepciones

Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.

—Franklin Delano Roosevelt

¡Oh! Arroja la peor parte de ello, y vive en forma más pura con la otra mitad.

—William Shakespeare

Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.

—Jerome David Salinger

¡Oh!.Infinita virtud! ¿Cómo sonríes desde la trampa más grande del mundo sin estar atrapada?

—William Shakespeare

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el manejo de excepciones y de errores.
- Utilizar `try`, `throw` y `catch` para detectar, indicar y manejar excepciones, respectivamente.
- Utilizar el bloque `finally` para liberar recursos.
- Comprender cómo la limpieza de la pila permite que las excepciones que no se atrapan en un alcance, se atrapen en otro.
- Comprender cómo ayuda la pila en la depuración.
- Comprender cómo se ordenan las excepciones en una jerarquía de clases de excepciones.
- Declarar nuevas clases de excepciones.
- Crear excepciones encadenadas que mantengan la información completa del rastreo de la pila.

Plan general

13.1 Introducción

En este capítulo presentaremos el **manejo de excepciones**. Una **excepción** es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre "excepción" implica que el problema ocurre con poca frecuencia; si la "regla" es que una instrucción generalmente se ejecuta en forma correcta, entonces la "excepción a la regla" es cuando ocurre un problema. El manejo de excepciones le permite crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir **programas tolerantes a fallas** y **robustos** (es decir, programas que traten con los problemas que puedan surgir sin dejar de ejecutarse). El estilo y los detalles sobre el manejo de excepciones en Java se basan, en parte, en el trabajo que Andrew Koenig y Bjarne Stroustrup presentaron en su artículo "Exception Handling for C++ (versión revisada)."[1]

Tip para prevenir errores 13.1

El manejo de excepciones ayuda a mejorar la tolerancia a fallas de un programa.

Ya vimos en capítulos anteriores una breve introducción a las excepciones. En el capítulo 7 aprendió que una excepción `ArrayIndexOutOfBoundsException` ocurre cuando hay un intento por acceder a un elemento más allá del fin del arreglo. Dicho problema puede ocurrir si hay un error de "desplazamiento por 1" en una instrucción `for` que manipula un arreglo. En el capítulo 10 presentamos la excepción `ClassCastException`, que ocurre cuando hay un intento por convertir un objeto que no tiene una relación *"es un"* con el tipo especificado en el operador de conversión. En el capítulo 11 hicimos una breve mención de la excepción `NullPointerException`, la cual ocurre cada vez que se utiliza una referencia `null` en donde se espera un objeto (por ejemplo, cuando hay un intento por adjuntar un componente de GUI a un objeto `Container`, pero el componente de GUI no se ha creado todavía). A lo largo de este libro ha utilizado también la clase `Scanner`; que, como veremos en este capítulo, también puede producir excepciones.

El capítulo empieza con un panorama general de los conceptos relacionados con el manejo de excepciones, y posteriormente se demuestran las técnicas básicas para el manejo de excepciones. Mostraremos estas técnicas

1. Koenig, A. y B. Stroustrup. "Exception Handling for C++ (versión revisada)", *Proceedings of the Usenix C++ Conference,* pp. 149-176, San Francisco, abril de 1990.

en acción, mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después del ejemplo, presentaremos varias clases de la parte superior de la jerarquía de clases de Java para el manejo de excepciones. Como verá posteriormente, sólo las clases que extienden a `Throwable` (paquete `java.lang`) en forma directa o indirecta pueden usarse para manejar excepciones. Después hablaremos sobre la característica de las excepciones encadenadas, que permiten a los programadores envolver la información acerca de una excepción que haya ocurrido en otro objeto de excepción, para proporcionar información más detallada acerca de un problema en un programa. Luego hablaremos sobre ciertas cuestiones adicionales sobre el manejo de excepciones, como la forma en que se deben manejar las excepciones que ocurren en un constructor. Presentaremos las precondiciones y poscondiciones, que deben ser verdaderas cuando se hacen llamadas a sus métodos y cuando esos métodos regresan, respectivamente. Por último presentaremos las aserciones, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código.

13.2 Generalidades acerca del manejo de excepciones

Con frecuencia, los programas evalúan condiciones que determinan cómo debe proceder la ejecución. Considere el siguiente seudocódigo:

```
Realizar una tarea
Si la tarea anterior no se ejecutó correctamente
    Realizar el procesamiento de los errores
Realizar la siguiente tarea
Si la tarea anterior no se ejecutó correctamente
    Realizar el procesamiento de los errores
…
```

En este seudocódigo empezamos realizando una tarea; después, evaluamos si esa tarea se ejecutó correctamente. Si no lo hizo, realizamos el procesamiento de los errores. De otra manera, continuamos con la siguiente tarea. Aunque esta forma de manejo de errores funciona, al entremezclar la lógica del programa con la lógica del manejo de errores el programa podría ser difícil de leer, modificar, mantener y depurar; especialmente en aplicaciones extensas.

Tip de rendimiento 13.1

Si los problemas potenciales ocurren con poca frecuencia, al entremezclar la lógica del programa y la lógica del manejo de errores se puede degradar el rendimiento del programa, ya que éste debe realizar pruebas (tal vez con frecuencia) para determinar si la tarea se ejecutó en forma correcta, y si se puede llevar a cabo la siguiente tarea.

El manejo de excepciones permite al programador remover el código para manejo de errores de la "línea principal" de ejecución del programa, lo cual mejora la claridad y capacidad de modificación del mismo. Usted puede optar por manejar las excepciones que elija: todas las excepciones, todas las de cierto tipo o todas las de un grupo de tipos relacionados (por ejemplo, los tipos de excepciones que están relacionados a través de una jerarquía de herencia). Esta flexibilidad reduce la probabilidad de que los errores se pasen por alto y, por consecuencia, hace que los programas sean más robustos.

Con lenguajes de programación que no soportan el manejo de excepciones, los programadores a menudo retrasan la escritura de código de procesamiento de errores, o algunas veces olvidan incluirlo. Esto hace que los productos de software sean menos robustos. Java permite al programador tratar con el manejo de excepciones fácilmente, desde el comienzo de un proyecto.

13.3 Ejemplo: división entre cero sin manejo de excepciones

Demostraremos primero qué ocurre cuando surgen errores en una aplicación que no utiliza el manejo de errores. En la figura 13.1 se pide al usuario que introduzca dos enteros y éstos se pasan al método `cociente`, que calcula el cociente y devuelve un resultado `int`. En este ejemplo veremos que las excepciones se **lanzan** (es decir, la excepción ocurre) cuando un método detecta un problema y no puede manejarlo.

La primera de las tres ejecuciones de ejemplo en la figura 13.1 muestra una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce el valor `0` como denominador. Observe que muestran varias líneas de

información en respuesta a esta entrada inválida. Esta información se conoce como el **rastreo de la pila**, la cual incluye el nombre de la excepción (`java.lang.ArithmeticException`) en un mensaje descriptivo, que indica el problema que ocurrió y la pila de llamadas a métodos completa (es decir, la cadena de llamadas) al momento en que ocurrió la excepción. El rastreo de la pila incluye la ruta de ejecución que condujo a la excepción, método por método. Esta información ayuda a depurar un programa. La primera línea especifica que ha ocurrido una

```
1  // Fig. 13.1: DivisionEntreCeroSinManejoDeExcepciones.java
2  // Una aplicación que trata de realizar una división entre cero.
3  import java.util.Scanner;
4
5  public class DivisionEntreCeroSinManejoDeExcepciones
6  {
7     // demuestra el lanzamiento de una excepción cuando ocurre una división entre cero
8     public static int cociente( int numerador, int denominador )
9     {
10       return numerador / denominador; // posible división entre cero
11    } // fin del método cociente
12
13    public static void main( String args[] )
14    {
15       Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada
16
17       System.out.print( "Introduzca un numerador entero: " );
18       int numerador = explorador.nextInt();
19       System.out.print( "Introduzca un denominador entero: " );
20       int denominador = explorador.nextInt();
21
22       int resultado = cociente( numerador, denominador );
23       System.out.printf(
24          "\nResultado: %d / %d = %d\n", numerador, denominador, resultado );
25    } // fin de main
26 } // fin de la clase DivisionEntreCeroSinManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionEntreCeroSinManejoDeExcepciones.cociente(
           DivisionEntreCeroSinManejoDeExcepciones.java:10)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
           DivisionEntreCeroSinManejoDeExcepciones.java:22)
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:840)
        at java.util.Scanner.next(Scanner.java:1461)
        at java.util.Scanner.nextInt(Scanner.java:2091)
        at java.util.Scanner.nextInt(Scanner.java:2050)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
           DivisionEntreCeroSinManejoDeExcepciones.java:20)
```

Figura 13.1 | División entera sin manejo de excepciones.

excepción `ArithmeticException`. El texto después del nombre de la excepción ("`/ by zero`") indica que esta excepción ocurrió como resultado de un intento de dividir entre cero. Java no permite la división entre cero en la aritmética de enteros. [*Nota:* Java *sí* permite la división entre cero con valores de punto flotante. Dicho cálculo produce como resultado el valor de infinito, que se representa en Java como un valor de punto flotante (pero en realidad aparece como la cadena `Infinity`)]. Cuando ocurre una división entre cero en la aritmética de enteros, Java lanza una excepción **`ArithmeticException`**. Este tipo de excepciones pueden surgir debido a varios problemas distintos en aritmética, por lo que los datos adicionales ("`/ by zero`") nos proporcionan más información acerca de esta excepción específica.

Empezando a partir de la última línea del rastreo de la pila, podemos ver que la excepción se detectó en la línea 22 del método `main`. Cada línea del rastreo de la pila contiene el nombre de la clase y el método (`DivideByZeroNoExceptionHandling.main`) seguido por el nombre del archivo y el número de línea (`DivideByZeroNoExceptionHandling.java:22`). Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en la línea 10, en el método `cociente`. La fila superior de la cadena de llamadas indica el **punto de lanzamiento**: el punto inicial en el que ocurre la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

En la tercera ejecución, el usuario introduce la cadena `"hola"` como denominador. Observe de nuevo que se muestra un rastreo de la pila. Esto nos informa que ha ocurrido una excepción `InputMismatchException` (paquete `java.util`). En nuestros ejemplos anteriores, en donde se leían valores numéricos del usuario, se suponía que éste debía introducir un valor entero apropiado. Sin embargo, algunas veces los usuarios cometen errores e introducen valores no enteros. Una excepción **`InputMismatchException`** ocurre cuando el método `nextInt` de `Scanner` recibe una cadena que no representa un entero válido. Empezando desde el final del rastreo de la pila, podemos ver que la excepción se detectó en la línea 20 del método `main`. Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en el método `nextInt`. Observe que en vez del nombre de archivo y del número de línea, se proporciona el texto `Unknown Source`. Esto significa que la JVM no tiene acceso al código fuente en donde ocurrió la excepción.

Observe que en las ejecuciones de ejemplo de la figura 13.1, cuando ocurren excepciones y se muestran los rastreos de la pila, el programa también termina. Esto no siempre ocurre en Java; algunas veces un programa puede continuar, aun cuando haya ocurrido una excepción y se haya impreso un rastreo de pila. En tales casos, la aplicación puede producir resultados inesperados. En la siguiente sección le mostraremos cómo manejar esas excepciones y mantener el programa ejecutándose sin problema.

En la figura 13.1, ambos tipos de excepciones se detectaron en el método `main`. En el siguiente ejemplo, veremos cómo manejar estas excepciones para permitir que el programa se ejecute hasta terminar de manera normal.

13.4 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`

La aplicación de la figura 13.2, que se basa en la figura 13.1, utiliza el manejo de excepciones para procesar cualquier excepción tipo `ArithmeticException` e `InputMismatchException` que pueda ocurrir. La aplicación todavía pide dos enteros al usuario y los pasa al método `cociente`, que calcula el cociente y devuelve un resultado `int`. Esta versión de la aplicación utiliza el manejo de excepciones de manera que, si el usuario comete un error, el programa **atrapa** y **maneja** (es decir, se encarga de) la excepción; en este caso, le permite al usuario tratar de introducir los datos de entrada otra vez.

```
1  // Fig. 13.2: DivisionEntreCeroConManejoDeExcepciones.java
2  // Un ejemplo de manejo de excepciones que verifica la división entre cero.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivisionEntreCeroConManejoDeExcepciones
7  {
8     // demuestra cómo se lanza una excepción cuando ocurre una división entre cero
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 1 de 3).

```
 9     public static int cociente( int numerador, int denominador )
10        throws ArithmeticException
11     {
12        return numerador / denominador; // posible división entre cero
13     } // fin del método cociente
14
15     public static void main( String args[] )
16     {
17        Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada
18        boolean continuarCiclo = true; // determina si se necesitan más datos de entrada
19
20        do
21        {
22           try // lee dos números y calcula el cociente
23           {
24              System.out.print( "Introduzca un numerador entero: " );
25              int numerador = explorador.nextInt();
26              System.out.print( "Introduzca un denominador entero: " );
27              int denominador = explorador.nextInt();
28
29              int resultado = cociente( numerador, denominador );
30              System.out.printf( "\nResultado: %d / %d = %d\n", numerador,
31                 denominador, resultado );
32              continuarCiclo = false; // entrada exitosa; termina el ciclo
33           } // fin de bloque try
34           catch ( InputMismatchException inputMismatchException )
35           {
36              System.err.printf( "\nExcepcion: %s\n",
37                 inputMismatchException );
38              explorador.nextLine(); // descarta entrada para que el usuario intente otra vez
39              System.out.println(
40                 "Debe introducir enteros. Intente de nuevo.\n" );
41           } // fin de bloque catch
42           catch ( ArithmeticException arithmeticException )
43           {
44              System.err.printf( "\nExcepcion: %s\n", arithmeticException );
45              System.out.println(
46                 "Cero es un denominador invalido. Intente de nuevo.\n" );
47           } // fin de catch
48        } while ( continuarCiclo ); // fin de do...while
49     } // fin de main
50  } // fin de la clase DivisionEntreCeroConManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0

Excepcion: java.lang.ArithmeticException: / by zero
Cero es un denominador invalido. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 2 de 3).

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola

Excepcion: java.util.InputMismatchException
Debe introducir enteros. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Figura 13.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException`. (Parte 3 de 3).

La primera ejecución de ejemplo de la figura 13.2 muestra una ejecución exitosa que no se encuentra con ningún problema. En la segunda ejecución, el usuario introduce un denominador cero y ocurre una excepción `ArithmeticException`. En la tercera ejecución, el usuario introduce la cadena `"hola"` como el denominador, y ocurre una excepción `InputMismatchException`. Para cada excepción, se informa al usuario sobre el error y se le pide que intente de nuevo; después el programa le pide dos nuevos enteros. En cada ejecución de ejemplo, el programa se ejecuta hasta terminar sin problemas.

La clase `InputMismatchException` se importa en la línea 3. La clase `ArithmeticException` no necesita importarse, ya que se encuentra en el paquete `java.lang`. El método `main` (líneas 15 a 49) crea un objeto `Scanner` en la línea 17. En la línea 18 se crea la variable `boolean` llamada `continuarCiclo`, la cual es verdadera si el usuario no ha introducido aún datos de entrada válidos. En las líneas 20 a 48 se pide repetidas veces a los usuarios que introduzcan datos, hasta recibir una entrada válida.

Encerrar código en un bloque `try`

Las líneas 22 a 33 contienen un **bloque `try`**, que encierra el código que podría lanzar (`throw`) una excepción y el código que no debería ejecutarse en caso de que ocurra una excepción (es decir, si ocurre una excepción, se omitirá el resto del código en el bloque `try`). Un bloque `try` consiste en la palabra clave `try` seguida de un bloque de código, encerrado entre llaves (`{}`). [*Nota:* el término "bloque `try`" se refiere algunas veces sólo al bloque de código que va después de la palabra clave `try` (sin incluir a la palabra `try`). Para simplificar, usaremos el término "bloque `try`" para referirnos al bloque de código que va después de la palabra clave `try`, incluyendo esta palabra]. Las instrucciones que leen los enteros del teclado (líneas 25 y 27) utilizan el método `nextInt` para leer un valor `int`. El método `nextInt` lanza una excepción `InputMismatchException` si el valor leído no es un entero válido.

La división que puede provocar una excepción `ArithmeticException` no se ejecuta en el bloque `try`. En vez de ello, la llamada al método `cociente` (línea 29) invoca al código que intenta realizar la división (línea 12); la JVM lanza un objeto `ArithmeticException` cuando el denominador es cero.

Observación de ingeniería de software 13.1

Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque `try`*, a través de llamadas a otros métodos, de llamadas a métodos con muchos niveles de anidamiento, iniciadas por código en un bloque* `try` *o desde la Máquina Virtual de Java, al momento en que ejecute códigos de byte de Java.*

Atrapar excepciones

El bloque `try` en este ejemplo va seguido de dos bloques `catch`: uno que maneja una excepción `InputMismatchException` (líneas 34 a 41) y uno que maneja una excepción `ArithmeticException` (líneas 42 a 47). Un **bloque `catch`** (también conocido como **cláusula `catch`** o **manejador de excepciones**) atrapa (es decir, recibe) y maneja una excepción. Un bloque `catch` empieza con la palabra clave `catch` y va seguido por un parámetro entre paréntesis (conocido como el parámetro de excepción, que veremos en breve) y un bloque de código encerrado entre llaves. [*Nota:* el término "cláusula `catch`" se utiliza algunas veces para hacer referencia a la palabra clave `catch`, seguida de un bloque de código, en donde el término "bloque `catch`" se refiere sólo al bloque de código que va después de la palabra clave `catch`, sin incluirla. Para simplificar, usaremos el término "bloque `catch`" para referirnos al bloque de código que va después de la palabra clave `catch`, incluyendo esta palabra].

Por lo menos un bloque `catch` o un **bloque `finally`** (que veremos en la sección 13.7) debe ir inmediatamente después del bloque `try`. Cada bloque `catch` especifica entre paréntesis un **parámetro de excepción**, que identifica el tipo de excepción que puede procesar el manejador. Cuando ocurre una excepción en un bloque `try`, el bloque `catch` que se ejecuta es aquél cuyo tipo coincide con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de la excepción que se lanzó, o es una superclase de ésta). El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada; por ejemplo, para invocar en forma implícita el método `toString` de la excepción que se atrapó (como en las líneas 37 y 44), que muestra información básica acerca de la excepción. La línea 38 del primer bloque `catch` llama al método `nextLine` de `Scanner`. Como ocurrió una excepción `InputMismatchException`, la llamada al método `nextInt` nunca leyó con éxito los datos del usuario; por lo tanto, leemos esa entrada con una llamada al método `nextLine`. No hacemos nada con la entrada en este punto, ya que sabemos que es inválida. Cada bloque `catch` muestra un mensaje de error y pide al usuario que intente de nuevo. Al terminar alguno de los bloques `catch`, se pide al usuario que introduzca datos. Pronto veremos con más detalle la manera en que trabaja este flujo de control en el manejo de excepciones.

Error común de programación 13.1

Es un error de sintaxis colocar código entre un bloque `try` y su correspondiente bloque `catch`.

Error común de programación 13.2

Cada instrucción `catch` sólo puede tener un parámetro; especificar una lista de parámetros de excepción separada por comas es un error de sintaxis.

Una **excepción no atrapada** ocurre y no hay bloques `catch` que coincidan. En el segundo y tercer resultado de ejemplo de la figura 13.1, vio las excepciones no atrapadas. Recuerde que cuando ocurrieron excepciones en ese ejemplo, la aplicación terminó antes de tiempo (después de mostrar el rastreo de pila de la excepción). Esto no siempre ocurre como resultado de las excepciones no atrapadas. Como aprenderá en el capítulo 23, Subprocesamiento múltiple, Java utiliza un modelo de ejecución de programas con subprocesamiento múltiple. Cada **subproceso** es una actividad paralela. Un programa puede tener muchos subprocesos. Si un programa sólo tiene un subproceso, una excepción no atrapada hará que el programa termine. Si un programa tiene varios subprocesos, una excepción no atrapada terminará sólo el subproceso en el cual ocurrió la excepción. Sin embargo, en dichos programas ciertos subprocesos pueden depender de otros, y si un subproceso termina debido a una excepción no atrapada, puede haber efectos adversos para el resto del programa.

Modelo de terminación del manejo de excepciones

Si ocurre una excepción en un bloque `try` (por ejemplo, si se lanza una excepción `InputMismatchException` como resultado del código de la línea 25 en la figura 13.2), el bloque `try` termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` en los que el tipo del parámetro de excepción coincide con el tipo de la excepción que se lanzó. En la figura 13.2, el primer bloque `catch` atrapa excepciones `InputMismatchException` (que ocurren si se introducen datos de entrada inválidos) y el segundo bloque `catch` atrapa excepciones `ArithmeticException` (que ocurren si hay un intento por dividir entre cero). Una vez que se maneja la excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado (y se han perdido sus variables locales). En vez de ello, el control se reanuda después del último bloque `catch`. Esto se conoce como el **modelo de terminación del manejo de excepciones**. [*Nota:* algunos lenguajes utilizan el **modelo de reanudación del manejo de excepciones** en el que, después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento].

Error común de programación 13.3

Pueden ocurrir errores lógicos si usted supone que después de manejar una excepción, el control regresará a la primera instrucción después del punto de lanzamiento.

Tip para prevenir errores 13.2

Con el manejo de excepciones, un programa puede seguir ejecutándose (en vez de terminar) después de lidiar con un problema. Esto ayuda a asegurar el tipo de aplicaciones robustas que contribuyen a lo que se conoce como computación de misión crítica, o computación crítica para los negocios.

Observe que nombramos a nuestros parámetros de excepción (`inputMismatchException` y `arithmeticException`) en base a su tipo. A menudo, los programadores de Java utilizan simplemente la letra `e` como el nombre de sus parámetros de excepción.

Buena práctica de programación 13.1

El uso del nombre de un parámetro de excepción que refleje el tipo del parámetro fomenta la claridad, al recordar al programador el tipo de excepción que se está manejando.

Después de ejecutar un bloque `catch`, el flujo de control de este programa procede a la primera instrucción después del último bloque `catch` (línea 48 en este caso). La condición en la instrucción `do...while` es `true` (la variable `continuarCiclo` contiene su valor inicial de `true`), por lo que el control regresa al principio del ciclo y se le pide al usuario una vez más que introduzca datos. Esta instrucción de control iterará hasta que se introduzcan datos de entrada válidos. En ese punto, el control del programa llega a la línea 32, en donde se asigna `false` a la variable `continuarCiclo`. Después, el bloque `try` termina. Si no se lanzan excepciones en el bloque `try`, se omiten los bloques `catch` y el control continúa con la primera instrucción después de los bloques `catch` (en la sección 13.7 aprenderemos acerca de otra posibilidad, cuando hablemos sobre el bloque `finally`). Ahora la condición del ciclo `do...while` es `false`, y el método `main` termina.

El bloque `try` y sus correspondientes bloques `catch` y/o `finally` forman en conjunto una **instrucción `try`**. Es importante no confundir los términos "bloque `try`" e "instrucción `try`"; el término "bloque `try`" se refiere a la palabra clave `try` seguida de un bloque de código, mientras que "instrucción `try`" incluye el bloque `try`, así como los siguientes bloques `catch` y/o un bloque `finally`.

Al igual que con cualquier otro bloque de código, cuando termina un bloque `try`, se destruyen las variables locales declaradas en ese bloque. Cuando termina un bloque `catch`, las variables locales declaradas dentro de este bloque (incluyendo el parámetro de excepción de ese bloque `catch`) también quedan fuera de alcance y se destruyen. Cualquier bloque `catch` restante en la instrucción `try` se ignora, y la ejecución se reanuda en la primera línea de código después de la secuencia `try...catch`; ésta será un bloque `finally`, en caso de que haya uno presente.

Uso de la cláusula `throws`

Ahora examinaremos el método `cociente` (figura 13.2; líneas 9 a 13). La porción de la declaración del método ubicada en la línea 10 se conoce como **cláusula `throws`**. Esta cláusula especifica las excepciones que lanza el método. La cláusula aparece después de la lista de parámetros del método y antes de su cuerpo. Contiene una lista separada por comas de las excepciones que lanzará el método, en caso de que ocurra un problema. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o mediante métodos que se llamen desde el cuerpo. Un método puede lanzar excepciones de las clases que se listen en su cláusula `throws`, o en la de sus subclases. Hemos agregado la cláusula `throws` a esta aplicación, para indicar al resto del programa que este método puede lanzar una excepción `ArithmeticException`. Por ende, a los clientes del método `cociente` se les informa que el método puede lanzar una excepción `ArithmeticException`. En la sección 13.6 aprenderá más acerca de la cláusula `throws`.

Tip para prevenir errores 13.3

Si sabe que un método podría lanzar una excepción, incluya el código apropiado para manejar excepciones en su programa, para que sea más robusto.

Tip para prevenir errores 13.4

Lea la documentación de la API en línea para saber acerca de un método, antes de usarlo en un programa. La documentación especifica la excepción lanzada por el método (si la hay), y también indica las razones por las que pueden ocurrir dichas excepciones. Después, incluya el código adecuado para manejar esas excepciones en su programa.

Tip para prevenir errores 13.5

Lea la documentación de la API en línea para buscar una clase de excepción, antes de escribir código para manejar ese tipo de excepciones. Por lo general, la documentación para una clase de excepción contiene las razones potenciales por las que podrían ocurrir dichas excepciones durante la ejecución de un programa.

Cuando se ejecuta la línea 12, si el `denominador` es cero, la JVM lanza un objeto `ArithmeticException`. Este objeto será atrapado por el bloque `catch` en las líneas 42 a 47, que muestra información básica acerca de la excepción, invocando de manera implícita al método `toString` de la excepción, y después pide al usuario que intente de nuevo.

Si el `denominador` no es cero, el método `cociente` realiza la división y devuelve el resultado al punto de la invocación, al método `cociente` en el bloque `try` (línea 29). Las líneas 30 y 31 muestran el resultado del cálculo y la línea 32 establece `continuarCiclo` en `false`. En este caso, el bloque `try` se completa con éxito, por lo que el programa omite los bloques `catch` y la condición falla en la línea 48, y el método `main` termina de ejecutarse en forma normal.

Observe que cuando `cociente` lanza una excepción `ArithmeticException`, `cociente` termina y no devuelve un valor, y las variables locales de `cociente` quedan fuera de alcance (y se destruyen). Si `cociente` contiene variables locales que sean referencias a objetos y no hay otras referencias a esos objetos, éstos se marcan para la recolección de basura. Además, cuando ocurre una excepción, el bloque `try` desde el cual se llamó `cociente` termina antes de que puedan ejecutarse las líneas 30 a 32. Aquí también, si las variables locales se crearon en el bloque `try` antes de que se lanzara la excepción, estas variables quedarían fuera de alcance.

Si se genera una excepción `InputMismatchException` mediante las líneas 25 o 27, el bloque `try` termina y la ejecución continúa con el bloque `catch` en las líneas 34 a 41. En este caso, no se hace una llamada al método `cociente`. Entonces, el método `main` continúa después del último bloque `catch` (línea 48).

13.5 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores sincrónicos**, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores que veremos en este libro son los índices fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango representable de valores), la división entre cero, los parámetros inválidos de método, la interrupción de subprocesos y la asignación fallida de memoria (debido a la falta de ésta). El manejo de excepciones no está diseñado para procesar los problemas asociados con los **eventos asíncronos** (por ejemplo, completar las operaciones de E/S de disco, la llegada de mensajes de red, clics del ratón y pulsaciones de teclas), los cuales ocurren en paralelo con, y en forma independiente de, el flujo de control del programa.

Observación de ingeniería de software 13.2

Incorpore su estrategia de manejo de excepciones en sus sistemas, partiendo desde el principio del proceso de diseño. Puede ser difícil incluir un manejo efectivo de las excepciones, después de haber implementado un sistema.

Observación de ingeniería de software 13.3

El manejo de excepciones proporciona una sola técnica uniforme para procesar los problemas. Esto ayuda a los programadores que trabajan en proyectos extensos a comprender el código de procesamiento de errores de los demás programadores.

Observación de ingeniería de software 13.4

Evite usar el manejo de excepciones como una forma alternativa de flujo de control. Estas excepciones "adicionales" pueden "estorbar" a las excepciones de tipos de errores genuinos.

Observación de ingeniería de software 13.5

El manejo de excepciones simplifica la combinación de componentes de software, y les permite trabajar en conjunto con efectividad, al permitir que los componentes predefinidos comuniquen los problemas a los componentes específicos de la aplicación, quienes a su vez pueden procesar los problemas en forma específica para la aplicación.

13.6 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.

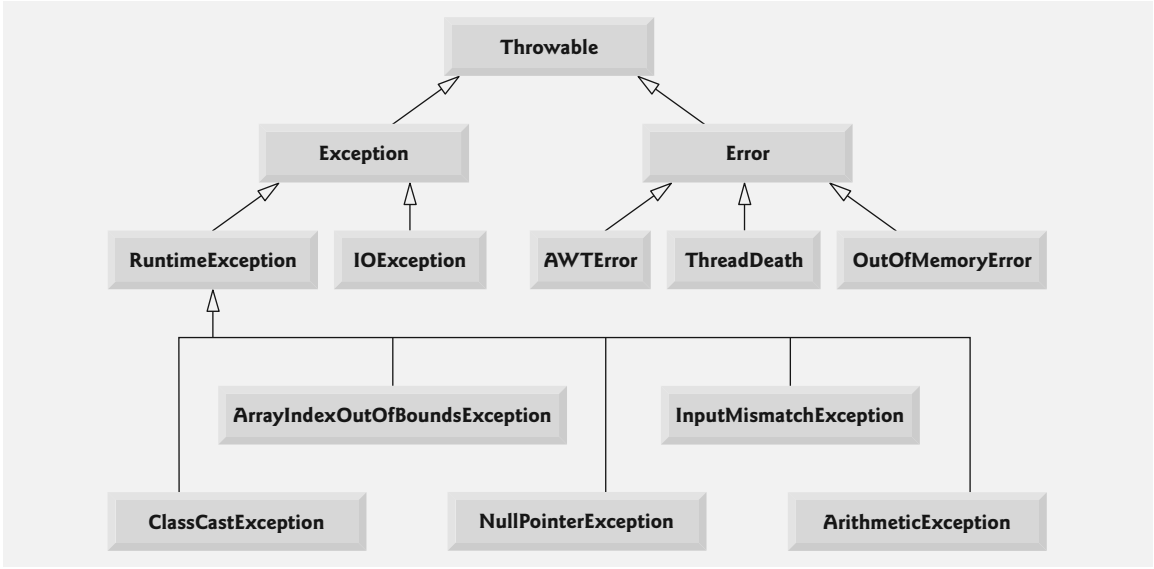

Figura 13.3 | Porción de la jerarquía de herencia de la clase `Throwable`.

La figura 13.3 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`, del paquete `java.lang`, e `IOException`, del 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 (por ejemplo, `OutOfMemoryError`) representan situaciones anormales que podrían ocurrir en la JVM. 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`. [*Nota:* 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 `java.sun.com/javase/6/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].

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 un **requerimiento de atrapar o declarar** para las excepciones verificadas. El tipo de una excepción determina si es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase **`RuntimeException`** (paquete `java.lang`) son excepciones no verificadas. Esto incluye a las excepciones que ya hemos visto, como las excepciones `ArrayIndexOutOfBoundsException` y `ArithmeticException` (que se muestran en la figura 13.3). Todas las clases que heredan de la clase `Exception` pero no de la clase `RuntimeException` se consideran como excepciones verificadas; y las que heredan de la clase `Error` se consideran como no verificadas. El compilador *verifica* cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza excepciones verificadas. De ser así, el compilador asegura que la excepción verificada sea atrapada o declarada en una cláusula `throws`. En la sección 13.4 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 uno de los tipos de su superclase). 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, indicando que la excepción debe ser atrapada o declarada. Esto obliga a los programadores a pensar acerca de los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza

excepciones verificadas. Las clases de excepciones se definen para verificarse cuando se consideran lo bastante importantes como para atraparlas o declararlas.

Observación de ingeniería de software 13.6

Los programadores se ven obligados a tratar con las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores pudieran simplemente ignorar las excepciones.

Error común de programación 13.4

Si un método intenta de manera explícita lanzar una excepción verificada (o si llama a otro método que lance una excepción verificada), y esa excepción no se lista en la cláusula `throws` *de ese método, se produce un error de compilación.*

Error común de programación 13.5

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase listar 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 13.7

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

A diferencia de las excepciones verificadas, el compilador de Java no verifica 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 `ArithmeticException` no verificada que lanza el método `cociente` (líneas 9 a 13) en la figura 13.2 puede evitarse si el método se asegura que el denominador no sea cero antes de tratar de realizar la división. No es obligatorio que se listen las excepciones no verificadas en la cláusula `throws` de un método; aun si se listan, no es obligatorio que una aplicación atrape dichas excepciones.

Observación de ingeniería de software 13.8

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 dichas excepciones podrían ocurrir. Por ejemplo, un programa podría procesar excepciones `NumberFormatException` *del método* `parseInt` *de la clase* `Integer`*, aun cuando las excepciones* `NumberFormatException` *(una subclase de* `RuntimeException`*) sean no verificadas. Esto hará que sus programas sean más robustos.*

Las clases de excepciones se pueden derivar de una superclase común. 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 los errores relacionados con una notación concisa, y permite el procesamiento polimórfico de las excepciones relacionadas. Evidentemente, se podría atrapar a cada uno de los tipos de las subclases en forma individual, si estas excepciones requirieran un procesamiento distinto. Atrapar excepciones relacionadas en un bloque `catch` tendría sentido solamente si el comportamiento del manejo fuera el mismo para todas las subclases.

Si hay varios bloques `catch` que coinciden con un tipo específico de excepción, sólo reejecuta el primer bloque `catch` que coincida cuando ocurra una excepción de ese tipo. Es un error de compilación tratar de 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 subclase de ese tipo. Por ejemplo, podríamos colocar un bloque `catch` para el tipo `ArithmeticException` después de un bloque `catch` para el tipo `Exception`; ambos coincidirían con las excepciones `ArithmeticException`, pero sólo se ejecutaría el primer bloque `catch` que coincidiera.

Tip para prevenir errores 13.6

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 demás bloques* `catch` *para todas las subclases de esa superclase aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.*

Error común de programación 13.6

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.*

13.7 Bloque finally

Los programas que obtienen ciertos tipos de recursos deben devolver esos recursos al sistema en forma explícita, 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 podrían no estar disponibles para su uso en otros programas.

Tip para prevenir errores 13.7

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`, como en la figura 13.4.

Java garantiza que un bloque `finally` (si hay uno presente en una instrucción `try`) se ejecutará, se lance o no una excepción en el bloque `try` correspondiente, o en cualquiera de sus bloques `catch` correspondientes. Java

```
try
{
    instrucciones
    instrucciones para adquirir recursos
} // fin del bloque try
catch ( UnTipoDeExcepción excepción1 )
{
    instrucciones para manejar excepciones
} // fin de bloque catch
.
.
.
catch ( OtroTipoDeExcepción excepción2 )
{
    instrucciones para manejar excepciones
} // fin de bloque catch
finally
{
    instrucciones
    instrucciones para liberar recursos
} // fin de bloque finally
```

Figura 13.4 | Una instrucción `try` con un bloque `finally`.

también garantiza que un bloque `finally` (si hay uno presente) 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 bloque `finally` *no* se ejecutará si la aplicación sale antes de tiempo de un bloque `try`, llamando al método `System.exit`. Este método, que demostraremos en el siguiente capítulo, termina de inmediato una aplicación.

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`, el programa ignora el resto de este bloque. Si el programa atrapa la excepción en uno de los bloques `catch`, procesa la excepción, después el bloque `finally` libera el recurso y el control pasa a la primera instrucción después del bloque `finally`.

Tip de rendimiento 13.2

Siempre debe liberar cada recurso de manera explícita y lo más pronto posible, una vez que ya no sea necesario. Esto hace que los recursos estén inmediatamente disponibles para que su programa (o cualquier otro programa) los reutilice, con lo cual se mejora la utilización de recursos.

Tip para prevenir errores 13.8

Como se garantiza que el bloque `finally` debe ejecutarse, ocurra o no una excepción en el bloque `try` correspondiente, este bloque es un lugar ideal para liberar los recursos adquiridos en un bloque `try`. Ésta es también una manera efectiva de eliminar las fugas de recursos. Por ejemplo, el bloque `finally` debe cerrar todos los archivos que estén abiertos en el bloque `try`.

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 ignora el resto del bloque `try` y el control procede al bloque `finally`. Después el programa pasa la excepción al siguiente bloque `try` (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`. También es posible que la excepción no se atrape.

Si un bloque `catch` lanza una excepción, el bloque `finally` de todas formas se ejecuta. Después, la excepción se pasa al siguiente bloque `try` exterior; de nuevo, en el método que hizo la llamada.

La figura 13.5 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 7 a 19), `lanzaExcepcion` (líneas 22 a 45) y `noLanzaExcepcion` (líneas 48 a 65). Los métodos `lanzaExcepcion` y `noLanzaExcepcion` se declaran como `static`, por lo que `main` puede llamarlos directamente sin instanciar un objeto `UsoDeExcepciones`.

```
1  // Fig. 13.5: UsoDeExcepciones.java
2  // Demostración del mecanismo de manejo de excepciones
3  // try...catch...finally.
4
5  public class UsoDeExcepciones
6  {
7     public static void main( String args[] )
8     {
9        try
10       {
11          lanzaExcepcion(); //llama al método lanzaExcepcion
12       } // fin de try
13       catch ( Exception excepcion ) // excepción lanzada por lanzaExcepcion
14       {
15          System.err.println( "La excepcion se manejo en main" );
16       } // fin de catch
17
18       noLanzaExcepcion();
19    } // fin de main
```

Figura 13.5 Mecanismo de manejo de excepciones `try`...`catch`...`finally`. (Parte 1 de 2).

```
20
21     // demuestra los bloques try...catch...finally
22     public static void lanzaExcepcion() throws Exception
23     {
24        try // lanza una excepción y la atrapa de inmediato
25        {
26           System.out.println( "Metodo lanzaExcepcion" );
27           throw new Exception(); // genera la excepción
28        } // fin de try
29        catch ( Exception excepcion ) // atrapa la excepción lanzada en el bloque try
30        {
31           System.err.println(
32              "La excepcion se manejo en el metodo lanzaExcepcion" );
33           throw excepcion; // vuelve a lanzar para procesarla más adelante
34
35           // no se llegaría al código que se coloque aquí, la excepción se vuelve a
             lanzar en el bloque catch
36
37        } // fin de catch
38        finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
39        {
40           System.err.println( "Se ejecuto finally en lanzaExcepcion" );
41        } // fin de finally
42
43        // no se llega al código que se coloque aquí, la excepción se vuelve a lanzar en
          el bloque catch
44
45     } // fin del método lanzaExcepcion
46
47     // demuestra el uso de finally cuando no ocurre una excepción
48     public static void noLanzaExcepcion()
49     {
50        try // el bloque try no lanza una excepción
51        {
52           System.out.println( "Metodo noLanzaExcepcion" );
53        } // fin de try
54        catch ( Exception excepcion ) // no se ejecuta
55        {
56           System.err.println( excepcion );
57        } // fin de catch
58        finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
59        {
60           System.err.println(
61              "Se ejecuto Finally en noLanzaExcepcion" );
62        } // fin de bloque finally
63
64        System.out.println( "Fin del metodo noLanzaExcepcion" );
65     } // fin del método noLanzaExcepcion
66  } // 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 13.5 Mecanismo de manejo de excepciones `try`...`catch`...`finally`. (Parte 2 de 2).

Observe el uso de **System.err** para imprimir datos en pantalla (líneas 15, 31–32, 40, 56, 60 y 61). De manera predeterminada, System.err.println, al igual que System.out.println, muestra los datos en el símbolo del sistema.

Tanto System.out como System.err son **flujos**: una secuencia de bytes. Mientras que System.out (conocido como el **flujo de salida estándar**) se utiliza para mostrar la salida de un programa, System.err (conocido como el **flujo de error estándar**) se utiliza para mostrar 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 información de 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 redimiremos la salida de System.err, sino que mostraremos dichos mensajes en el símbolo del sistema. En el capítulo 14, Archivos y flujos, aprenderá más acerca de los flujos.

Lanzar excepciones mediante la instrucción **throw**

El método main (figura 13.5) empieza a ejecutarse, entra a su bloque try y de inmediato llama al método lanzaExcepcion (línea 11). El método lanzaExcepcion lanza una excepción tipo Exception. La instrucción en la línea 27 se conoce como **instrucción throw**; esta instrucción se ejecuta para indicar que ha ocurrido 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 ha ocurrido un error. Una instrucción throw especifica un objeto que se lanzará. El operando de throw puede ser de cualquier clase derivada de Throwable.

Observación de ingeniería de software 13.9

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

Observación de ingeniería de software 13.10

Un objeto 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 13.11

Las excepciones pueden lanzarse desde constructores. Cuando se detecta un error en un constructor, debe lanzarse una excepción en vez de crear un objeto formado en forma inapropiada.

Volver a lanzar excepciones

La línea 33 de la figura 13.5 **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 parcialmente. 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. Observe que las excepciones no se pueden volver a lanzar desde un bloque finally, ya que el parámetro de la excepción del bloque catch ha expirado.

Cuando se vuelve a lanzar una excepción, el siguiente bloque try 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 9 a 12 en el método main. Sin embargo, antes de manejar la excepción que se volvió a lanzar, se ejecuta el bloque finally (líneas 38 a 41). Después, el método main detecta la excepción que se volvió a lanzar en el bloque try, y la maneja en el bloque catch (líneas 13 a 16).

A continuación, main llama al método noLanzaExcepcion (línea 18). Como no se lanza una excepción en el bloque try de noLanzaExcepcion (líneas 50 a 53), el programa ignora el bloque catch (líneas 54 a 57), pero el bloque finally (líneas 58 a 62) se ejecuta de todas formas. El control pasa a la instrucción que está después del bloque finally (línea 64). Después, el control regresa a main y el programa termina.

Error común de programación 13.7

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`*, se perderá la primera excepción y se devolverá la excepción del bloque* `finally` *al método que hizo la llamada.*

Tip para prevenir errores 13.9

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

Error común de programación 13.8

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 13.2

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` *alrededor de una porción considerable de su código, y después de ese bloque* `try` *coloque bloques* `catch` *para manejar cada posible excepción, y después de esos bloques* `catch` *coloque un solo bloque* `finally` *(si se requiere).*

13.8 Limpieza de la pila

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, todas las variables en ese método quedan fuera de alcance y 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 (`catch`) esa excepción. Si un bloque `try` no encierra a esa instrucción, se lleva a cabo la limpieza de la pila otra vez. Si ningún bloque `catch` atrapa a esta excepción, y la excepción es verificada (como en el siguiente ejemplo), al compilar el programa se producirá un error. El programa de la figura 13.6 demuestra la limpieza de la pila.

```
1  // Fig. 13.6: UsoDeExcepciones.java
2  // Demostración de la limpieza de la pila.
3
4  public class UsoDeExcepciones
5  {
6     public static void main( String args[] )
7     {
8        try // llama a lanzaExcepcion para demostrar la limpieza de la pila
9        {
10          lanzaExcepcion();
11       } // fin de try
12       catch ( Exception excepcion ) // excepción lanzada en lanzaExcepcion
13       {
14          System.err.println( "La excepcion se manejo en main" );
15       } // fin de catch
16    } // fin de main
17
18    // lanzaExcepcion lanza la excepción que no se atrapa en este método
19    public static void lanzaExcepcion() throws Exception
```

Figura 13.6 | Limpieza de la pila. (Parte 1 de 2).

```
20     {
21        try // lanza una excepción y la atrapa en main
22        {
23           System.out.println( "Metodo lanzaExcepcion" );
24           throw new Exception(); // genera la excepción
25        } // fin de try
26        catch ( RuntimeException runtimeException ) // atrapa el tipo incorrecto
27        {
28           System.err.println(
29              "La excepcion se manejo en el metodo lanzaExcepcion" );
30        } // fin de catch
31        finally // el bloque finally siempre se ejecuta
32        {
33           System.err.println( "Finally siempre se ejecuta" );
34        } // fin de finally
35     } // fin del método lanzaExcepcion
36  } // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
Finally siempre se ejecuta
La excepcion se manejo en main
```

Figura 13.6 | Limpieza de la pila. (Parte 2 de 2).

Cuando se ejecuta el método `main`, la línea 10 en el bloque `try` llama al método `lanzaExcepcion` (líneas 19 a 35). En el bloque `try` del método `lanzaExcepcion` (líneas 21 a 25), la línea 24 lanza una excepción `Exception`. Esto termina el bloque `try` de inmediato, y el control ignora el bloque `catch` en la línea 26, debido a que el tipo que se está atrapando (`RuntimeException`) no es una coincidencia exacta con el tipo lanzado (`Exception`) y no es una superclase del mismo. El método `lanzaExcepcion` termina (pero no hasta que se ejecute su bloque `Finally`) y devuelve el control a la línea 10; el punto desde el cual se llamó en el programa. La línea 10 es un bloque `try` circundante. La excepción no se ha manejado todavía, por lo que el bloque `try` termina y se hace un intento por atrapar la excepción en la línea 12. El tipo que se atrapará (`Exception`) no coincide con el tipo lanzado. En consecuencia, el bloque `catch` procesa la excepción y el programa termina al final de `main`. Si no hubiera bloques `catch` que coincidieran, 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.

13.9 printStackTrace, getStackTrace y getMessage

En la sección 13.6 vimos que las excepciones se derivan de la clase `Throwable`. Esta clase ofrece un método llamado **`printStackTrace`**, que envía al flujo de error estándar la pila de llamadas a métodos (lo cual se describe en la sección 13.3). A menudo, esto ayuda en la prueba y 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. El ejemplo de esta sección, considera estos tres métodos.

Tip para prevenir errores 13.10

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 subproceso de ejecución, la aplicación termina. En una aplicación con varios subprocesos, termina el subproceso que produjo la excepción.

Tip para prevenir errores 13.11

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

En la figura 13.7 se 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. En el capítulo 14, Archivos y flujos, veremos cómo enviar datos a otros flujos.

En `main`, el bloque `try` (líneas 8 a 11) llama a `metodo1` (declarado en las líneas 35 a 38). A continuación, `metodo1` llama a `metodo2` (declarado en las líneas 41 a 44), que a su vez llama a `metodo3` (declarado en las líneas 47 a 50). En la línea 49 de `metodo3` se lanza un objeto `Exception`; éste es el punto de lanzamiento. Como la instrucción `throw` de la línea 49 no va encerrada en ningún bloque `try`, se lleva a cabo la limpieza de la pila; `metodo3` termina en la línea 49 y después regresa el control a la instrucción en `metodo2` que invocó a `metodo3` (es decir, la línea 43). Como ningún bloque `try` encierra a la línea 43, se lleva a cabo la limpieza de la pila otra vez; `metodo2` termina en la línea 43 y regresa el control a la instrucción en `metodo1` que invocó a `metodo2` (es decir, la línea 37). Como ningún bloque `try` encierra a la línea 37, se lleva a cabo la limpieza de la pila una vez más; `metodo1` termina en la línea 37 y regresa el control a la instrucción en `main` que invocó a `metodo1` (es decir, la línea 10). El bloque `try` de las líneas 8 a 11 encierra a esta instrucción. La excepción no ha sido manejada, por lo que el bloque `try` termina y el primer bloque `catch` concordante (líneas 12 a 31) atrapa y procesa la excepción.

```
1  // Fig. 13.7: UsoDeExcepciones.java
2  // Demostración de getMessage y printStackTrace de la clase Exception.
3
4  public class UsoDeExcepciones
5  {
6     public static void main( String args[] )
7     {
8        try
9        {
10          metodo1(); // llama a metodo1
11       } // fin de try
12       catch ( Exception excepcion ) // atrapa la excepción lanzada en metodo1
13       {
14          System.err.printf( "%s\n\n", excepcion.getMessage() );
15          excepcion.printStackTrace(); // imprime el rastreo de la pila de la excepción
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:" );
21          System.out.println( "Clase\t\t\tArchivo\t\t\tLinea\tMetodo" );
22
23          // itera a través de elementosRastreo para obtener la descripción de la
               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          } // fin de for
31       } // fin de catch
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    } // fin del método metodo1
39
```

Figura 13.7 | Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 1 de 2).

```
40      // llama a metodo3; lanza las excepciones de vuelta a metodo1
41      public static void metodo2() throws Exception
42      {
43         metodo3();
44      } // fin del método metodo2
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      } // fin del método metodo3
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 13.7 | Los métodos `getMessage`, `getStackTrace` y `printStackTrace` de `Throwable`. (Parte 2 de 2).

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 de la figura 13.7 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, y *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 13.1. 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 13.12

Nunca ignore una excepción que atrape. Por lo menos, use el método `printStackTrace` para imprimir un mensaje de error. Esto informará a los usuarios que existe un problema, para que puedan tomar las acciones apropiadas.

13.10 Excepciones encadenadas

Algunas veces un bloque `catch` atrapa un tipo de excepción y después lanza una nueva excepción de un tipo distinto, para indicar que ocurrió una excepción específica del programa. En las primeras versiones de Java, no había mecanismo para envolver la información de la excepción original con la información de la nueva excepción, para proporcionar un rastreo completo de la pila, 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 de excepción mantenga la información completa sobre el rastreo de la pila. En la figura 13.8 se demuestran las excepciones encadenadas.

```
1  // Fig. 13.8: UsoDeExcepcionesEncadenadas.java
2  // Demostración de las excepciones encadenadas.
3
4  public class UsoDeExcepcionesEncadenadas
5  {
6     public static void main( String args[] )
7     {
8        try
9        {
10          metodo1(); // llama a metodo1
11       } // fin de try
12       catch ( Exception excepcion ) // excepciones lanzadas desde metodo1
13       {
14          excepcion.printStackTrace();
15       } // fin de catch
16    } // fin de main
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(); // llama a 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       } // fin de try
29    } // fin del método metodo1
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(); // llama a metodo3
37       } // fin de try
38       catch ( Exception excepcion ) // excepción lanzada desde metodo3
39       {
40          throw new Exception( "La excepcion se lanzo en metodo2", excepcion );
41       } // fin de catch
42    } // fin del método metodo2
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    } // fin del método metodo3
49 } // fin de la clase UsoDeExcepcionesEncadenadas
```

Figura 13.8 | Excepciones encadenadas. (Parte 1 de 2).

```
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 13.8 | Excepciones encadenadas. (Parte 2 de 2).

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`. Como 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.

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 23). 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, muestra cuántas excepciones encadenadas más quedan pendientes.

13.11 Declaración de nuevos tipos de excepciones

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) para crear aplicaciones de Java. Los métodos de esas clases por lo general se declaran para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, para 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.

Observación de ingeniería de software 13.13

De ser posible, indique las excepciones de sus métodos mediante el uso de las clases de excepciones existentes, en vez de crear nuevas clases de excepciones. La API de Java contiene muchas clases de excepciones que podrían ser adecuadas para el tipo de problema que su método necesite indicar.

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. Al igual que cualquier otra clase, una clase de excepción puede contener campos y métodos. Sin embargo, una nueva clase de excepción, por lo general, contiene sólo dos constructores; uno que no toma argumentos y pasa un mensaje de excepción predeterminado al constructor de la superclase, y otro que recibe un mensaje de excepción personalizado como una cadena y lo pasa al constructor de la superclase.

Buena práctica de programación 13.3

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

Observación de ingeniería de software 13.14

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. La nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a* `Exception` *pero no a* `RuntimeException`*) los posibles clientes deben manejar la excepción. La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. La nueva clase de excepción debe extender a* `RuntimeExcepcion` *si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada).*

En el capítulo 17, Estructuras de datos, 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` no se permitirán si la `Lista` está vacía, como eliminar un elemento de la parte frontal o posterior de la lista (es decir, no pueden eliminarse elementos, ya que la `Lista` no contiene ningún elemento en ese momento). Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.

Buena práctica de programación 13.4

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

13.12 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, comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

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. 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. Sin embargo, nunca hay que confiar en las precondiciones o esperar un comportamiento consistente, si éstas no se cumplen.

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

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 a 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 obtiene en realidad el carácter en el índice.

Algunos programadores establecen las precondiciones y poscondiciones de manera informal, como parte de la especificación general del método, mientras que otros prefieren un enfoque más formal, al definirlas de manera explícita. Al diseñar sus propios métodos, usted debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método, de cualquier forma que guste. Establecer las precondiciones y poscondiciones antes de escribir un método también nos ayuda a guiarnos a medida que implementamos el método.

13.13 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 acerca del estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones acerca del estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo, Java incluye dos versiones de la instrucción **`assert`** para validar aserciones mediante la programación. La instrucción `assert` evalúa una expresión `boolean` y determina si es verdadera o falsa. La primera forma de la instrucción `assert` es

```
assert expresión;
```

Esta instrucción evalúa *expresión* y lanza una excepción `AssertionError` si la expresión es `falsa`. La segunda forma es

```
assert expresión1 : expresión2;
```

Esta instrucción 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. El ejemplo de la figura 13.9 demuestra la funcionalidad de 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 de la línea de comandos. La instrucción `assert` en la línea 15 determina si el usuario introdujo un número dentro del rango válido. Si el número está fuera de rango, entonces el programa reporta un error; en caso contrario, continúa en forma normal.

```
1  // Fig. 13.9: PruebaAssert.java
2  // Uso de assert para verificar que un valor absoluto sea positivo
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 absoluto sea >= 0
15       assert ( numero >= 0 && numero <= 10 ) : "numero incorrecto: " + numero;
```

Figura 13.9 | Verificar con `assert` que un valor se encuentre dentro del rango. (Parte 1 de 2).

```
16
17         System.out.printf( "Usted escribio %d\n", numero );
18      } // fin de main
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 13.9 | Verificar con `assert` que un valor se encuentre dentro del rango. (Parte 2 de 2).

El programador utiliza las aserciones principalmente para depurar e identificar errores lógicos en una aplicación. De manera predeterminada, las aserciones están deshabilitadas al ejecutar un programa, ya que reducen el rendimiento y son innecesarias para el usuario del programa. Para habilitar las aserciones en tiempo de ejecución, use la opción de línea de comandos `-ea` del comando `java`. Para ejecutar el programa de la figura 13.9 con las aserciones habilitadas, escriba

```
java -ea PruebaAssert
```

No debe encontrar ningún error tipo `AssertionError` durante la ejecución normal de un programa escrito en forma apropiada. Dichos errores sólo deben indicar errores en la implementación. Como resultado, nunca se debe atrapar una excepción tipo `AssertionError`. En vez de ello, debemos permitir que el programa termine al ocurrir el error, para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. Como los usuarios de las aplicaciones pueden elegir no habilitar las aserciones en tiempo de ejecución, no debemos usar la instrucción `assert` para indicar problemas en tiempo de ejecución en el código de producción. En vez de ello, debemos usar el mecanismo de las excepciones para este fin.

13.14 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores en una aplicación. Aprendió que el manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa. Vio el manejo de errores en el contexto de un ejemplo de división entre cero. Aprendió a 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, que indica que una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento. Conoció la diferencia entre las excepciones verificadas y no verificadas, y cómo especificar mediante la cláusula `throws` que las excepciones específicas que ocurran en un método serán lanzadas por ese método al método que lo llamó. 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 acerca de una excepción, mediante el uso de los métodos `printStackTrace`, `getStackTrace` y `getMessage`. El capítulo continuó con una discusión sobre las excepciones encadenadas, que permiten a los programadores envolver la información de la excepción original con la información de la nueva excepción. Después, vimos las generalidades acerca de cómo 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. Cuando no se cumplen las precondiciones y poscondiciones, los métodos generalmente lanzan excepciones. Por último, 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. En el siguiente capítulo aprenderá acerca del procesamiento de archivos, incluyendo la forma en que se almacenan los datos persistentes y cómo se manipulan.

Resumen

Sección 13.1 Introducción

- Una excepción es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver las excepciones.

Sección 13.2 Generalidades acerca del manejo de excepciones

- El manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa, mejorando su claridad y capacidad de modificación.

Sección 13.3 Ejemplo: división entre cero sin manejo de excepciones

- Las excepciones se lanzan cuando un método detecta un problema y no puede manejarlo.
- El rastreo de la pila de una excepción incluye el nombre de la excepción en un mensaje descriptivo, el cual indica el problema que ocurrió y la pila de llamadas a métodos completa (es decir, la cadena de llamadas), en el momento en el que ocurrió la excepción.
- El punto en el programa en el cual ocurre una excepción se conoce como punto de lanzamiento.

Sección 13.4 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`

- Un bloque `try` encierra el código que podría lanzar una excepción, y el código que no debe ejecutarse si se produce esa excepción.
- Las excepciones pueden surgir a través de código mencionado explícitamente en un bloque `try`, a través de llamadas a otros métodos, o incluso a través de llamadas a métodos anidados, iniciadas por el código en el bloque `try`.
- Un bloque `catch` empieza con la palabra clave `catch` y un parámetro de excepción, seguido de un bloque de código que atrapa (es decir, recibe) y maneja la excepción. Este código se ejecuta cuando el bloque `try` detecta la excepción.
- Una excepción no atrapada es una excepción que ocurre y para la cual no hay bloques `catch` que coincidan.
- Una excepción no atrapada hará que un programa termine antes de tiempo, si éste sólo contiene un subproceso. Si el programa contiene más de un subproceso, sólo terminará el subproceso en el que ocurrió la excepción. El resto del programa se ejecutará, pero puede producir efectos adversos.
- Justo después del bloque `try` debe ir por lo menos un bloque `catch` o un bloque `finally`.
- Cada bloque `catch` especifica entre paréntesis un parámetro de excepción, el cual identifica el tipo de excepción que puede procesar el manejador. El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada.
- Si ocurre una excepción en un bloque `try`, éste termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` cuyo parámetro de excepción coincida con el tipo de la excepción que se lanzó.
- Una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado. A esto se le conoce como el modelo de terminación del manejo de excepciones.
- Si hay varios bloques `catch` que coinciden cuando ocurre una excepción, sólo se ejecuta el primero.
- Después de ejecutar un bloque `catch`, el flujo de control del programa pasa a la siguiente instrucción después del último bloque `catch`.
- Una cláusula `throws` especifica las excepciones que lanza el método, y aparece después de la lista de parámetros del método, pero antes de su cuerpo.
- La cláusula `throws` contiene una lista separada por comas de excepciones que lanzará el método, en caso de que ocurra un problema cuando el método se ejecute.

Sección 13.5 Cuándo utilizar el manejo de excepciones

- El manejo de excepciones está diseñado para procesar errores sincrónicos, que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los problemas asociados con eventos asíncronos, que ocurren en paralelo con (y en forma independiente de) el flujo de control del programa.

Sección 13.6 Jerarquía de excepciones de Java

- Todas las clases de excepciones de Java heredan, ya sea en forma directa o indirecta, de la clase `Exception`. Debido a esto, las clases de excepciones de Java forman una jerarquía. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

- La clase `Throwable` es la superclase de la clase `Exception` y, por lo tanto, es también la superclase de todas las excepciones. 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 representan situaciones excepcionales que podrían ocurrir en un programa de Java y ser atrapados por la aplicación.
- La clase `Error` y todas sus subclases representan situaciones excepcionales que podrían ocurrir en el sistema en tiempo de ejecución de Java. Los errores tipo `Error` ocurren con poca frecuencia y, por lo general, no deben ser atrapados por una aplicación.
- Java clasifica a las excepciones en dos categorías: verificadas y no verificadas.
- A diferencia de las excepciones verificadas, el compilador de Java no verifica el código para determinar si una excepción no verificada se atrapa o se declara. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada.
- El tipo de una excepción determina si ésta es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase `RuntimeException` son excepciones no verificadas. Todos los tipos de excepciones que heredan de la clase `Exception` pero no de `RuntimeException` son verificadas.
- Varias clases de excepciones pueden derivarse de una superclase común. Si se escribe un bloque `catch` para atrapar los objetos de excepción de un tipo de la superclase, también puede atrapar a todos los objetos de las subclases de esa clase. Esto permite el procesamiento polimórfico de las excepciones relacionadas.

Sección 13.7 Bloque `finally`

- Los programas que obtienen ciertos tipos de recursos deben devolverlos al sistema de manera explícita, para evitar las denominadas fugas de recursos. Por lo general, el código para liberar recursos se coloca en un bloque `finally`.
- El bloque `finally` es opcional. Si está presente, se coloca después del último bloque `catch`.
- Java garantiza que si se proporciona un bloque `finally`, se ejecutará sin importar que se lance o no una excepción en el bloque `try` correspondiente, o en uno de sus correspondientes bloques `catch`. Java también garantiza que un bloque `finally` se ejecutará si un bloque `try` sale mediante el uso de una instrucción `return`, `break` o `continue`.
- Si una excepción que ocurre en el bloque `try` no se puede atrapar mediante uno de los manejadores `catch` asociados a ese bloque `try`, el programa ignora el resto del bloque `try` y el control pasa al bloque `finally`, que libera el recurso. Después, el programa pasa al siguiente bloque `try` exterior; por lo general, en el método que hace la llamada.
- Si un bloque `catch` lanza una excepción, de todas formas se ejecuta el bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior; por lo general, en el método que hizo la llamada.
- Los programadores pueden lanzar excepciones mediante el uso de la instrucción `throw`.
- Una instrucción `throw` especifica un objeto a lanzar. El operando de una instrucción `throw` puede ser de cualquier clase que se derive de `Throwable`.

Sección 13.8 Limpieza de la pila

- Las excepciones se vuelven a lanzar cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesarla, o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción se difiere el manejo de excepciones (o tal vez una parte de éste) a otro bloque `catch`.
- Cuando se vuelve a lanzar una excepción, el siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y los bloques `catch` de ese bloque `try` tratan de manejarla.
- Cuando se lanza una excepción, pero no se atrapa en un alcance específico, se limpia la pila de llamadas a métodos y se hace un intento por atrapar la excepción en la siguiente instrucción `try` exterior. A este proceso se le conoce como limpieza de la pila.

Sección 13.9 `printStackTrace`, `getStackTrace` *y* `getMessage`

- La clase `Throwable` ofrece un método `printStackTrace`, que imprime la pila de llamadas a métodos. A menudo, esto es útil en la prueba y depuración.
- La clase `Throwable` también proporciona un método `getStackTrace`, que obtiene información de rastreo de la pila, que `printStackTrace` imprime.
- El método `getMessage` de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción.
- El método `getStackTrace` obtiene la información de rastreo de la pila como un arreglo de objetos `StackTraceElement`. Cada objeto `StackTraceElement` representa una llamada a un método en la pila de llamadas a métodos.
- Los métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` de la clase `StackTraceElement` obtienen el nombre de la clase, el nombre de archivo, el número de línea y el nombre del método, respectivamente.

Sección 13.10 Excepciones encadenadas
- Las excepciones encadenadas permiten que un objeto de excepción mantenga la información de rastreo de la pila completa, incluyendo la información acerca de las excepciones anteriores que provocaron la excepción actual.

Sección 13.11 Declaración de nuevos tipos de excepciones
- Una nueva clase de excepción debe extender a una clase de excepción existente, para asegurar que la clase pueda usarse con el mecanismo de manejo de excepciones.

Sección 13.12 Precondiciones y poscondiciones
- La precondición de un método es una condición que debe ser verdadera al momento de invocar el método.
- La poscondición de un método es una condición que es verdadera una vez que regresa el método con éxito.
- Al diseñar sus propios métodos, debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método.

Sección 13.13 Aserciones
- Dentro de una aplicación, los programadores pueden establecer condiciones que asuman como verdaderas en un punto específico. Estas condiciones, conocidas como aserciones, ayudan a asegurar la validez de un programa al atrapar errores potenciales e identificar posibles errores lógicos.
- Java incluye dos versiones de una instrucción `assert` para validar las aserciones mediante la programación.
- Para habilitar las aserciones en tiempo de ejecución, use el modificador `-ea` al ejecutar el comando `java`.

Terminología

`ArithmeticException`, clase
aserción
`assert`, instrucción
atrapar una excepción
bloque `try` circundante
`catch`, bloque
`catch`, cláusula
error sincrónico
`Error`, clase
evento asíncrono
Excepción
excepción encadenada
excepción no atrapada
excepción verificada
excepciones no verificadas
`Exception`, clase
falla en el constructor
`finally`, bloque
`finally`, cláusula
flujo de error estándar
flujo de salida estándar
fuga de recursos
`getClassName`, método de la clase `StackTraceElement`
`getFileName`, método de la clase `StackTraceElement`
`getLineNumber`, método de la clase `StackTraceElement`
`getMessage`, método de la clase `Throwable`
`getMethodName`, método de la clase `StackTraceElement`
`getStackTrace`, método de la clase `Throwable`
`InputMismatchException`, clase
lanzar una excepción
liberar un recurso
limpieza de la pila
manejador de excepciones
manejo de excepciones
modelo de reanudación del manejo de excepciones
modelo de terminación del manejo de excepciones
parámetro de excepción
poscondición
precondición
`printStackTrace`, método de la clase `Throwable`
programa tolerante a fallas
punto de lanzamiento
rastreo de la pila
requerimiento de atrapar o declarar
`RuntimeException`, clase
`StackTraceElement`, clase
`System.err`, flujo
`throw`, instrucción
`throw`, palabra clave
`Throwable`, clase
`throws`, cláusula
`try`, bloque
`try`, instrucción
`try...catch...finally`, mecanismo para manejar excepciones
volver a lanzar una excepción

Ejercicios de autoevaluación

13.1 Enliste cinco ejemplos comunes de excepciones.

13.2 Dé varias razones por las cuales no deban utilizarse las técnicas de manejo de excepciones para el control convencional de los programas.