Manejo de excepciones: un análisis más detallado

11.1 Introducción

Como vimos en el capítulo 7, una excepción es la indicación de un problema que ocurre durante la ejecución de un programa. 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. Las características que presentamos en este capítulo permiten a los programadores escribir programas *tolerantes a fallas* y *robustos*, que traten con los problemas que puedan surgir sin dejar de ejecutarse o que *terminen sin causar estragos*. El manejo de excepciones en Java se basa, en parte, en el trabajo de Andrew Koenig y Bjarne Stroustrup.[1]

Primero demostraremos las técnicas básicas de manejo de excepciones mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después 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 le mostraremos cómo usar *excepciones encadenadas* (cuando invocamos a un método que indica una excepción, podemos lanzar otra excepción y encadenar la original a la nueva). Esto nos permite agregar información específica de la aplicación a la excepción original. Luego le presentaremos las *precondiciones y poscondiciones*, que deben ser verdaderas cuando se hacen llamadas a sus métodos y cuando éstos regresan, respectivamente. A continuación presentaremos las *aserciones*, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código. También hablaremos de dos nuevas características de manejo de excepciones que se introdujeron en Java SE 7: atrapar varias excepciones con un solo manejador **`catch`** y la nueva instrucción **`try`** con recursos, que libera automáticamente un recurso después de usarlo en el bloque try.

Este capítulo se enfoca en los conceptos de manejo de excepciones y presenta varios ejemplos mecánicos que demuestran diversas características. Como veremos en capítulos posteriores, muchos métodos de las API de Java lanzan excepciones que manejamos en nuestro código. La figura 11.1 muestra algunos de los tipos de excepciones que ya hemos visto y otros que verá más adelante.

[1] A. Koenig y B. Stroustrup, “Exception Handling for C++ (revised)”, *Proceedings of the Usenix C++ Conference*, págs. 149-176, San Francisco, abril de 1990.

Capítulo	Ejemplo de excepciones utilizadas
Capítulo 7	ArrayIndexOutOfBoundsException
Capítulos 8 a 10	IllegalArgumentException
Capítulo 11	ArithmeticException, InputMismatchException
Capítulo 15	SecurityException, FileNotFoundException, IOException, ClassNotFoundException, IllegalStateException, FormatterClosedException, NoSuchElementException
Capítulo 16	ClassCastException, UnsupportedOperationException, NullPointerException, tipos de excepciones personalizadas
Capítulo 20	ClassCastException, tipos de excepciones personalizadas
Capítulo 21	IllegalArgumentException, tipos de excepciones personalizadas
Capítulo 23	InterruptedException, IllegalMonitorStateException, ExecutionException, CancellationException
Capítulo 28	MalformedURLException, EOFException, SocketException, InterruptedException, UnknownHostException
Capítulo 24	SQLException, IllegalStateException, PatternSyntaxException
Capítulo 31	SQLException

Fig. 11.1 Varios tipos de excepciones que verá en este libro

11.2 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 11.2 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.

```
 1. // DivisionEntreCeroSinManejoDeExcepciones.java
 2. // División de enteros sin manejo de excepciones.
 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.    }
12.
13.    public static void main(String[] args)
14.    {
15.       Scanner explorador = new Scanner(System.in);
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.    }
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(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
```

```
at java.util.Scanner.nextInt(Unknown Source)
at DivisionEntreCeroSinManejoDeExcepciones.main(
DivisionEntreCeroSinManejoDeExcepciones.java:20)
```

Figura 1.12 División de enteros sin manejo de excepciones

Rastreo de la pila

La primera de las tres ejecuciones de ejemplo anterior muestran una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce el valor 0 como denominador. Se muestran varias líneas de información en respuesta a esta entrada inválida.

Esta información se conoce como el **rastreo de la pila**, la cual lleva 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 (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 nos ayuda a depurar un programa.

Rastreo de la pila para una excepción `ArithmeticException`

La primera línea especifica que ocurrió una 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. Cuando ocurre esto, 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 información más específica. Java *sí* permite la división entre cero con valores de punto flotante. Dicho cálculo produce como resultado el valor de infinito positivo o negativo, que se representa en Java como un valor de punto flotante (pero en realidad aparece como la cadena `Infinity` o `-Infinity`). Si se divide 0.0 entre 0.0, el resultado es *NaN* (no es un número), que también se representa en Java como un valor de punto flotante (pero se visualiza como *NaN*). Si necesita comparar un valor de punto flotante con *NaN*, use el método `isNaN` de la clase `Float` (para valores **float**) o de la clase `Double` (para valores **double**). Las clases `Float` y `Double` están en el paquete `java.lang`.

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**, que es el punto inicial en el que ocurrió la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

Rastreo de la pila para una excepción `InputMismatchException`

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 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 (`string`) 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 a los supuestos *símbolos de depuración* que proveen la información sobre el nombre del archivo y el número de línea para la clase de ese método (por lo general, éste es el caso para las clases de la API de Java). Muchos IDE tienen acceso al código fuente de la API de Java, por lo que muestran los nombres de archivos y números de línea en los rastreos de la pila.

Terminación del programa

En las ejecuciones de ejemplo de la figura 11.2, 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 imprima un rastreo de pila. En tales casos, la aplicación puede producir resultados inesperados. Por ejemplo, una aplicación de interfaz gráfica de usuario (GUI) por lo general se seguirá ejecutando. En la figura 11.2, 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.

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

La aplicación de la figura 11.3, que se basa en la figura 11.2, 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, permite al usuario tratar de introducir los datos de entrada otra vez.

```
 1. // DivisionEntreCeroConManejoDeExcepciones.java
 2. // Manejo de excepciones ArithmeticException e InputMismatchException.
 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
 9.    public static int cociente(int numerador, int denominador)
10.       throws ArithmeticException
11.    {
12.       return numerador / denominador; // posible división entre cero
13.    }
14.
15.    public static void main(String[] args)
16.    {
17.       Scanner explorador = new Scanner(System.in);
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.             }
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\n");
41.             }
42.             catch (ArithmeticException arithmeticException)
43.             {
44.                System.err.printf("\nExcepcion: %s\n", arithmeticException);
45.                System.out.printf(
46.                    "Cero es un denominador invalido. Intente de nuevo.\n\n");
47.             }
48.          } while (continuarCiclo);
49.       }
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


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 11.3 Manejo de excepciones `ArithmeticException` e `InputMismatchException`

La primera ejecución de ejemplo de la figura 11.3 es 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`**. En la línea 18 se crea la variable boolean llamada **`continuarCiclo`**, la cual es **`true`** 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`** en sí). 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.

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 11.1

> *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, mientras que 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].

Al menos **debe** ir un bloque **catch** o un **bloque finally** (que veremos en la sección 11.6) justo 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 el primero 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 directa o indirecta 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. Observe que usamos el objeto **System.err** (flujo de error estándar) para mostrar los mensajes de error en pantalla. Los métodos print de **System.err**, al igual que los de System.out, muestran datos en el *símbolo del sistema* de manera predeterminada.

La línea 38 del primer bloque **catch** llama al método **nextLine** de **Scanner**. Como ocurrió una excepción del tipo **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 11.1

*Es un error de sintaxis colocar código entre un bloque **try** y sus correspondientes bloques **catch**.*

Multi-catch

Es relativamente común que después de un bloque `try` vayan varios bloques **catch** para manejar diversos tipos de excepciones. Si los cuerpos de varios bloques **catch** son idénticos, puede usar la característica **multi-catch** (introducida en Java SE 7) para atrapar esos tipos de excepciones en un solo manejador **catch** y realizar la misma tarea. La sintaxis para una instrucción **multi-catch** es:

```
catch (tipo1 | tipo2 | tipo3 e)
```

Cada tipo de excepción se separa del siguiente con una barra vertical (|). La línea anterior de código indica que *cualquiera* de los tipos (o sus subclases) pueden atraparse en el manejador de excepciones. En una instrucción multi-catch puede especificarse cualquier número de tipos **Throwable**.

Excepciones no atrapadas

Una **excepción no atrapada** es una para la que no hay bloques **catch** que coincidan. En el segundo y tercer resultado de ejemplo de la figura 11.2, 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. Java utiliza un modelo "multihilos" de ejecución de programas, en el que cada **hilo** es una *actividad concurrente*. Un programa puede tener muchos hilos. Si un programa sólo tiene un hilo, una excepción no atrapada hará que el programa termine. Si un programa tiene *múltiples* hilos, una excepción no atrapada terminará *sólo* el hilo en el cual ocurrió la excepción. Sin embargo, en dichos programas ciertos hilos pueden depender de otros, y si un hilo termina debido a una excepción no atrapada, puede haber efectos adversos para el resto del programa. En el capítulo 23 en línea, Concurrency, analizaremos estas cuestiones con detalle.

Modelo de terminación del manejo de excepciones

Si ocurre una excepción en un bloque **try** (por ejemplo, si se lanza una excepción **Input- MismatchException** como resultado del código de la línea 25 en la figura 11.3), 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 11.3, 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. 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*.

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

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 11.6 aprenderemos sobre otra posibilidad, cuando hablemos del bloque *finally*). Ahora la condición del ciclo **do...while** es **false**, y el método **main** termina.

El bloque **try** y sus correspondientes bloques **catch** o **finally** forman en conjunto una instrucción **try**. Es importante no confundir los términos "*bloque try*" e "instrucción **try**; esta última incluye el bloque **try**, así como los siguientes bloques **catch** o un bloque **finally**.

Al igual que con cualquier otro bloque de código, cuando termina un bloque **try**, las *variables locales* declaradas en ese bloque *quedan fuera de alcance* y ya no son accesibles; por ende, las variables locales de un bloque **try** no son accesibles en los correspondientes bloques **catch**. 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`

En el método cociente (figura 11.3; líneas 9 a 13), la línea 10 se conoce como cláusula throws. Esta cláusula especifica las excepciones que el método podría lanzar si ocurren problemas. La cláusula, que debe aparecer después de la lista de parámetros del método y antes de su cuerpo, contiene una lista separada por comas de los tipos de excepciones. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o a través de métodos que se llamen desde ahí. Hemos agregado la cláusula throws a esta aplicación, para indicar que este método puede lanzar una excepción **ArithmeticException**. Por ende, a los clientes del método cociente se les informa que el método podría lanzar una excepción **ArithmeticException**. Algunos tipos de excepciones, como **ArithmeticException**, no tienen que aparecer en la lista de la cláusula **throws**. Para las que sí deben aparecer, el método puede lanzar excepciones que tengan la relación *es un(a)* con las clases en la lista de la cláusula **throws**. En la sección 11.5 aprenderá más acerca de esto.

Tip para prevenir errores 11.1

Antes de usar un método en un programa, lea la documentación en línea de la API para saber acerca del mismo. La documentación especifica las excepciones que lanza el método (si las hay), y también indica las razones por las que pueden ocurrir dichas excepciones. Después, lea la documentación de la API en línea para ver las clases de excepciones especificadas. Por lo general, la documentación para una clase de excepción contiene las razones potenciales por las que pueden ocurrir dichas excepciones. Por último, incluya el código adecuado para manejar esas excepciones en su programa.

Cuando se ejecuta la línea 12, si el denominador es cero, la **JVM** lanza un objeto **Arith- meticException**. 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 hace fallar la condición en la línea 48, con lo cual el método **`main`** termina de ejecutarse en forma normal.

Cuando cociente lanza una excepción **`ArithmeticException`**, cociente *termina* y no devuelve un valor, por lo que sus variables locales 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).

11.4 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 un método* y la *interrupción de hilos* (como veremos en el capítulo 23), así como 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 e *independientemente* del flujo de control del programa.

Observación de ingeniería de software 11.2

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

Observación de ingeniería de software 11.3

El manejo de excepciones proporciona una sola técnica uniforme para documentar, detectar y recuperarse de los errores. 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 11.4

Hay una gran variedad de situaciones que generan excepciones: es más fácil recuperarse de unas que de otras.