Empezar a Programar Usando Java

Natividad Prieto
Francisco Marqués
Isabel Galiano
Jorge González
Carlos Martínez-Hinarejos
Javier Piris
Assumpció Casanova
Marisa Llorens
Jon Ander Gómez
Carlos Herrero
Germán Moltó

Capítulo 14

Tratamiento de errores

El objetivo básico que se pretende cubrir en este capítulo es el de incorporar al proceso de diseño de una aplicación Java las tareas de previsión y tratamiento de las anomalías o fallos que se pueden producir durante su ejecución, evitando así, en la medida de lo posible, su terminación abrupta o que obtenga resultados incorrectos. A los fallos o condiciones anormales que ocurren durante la ejecución de un segmento de código se las denomina *excepciones*. Al ocurrir una excepción, ésta debe ser tratada porque, de lo contrario, la ejecución de la secuencia de código se detendrá dando un error en tiempo de ejecución. En el lenguaje Java, las excepciones son objetos que describen una condición excepcional que se produce durante la ejecución de un fragmento de código. Dicho objeto se envía al causante de la excepción, que puede tratar la aparición de la misma (mediante instrucciones explícitas de tratamiento), o no tratarla dejando que sea el sistema (dando un error) el encargado de ello.

Antes de entrar en detalle con la gestión de excepciones, se definen y clasifican los diversos fallos o anomalías que se pueden producir durante la ejecución de una aplicación, presentando la jerarquía `Throwable` como el instrumento que proporciona el estándar de Java para su representación, tanto la de los fallos que sí son recuperables (*excepciones*) como la de los que no lo son (*errores*); en particular, se indica cómo reutilizar vía herencia esta jerarquía para diseñar las denomina-das *excepciones de usuario* (en inglés, *user-defined exceptions*) o excepciones *ad hoc* para una aplicación, con las que prevenir y tratar los resultados atípicos que obtienen, en determinadas circunstancias, algunos de sus métodos.

14.1 Fallos de ejecución y su modelo Java

Durante la ejecución de una aplicación pueden producirse fallos o anomalías que conducen a su terminación abrupta o, lo que es aún peor, a la obtención de resultados incorrectos. Incluso a un programador novel le resultan familiares fallos como los de entrada/salida que, por ejemplo, resultan de leer datos cuyo formato o rango no se ajusta al previsto –un valor negativo como radio de un círculo, un carácter como opción de un menú numérico–, de un acceso indebido (sin los permisos oportunos o empleando un nombre incorrecto) a un fichero, etc. También son muy habituales, cuando se aprende a programar, fallos en la lógica de la aplicación provocados por los resultados atípicos que pueden obtener algunos de sus métodos como, por ejemplo, los que se producen en la aplicación que manipula una secuencia de círculos cuando se busca o borra un círculo dado que no está en una secuencia, y fallos de programación como dividir por cero, calcular la raíz cuadrada de un valor negativo y acceder a una posición inexistente de un array o a un objeto `null`. A pesar de ser de tipos diferentes, los fallos de ejecución descritos se clasifican como *excepciones* porque resulta factible o razonable evitar sus consecuencias y reconducir la ejecución de la aplicación una vez se producen; justo lo contrario sucede con los denominados *errores*, fallos de ejecución que por ser muy severos o por involucrar al soporte *hardware* de la aplicación (por ejemplo, agotar la memoria o acceder indebidamente a una de sus zonas) resultan de difícil o imposible solución y, por ello, irrecuperables.

Pues bien, asumiendo que una estricta disciplina de programación y prueba (*debugging*) elimina los fallos de programación, es más que razonable pensar que ya en el proceso de diseño de una buena aplicación Java se debe advertir de la presencia y gestionar el tratamiento de aquellas excepciones que puedan producirse al ejecutarla; para ello, antes es imprescindible introducir los instrumentos que el lenguaje Java proporciona para la representación y clasificación de cualquier fallo de ejecución.

14.1.1 La jerarquía `Throwable`

El lenguaje Java proporciona, en su paquete estándar `java.lang`, la clase `Throwable`, derivada de `Object`, que representa cualquier fallo de ejecución independientemente de su tipo. Es más, la reutilización vía herencia de la clase `Throwable` resulta obligatoria para organizar y diseñar todas las clases que componen la jerarquía `Throwable`, jerarquía que reproduce la clasificación de los fallos de ejecución anteriormente citados.

Como se detalla a continuación, es el propio lenguaje Java quien establece implícitamente el procedimiento a seguir y las características que, como mínimo, debe exhibir cualquier clase de esta jerarquía al ofrecer, ya diseñadas en distintos paquetes de su estándar, la mayoría de las subclases de `Throwable`. Para empezar por el

ejemplo quizás más completo y relevante en lo que a organización y representación de fallos de ejecución se refiere, cabe detenerse a observar en la documentación del *API* de Java [Ora16b] el árbol de la jerarquía `Throwable` en el paquete estándar `java.lang` mostrado en la figura 14.1.

Se puede observar que, en efecto, para poder incluir cualquier fallo de ejecución entre las variables que crea y manipula una aplicación, y así poder tratarlos según convenga, el lenguaje Java deriva de `Object` la clase `Throwable`; a su vez, visto que una excepción es un fallo de ejecución -recuperable- y que también un error es un fallo de ejecución -aunque irrecuperable-, las clases `Error` y `Exception` se diseñan como derivadas directas de `Throwable`, la base de la jerarquía Java de errores y excepciones.

También es importante destacar que `Exception` y `Error` son además las raíces de sendas jerarquías donde coexisten clases que representan tipos concretos de excepciones y errores junto con subárboles de clases que representan subtipos de éstos; así, por ejemplo, de `Exception` se derivan tanto `ClassNotFoundException` que representa la excepción que se produce al cargar una clase empleando un nombre incorrecto como `RuntimeException` que es la clase raíz de la jerarquía que contiene todos los fallos de programación, tan conocidos por otra parte que para identificarlos basta con fijarse en sus nombres, por ejemplo:

```
ArithmeticException
ClassCastException
IllegalArgumentException:
    NumberFormatException
IndexOutOfBoundsException:
    ArrayIndexOutOfBoundsException
    StringIndexOutOfBoundsException
NullPointerException
```

Pero, como se ha comentado anteriormente, `java.lang` no es el único paquete estándar donde se ubican componentes de la jerarquía `Throwable`. Por no alargar demasiado esta exposición, se analiza ahora el rol y la organización de algunas clases de esta jerarquía únicamente en `java.util` y `java.io`, aunque se recomienda investigar en la documentación del *API* de Java [Ora16b] para averiguar algo más sobre el tema; así pues, en las figuras 14.2 y 14.3 se muestran los árboles `Throwable` que la documentación del *API* proporciona para los dos paquetes estándar citados.

De todas las clases que aparecen en estas figuras, dos en particular resultan reseñables por su importancia a la hora de representar las excepciones de entrada/salida. Mencionar en primer lugar la clase `IOException` en `java.io`, pues representa cualquier excepción provocada por un fallo o una interrupción en una operación de entrada o salida, como se verá en el capítulo 15. En segundo lugar, cabe destacar la clase `InputMismatchException` en `java.util`, que representa la excepción que

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.lang.**AssertionError**
    - java.lang.**LinkageError**
      - java.lang.**BootstrapMethodError**
      - java.lang.**ClassCircularityError**
      - java.lang.**ClassFormatError**
        - java.lang.**UnsupportedClassVersionError**
      - java.lang.**ExceptionInInitializerError**
      - java.lang.**IncompatibleClassChangeError**
        - java.lang.**AbstractMethodError**
        - java.lang.**IllegalAccessError**
        - java.lang.**InstantiationError**
        - java.lang.**NoSuchFieldError**
        - java.lang.**NoSuchMethodError**
      - java.lang.**NoClassDefFoundError**
      - java.lang.**UnsatisfiedLinkError**
      - java.lang.**VerifyError**
    - java.lang.**ThreadDeath**
    - java.lang.**VirtualMachineError**
      - java.lang.**InternalError**
      - java.lang.**OutOfMemoryError**
      - java.lang.**StackOverflowError**
      - java.lang.**UnknownError**
  - java.lang.**Exception**
    - java.lang.**CloneNotSupportedException**
    - java.lang.**InterruptedException**
    - java.lang.**ReflectiveOperationException**
      - java.lang.**ClassNotFoundException**
      - java.lang.**IllegalAccessException**
      - java.lang.**InstantiationException**
      - java.lang.**NoSuchFieldException**
      - java.lang.**NoSuchMethodException**
    - java.lang.**RuntimeException**
      - java.lang.**ArithmeticException**
      - java.lang.**ArrayStoreException**
      - java.lang.**ClassCastException**
      - java.lang.**EnumConstantNotPresentException**
      - java.lang.**IllegalArgumentException**
        - java.lang.**IllegalThreadStateException**
        - java.lang.**NumberFormatException**
      - java.lang.**IllegalMonitorStateException**
      - java.lang.**IllegalStateException**
      - java.lang.**IndexOutOfBoundsException**
        - java.lang.**ArrayIndexOutOfBoundsException**
        - java.lang.**StringIndexOutOfBoundsException**
      - java.lang.**NegativeArraySizeException**
      - java.lang.**NullPointerException**
      - java.lang.**SecurityException**
      - java.lang.**TypeNotPresentException**
      - java.lang.**UnsupportedOperationException**

Figura 14.1: Jerarquía `Throwable` en `java.lang`.

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.io.**IOError**
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.io.**CharConversionException**
      - java.io.**EOFException**
      - java.io.**FileNotFoundException**
      - java.io.**InterruptedIOException**
      - java.io.**ObjectStreamException**
        - java.io.**InvalidClassException**
        - java.io.**InvalidObjectException**
        - java.io.**NotActiveException**
        - java.io.**NotSerializableException**
        - java.io.**OptionalDataException**
        - java.io.**StreamCorruptedException**
        - java.io.**WriteAbortedException**
      - java.io.**SyncFailedException**
      - java.io.**UnsupportedEncodingException**
      - java.io.**UTFDataFormatException**
    - java.lang.**RuntimeException**
      - java.io.**UncheckedIOException**

Figura 14.2: Jerarquía `Throwable` en `java.io`.

se produce cuando con un objeto `Scanner` se lee un dato cuyo formato o rango no se corresponde con el del tipo esperado como, por ejemplo, cuando se lee con `nextInt()` un número real o una cadena de caracteres en lugar de un valor `int`.

Vista su organización, restan ahora por introducir la funcionalidad básica y las características de las que dota el lenguaje Java a cualquier clase de la jerarquía `Throwable`. En lo que a funcionalidad respecta, la clase `Throwable` y sus derivadas suelen constar de dos constructores, uno con un único parámetro de tipo `String` y el constructor por defecto (sin parámetros); cualquiera de ellos construye un objeto que almacena el estado de la ejecución en el instante en el que se produce el fallo que representa. Cualquier excepción tiene una variable de instancia de tipo `String` con un mensaje que, normalmente, identifica el motivo de la excepción. El `String` argumento del primer constructor inicializa dicha variable de instancia, es decir, se utiliza para añadir información sobre el fallo acaecido. De todos los métodos heredados de la clase `Throwable` cabe destacar, por su utilidad, los métodos `getMessage()`, `printStackTrace()` y `toString()`. El método `getMessage()` devuelve el `String` con el mensaje de la excepción. El método `printStackTrace()` imprime el tipo de la excepción y su traza (secuencia de llamadas en la pila de ejecución) en la salida de error estándar. Suele usarse en tareas de depuración, facilitando al programador la localización del fallo producido. El método `toString()` sólo proporciona una muy breve descripción de la información que almacena el objeto sobre el que se aplica. Si el objeto se ha creado haciendo uso del constructor sin argumentos, devuelve el tipo de la excepción. Si se ha creado mediante el constructor con un argumento, entonces el resultado es la concatenación de tres

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.util.**ServiceConfigurationError**
  - java.lang.**Exception**
    - java.io.**IOException**
      - java.util.**InvalidPropertiesFormatException**
    - java.lang.**RuntimeException**
      - java.util.**ConcurrentModificationException**
      - java.util.**EmptyStackException**
      - java.lang.**IllegalArgumentException**
        - java.util.**IllegalFormatException**
          - java.util.**DuplicateFormatFlagsException**
          - java.util.**FormatFlagsConversionMismatchException**
          - java.util.**IllegalFormatCodePointException**
          - java.util.**IllegalFormatConversionException**
          - java.util.**IllegalFormatFlagsException**
          - java.util.**IllegalFormatPrecisionException**
          - java.util.**IllegalFormatWidthException**
          - java.util.**MissingFormatArgumentException**
          - java.util.**MissingFormatWidthException**
          - java.util.**UnknownFormatConversionException**
          - java.util.**UnknownFormatFlagsException**
      - java.lang.**IllegalStateException**
        - java.util.**FormatterClosedException**
      - java.util.**IllformedLocaleException**
      - java.util.**MissingResourceException**
      - java.util.**NoSuchElementException**
        - java.util.**InputMismatchException**
    - java.util.**TooManyListenersException**

Figura 14.3: Jerarquía `Throwable` en `java.util`.

`String`: el tipo de la excepción, ": " y el resultado de `getMessage()` para este objeto. Por ello, lo más aconsejable, en la mayoría de las ocasiones, es usar el constructor con un argumento; para ilustrar este punto basta con ejecutar el siguiente código:

```
Scanner teclado = new Scanner(System.in).useLocale(Locale.US);
System.out.print("Introduce el número 0.3: ");
String leido = teclado.next();
if (leido.equals("o.3")) {
    System.out.print("La lectura con teclado.nextInt() ");
    System.out.println("provocaría un:");
    System.out.println(new InputMismatchException());
    System.out.println("En concreto: ");
    System.out.println(new InputMismatchException("al leer "
        + leido + " por 0.3"));
}
```

y observar atentamente su resultado, que es:

Entrada/Salida Estándar

```
Introduce el número 0.3: o.3
La lectura del dato con teclado.nextInt() provocaría un:
java.util.InputMismatchException
En concreto:
java.util.InputMismatchException: al leer o.3 por 0.3
```

Nótese entonces que el constructor sin parámetros de una clase de la jerarquía `Throwable` tiene asociado por defecto el mensaje `null`.

Pero aún más que su funcionalidad mínima, se debe destacar la característica más novedosa que un objeto `Throwable` tiene frente a los que no lo son: en lugar de ser devuelto (*return*), un objeto `Throwable` se lanza (*throw*) como resultado del método si en su ejecución se origina la excepción, para ello se utiliza la cláusula `throw` que se estudiará en la siguiente sección del capítulo. Esta característica de ser lanzado fuera de la secuencia normal de retorno de datos es precisamente la que da nombre a la clase raíz de la jerarquía y, gracias a ella, es posible ya en la fase de diseño de una aplicación advertir dónde, cuándo y qué fallo de ejecución se ha producido y, en su caso, cuál es la gestión a realizar para evitar sus consecuencias; es por ello también que en la documentación del *API* de Java [Ora16b] de muchos métodos aparece precedida por la palabra **Throws** la información relativa a sus resultados en condiciones anómalas, como se muestra en la figura 14.4 para el método `nextDouble()` de la clase `Scanner` ubicada en `java.util`.

nextDouble

```
public double nextDouble()
```

Scans the next token of the input as a `double`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid double value. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Float* regular expression defined above then the token is converted into a `double` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Double.parseDouble`. If the token matches the localized NaN or infinity strings, then either "Nan" or "Infinity" is passed to `Double.parseDouble` as appropriate.

Returns:

the double scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Float* regular expression, or is out of range

`NoSuchElementException` - if the input is exhausted

`IllegalStateException` - if this scanner is closed

Figura 14.4: Documentación del método `nextDouble()` en `java.util.Scanner`.

14.1.2 Ampliación de la jerarquía `Throwable` con excepciones de usuario

Como ya se ha comentado previamente, la jerarquía `Throwable` del estándar de Java no contempla la representación de los fallos en la lógica de una aplicación precisamente por su carácter *ad hoc*. Una excepción de este tipo se produce al ignorar el resultado atípico que obtiene en ciertas circunstancias alguno de los métodos que usa una aplicación dada. Además, existe otro tipo de errores que tampoco serían advertidos por el compilador ni se considerarían fallos de programación.

En el programa de la figura 14.5 se realiza la lectura de una serie de mediciones de lluvia que se almacenan en un array. En dicho programa pueden tener lugar dos posibles errores, uno de ejecución y otro de la semántica interna del problema. El primero sería un error de ejecución que se produciría en la lectura desde teclado de un `int`, si el usuario introdujese un entero negativo como número de mediciones. En efecto, se produciría la excepción `java.lang.NegativeArraySizeException`.

El segundo error es más sutil. El usuario podría introducir alguna de las mediciones negativa. Esto no provocaría un error de ejecución (en este punto) pero sí podría dar lugar a algún error en otro punto del programa, ya que resulta absurdo que haya llovido, por ejemplo, -5 litros.

Podría resultar útil tener una excepción que indicase precisamente esta circunstancia, por ejemplo, `ExcepcionNumeroNegativo` que se pudiera crear cuando cualquiera de ambos errores se hubiese producido, sin necesidad de esperar a que se produjese un error de ejecución en otro punto del programa debido a dichos errores.

Para poder ser tratado en Java, cualquier fallo en la lógica de una aplicación debe ser representado mediante una nueva clase de la jerarquía `Throwable`, en concreto, como una nueva clase derivada de `Exception`. A estas nuevas clases, para distinguirlas de las que ya figuran en la jerarquía `Throwable` del estándar de Java, se las denomina *excepciones de usuario* (en inglés, *user-defined exceptions*) aunque, en principio, poseen la misma funcionalidad básica y características que cualquier otra con los métodos que heredan de `Exception`.

La clase `ExcepcionNumeroNegativo`, que se define en la figura 14.6, sería una excepción de usuario. Así, el código de la clase `Mediciones` podría lanzar dicha excepción al detectar el número negativo antes de tratar de crear el array o de almacenar un valor negativo en una componente del mismo.

Sin embargo, más allá de detectar o crear las excepciones, lo más importante de las mismas es el concepto de *tratamiento* de una excepción que se aborda en la siguiente sección.

```
import java.util.Locale;
import java.util.Scanner;
/** Clase Mediciones: permite la lectura de mediciones de lluvia
 *  que se almacenan en un array.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class Mediciones {
    /** Método principal.
     *  @param args String[], argumentos del programa.
     */
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in).useLocale(Locale.US);
        double[] elArray = recabarDatosLluvia(teclado);
        System.out.print("Array Mediciones: { ");
        for (int i = 0; i < elArray.length; i++) {
            if (i != elArray.length - 1) {
                System.out.print(elArray[i] + ", ");
            }
            else { System.out.print(elArray[i] + " }"); }
        }
    }

    /** Lectura de las mediciones y construcción del array.
     *  @param t Scanner, desde el que se realiza la lectura.
     *  @return double[], array en el que se almacenan las mediciones.
     */
    public static double[] recabarDatosLluvia(Scanner t) {
        System.out.print("Introduzca el número de mediciones: ");
        int num = t.nextInt();
        double[] resultado = new double[num];
        System.out.println("Introduzca las precipitaciones medidas:");
        for (int i = 0; i < num; i++) {
            System.out.print("Litros de la medición " + (i + 1) + ": ");
            resultado[i] = t.nextDouble();
        }
        return resultado;
    }
}
```

Figura 14.5: Clase `Mediciones`.

14.2 Tratamiento de excepciones

Se ha visto que una excepción provoca la terminación abrupta del programa. Sin embargo, una excepción puede tratarse para evitar dicha terminación abrupta. Existen dos maneras de tratar una excepción: *capturarla* y *propagarla*. Ambos conceptos son sencillos de entender con ejemplos:

- Supóngase que en una base de datos hay que realizar un proceso muy delicado que puede que no salga bien. Si el proceso sufre algún error se perdería todo el trabajo hecho. Por ejemplo, al actualizar la base de datos, si el proceso se interrumpe por un error, la base de datos puede quedar inconsistente.

```
/** Clase ExcepcionNumeroNegativo: lanzada desde clases en las que
 *  se realiza una lectura de datos cuando el token leído no es un
 *  valor numérico >= 0.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ExcepcionNumeroNegativo extends Exception {

    /** Crea un ExcepcionNumeroNegativo con null como mensaje de error. */
    public ExcepcionNumeroNegativo() { super(); }

    /** Crea un ExcepcionNumeroNegativo con msg como mensaje de error.
     *  @param msg String, el mensaje de error.
     */
    public ExcepcionNumeroNegativo(String msg) { super(msg); }
}
```

Figura 14.6: Clase `ExcepcionNumeroNegativo`.

- Supóngase que en una empresa un empleado detecta un error gravísimo en la contabilidad de la misma; sus alternativas son:
  - tratar de arreglar el problema que se ha producido,
  - informar a su superior (el contable) que se encargará del problema. Éste, a su vez, puede considerar que el error es leve y solucionarlo él mismo o, por el contrario, que no tiene autoridad suficiente para resolverlo, con lo cual informará a su superior, el jefe de contabilidad que puede hacerse cargo del problema o informar al subdirector financiero y así sucesivamente.

En el primer caso, lo ideal sería que los cambios de la actualización no se hicieran efectivos hasta que el proceso se hubiese completado con éxito. De esa manera si al *tratar* de hacer el proceso hay algún problema, en realidad, no se ha perdido el resto de la información.

En el segundo caso, el empleado puede *propagar* el informe de detección del problema por toda la escala de mando de la empresa hasta que llegue al nivel adecuado en la jerarquía para resolverlo.

14.2.1 Captura de excepciones: `try/catch/finally`

En el primer ejemplo planteado, se pretendía resaltar que es posible que algunas de las acciones que se realizan en un programa pueden quedar comprometidas por la aparición de una excepción. La solución consiste en capturar dicha excepción y marcar el fragmento de código que se quiere que sólo se realice si la operación que puede lanzar la excepción no se produce. Por ejemplo, en la figura 14.5, la lectura

del `int` que representa el número de mediciones podría realizarse, en el método `recabarDatosLluvia(Scanner)`, como sigue:

```
System.out.print("Introduzca el número de mediciones: ");
int num = t.nextInt();
double[] resultado;
try {
    resultado = new double[num];
} catch (NegativeArraySizeException e) {
    System.out.println("Se ha producido el error " + e);
    System.out.println(num + "<0 --> se le cambia el signo.");
    System.out.println("Array de " + (-num) + " elementos.");
    num *= -1;
    resultado = new double[num];
}
```

La interpretación del código anterior sería la siguiente: se intenta (`try`) efectuar un bloque de instrucciones que puede provocar una excepción. Si se tiene éxito, la ejecución del programa prosigue pero si se produce la excepción esperada, se captura (`catch`) y se arregla el problema antes de continuar; el programa no termina de forma abrupta al producirse el error sino que, simplemente, se guardan valores válidos para continuar, de manera que el resto de trabajo realizado no se pierda.

La sintaxis de las cláusulas `try/catch/finally` es la que sigue:

```
try {

  ... // instrucciones que pueden provocar alguna excepción

} catch (NombreDeExcepcion1 nomE1) {
    ... // instrucciones a realizar si se produce
        // la excepción NombreDeExcepcion1
}

... // puede haber varios catch por cada try

} catch (NombreDeExcepcionX nomEX) {
    ... // instrucciones a realizar si se produce
        // la excepción NombreDeExcepcionX
} finally {
    ... // bloque opcional, instrucciones a realizar tanto
    ... // si se ha producido una excepción como si no
}
```

Básicamente, el significado es como sigue: se intenta (`try`) ejecutar un bloque de código en el que pueden ocurrir errores que se representan con alguna de las

excepciones que se explicitan en los bloques `catch`. Si se produce un error, el sistema lanza una excepción (`throws`) que puede ser capturada (`catch`) en base al tipo de excepción, ejecutándose las instrucciones correspondientes. Finalmente, tanto si se ha producido o no una excepción y si ésta ha sido o no tratada, se ejecutan las instrucciones asociadas a la cláusula `finally`. Al finalizar todo el bloque, la ejecución se reanuda del modo habitual.

A partir de la versión 7.0 de Java, un único bloque `catch` puede utilizarse para capturar más de un tipo de excepción, separando estas por una barra vertical (|), siempre que no haya ninguna que sea subclase de otra.

Siempre que aparece una cláusula `try`, debe existir al menos una cláusula `catch` o `finally`. Nótese que, para una única cláusula `try`, pueden existir tantas `catch` como sean necesarias para tratar las excepciones que se puedan producir en el bloque de código del `try`. Cuando en el bloque `try` se lanza una excepción, los bloques `catch` se examinan en orden, y el primero que se ejecuta es aquel cuyo tipo sea compatible con el de la excepción lanzada. Así pues, el orden de los bloques `catch` es importante. Por ejemplo, el orden en los bloques `catch` que siguen no sería adecuado:

```
} catch (Exception e) {
    ...
} catch (ExcepcionNumeroNegativo e) {
    ...
}
```

Con este orden, el segundo bloque `catch` nunca se alcanzaría, puesto que todas las excepciones serían capturadas por el primero, ya que la excepción `ExcepcionNumeroNegativo` se deriva de la clase `Exception`. Afortunadamente, el compilador advierte sobre esto. El orden correcto consiste en invertir los bloques `catch` para que la excepción más específica aparezca antes que cualquier excepción de una clase antecesora.

El bloque precedido por `finally` es opcional si hay al menos un `catch`. Contiene un grupo de instrucciones que se ejecutarán tanto si el `try` tiene éxito como si no. Aparentemente funciona igual si se situa una instrucción después del último `catch` o dentro del `finally`, como en el ejemplo que sigue:

```
try {
    ... // instrucciones que pueden generar una excepción
    System.out.println("Intento logrado");
} catch (NombreExcepcion e) {
    System.out.println("Capturada!");
} finally {
    System.out.println("Y el finally!");
}
```

En el ejemplo, los puntos suspensivos representan las instrucciones que pueden producir la excepción `NombreExcepcion` capturada en el `catch`. Tanto en un caso como en otro, el final de la ejecución del fragmento sería la escritura del texto "`Y el finally!`". Sin embargo, ¿qué ocurriría si durante la ejecución de los puntos suspensivos se produjese una excepción diferente a `NombreExcepcion`, por ejemplo `OtraExcepcion`? La respuesta es que aún así, antes de la terminación abrupta del código, se efectuaría el `finally`.

```
(sin Excepcion):     (con NombreExcepcion):    (con OtraExcepcion):
  Intento logrado      Capturada!                Y el finally!
  Y el finally!        Y el finally!
```

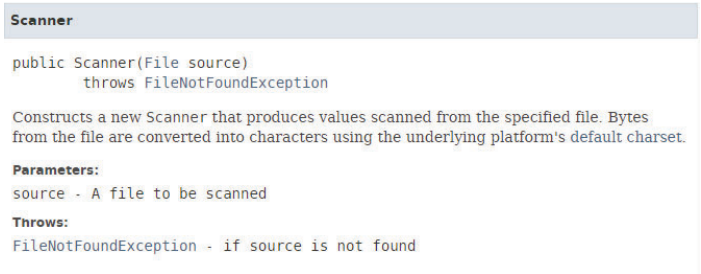
Esto puede quedar más claro si se piensa que en el `finally` pueden ir instrucciones como guardar datos en ficheros o bases de datos, etc. Es decir, aquellas instrucciones que salvaguardan la integridad de la información o de la ejecución.

Ejemplo 14.1. Supóngase que se desea definir un método para leer un entero positivo desde la entrada estándar. Dos son los errores que se prevé puedan producirse: el primero, un error de ejecución si el valor leído no es un entero y el segundo, un error en la lógica de la aplicación si el usuario introduce un entero pero negativo. El siguiente método resuelve el problema capturando la excepción `InputMismatchException` para el primer caso y utilizando una instrucción condicional para el segundo, dando opción en ambos casos a que el usuario introduzca de nuevo un valor. Nótese que, para cualquier posible lectura, siempre se ejecuta la instrucción `tec.nextLine()` de la cláusula `finally`, permitiendo descartar el salto de línea que se se almacena en el buffer de entrada cuando el usuario pulsa la tecla *Enter*.

```
public static int leerEnteroPositivo(Scanner t) {
    boolean salir = false;
    int leido = 0;
    do {
        try {
            System.out.print("Introduce un entero positivo: ");
            leido = t.nextInt();
            if (leido < 0) {
                System.out.println("Error: no es positivo");
            } else { salir = true; }
        } catch (InputMismatchException e) {
            System.out.println("Error: no es un entero");
        } finally {
            tec.nextLine();
        }
    } while (!salir);
    return leido;
}
```

14.2.2 Propagación de excepciones: `throw` versus `throws`

Si la excepción se produce dentro de un método, puede elegirse dónde se trata. Si se trata dentro del método se utilizan las cláusulas ya vistas `try/catch/finally` pero si no, puede decidirse su propagación hacia el punto del programa desde donde se invocó el método. Para hacer esto es necesario que el método tenga en su cabecera la cláusula `throws` y el nombre de la clase de excepción (o clases, separadas por comas) que se propagarán desde dicho método. De este modo, si se produce una excepción dentro del cuerpo del método se abortará la ejecución del mismo y se lanzará la excepción hacia el punto desde donde se invocó al método. En el capítulo 15, se verán distintos métodos que propagan excepciones de entrada/salida como, por ejemplo, el constructor `Scanner(File)`, uno de los constructores de la clase `Scanner`, que propaga la excepción `FileNotFoundException` si el `File` no se encuentra, como se puede ver en la documentación de dicho método en el API de Java [Ora16b], que se muestra en la figura 14.7.



Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

Figura 14.7: Documentación del método `Scanner(File)` en `java.util.Scanner`.

No sólo se puede escoger que el método propague una excepción; también se puede escoger el punto y las condiciones bajo las cuales se produce dicha excepción. Antes de construir (`new`) la nueva excepción se escribe la cláusula `throw` (sin "s" final) de manera que se indica el punto exacto donde se origina y propaga la excepción, como puede verse en el código de la figura 14.8, donde cuando el usuario introduce un entero negativo se lanza la excepción `ExcepcionNumeroNegativo` definida en la figura 14.6. Pese a que no se puedan encontrar en este fragmento de código las cláusulas `try/catch/finally`, la ejecución no termina de forma abrupta en este punto. Lo que sí que sucede es que el método termina sin llegar al `return`, es decir, el método sí termina pero el programa no. La excepción se propaga (en la cabecera del método aparece la cláusula `throws`) hasta el `main`. El bucle `while` no tiene terminación abrupta porque se produzca una excepción en la llamada a `leerEnteroPositivo(Scanner)` al introducir `String` en lugar de `int` o al introducir un entero negativo, sino que se captura (`InputMismatchException` o `ExcepcionNumeroNegativo`, respectivamente) y prosigue hasta que se introduzca un valor válido y se incumpla la condición de continuación del bucle.

```
import java.util.InputMismatchException;
import java.util.Locale;
import java.util.Scanner;
/** Clase LecturaEnteroPositivo: realiza la lectura de un entero
 *  positivo, tratando las excepciones ExcepcionNumeroNegativo
 *  y InputMismatchException que se pueden producir.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class LecturaEnteroPositivo {
    /** Método principal.
     *  @param args String[], argumentos del programa.
     */
    public static void main(String[] args) {
        Scanner tec = new Scanner(System.in).useLocale(Locale.US);
        boolean salir = false;
        while (!salir) {
            try {
                int resp = leerEnteroPositivo(tec);
                System.out.println("Se ha leído el número " + resp);
                salir = true;
            } catch (InputMismatchException e) {
                System.out.println(e + ": Error: no es un entero");
            } catch (ExcepcionNumeroNegativo e) {
                System.out.println(e + ": Error: no es positivo");
            } finally {
                tec.nextLine();
            }
        }
    }

    /** Lectura de un entero >= 0 desde la entrada estándar.
     *  @param t Scanner, desde el que se realiza la lectura.
     *  @return int, entero leído.
     *  @throws InputMismatchException si el token leído no
     *  es un Integer o está fuera de rango.
     *  @throws ExcepcionNumeroNegativo si el token leído no
     *  es un entero >= 0.
     */
    private static int leerEnteroPositivo(Scanner t)
        throws ExcepcionNumeroNegativo {
        System.out.print("Introduce un entero positivo: ");
        int leido = t.nextInt();
        if (leido < 0) {
            throw new ExcepcionNumeroNegativo("número " + leido);
        }
        return leido;
    }
}
```

Figura 14.8: Clase `LecturaEnteroPositivo`.

Es importante no confundir las dos cláusulas: `throws NombreExcepcion` se sitúa en la cabecera del método e indica que, si se produce una excepción de ese tipo, el método la propagará al punto de invocación del mismo; `throw` indica que se lanza la excepción que se crea a continuación.

Un ejemplo de ejecución del método `main` de la clase de la figura 14.8 sería el siguiente:

Entrada/Salida Estándar

```
Introduce un entero positivo: Hola
java.util.InputMismatchException: Error: no es un entero
Introduce un entero positivo: -7
ExcepcionNumeroNegativo: número: -7: Error: no es positivo
Introduce un entero positivo: 10
Se ha leído el número 10
```

Ejemplo 14.2. Para corregir los dos errores, comentados en la sección 14.1.2, que se pueden dar en el método `recabarDatosLluvia(Scanner)` de la clase `Mediciones` de la figura 14.5, se propone el código que se muestra en la figura 14.9. El error de ejecución que se produciría en la lectura desde teclado del número de mediciones, si el usuario: (a) no introdujese un entero o (b) introdujese un entero negativo o cero, se corrige, de manera similar a lo visto en el método `leerEnteroPositivo(Scanner)` del ejemplo 14.1, capturando la excepción `InputMismatchException` en el caso (a) y utilizando una instrucción condicional en el caso (b), dando opción en ambos casos a que el usuario introduzca de nuevo un valor para el número de mediciones. Una vez creado el array, el error que se produciría si el usuario introdujese una medición negativa se corrige lanzando la excepción `ExcepcionNumeroNegativo` y capturándola en el propio método `recabarDatosLluvia(Scanner)`, dando también la opción a que el usuario introduzca una nueva medición.

14.2.3 Excepciones *checked*/*unchecked*

Como ya se ha comentado, todas las situaciones de error se representan en Java como excepciones que hay que tratar. Sin embargo, en los capítulos previos a éste aparecen muchas líneas de código sin tratamiento de excepciones. Esto se explica porque en Java se pueden distinguir dos tipos de excepciones:

- las excepciones *unchecked* o "no comprobadas" que heredan de `RuntimeException` y
- las excepciones *checked* o "comprobadas".

La diferencia está en que para las *checked* el compilador de Java obliga a tratar la excepción en un bloque `catch` o bien propagar la excepción en el `throws` para

```
/** Lectura de las mediciones y construcción del array.
 *  @param t Scanner, desde el que se realiza la lectura.
 *  @return double[], array en el que se almacenan las mediciones.
 */
public static double[] recabarDatosLluvia(Scanner t) {
    boolean salir = false;
    int num = 0;
    do {
        try {
            System.out.print("Introduzca el número de mediciones: ");
            num = t.nextInt();
            if (num <= 0) {
                System.out.println("Error: no es positivo");
            }
            salir = true;
        } catch (InputMismatchException e) {
            System.out.println("Error: no es un entero");
        } finally { t.nextLine(); }
    } while (!salir);

    double[] resultado = new double[num];
    salir = false;
    System.out.println("Introduzca las precipitaciones medidas:");
    for (int i = 0; i < num; i++) {
        do {
            try {
                System.out.print("Litros de la medición "
                    + (i + 1) + ": ");
                resultado[i] = t.nextDouble();
                if (resultado[i] < 0) {
                    String msg = "Error: no es positivo o cero";
                    throw new ExcepcionNumeroNegativo(msg);
                }
                salir = true;
            } catch (InputMismatchException e) {
                System.out.println("Error: no es un real");
            } catch (ExcepcionNumeroNegativo e) {
                System.out.println(e.getMessage());
            } finally { t.nextLine(); }
        } while (!salir);
    }

    return resultado;
}
```

Figura 14.9: Método `recabarDatosLluvia(Scanner)` con tratamiento de excepciones.

obligar al método que invoca a tratarla o propagarla a su vez. Para las *unchecked* esto no es obligatorio, aunque sí es posible. Por ejemplo, en la cláusula `throws` del método `leerEnteroPositivo(Scanner)` de la figura 14.8 podría aparecer la excepción `InputMismatchException` pero no se considera una buena práctica de programación.

Las clases derivadas de `java.lang.Error` también pueden considerarse como *unchecked* puesto que su tratamiento o propagación no es obligatorio.

Nótese que si una excepción de usuario se define como derivada directa de `RuntimeException` debe tratarse como una excepción *unchecked* mientras que si se define como derivada directa de `Exception` debe tratarse como una excepción *checked*. Este es el caso de la excepción de usuario `ExcepcionNumeroNegativo` que:

- en el método `leerEnteroPositivo(Scanner)` de la figura 14.8 se propaga, lo que obliga a tratarla en el `main` de la clase `LecturaEnteroPositivo` y
- en el método `recabarDatosLluvia(Scanner)` de la figura 14.9 se trata en el mismo método.

Si no se propagara o no se tratara, se produciría el error de compilación `unreported exception ExcepcionNumeroNegativo; must be caught or declared to be thrown`.

14.2.4 Documentación de excepciones con la etiqueta `@throws`

La etiqueta `@throws` en los comentarios de documentación [Ora16a; Ora16c] de un método se utiliza para indicar qué excepciones debe capturar el programador (las excepciones *checked*) o puede querer capturar (las excepciones *unchecked*). Por ejemplo, en la documentación del método `leerEnteroPositivo(Scanner)` de la figura 14.8 se ha documentado tanto la excepción *checked* `ExcepcionNumeroNegativo` como la *unchecked* `InputMismatchException`. Después de la etiqueta `@throws` se indica el nombre de la excepción y el contexto que causará que dicha excepción ocurra en el método.

En general, es deseable documentar las excepciones *unchecked* que se puedan producir en un método; esto permite (pero no requiere) que, desde donde se invoque el método, dichas excepciones se puedan gestionar. Pero son tantas las excepciones *unchecked* que se pueden producir por fallos de programación que es difícil poder documentarlas todas y, por ello, no hay que presuponer que un método no pueda lanzar una excepción *unchecked* que no aparezca documentada. Por ejemplo, si el argumento del método `leerEnteroPositivo(Scanner)` es `null`, el método puede lanzar una `NullPointerException`, que es una excepción *unchecked* no documentada.

14.3 Problemas propuestos

1. Escribir una clase de utilidades `LecturaValidada` que permita:

   a) Leer un número entero.

   b) Leer un número entero en un rango determinado.

   c) Leer un número real.

   d) Leer un número real positivo.

   e) Leer un número real en un rango determinado.

   Se deben preveer todos los errores posibles que puedan suceder y tratarlos en los métodos correspondientes.

2. Dada la clase `Figura` que contiene los atributos `String color`, `String tipo` y el método `area()`, se ha definido el método `equals(Object)` de la manera siguiente:

```
public boolean equals(Object o) {
    Figura f = (Figura) o;
    return this.color.equals(f.color)
        && this.nombre.equals(f.nombre)
        && this.area() == f.area();
}
```

   Sin embargo, esto no es del todo correcto. Si se ejecutasen las siguientes instrucciones:

```
Figura f1 = new Figura("rojo", "cuadrado");
Figura f2 = new Figura("rojo", "cuadrado");
Double d = new Double(1.0);
String k = "Hello";
boolean b1 = f1.equals(f2);
boolean b2 = d.equals(k);
boolean b3 = k.equals(f2);
boolean b4 = f1.equals(d);
```

   `b1`, `b2` y `b3` se evaluarían, respectivamente, a `true`, `false` y `false` pero al evaluar `b4` se produciría la excepción *unchecked* `ClassCastException`.

   Modificar el método `equals(Object)` para que si se produce la excepción *unchecked* `ClassCastException` ésta sea capturada y tratada en el cuerpo de dicho método y se comporte como el `equals(Object)` definido en la clase `Object`.

3. Se está implementando una aplicación para la gestión de una agenda telefónica. Se dispone de un método `menu(Scanner)` que muestra por pantalla un menú de opciones:

```
private int menu(Scanner teclado) {
    System.out.println("      Menú de Agenda ");
    System.out.println("--------------------------");
    System.out.println("1.- Cargar Fichero Agenda");
    System.out.println("2.- Guardar Fichero Agenda");
    System.out.println("3.- Buscar Nombre");
    System.out.println("4.- Insertar Nuevo Nombre");
    System.out.println("5.- Eliminar Nombre");
    System.out.println("0.- Salir");
    System.out.print("Seleccione [0..5]: ");
    return teclado.nextInt();
}
```

Dicho método es invocado desde otro método como sigue:

```
Scanner tec = new Scanner(System.in).useLocale(Locale.US);
        ...
int opcion = menu(tec);
switch (opcion) {
    case 0:
        ...
    case 1:
        ...
    case 2:
        ...
    case 3:
        ...
    case 4:
        ...
    case 5:
        ...
}
```

Cuando el programa es probado por el usuario, se detecta que, en ocasiones, el programa aborta su ejecución porque el usuario se equivoca y escribe números que no están en el menú o texto en lugar de dichos números.

Se pide:

a) Definir una nueva excepción de usuario `NumeroFueraDeRango` para identificar el error de que el usuario haya escrito un número de opción en el menú fuera del rango [0..5].

b) Modificar el método `menu(Scanner)` para que, en caso de que el número introducido esté fuera del rango [0..5], se lance la nueva excepción conteniendo como mensaje "la opción elegida ha sido X".

c) Modificar el fragmento de código dado para que se detecte si el número introducido está en el rango correcto y si no es así se vuelva a presentar el menú hasta que el usuario acierte.

d) Modificar el fragmento de código dado para que también se detecte si el usuario ha introducido un valor que no sea un entero y si no es así se vuelva a presentar el menú.

4. Dado el siguiente fragmento de código:

```
public static void metodo1() throws Excepcion1, Excepcion2 {
    ... (1)
}

public static void metodo2(){
    try {
        ... (2)
    } catch (IndexOutOfBoundsException e) {
        System.out.println("texto0");
    }
}

public static void metodo3() throws Excepcion3, Excepcion1 {
    ... (3)
}

public static void main(String[] args) {
    try {
        metodo1();
        metodo2();
        metodo3();
    } catch (Excepcion1 e) {
        System.out.println("texto1");
    } catch (Excepcion2 e) {
        System.out.println("texto2");
    } catch (Excepcion3 e) {
        System.out.println("texto3");
    } catch (InputFormatException e) {
        System.out.println("texto4");
    } finally {
        System.out.println("texto5");
    }
}
```

Indicar qué aparece por pantalla si en los puntos suspensivos marcados se producen las siguientes excepciones:

a) En `(1)` la excepción de usuario `Excepcion1`.

b) En `(1)` la excepción *unchecked* `IndexOutOfBoundsException`.

c) En `(1)` la excepción de usuario `Excepcion2`.

d) En `(2)` la excepción *unchecked* `InputFormatException`.

e) En `(2)` la excepción *unchecked* `IndexOutOfBoundsException`.

f) En `(3)` la excepción de usuario `Excepcion3`.

g) En `(3)` la excepción *unchecked* `InputFormatException`.

5. Ampliar la clase `LecturaValidada` del ejercicio 1 para que permita leer un `String` que deberá pertenecer a los existentes previamente en un array de elementos de dicho tipo. En caso de que el `String` leído exista en el array, el método de lectura que se construya deberá devolver el índice con la posición en el array del `String` leído o, en caso de no existir, deberá lanzar la excepción `ElementoNoExistente` que se habrá definido previamente.

   Por ejemplo, dada la declaración de la constante array:

   ```
   public static final String[] COMPOSITORES = {"Bach", "Haydn",
       "Mozart", "Beethoven", "Brahms", "Mahler", "Bartok"};
   ```

   El método que se pide deberá, si se utiliza el array `COMPOSITORES` al ejecutarlo, leer un `String` desde el teclado, devolviendo la posición del texto encontrado en el array (en el caso del ejemplo, el nombre del compositor) o lanzar la excepción correspondiente (`ElementoNoExistente`), en caso de que éste no exista.

Más información

[Eck15] D.J. Eck. *Introduction to Programming Using Java, Seventh Edition*. 2015. URL: `http://math.hws.edu/javanotes/`. Capítulo 3 (3.7) y Capítulo 8 (8.3).

[GJo15] J. Gosling, B. Joy y otros. *The Java® Language Specification, Java SE 8 Edition*, 2015. URL: `https://docs.oracle.com/javase/specs/jls/se8/html/`.

[Ora16a] Oracle. *How to Write Doc Comments for the Javadoc Tool*, 2016. URL: `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`.

[Ora16c] Oracle. *Javadoc Tool*, 2016. URL: `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`.

[Ora15] Oracle. *The Java^{TM} Tutorials*, 2015. URL: `http://download.oracle.com/javase/tutorial/`. Trail: Essential Java Classes. Lesson: Exceptions.

[SM16] W.J. Savitch y K. Mock. *Absolute Java, Sixth Edition*. Pearson Education, 2016. Capítulo 9.

[WP00] R. Wiener and L.J. Pinson. *Fundamentals of OOP and data structures in Java*. Cambridge University Press, 2000. Capítulo 7.