Introducción

En la mayoría de los programas, en un momento u otro hay que interaccionar con alguna fuente o soporte de datos, como un archivo en un dispositivo de almacenamiento de datos o un dispositivo de red, ya sea para guardar información o para recuperarla. Java implementa una serie de clases llamadas *flujos,* encargadas de comunicarse con estos dispositivos. El funcionamiento de estos flujos, desde el punto de vista del programador, no depende del tipo del dispositivo hardware con el que está asociado, lo que nos liberará del trabajo que supone tener en cuenta las características físicas de cada uno de ellos.

Los flujos pueden ser de entrada o de salida, según sean para recuperar o guardar información. Por otra parte, atendiendo a la clase de datos que se transmiten, los flujos son de dos tipos:

- **Carácter:** si se asocian a archivos u otras fuentes de texto.
- **Binarios:** si transmiten bytes, cuyos valores están comprendidos entre 0 y 255. Estos, en realidad, permiten manipular cualquier tipo de datos.

Vamos a empezar por los flujos de tipo texto, mientras que en la siguiente unidad nos dedicaremos a los archivos —flujos— de tipo binario. Pero, dado que en los flujos de datos entre el ordenador y los dispositivos de almacenamiento se producen errores frecuentes, como intentos de acceso a ficheros inexistentes o con nombres mal escritos, tendremos que estudiar las **excepciones,** que es como se llaman los errores en Java.

Argot técnico

Un flujo (*stream,* en inglés) es una abstracción que permite que un programa se conecte con un dispositivo físico (un disco duro, un puerto de red, un DVD, etc.) para recibir o enviar información.

No debemos confundir los flujos de datos con los objetos de tipo Stream, que estudiaremos en una unidad posterior.

10.1. Excepciones

Los errores en los programas son prácticamente inevitables, tanto si están originados por códigos deficientes, entrada de datos, parámetros incorrectos o archivos inexistentes como si lo están por discos defectuosos, entre otros.

En la mayoría de los lenguajes de programación, la manipulación de los errores suele ser complicada y confusa. Su código se mezcla con el del resto del programa, haciéndolo poco claro, y suele estar destinado solo a evitar el error y no a controlar la situación una vez que dicho error se ha producido.

Java es un lenguaje que se adapta a las condiciones de internet, y sus programas se ejecutarán en máquinas remotas, casi siempre manipuladas por personal no cualificado.

Esto obliga a adoptar un enfoque nuevo, en el que se trata de evitar, en lo posible, que el programa se interrumpa a causa del error. Para ello no basta con evitar que se produzcan los errores, sino que hay que implementar los medios para que un programa se recupere de las condiciones generadas por un error que no se ha podido evitar. Eso depende del tipo de error, y no siempre es posible.

Cuando, en la ejecución de un programa, se produce una situación anormal —un error— que interrumpe el flujo normal de ejecución, el método que se esté ejecutando genera un objeto de la clase `Throwable` («arrojable»), que contiene información del error, de su causa y del contexto del programa en el momento en que se produce, y lo entrega (lo «arroja») al sistema de tiempo de ejecución —la máquina virtual—. Este objeto es susceptible de ser capturado —ya veremos cómo— por el programa y analizado para dar una respuesta, si procede. En caso contrario, el programa se interrumpe y el sistema de tiempo de ejecución muestra una serie de mensajes que describen el error, como ocurre en otros lenguajes de programación.

Hay errores lo bastante graves para que sea preferible que el programa se interrumpa. Por ejemplo, errores derivados de problemas de hardware. Si intentamos leer de un disco defectuoso, se producirán errores de los que es difícil, si no imposible, recuperarse. Estos y otros errores relacionados directamente con la máquina virtual, y no con el código de nuestro programa, arrojan objetos de la clase `Error`, una subclase de `Throwable`. De ellos no nos vamos a ocupar, ya que poco se puede hacer con estos errores a la hora de programar. Tampoco nos ocuparemos de las excepciones llamadas *de tiempo de ejecución* (`runtime exceptions`), que proceden de líneas de código equivocadas. La solución a estos errores es corregir el código.

Pero hay otra clase de errores más habituales y menos graves, como entradas de datos de tipo equivocado, aperturas de ficheros con ruta de acceso errónea u operaciones aritméticas no permitidas. A estos errores se les llama *excepciones,* y producen un objeto de la clase `Exception`, otra subclase de `Throwable`. A continuación explicaremos estas excepciones, ya que son manipulables a través del código. Para ello se usan los bloques `try`, `catch` y `finally`.

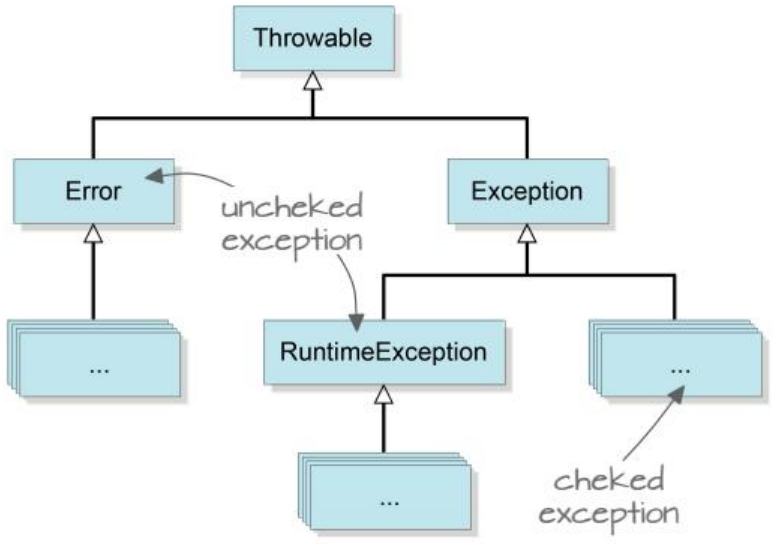

Figura 10.1. *Estructura de las clases que forman el sistema de excepciones de Java.*

Cuando sabemos que en un determinado fragmento de código se puede producir una excepción, lo encerramos dentro de un bloque `try`. Por ejemplo, si en una división entre las variables `a` y `b` sospechamos que el divisor `b` podría hacerse cero en algún momento, escribimos

```
try {
        int c;
        c = a / b;
   }
```

Con esta estructura estamos sometiendo a observación al bloque de código encerrado entre llaves. Si salta una excepción en ese bloque, deberá ser capturada por un bloque `catch` de la siguiente forma:

```
catch(ArithmeticException e) {
        System.out.println("Error: división por cero");
}
```

Si se produce la excepción y es capturada, se interrumpe la ejecución del código del bloque `try`, saltando a la primera línea del bloque `catch`. Cuando se termina de ejecutar dicho bloque, continúa en la línea inmediatamente posterior a la estructura `try-catch`.

La palabra reservada `catch` va seguida de unos paréntesis donde se encierra un parámetro de la clase de excepción que puede atrapar; en este caso, una excepción aritmética. El parámetro `e` se puede usar como variable local dentro del bloque. Hace referencia al objeto de la excepción y contiene toda la información sobre el error que la ha producido. Entre sus métodos está `getMessage()`, que nos muestra un mensaje descriptivo del error. Podríamos haber puesto

```
catch(ArithmeticException e) {
        System.out.println(e.getMessage());
}
```

o incluso

```
catch(ArithmeticException e) {
        System.out.println(e);
}
```

que hace una llamada a `toString()` de la clase de la variable `e`, donde también se describe el error (en inglés).

Los bloques `try` y `catch` deben ir uno a continuación del otro, sin ningún código en medio.

```
try {
        //…bloque try
} catch(TipoExcepción nombreParámetro) {
        //…bloque catch
}
```

En el bloque `catch` solo se recogerán las excepciones del tipo declarado entre paréntesis o de una subclase. En el ejemplo anterior podríamos haber escrito lo siguiente:

```
try {
        c = a / b;
} catch(Exception e) {
        System.out.println("Error: división por cero");
}
```

ya que `ArithmeticException` es una subclase de `Exception`.

Esto nos permitiría recoger otros tipos de excepción en el mismo `catch`, pero, para ese caso, es mejor escribir más de un bloque `catch`. Con un mismo bloque `try` se pueden poner tantos bloques `catch` como deseemos, siempre que vayan seguidos,

```
try {
        //…bloque try
} catch(tipoExcepción1 nombreParámetro1) {
        //…bloque catch
} catch(tipoExcepción2 nombreParámetro2) {
        //…bloque catch
} ...
```

Cuando se produce una excepción en el bloque `try`, se compara con el tipo del primer bloque `catch`. Si coincide con él o con una subclase, se ejecuta dicho bloque y continúa el programa después del último bloque `catch`. Si no coincide, se compara con el tipo del segundo bloque, y así sucesivamente hasta que se encuentra un bloque cuyo parámetro coincida o sea una superclase de la excepción producida, de forma que solo se ejecuta un bloque `catch`, el primero cuyo tipo sea compatible. Si la excepción no coincide con ninguno de los parámetros de los bloques, no es capturada y se interrumpe la ejecución del programa. Aquí hay que tener cuidado de no poner un bloque `catch` con una excepción que sea superclase de otra que vaya más abajo, pues el bloque de esta última nunca se ejecutará.

Por ejemplo:

```
try {
        c = a / b;
} catch(Exception e) {
        System.out.println("Estoy en el primer catch");
} catch(ArithmeticException e) {
        System.out.println("Estoy en el segundo catch");
}
```

Si `b` vale cero, se producirá una excepción de tipo `ArithmeticException`, pero, al ser un subtipo de `Exception`, será capturada en el primer `catch`, cuyo bloque será el que se ejecute, apareciendo el mensaje: «Estoy en el primer `catch`». La ejecución seguirá después del último bloque, de modo que el segundo bloque `catch` es inútil, ya que jamás se va a ejecutar. De hecho, en nuestro ejemplo —como todo tipo de excepción es subclase de `Exception`— cualquier excepción que se produzca en el bloque `try` será capturada en el primer bloque `catch` y ningún bloque que pongamos después se va a ejecutar nunca.

Por otra parte, existe la posibilidad de capturar más de un tipo de excepción con un único bloque `catch`.

```
catch(tipoExcepción1 | tipoExcepción2 | ... e) {
        ...
}
```

Aquí, la barra vertical equivale a una disyunción lógica `O`. Se pueden añadir tantos tipos de excepción como se quiera, separados por barras verticales. El significado es: «si se arroja una excepción del tipo `tipoExcepción1` o del tipo `tipoExcepción2` o `. . .`, ejecutar este bloque `catch`, donde la excepción será referenciada con la variable `e`».

Todo código en Java forma parte de algún método que, en última instancia, puede ser el método `main()`. Si desde un método `metodo2()` se invoca otro método `metodo1()` y salta una excepción durante la ejecución de este último, podemos capturarla y manipularla, como hemos hecho hasta ahora, con una estructura `try-catch` en el propio código de `metodo1()`. Pero hay un enfoque alternativo. Podemos declarar, en la definición de `metodo1()`, que en su ejecución puede producirse dicha excepción. Para ello usaremos la palabra clave `throws`.

```
tipo metodo1(tipo1 parametro1,...) throws tipoExcepción {
        ...
}
```

En el cuerpo de `metodo1()` ya no tenemos que insertar los bloques `try-catch` para ese tipo de excepción. En cambio, `metodo2()` será el encargado de gestionarla cuando invoque a `metodo1()`.

```
tipo metodo2(tipo1 parametro1, tipo2 parametro2...) {
        ...
        try {
                metodo1(); //llamada al metodo1()
        } catch (tipoExcepción e) {
                //…tratamiento de la excepción
        }
}
```

Veámoslo con un ejemplo. Supongamos que en método `metodo1()` se puede dar una división por cero. Por otra parte, un segundo método `metodo2()` llama a `metodo1()`. Lo que hemos hecho hasta ahora es:

```
void metodo1(int a, int b) {
        int c;
        try {
                c = a / b;
        } catch(ArithmeticException e) {
                System.out.println("División por cero");
        }
        System.out.println("a/b = " + c);
}
```

`metodo2()`, que llama a `metodo1()`, podría ser:

```
void metodo2() {
        int x, y;
        ...
```

```
        metodo1(x, y);
        ...
}
```

Si la variable `y` es cero, la excepción producida es capturada y manipulada en el lugar donde se ha intentado la división —en `metodo1()`— antes de que la ejecución vuelva al método que lo llama —`metodo2()`—.

Pero podríamos hacerlo de otra forma. Primero, eliminamos el bloque `try-catch` de `metodo1()` y declaramos a este último como susceptible de producir una `ArithmeticException`. Esto se implementa por medio de la palabra clave `throws` en su encabezamiento,

```
void metodo1(int a, int b) throws ArithmeticException {
        int c;
        c = a / b;
        System.out.println("a/b = " + c);
}
```

Con esto estamos declarando que, dentro del método, puede producirse una excepción aritmética y que deberá ser manipulada por código externo, en el método que llame a `metodo1()`, en nuestro caso `metodo2()`. Además, esta particularidad, que forma parte de la definición de `metodo1()`, deberá constar en la documentación que la acompaña para que los usuarios del método sepan a qué atenerse. La implementación de `metodo2()`, por tanto, deberá hacerse cargo de la excepción,

```
void metodo2() {
        int x, y;
        ...
        try {
                metodo1(x, y);
        } catch(ArithmeticException e) {
                System.out.println("División por cero");
        }
        ...
}
```

Por supuesto, `metodo2()` podría «pasar» la excepción a un tercer método que lo invoque, y así sucesivamente.

Una estructura `try-catch` supone una bifurcación en el programa. Pero a menudo estamos interesados en que una serie de líneas de código se ejecuten, tanto si se produce una excepción como si no; por ejemplo, para cerrar un archivo en el que estábamos escribiendo.

Para esto se define el bloque opcional `finally` que, cuando está presente, se coloca al final de una estructura `try-catch` —es posible una estructura `try-finally`, sin bloque `catch`— y se ejecuta independientemente de si en el bloque `try` se ha lanzado una excepción o no, y de si la excepción ha sido capturada o no. Se ejecutará antes incluso que cualquier sentencia `return` que aparezca en los bloques `try` o `catch`. Esto nos asegura que, en circunstancias anómalas, se van a ejecutar determinadas tareas, como cerrar ficheros abiertos, antes de que termine la ejecución del programa. En el siguiente trozo de código:

```
try {
        //…bloque que trabaja con archivos
        return
} catch(IOException e) {
        //…bloque si salta excepción
} finally {
        //…código para cerrar archivos
}
...
return;
```

el bloque `finally` se ejecuta incluso antes de ejecutarse el `return` del bloque `try`, a pesar de que figura después de dicha sentencia, tanto si se produce una excepción como si no.

10.1.1. Requisito de captura o especificación

Al principio distinguimos entre las excepciones —clase `Exception`— y los errores propiamente dichos —clase `Error`—. Pero, entre las excepciones, hay un grupo especialmente importante, ya que son predecibles a partir del código y es fácil recuperarse de ellas. Tanto es así que el propio compilador sabe dónde se pueden producir y nos obliga a manipularlas, ya sea por medio de estructuras `try-catch` o declarándolas en el encabezamiento de los métodos (mediante `throws`).

Este grupo de excepciones, llamadas **excepciones comprobadas** —*checked exceptions*—, entre las que se encuentran las más comunes, generalmente con un origen fuera del programa (entradas de datos, nombres de ficheros incorrectos, etc.), se dice que están sometidas al requisito de «capturar o especificar» (*catch or specify*); es decir, o implementamos el bloque `try-catch`, o especificamos la excepción en la declaración del método por medio de `throws`. Como el compilador nos exige su tratamiento (si no lo hacemos, genera un error de compilación), no tenemos que preocuparnos por saber cuáles son exactamente, aunque con la práctica acabamos familiarizándonos con las más importantes: `IOException`, `FileNotFoundException`, `NumberFormatException`, `ClassCastException`, etcétera.

Junto a ellas se encuentran las excepciones **no comprobadas** (*unchecked exceptions*), como `ArithmeticException`, producidas por errores aritméticos, o `ArrayIndexOutOfBoundsException`, que se produce cuando intentamos salirnos de los límites de una tabla. Dichas excepciones suelen estar asociadas a malas prácticas de programación. Por tanto, más que tratarlas con una estructura `try-catch`, hay que corregir el código para evitar que se produzcan.

Argot técnico

Se llaman *excepciones comprobadas* o *checked exception* a aquellas cuya posible ocurrencia es predecible y, por tanto, anticipable por el compilador, que nos exige su tratamiento por medio de una estructura `try-catch` adecuada o por la cláusula `throws`.

Actividad resuelta 10.1

Escribir el método

```
Integer leerEntero(),
```

que pide un entero por consola, lo lee del teclado y lo devuelve. Si la cadena introducida por consola no tiene el formato correcto, muestra un mensaje de error y vuelve a pedirlo.

Solución

```
/* Establecemos un bucle infinito del que solo nos puede sacar el break que, por
otra parte, solo se ejecutará si se produce la lectura de Scanner sin que salte
una excepción InputMismatchException por una entrada de tipo erróneo. */
static Integer leerEntero () {
        Integer resultado;
        while (true) {
                try {
                        System.out.print("Introducir entero: ");
                        resultado = new Scanner(System.in).nextInt();
                        break; /*aquí se llega solo si la lectura del Scanner ha
                        sido correcta*/
                } catch (InputMismatchException ex) {
                        System.out.println("Tipo erróneo");
                }
        }
        return resultado;
}
```

10.1.2. Excepciones de usuario

Hasta ahora, todas las excepciones que hemos visto vienen predefinidas en Java. Pero podemos implementar las nuestras propias con tan solo heredar de alguna que ya exista. Además, es posible lanzarlas cuando nos interese por medio de la palabra reservada `throw`.

Por ejemplo, si estamos introduciendo por teclado un valor entero que representa una edad, no tiene sentido un número negativo. Normalmente, en estos casos nos conformamos con un condicional y un simple mensaje de error. Pero también podemos crear con `new` y lanzar con `throw` una excepción definida por nosotros.

En la definición de una excepción de usuario no necesitamos implementar ningún método; basta con heredar de `Exception`. En todo caso, podemos sustituir el método `toString()` por uno que represente mejor nuestro caso.

```
public class ExcepcionEdadNegativa extends Exception {
        public String toString(){
                return "Edad negativa";
        }
}
```

Veamos un ejemplo donde se usa esta excepción:

```
try {
        System.out.print("Introducir edad: ");
        int edad = new Scanner(System.in).nextInt();
        if (edad < 0) {
                throw new ExcepcionEdadNegativa();
        } else {
                //cualquier código donde se use edad, por ejemplo:
                System.out.println("edad correcta: " + edad);
        }
} catch (ExcepcionEdadNegativa ex) {
        System.out.println(ex);
}
```

10.2. Flujos de entrada de texto

Los flujos de entrada de tipo texto heredan de la clase `InputStreamReader`. Todas las clases que vamos a usar para trabajar con ficheros se encuentran en el paquete `java.io`. Podemos importarlas todas con la sentencia,

```
import java.io.*;
```

Las clases de entrada de texto tienen siempre un nombre que termina en `Reader`. Nosotros usaremos flujos del tipo `FileReader`. El constructor es:

```
FileReader(String nombreArchivo)
```

al que se le pasa, como parámetro, el nombre del archivo al que se quiere asociar un flujo para su lectura. Este nombre puede llevar incluida la ruta de acceso si el archivo no está en la carpeta de trabajo. Por ejemplo,

```
FileReader in = new FileReader("C:\\programas\\prueba.txt");
```

crea un flujo de texto asociado al archivo *prueba.txt,* que se halla en la carpeta *programas* de la unidad C. No hay que olvidar que, para imprimir la barra invertida, hay que escribirla doble (\\), ya que escrita de forma simple se emplea para las secuencias de escape (por ejemplo, «\n» significa un cambio de línea). Cuando estamos trabajando en una plataforma Linux, las rutas de acceso son distintas. Por ejemplo,

```
FileReader in = new FileReader("/home/pedro/programas/prueba.txt");
//la barra hacia delante (/) se puede escribir simple
```

La apertura de un fichero puede arrojar una excepción del tipo `IOException` —Input-Output Exception, excepción de entrada y salida— o alguna subclase, cuando el archivo no se abre por alguna razón. Por ejemplo, el constructor de FileReader puede arrojar una excepción FileNotFoundException, que hereda de IOException si el archivo que queremos abrir para lectura no existe, o no se encuentra en la ruta de acceso especificada. Por ello, dicha operación siempre deberá ir dentro de la estructura `try-catch` correspondiente. Cuando se abre un archivo, un cursor se posiciona al principio, apuntando al primer carácter, e irá avanzando por el archivo a medida que vayamos leyendo de él.