Excepciones

Introducción

Una excepción es cualquier situación que surge durante la ejecución de un programa que impide continuar con el algoritmo en curso, al causar que se incumplan las condiciones para su funcionamiento. Consideremos, por ejemplo, la siguiente clase, cuyo método *main* ejecuta un sencillo algoritmo que pide un número al usuario, calcula su factorial, llamando al método auxiliar *fact*, y lo muestra en panatalla:

```
 1.  import java.util.Scanner;
 2.
 3.  public class Excepciones {
 4.    private static int fact(int n) {
 5.      if (n == 0) {
 6.        return 1;
 7.      } else {
 8.        return n * fact(n - 1);
 9.      }
10.    }
11.
12.    public static void main(String[] args) {
13.      Scanner input = new Scanner(System.in);
14.      System.out.print("Entra un número entero no negativo: ");
15.      int num = input.nextInt();
16.      System.out.println("El factorial de " + num + " es " + fact(num));
17.      input.close();
18.    }
19.  }
```

Si, al ejecutarlo, el usuario, por error o intencionadamente, teclea algo que no se puede representar como un valor de tipo *int*, al intentar leerlo en la línea 15, se produciría el siguiente resultado:

```
Entra un número entero no negativo: a
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 Excepciones.main(Excepciones.java:15)
```

Este mensaje indica que en la línea 15 del fichero *Excepciones.java*, en el método *main* de la clase Excepciones, se ha producido una excepción de la clase *java.util.InputMismatchException* (el formato

de la entrada no coincide con lo que se espera). La ejecución del programa se aborta al no poder continuar con las acciones previstas.

Cómo señalar la ocurrencia de una excepción

InputMismatchException es una clase de excepción definida en el paquete *java.util*. Cuando, en un programa Java, se detecta una situación anómala, se construye un objeto en el que se almacena toda la información sobre la excepción y se "lanza" para que, en su caso, alguna otra parte del programa pueda recogerlo y usar dicha información para intentar reconducir la situación. Los objetos que reúnen información sobre excepciones son conocidos asimismo como excepciones, con lo que tenemos dos definiciones para el término excepción:

1. Una situación anómala que ocurre durante la ejecución de un programa
2. Un objeto que reúne información sobre una situación anómala ocurrida durante la ejecución de un programa

Los objetos que se construyen para almacenar información sobre las excepciones deben ser de alguna clase derivada, directa o indirectamente, de la clase *java.lang.Throwable*, cuya jerarquía se muestra en la Ilustración 1.

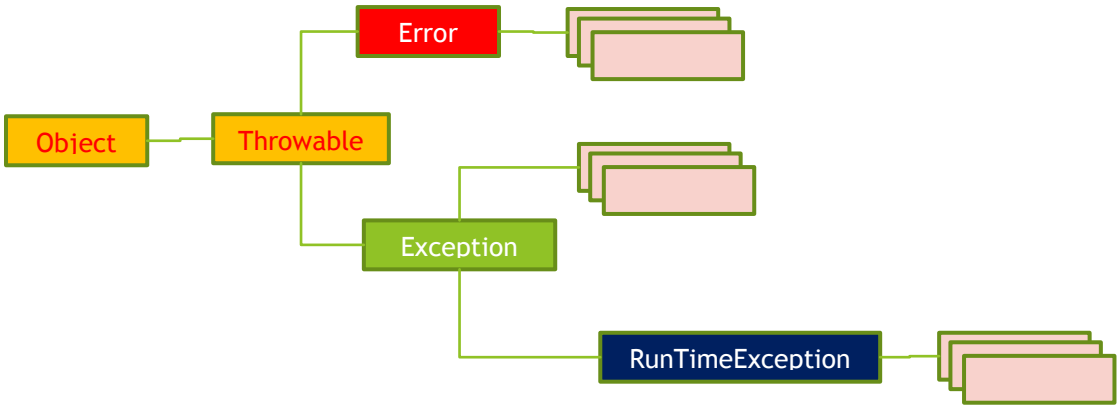

Ilustración 1

En los diferentes paquetes de Java ya hay definidas numerosas clases de excepciones, y el programador puede definir otras nuevas, específicas de cada problema. De la jerarquía que muestra la Ilustración 1, la clase *Error* y sus derivadas se reservan para errores graves en el funcionamiento de la máquina virtual de Java. Por su parte, la clase *RunTimeException* y sus derivadas se reservan para situaciones

relacionadas con el mal uso de la API (intentar acceder a un objeto usando una referencia nula, *NullPointerException*; usar un índice erróneo para acceder a un array, *IndexOutOfBoundsException*; intentar leer datos con un formato erróneo, *InputMismatchException*;…). Las excepciones específicas de un problema se derivan normalmente de la clase *Exception* o clases derivadas de ella que no sean derivadas de *RunTimeException*.

La clase *java.lang.Throwable* ofrece constructores para crear un objeto con un mensaje y, en su caso, la causa de la excepción, así como otras para almacenar información y acceder a la información almacenada.

Constructores	
Throwable()	Throwable(String message)
Throwable(String message, Throwable cause)	Throwable(Throwable cause)
Métodos	
Throwable fillInStackTrace()	Throwable initCause(Throwable cause)
Throwable getCause()	void printStackTrace()
String getLocalizedMessage()	void printStackTrace(PrintStream s)
String getMessage()	void printStackTrace(PrintWriter s)
StackTraceElement[] getStackTrace()	void setStackTrace(StackTraceElement[] stackTrace)

Un programador puede definir nuevas clases de excepciones, derivándolas, normalmente, de la clase *java.lang.Exception*, o de una derivada de esta, para representar excepciones específicas que pueden surgir en el contexto de un problema concreto.

Por ejemplo, en el caso del cálculo del factorial, el método *fact* solo puede funcionar correctamente si se le pasa un valor no negativo. Si se le pasa un valor negativo, se obtendría el resultado que se muestra a continuación:

```
Entra un número entero no negativo: -1
Exception in thread "main" java.lang.StackOverflowError
      at Excepciones.fact(Excepciones.java:5)
      at Excepciones.fact(Excepciones.java:8)
      at Excepciones.fact(Excepciones.java:8)
      at Excepciones.fact(Excepciones.java:8)
      …
```

Al recibir un valor negativo, el método no converge nunca hacia el caso base, dando lugar a una secuencia infinita de llamadas recursivas que se ve interrumpida cuando se llena la pila de ejecución (*stack*), lo que se identifica como *StackOverflowError* (error de desbordamiento de la pila).

Podemos definir una clase de excepción específica para lidiar con esta situación:

```
public class NegativeNumberException extends Exception {
    private int num;

    public NegativeNumberException(int num) {
        super("No se puede calcular el factorial de un número
negativo");
        this.num = num;
    }

    public String getMessage() {
        return super.getMessage() + ": " + num;
    }
}
```

La excepción *NegativeNumberException* se define como una clase derivada de *Exception* con un campo, *num*, donde se almacenará el número que ha causado el problema. Se le ha dotado de un constructor que almacena un mensaje en la clase base y almacena localmente el número, y de una operación *getMessage*, que compone un mensaje combinando el mensaje almacenado en la clase base, con el número almacenado localmente. Además, la clase *NegativeNumberException* puede hacer uso del resto de los constructores y métodos de la clase *Exception*.

Ahora, podemos modificar el método *fact* para que notifique esta excepción cuando se dé el caso:

```
private static int fact(int n) throws NegativeNumberException {
    if (n < 0) {
        throw new NegativeNumberException(n);
    } else if (n == 0) {
        return 1;
    } else {
        return n * fact (n - 1);
    }
}
```

Lo que se hace es comprobar antes que nada si el número es negativo, y si es así, en vez de intentar el cálculo, se construye y se lanza (*throw*) un objeto de la clase *NegativeNumberException*. Además, se ha tenido que modificar la cabecera del método para indicar que puede lanzar (*throws*) una excepción del tipo *NegativeNumberException*.

En principio, se supone que un programa tiene pocas posibilidades de recuperarse de un problema de la clase *Error* y no muchas de uno de la clase *RunTimeExeception*. Sin embargo, en general, debería controlar cualquier otro problema de la clase *Exception*. En consecuencia, un método, como *fact*, en

que se pueda producir una excepción derivada de la clase *Exception*, debe incluir código para controlarla o, si la "deja pasar" para que llegue a quién lo llamó, indicarlo añadiendo la cláusula *throws* en su interfaz. Si un método "deja pasar" o lanza una excepción, el método que lo llamó debe controlarla, o añadir, a su vez la correspondiente cláusula *throws* en su interfaz. En última instancia, la excepción llegará al método *main*, que deberá controlarla.

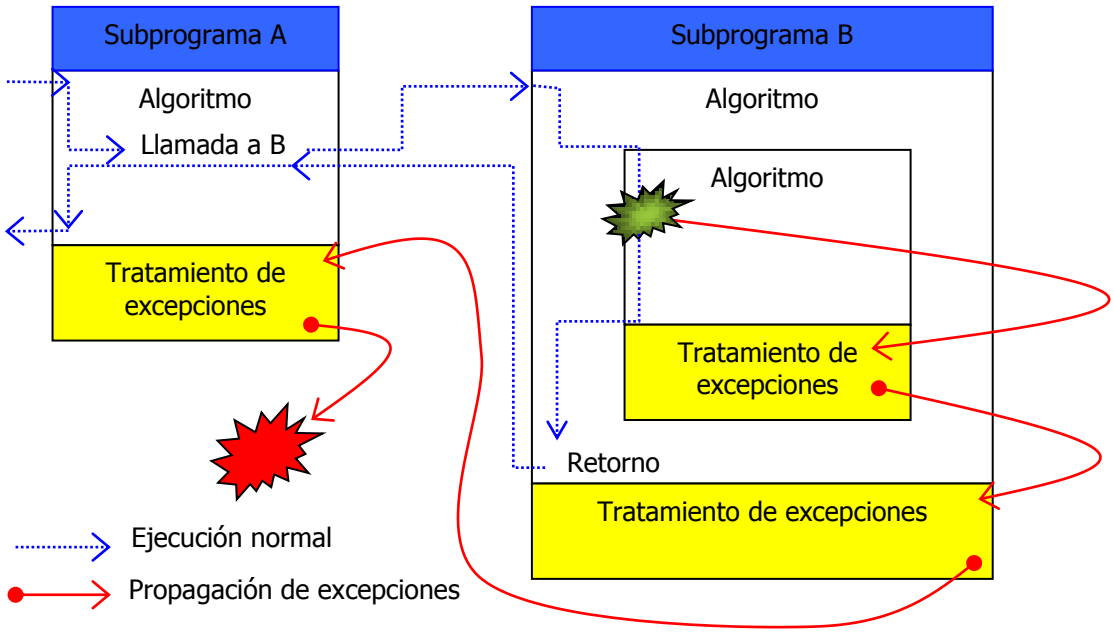

Ilustración 2

Las excepciones que deriven de *RunTimeException* no requieren que se añada la cláusula *throws* cuando un método no las controla.

Cómo controlar las excepciones

Controlar una excepción consiste en detener su propagación e intentar reconducir el programa. Esta reconducción puede, según la situación, conseguir que el programa continúe como si no hubiera pasado nada, o hacer que termine de forma controlada.

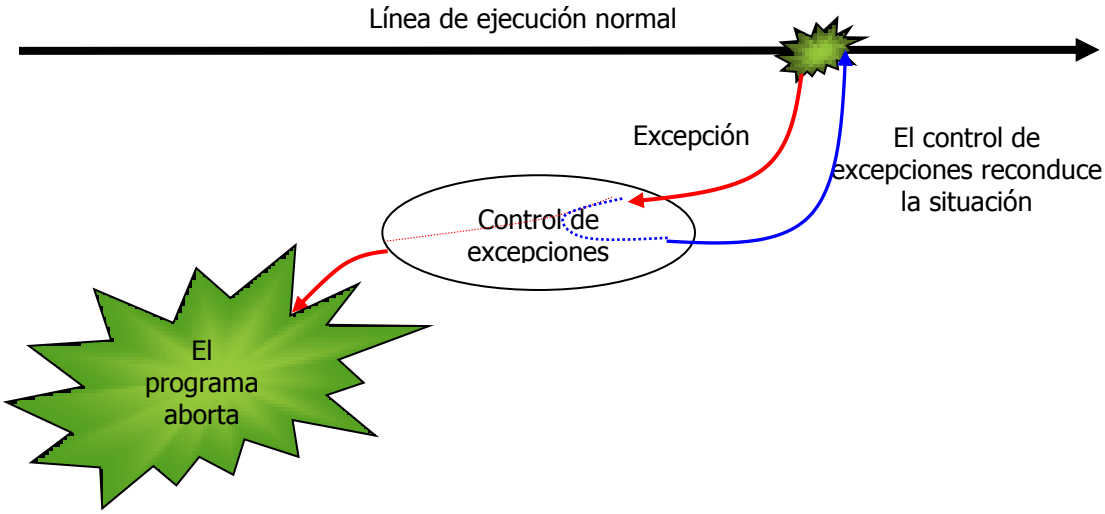

Ilustración 3

Para poder controlar las excepciones, el primer paso es identificar los segmentos de código donde se pueden producir, y encerrarlos en un bloque *try*:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    try {
        System.out.print("Entra un número entero no negativo: ");
        int num = input.nextInt();
        System.out.println("El factorial de " +
                           num + " es " + fact(num));
    }
    …
```

Detrás del bloque *try*, hay que poner uno o varios bloques *catch* (por cada excepción diferente que se quiere controlar). En los bloques *catch* es donde se pone el código que se quiere que se ejecute cuando se produce la excepción:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    try {
        System.out.print("Entra un número entero no negativo: ");
        int num = input.nextInt();
        System.out.println("El factorial de " +
                           num + " es " + fact(num));
    }
    catch (NegativeNumberException e) {
        System.out.println(e.getMessage());
    }
```

En este caso, el control que captura la excepción *NegativeNumberException* se limita a mostrar un mensaje indicando que no se puede calcular el factorial de un número negativo y terminar el programa, evitando que el usuario tenga que enfrentarse a un mensaje críptico sobre un *StackOverflowError*, que, en realidad, describe el resultado del problema, no su causa.

Al comienzo de este documento habíamos hablado de otra excepción que se puede producir, *InputMismatchException*. Vamos a controlarla también, pero en este caso, vamos a intentar que el programa se recupere y continúe funcionando, en vez de terminar.

Lo primero que vamos a hacer en este caso es desarrollar un método auxiliar que encapsule la llamada a *nextInt* e intente recuperarse de la excepción en caso de que se produzca:

```
private static int secureNextInt(Scanner input) {
    do {
        try {
            return input.nextInt();
        }
        catch (InputMismatchException e) {
            input.nextLine(); // Vacía el buffer de entrada
            System.out.println("No has tecleado un valor numérico.");
            System.out.println("Por favor, entra un número: ");
        }
    } while (true);
}
```

Lo que se ha hecho en este método es "encerrar" la llamada a *nextInt* en un bloque *try*, y añadir un bloque *catch* para controlar la excepción *InputMismatchException*. La primera instrucción del bloque

catch salta una línea para eliminar la entrada errónea, que no ha podido leerse, e invitar a continuación al usuario a que introduzca una entrada correcta. Como el conjunto está encerrado en un bucle sin fin, tras producirse la excepción se vuelve a intentar leer el dato. Cuando no se produzca la excepción, el dato se leerá y se ejecutará la instrucción *return*, abandonando el bucle y el método (una forma alternativa sería leer el dato en una variable y a continuación abandonar el bucle con una instrucción *break*). El bucle es lo que hace que el programa se recupere e intente continuar su trabajo después de controlar la excepción.

Tal como está, el método no termina hasta que consigue que el usuario entre un número. Esto puede resultar excesivo, por lo que ofrecemos una versión alternativa, que ofrece un número limitado de intentos:

```
private static int secureNextInt(Scanner input) throws TooManyAttempts {
    int attempts = 0;
    do {
        try {
            return input.nextInt();
        }
        catch (InputMismatchException e) {
            attempts++;

            if (attempts < 3) {
                input.nextLine();
                System.out.println("No    has    tecleado    un    valor
numérico.");
                System.out.println("Por favor, entra un número: ");
            } else {
                throw new TooManyAttempts(e);
            }
        }
    } while (true);
}
```

En esta versión, se cuenta el número de intentos y, cuando se supera un límite, se lanza una nueva excepción, de la clase *TooManyAttempts*, que se tendrá que haber definido, como se hizo antes con la *NegativeNumberException*. Al constructor de esta nueva excepción se le ha pasado, como causa, la excepción original, e, de forma que quien controle la excepción pueda tener información de "la excepción que desencadenó la excepción", si lo necesita.

Hay que modificar al método llamador (*main*) para incluir la llamada a *secureNextInt*, en lugar de a *nextInt*, y controlar la excepción *TooManyAttemps*:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    try {
        System.out.print("Entra un número entero no negativo: ");
        int num = secureNextInt(input);
        System.out.println("El factorial de " + num + " es " +
fact(num));
    }
    catch (NegativeNumberException e) {
        System.out.println(e.getMessage());
    }
    catch (TooManyAttemps e) {
        System.out.print("Ha habido varios intentos con el resultado ");
        System.out.println(e.getCause().toString());
        System.out.println("Los sentimos no podemos continuar");
    }
    …
```

Como se ve, podemos tener varios bloques *catch*. Solo hemos de tener en cuenta que, si ponemos bloques para una clase de excepción y otras que derivan de ellas, los bloques de las derivadas deben ir antes que el bloque de la superclase, dado que la excepción se intenta casar con un bloque que la controle en el orden en que aparecen los bloques. Si por ejemplo, quisiésemos poner un bloque para controlar cualquier otra excepción no tenida en cuenta, de la clase *Exception*, debería ir en último lugar, para que primero se mirasen las excepciones más específicas. En cualquier caso, es aconsejable ser lo más específico posible en el control de excepciones.

Detrás de los bloques *catch* (o incluso en vez de ellos), se puede añadir un bloque *finally* en el que incluiríamos instrucciones que queremos que se ejecuten siempre al final, haya ocurrido, o no, una excepción:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    try {
        System.out.print("Entra un número entero no negativo: ");
        int num = secureNextInt(input);
        System.out.println("El factorial de " +
                           num + " es " + fact(num));
    }
    catch (NegativeNumberException e) {
        System.out.println(e.getMessage());
```

```
        }
        catch (TooManyAttempts e) {
            System.out.print("Ha habido varios intentos con el resultado ");
            System.out.println(e.getCause().toString());
            System.out.println("Lo sentimos, no podemos continuar");
        }
        finally {
           System.out.println("Gracias por usar este programa");
           input.close();
        }
}
```