Tema 8: Manejo de excepciones

A la hora de escribir programas, uno de los mayores esfuerzos del programador debe ser evitar que se produzcan errores durante la ejecución. A pesar de esos esfuerzos, los errores suelen ocurrir (es una de las reglas de oro de la programación). Algunos lenguajes de programación responden a esos errores con la finalización repentina de la ejecución del programa o incluso puede que la ejecución continúe pero de una forma impredecible. El lenguaje Java usa las *excepciones* para el tratamiento y manipulación de los errores.

8.1. ¿QUÉ ES UNA EXCEPCIÓN.-

Se podría definir una excepción como un suceso que ocurre durante la ejecución del programa, y que rompe el flujo normal del mismo. Un ejemplo de este tipo de violaciones es intentar acceder a un elemento de una matriz con un índice mayor que su dimensión. Cuando ocurre un error de este tipo, se crea una excepción y el control del programa pasa desde el punto donde el error se produjo a otro punto especificado por el programador. En tal caso, se dice que la excepción es "lanzada" (*thrown*) desde el punto donde se creó y es "recogida" (*caugth*) en el punto donde se transfiere el control, Java hereda la terminología y la sintaxis del tratamiento de excepciones del lenguaje C++.

Toda excepción es representada por un objeto de la clase ***Throwable*** o cualquier subclase suya, este objeto contiene información sobre el tipo de la excepción y el estado del programa cuando se produjo. Además, puede ser utilizado para enviar información desde el punto donde el error se produjo hasta el punto donde será tratado.

Una vez que una excepción ha sido lanzada desde el interior de un método, el sistema se encarga automáticamente de buscar un manipulador de excepciones (exception handler) que sea capaz de responder a esa excepción. La búsqueda comienza en el método donde se produjo el error, y en caso de no encontrarse dicho manipulador, la búsqueda continua en el método en el que se hizo la llamada al método que provocó la excepción y así sucesivamente. De esta forma se continúa en una búsqueda ascendente hasta encontrar el manipulador apropiado. Un manipulador de excepciones es considerado apropiado si coinciden los tipos de la excepción producida y la excepción a la que responde el manipulador. En el caso de que el sistema termine la búsqueda y no sea capaz de encontrar el manipulador adecuado, la ejecución del programa terminará.

8.2. CAUSAS DE EXCEPCIONES.-

En un programa Java las excepciones pueden provocarse por el propio sistema cuando detecta un error o una condición de ejecución anormal, o explícitamente por el programador, utilizando la sentencia ***throw***. Dentro del primer caso podrían identificarse las siguientes causas:

- Una operación que provoca una violación de las normas semánticas del lenguaje. Por ejemplo, índice fuera de límites en una matriz.
- Un error cuando se carga una parte del programa.
- Un exceso en el uso de un cierto recurso. Por ejemplo, agotar la memoria.
- Un error interno en la máquina virtual.

8.3. ¿POR QUÉ USAR EXCEPCIONES?.-

La principal ventaja del uso de excepciones es la robustez del programa, previniéndole de errores que podrían provocar una finalización inesperada. Pero se podrían destacar otras características que hacen del uso y tratamiento de excepciones una costumbre plenamente recomendada:

- En primer lugar, permiten separar el código específico de tratamiento de errores del resto del código. Es muy común en otros lenguajes de programación que no incorporan el tratamiento de excepciones, insertar en el código sentencias condicionales para detectar posibles errores; esto complica innecesariamente la lectura del código y aumenta su complejidad.

- Permite propagar los errores a otros métodos dentro de la línea de llamadas. Por ejemplo, si un método *A()* efectúa una llamada a otro método *B()* y a su vez éste a un tercero *C()* y la excepción se produce en *C(),* podría resultar adecuado que el método *A()* sea el encargado de tratar dicha excepción.

- Una tercera característica es la posibilidad de crear grupos de excepciones y diferenciar unas excepciones de otras. Esto se consigue creando subclases de la clase ***Throwable***. Por ejemplo, se podrían agrupar todas las excepciones relativas a gestión de ficheros, como fichero no existente, fichero dañado, fichero abierto sin permiso de escritura, etc., en una categoría, definiendo una subclase *FileException* de la clase ***Throwable***.

8.4. TIPOS DE EXCEPCIONES.-

Como se ha indicado previamente, toda excepción es un objeto de la clase ***Throwable*** o de alguna de sus subclases, bien sean incorporadas por librerías de Java o declaradas en el propio programa. Como norma recomendada, podría indicarse que todos los nombres de las clases de excepciones terminen con un palabra *Exception*. Existen subclases de particular interés como la clase *Error* y la clase *Exception*, además de la clase *RuntimeException* que es, a su vez, subclase de la clase *Exception*. El programador puede crear su propios tipos de excepciones construyendo subclases de la clse *Throwable* que se adapten mejor a sus necesidades particulares.

Dependiendo de la clase a la que pertenezcan, las excepciones se dividen en dos grupos:

- *Excepción no comprobada*: pertenece a las clase *RuntimeException*, *Error* o cualquiera de sus subclases.

- *Excepción comprobada*: pertenece a una subclase de *Throwable* que no se *Error*, *RuntimeException*, ni ninguna de su subclases.

8.4.1. EXCEPCIONES EN TIEMPO DE EJECUCIÓN (runtime).-

Las subclases estándar de la clase ***RuntimeException*** que corresponden a excepciones para las cuales no es obligada la presencia de un manipulador, se encuentran definidas en el paquete *java.lang* y son las siguientes:

- ***ArithmeticException***: provocada por una operación aritmética ilegal. Por ejemplo, división por cero.

- ***ArrayStoreException***: intento de asignar a uno de los elementos de una matriz un valor con tipo no compatible.

- ***ClassCastException***: conversión de tipo ilegal.

- ***IllegalArgumentException***: llamada a un método con un argumento inapropiado. Existen, a su vez, dos subclases de esta clase:

  - ***IllegalThreadStateException***: se efectúa una llamada no permitida a un método thread.

- ***NumberFormatException***: intento de convertir una cadena que no tiene el formato adecuado a un determinado tipo numérico.

- ***IndexOutOfBoundsException***: índice fuera de rango.

- ***NegativeArraySizeException***: intento de crear una matriz cuya longitud es negativa.

- ***NullPointerException***: uso de una referencia nula donde se requería la referencia a un objeto.

8.4.2. EXCEPCIONES DE OBLIGADA RESPUESTA.-

Además de la clase *RuntimeException*, existen otras subclases de la clase *Exception* del tipo comprobada. Es importante destacar de nuevo que todo método en el que se puede producir una de estas excepciones, debe estar obligatoriamente preparado para responder a ella o, en todo caso, utilizar la sentencia ***throw*** para indicar que la excepción puede producirse pero no es ese método el encargado de darle respuesta.

Dentro del paquete java.lang se encuentran definidas las siguientes excepciones:

- ***ClassNotFountException***: clase o interfaz no encontrada cuando para localizarla se utiliza el método *forName()* indicando su nombre mediante una cadena de caracteres.

- ***CloneNotSupportedException***: se ha producido un error al utilizar el método *clone()* de la clase *Object* para hacer una copia de un objeto.

- ***IllegalAccessException***: intento fallido de cargar una clase desde un método que no tiene acceso a ella porque no es de tipo *public* y se encuentra en otro paquete.

- ***InstantionException***: intento de crear un ejemplar de una clase utilizando el método *newInstance(),* que no puede crearse porque es una interfase o de tipo *abstract*.

- ***InterruptedException***: un proceso ligero interrumpe a otro que se encontraba en situación de espera,.

Dentro del paquete *java.io* se encuentra definida la clase *IOException* a la que pertenecen todas las excepciones provocadas por fallos en operaciones de entrada/salida. Posee las siguientes subclases dentro de ese mismo paquete:

- ***java.io.EOFException***: alcanzado el fin de fichero.

- ***java.io.FileNotFoundException***: no se encuentra el fichero.

- ***java.io.InterruptedIOException***: un proceso ligero interrumpe a otro que se encontraba en situación de espera para completar una operación de *Input/Output*, utlizando el método *interrupt()*.

8.4.3. LA CLASE Error.-

Como se ha indicado, la clase Error es una subclase de la clase *Throwable*. En ella se agrupan todos los errores "graves" del sistema que conducen a situaciones de difícil solución, por lo tanto, es poco probable que dentro de un programa se intente hacer un tratamiento de este tipo de errores. Por ejemplo, en el caso de que al crear un objeto no se disponga de memoria suficiente, se produciría un error del tipo *OutOfMemoryError*.

8.5. TRAMIENTO DE LAS EXCEPCIONES.-

Una vez analizada la importancia de las excepciones, sus tipos y la necesidad de responder en algunos casos a cada excepción, esta sección se dedica a la exposición de cómo hacerlo. Para ello se parte del programa siguiente, en el que se define una clase *División*, en cuyo método *main()* se está leyendo, desde un fichero llamado *Datos*, dos números, se calcula su cociente y se presenta el resultado en pantalla.

```
import java.io.*;

class Division
{
   public static void main (String args[])
   {
      FileInputStream f;
      float x, y;

      f = new FileInputStream ("Datos");
      x = f.read ();
      y = f.read ();

      System.out.println ("Resultado = " + x/y);

   }
}
```

Si se intenta compilar este programa tal como ha sido presentado, se observará que se producen varios errores de compilación, avisando de que existen excepciones que deben ser tratadas. En concreto, en la instrucción

```
   f = new FileInputStream ("Datos");
```

se ordena abrir el fichero de nombre *Datos* por lo que, si el fichero no se encontrara, se produciría una excepción del tipo *FileNotFoundException*. Pero además podrían provocarse excepciones del tipo *IOException* en la instrucciones:

```
x = f.read();
y = f.read ();
```

en el caso de no poder leer los datos, porque el fichero esté vacío, por ejemplo.

Analizando el resto del código, puede detectarse otro punto que pudiera provocar una excepción:

```
System.out.println ("Resultado = " + x/y);
```

En esta instrucción se está ordenando la escritura por pantalla del resultado de dividir *x* entre *y*, pero ¿qué ocurriría si *y* tomara el valor *0*?, lo que ocurriría es que se produciría una excepción del tipo *ArithmeticException*. Sin embargo, el compilador no avisa de que esa excepción pudiera ocurrir y, por lo tanto, no es necesario tratarla. En caso de que el programador desee no hacerlo, deberá asumir los riesgos que eso supone. ¿Qué es lo que diferencia esta excepción de las anteriores? Lo que la diferencia es el hecho de que esta última es una subclase de las excepciones *RuntimeException*. No ocurre lo mismo con la excepciones anteriores.
Un método puede responder a una excepción de dos formas: incorporando un manipulador de excepciones apropiado al tipo de la misma, o bien, declarándola y dejando que sea alguno de los métodos de los que provino quien se encargue de responderla.

8.5.1. Construcción de un manipulador de excepciones.-

Esta sección pretende mostrar la forma de construir un manipulador de excepciones (exception handler) en un método y sus tres componentes básicas:

- El blque **`try`**.
- El bloque o bloques **`catch`**
- El bloque **`finally`**.

El primer bloque agrupa a todas las sentencias del método que pueden provocar una excepción. Puede haber uno o varios bloques *catch* inmediatamente a continaución de cada bloque *try*. Por último, el bloque *finally* es opcional.

La estructura general de un manipulador de excepciones es la siguiente:

```
try
   {
   ...
   }
catch (....)
   {
   ...
   }
catch (....)
   {
   ....
   }
finallly
   {
   ...
   }
```

No puede haber otra sentencia entre el final del bloque *try* y el comienzo del primer bloque *catch*.

Cuando existen varias sentencias del método que potencialmente pueden provocar una excepción, hay dos posibilidades:

- Agrupar todas las sentencias en un solo bloque *try* con varios bloques *catch* para responder a cada una de la excepciones.

- Agrupar cada sentencia individual en un bloque *try* con su bloque *catch* correspondiente.

Todo bloque *try* define el alcance del manipulador de excepciones y debe ir acompañado de al menos un bloque *catch* o el bloque *finally*.

Cada bloque *catch* requiere un argumento que se declara de la misma forma que la declaración de argumentos de un método. Este argumento, que acompaña a la sentencia *catch*, indica el tipo de excepción que es tratada en el correspondiente bloque y siempre debe ser una subclase de la calse *Throwable*, que se encuentra definida en el paquete *java.lang*. Además de la clase de la excepción debe darse un nombre que puede ser usado para referirse a ella en las sentencias del bloque. A través de este nombre puede accederse a variables miembro o métodos de la clase de forma similar a como se haría en cualquier otra parte del método. Uno de tales métodos es *getMssage()*, que escribe información adicional sobre el error que provocó la excepción.

El orden en que se coloquen los distintos bloques *catch* es importante, ya que cuando se produce la excepción en una instrucción, el sistema abandona el flujo de ejecución normal y busca automáticamente el primer bloque *catch* cuyo tipo de excepción se ajusta al de la que se ha producido, ejecutando únicamente las sentencias de ese bloque *fnally* o de dar por concluido el programa. La búsqueda del bloque comienza en el mismo método en el que produjo la excepción y continúa por los métodos de los que provenía la llamada.

A continuación, se muestra el programa anterior, incorporando los correspondientes manipuladores para las dos clases de excepciones de tipo diferente al d la runtime que pueden producirse:

```
import java.io.*;

class Division
{
   public static void main (String args[])
   {
      FileInputStream f;
      float x, y;

      try
      {
         f = new FileInputStream (“Datos”);
         x = f.read ();
         y = f.read ();

         System.out.println (“Resultado = “ + x/y);
      }
      catch (FileNotFoundException e)
      {
         System.err.println (“No encontrado el fichero”);
      }
      catch (IOException e)
```

```
      {
         System.err.println (“Error de lectura”);
      }
   }
}
```

De esta forma, el programa podría compilarse correctamente. Obsérvese que no ha sido incluido el bloque catch para la excepción *ArithmeticException* aunque podría hacerse si así se desea.

Es importante destacar lo siguiente ,¿qué ocurre si por algún motivo no puede abirse el fichero *Datos*? Está claro que se produce inmediatamente una excepción del tipo *FileNotFoundException* y, automaticamente se interrumpe la ejecución del bloque *try*. Pero ¿dónde se produce realmente la excepción? Ésta se produce en el constructor de la clase *FileInputStream* y es ahí donde comienza la búsqueda del bloque *catch* que responda ala excepción. En ese constructor no existe dicho bloque, de manera que se busca en el método del que procedía la llamada: el método *main()* de la clase *Division*, encontrándose en ese caso el *catch* correspondiente.
En el programa anterior han sido construidos dos manipuladores de excepciones, especializados en responder cada uno de ellos a una sola excepción. Otra posibilidad es la de tratar las dos excepciones en solo bloque *catch*. Para ello bastaría con definir el tipo de excepción como una clase que tenga como subclases las correspondientes a las dos excepciones que quieren ser tratadas. Con esta modificación, el programa podría quedar de la siguiente forma:

```
import java.io.*;

class Division
{
   public static void main (String args[])
   {
      FileInputStream f;
      float x, y;

      try
      {
         f = new FileInputStream (“Datos”);
         x = f.read ();
         y = f.read ();

         System.out.println (“Resultado = “ + x/y);
      }
      catch (Exception e)
      {
         System.err.println (“Error: ”+ e.getMessage());
      }
   }
}
```

Ahora se ha utilizado la clase *Exception* como clase general, de forma que, indirectamente se está dando tratamiento a la excepción *ArithmeticException*, ya que ésta es también una subclase de la clase *Exception*. En general, resulta más adecuado crear manipuladores más especializados, aunque una situación intermedia consistiría en definir clase de excepciones propias con diferentes subclases, y de esta forma poder agrupar, en diferentes categorías, los errores que podrían provocar excepciones.

Opcionalmente, tras el último bloque *catch*, puede colocarse un bloque *finally* y que puede ser un

mecanismo para efectuar una “limpieza” en el estado del método antes de pasar el control a otra parte del programa. Por ejemplo, si un fichero ha sido abierto para lectura y se produce una excepción, podría resultar interesante cerrarlo antes de salir del método. Esto puede hacerse colocando esa instrucción dentro del bloque *finally*. Todas las instrucciones que se encuentren dentro de este bloque serán ejecutadas siempre, tanto si el desarrollo del método ha sido el normal como si se ha producido una excepción; en este último caso se ejecutan tras las sentencias del bloque catch correspondiente. Otra de las utilidades del bloque *finally* es la de evitar la repetición de código innecesaria.

8.5.2. DECLARACIÓN DE EXCEPCIONES.-

Una alternativa a la construcción de un manipulador para una excepción es dejar que un método anterior en la sucesión de llamadas se encargue de responderla, utilizando el comando **`throws`**. Cuando se quiere que un método declare varias excepciones pero que no responda a ellas con el o los correspondientes manipuladores de excepciones, debe colocarse, a continuación del nombre del método y de la lista de sus argumentos, la palabra clave ***throws*** seguida de una lista separada por comas, de las excepciones. Por ejemplo, un método definido de la forma siguiente:

```
public void S (int []) throws IndexOutOfBoundsException {
   ....
}
```

indica que en el método puede generarse una excepción *IndexOfBoundsException*, pero en ese caso, el bloque *catch* para tratarla debe buscarse no el método *S()* sino a partir del método en el que se hizo la llamada a él. Conviene recordar que la excepción IndexOutOfBounds es de tipo runtime y por tanto no sería necesario declararla, aunque puede hacerse.

8.6. LA SENTENCIA throw.-

La sentencia **`throw`** se utiliza dentro de los métodos para “lanzar” excepciones, es decir, cuando se detecta una situación que provocaría un error, debe crearse un objeto de la clase de excepción correspondiente, e indicar que la excepción se ha producido mediante una sentencia de la forma:

```
throw Objeto;
```

donde *Objeto* debe pertenecer a alguna de las subclases de la clase *Throwable*.

Muchas veces, la sentencia *throw* es utilizada implícitamente por el sistema cuando se detectan algunos errores estándar. Otras veces se utliza dentro de los paquetes del entorno Java en incluso el programador puede utlizarla para “lanzar” sus propias excepciones de forma explícita.

No debe confundirse las palabras claves *throws* y *throw* que, aunque muy similares tienen diferente significado. La primera se utiliza para declarar que un método puede provocar una excepción y la segunda es la que realmente provoca dicha excepción.

8.7. EJERCICIOS PROPUESTOS.-

❶ *Se desea definir la función:*

```
F(a,b,c) = sqrt ((a-log b) / c)
```

Utilizar excepciones para construir un método en el que el tratamiento de errores quede separado claramente del resto del código. Crear para ello tres tipos de excepciones.

❷ *Escribir un programa en Java para calcular la intersección de dos rectas en el espacio. Crear dos excepciones como subclases de la clase Throwable para responder a los casos de rectas paralelas o rectas que se cruzan, avisando con los correspondientes mensajes.*

❸ *Se quiere escribir un programa en Java que resuelva la ecuación de segundo grado*

$$ax^2 + bx + c = 0$$

dándole como datos los valores de a, b y c. Se supone que el programa va a utilizarse para el caso de raíces reales.

Se pide:

a) Detectar las posibles excepciones que pueden generarse.
b) Clasificar las mismas, indicando cuáles son de tipo runtime y cuales no.
c) Indicar como tratarlas.
d) Escribir el programa completo.