CLASE 5. EXCEPCIONES

A diferencia de otros lenguajes de programación orientados a objetos como Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje Java, una Exception es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa sin embargo, la aplicación no debería morirse y generar un core (o un crash en caso del DOS). En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido en cambio, en otros casos, como por ejemplo el no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables y el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).
Utilizadas en forma adecuada, las excepciones aumentan en gran medida la robustez de las aplicaciones.

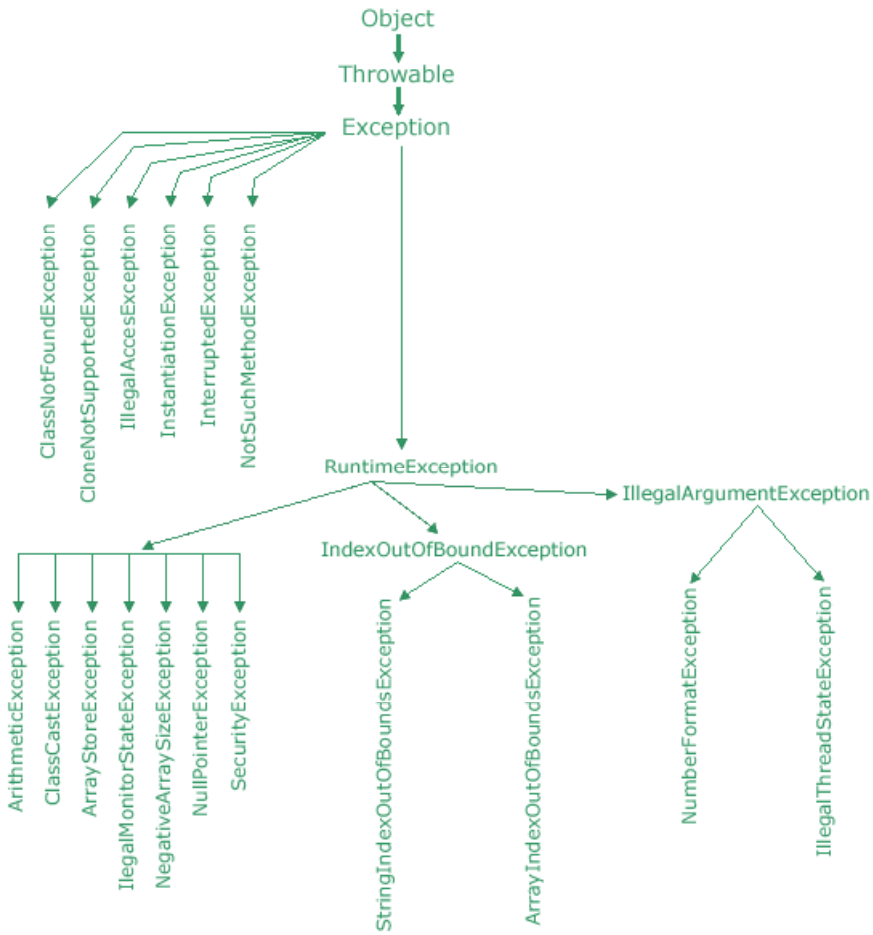
Un buen programa debe gestionar correctamente todos o la mayor parte de los errores que se pueden producir. Hay dos "estilos" de hacer esto:

> **1.** De manera antigua: los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de if elseif …, gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.
>
> **2.** Con soporte en el propio lenguaje: En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o Exceptions. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y Java.

En los siguientes apartados se examina cómo se trabaja con los bloques y expresiones try, catch, throw, throws y finally, cuándo se deben lanzar excepciones, cuándo se deben capturar y cómo se crean las clases propias de tipo Exception.

5.1 EXEPCIONES ESTÁNDAR DE JAVA

Los errores se representan mediante dos tipos de clases derivadas de la clase Throwable: Error y Exception.
A continuación se muestra parcialmente la jerarquía de clases relacionada con Throwable.



La clase Exception tiene más interés. Dentro de ella se puede distinguir:

1. RuntimeException: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar excepciones implícitas.

2. Las demás clases derivadas de Exception son excepciones explícitas. Java obliga a tenerlas en cuenta y chequear si se producen.

El caso de RuntimeException es un poco especial. El propio lenguaje Java durante la ejecución de un programa chequea y lanza automáticamente las excepciones que derivan de RuntimeException. El programador no necesita establecer los bloques try/catch para controlar este tipo de excepciones.

Representan dos casos de errores de programación:

1. Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia null en un método.

2. Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un

array (genera un ArrayIndexOutOfBoundsException automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequear continuamente todo tipo de errores (que las referencias son distintas de null, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de Exception pueden pertenecer a distintos packages de Java. Algunas perenecen a java.lang (Throwable, Exception, RuntimeException, …); otras a java.io (EOFException, FileNotFoundException, ...) o a otros packages. Por heredar de Throwable todos los tipos de excepciones pueden usar los métodos siguientes:

1. String getMessage() Extrae el mensaje asociado con la excepción.

2. String toString() Devuelve un String que describe la excepción.

3. void printStackTrace() Indica el método donde se lanzó la excepción.

Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:
ArithmeticException: Las excepciones aritméticas son típicamente el resultado de una división por 0:
int i = 12 / 0;

NullPointerException: Se produce cuando se intenta acceder a una variable o método antes de ser definido:

IncompatibleClassChangeException: El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

ClassCastException: El intento de convertir un objeto a otra clase que no es válida.
y = (Prueba)x; // donde x no es de tipo Prueba

NegativeArraySizeException: Puede ocurrir si hay un error aritmético al intentar cambiar el tamaño de un array.

OutOfMemoryException: ¡No debería producirse nunca! El intento de crear un objeto con el operador new ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el garbage collector se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

NoClassDefFoundException: Se referenció una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException: Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

UnsatisfiedLinkException: Se hizo el intento de acceder a un método nativo que no existe. Aquí no existe un método a.kk

```
class A {
   native void kk();
   }
```

y se llama a a.kk(), cuando debería llamar a A.kk().

InternalException: Este error se reserva para eventos que no deberían ocurrir. Por definición, el usuario nunca debería ver este error y esta excepción no debería lanzarse.

5.2 ¿CÓMO LANZAR UNA EXCEPCIÓN?

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzar una excepción es el siguiente:

1. Se crea un objeto Exception de la clase adecuada.

2. Se lanza la excepción con la sentencia throw seguida del objeto Exception creado.

```
// Código que lanza la excepción MyException una vez detectado el error
MyException me = new MyException("MyException message");
throw me;
```

Esta excepción deberá ser capturada (catch) y gestionada en el propio método o en algún otro lugar del programa (en otro método anterior en la pila o stack de llamadas), según se explica en el Apartado 5.3.

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques try/catch/finally se ejecutará el bloque catch que la captura o el bloque finally (si existe).

Todo método en el que se puede producir uno o más tipos de excepciones (y que no utiliza directamente los bloques try/catch/finally para tratarlos) debe declararlas en el encabezamiento de la función por medio de la palabra throws. Si un método puede lanzar varias excepciones, se ponen detrás de throws separadas por comas, como por ejemplo:

```
public void leerFichero(String fich) throws EOFException,
FileNotFoundException {…}
```

Se puede poner únicamente una superclase de excepciones para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. El caso anterior sería equivalente a:
public void leerFichero(String fich) throws IOException {…}
Las excepciones pueden ser lanzadas directamente por leerFichero() o por alguno de los métodos llamados por leerFichero(), ya que las clases EOFException y FileNotFoundException derivan de IOException.

Se recuerda que no hace falta avisar de que se pueden lanzar objetos de la clases Error o RuntimeException (excepciones implícitas).

5.3 ¿CÓMO CAPTURAMOS UNA EXCEPCIÓN?

Como ya se ha visto, ciertos métodos de los packages de Java y algunos métodos creados por cualquier programador producen ("lanzan") excepciones. Si el usuario llama a estos métodos sin tenerlo en cuenta se produce un error de compilación con un mensaje del tipo: "… Exception java.io.IOException must be caugth or it must be declared in the throws clause of this method".

El programa no compilará mientras el usuario no haga una de estas dos cosas:

> **1.** Gestionar la excepción con una construcción del tipo try {…} catch {…}.
>
> **2.** Re-lanzar la excepción hacia un método anterior en el stack, declarando que su método también lanza dicha excepción, utilizando para ello la construcción throws en el header del método.

El compilador obliga a capturar las llamadas excepciones explícitas, pero no protesta si se captura y luego no se hace nada con ella. En general, es conveniente por lo menos imprimir un mensaje indicando qué tipo de excepción se ha producido.

5.3.1 Bloques try y catch

En el caso de las excepciones que no pertenecen a las RuntimeException y que por lo tanto Java obliga a tenerlas en cuenta habrá que utilizar los bloques try, catch y finally. El bloque try es el bloque de código donde se prevé que se genere una excepción. Es como si dijésemos "intenta estas sentencias y mira a ver si se produce una excepción". Si se produce una situación anormal y se lanza por lo tanto una excepción, el control salta o sale del bloque try y pasa al bloque catch, que se hace cargo de la situación y decide lo que hay que hacer. El código del bloque catch es el que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

catch(Excepcion e) { ...

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque.

```
int valor;
try {
  for( x=0,valor = 100; x < 100; x ++ )
      valor /= x;
  }
catch( ArithmeticException e ) {
  System.out.println( "Matemáticas locas!" );
  }
catch( Exception e ) {
  System.out.println( "Se ha producido un error" );
  }
```

Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas. Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}
  .
  .
  .
  try {
    if( temp > 40 )
      throw( new demasiadoCalor() );
    if( dormir < 8 )
      throw( new demasiado Cansado() );
  } catch( Limites lim ) {
    if( lim instanceof demasiadoCalor )
      {
      System.out.println( "Capturada excesivo calor!" );
      return;
      }
    if( lim instanceof demasiadoCansado )
      {
      System.out.println( "Capturada excesivo cansancio!" );
      return;
      }
  } finally
    System.out.println( "En la clausula finally" );
```

El bloque finally es el bloque de código que se ejecuta siempre, haya o no excepción. Hay una cierta controversia entre su utilidad, pero, por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejarnos grabado si se producen excepciones y nos hemos recuperado de ellas o no.

Este bloque puede sernos útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque try.

Cuando vamos a tratar una excepción, se nos plantea el problema de qué acciones vamos a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al usuario y un mensaje avisándolo de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa. El bloque finally se ejecuta aunque en el bloque try haya un return.

En el siguiente ejemplo se presenta un método que debe "controlar" una IOException relacionada con la lectura ficheros y una MyException propia:

```
void metodo1(){
...
try {
// Código que puede lanzar las excepciones IOException y MyException
}
catch (IOException e1) {
// Se ocupa de IOException simplemente dando aviso
System.out.println(e1.getMessage());
}
catch (MyException e2) {
// Se ocupa de MyException dando un aviso y finalizando la función
System.out.println(e2.getMessage()); return;
}
finally { // Sentencias que se ejecutarán en cualquier caso
...
}
...
} // Fin del metodo1
```

5.3.2 Método finally {...}

El bloque finally {...} debe ir detrás de todos los bloques catch considerados. Si se incluye (ya que es opcional) sus sentencias se ejecutan siempre, sea cual sea el tipo de excepción que se produzca, o incluso si no se produce ninguna. El bloque finally se ejecuta incluso si dentro de los bloques try/catch hay una sentencia continue, break o return. La forma general de una sección donde se controlan las excepciones es por lo tanto:

```
try {
// Código "vigilado" que puede lanzar una excepción de tipo A,B o C
} catch (A a1) {
// Se ocupa de la excepción A
```

```
} catch (B b1) {
// Se ocupa de la excepción B
} catch (C c1) {
// Se ocupa de la excepción C
} finally {
// Sentencias que se ejecutarán en cualquier caso
}
```

El bloque finally es necesario en los casos en que se necesite recuperar o devolver a su situación original algunos elementos. No se trata de liberar la memoria reservada con new ya que de ello se ocupará automáticamente el garbage collector.

Como ejemplo se podría pensar en un bloque try dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el fichero dentro del bloque finally.

8.4 ¿CÓMO CREAMOS NUEVAS EXCEPCIONES?

El programador puede crear sus propias excepciones sólo con heredar de la clase Exception o de una de sus clases derivadas. Lo lógico es heredar de la clase de la jerarquía de Java que mejor se adapte al tipo de excepción. Las clases Exception suelen tener dos constructores:

1. Un constructor sin argumentos.

2. Un constructor que recibe un String como argumento. En este String se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva super(String).

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
class MiExcepcion extends Exception {
public MiExcepcion() { // Constructor por defecto
super();
}
public MiExcepción(String s) { // Constructor con mensaje
super(s);
}
}
```

8.5 PROPAGACIÓN DE EXCEPCIONES

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque y sigue el flujo de control por el bloque finally (si lo hay) y concluye el control de

la excepción.

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula finally (en caso de que la haya). Lo que ocurre en este caso, es exactamente lo mismo que si la sentencia que lanza la excepción no se encontrase encerrada en el bloque try.

El flujo de control abandona este método y retorna prematuramente al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia try, entonces se vuelve a intentar el control de la excepción, y así continuamente.

Veamos lo que sucede cuando una excepción no es tratada en la rutina en donde se produce. El sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, podemos suponer que Java nos está proporcionando un bloque catch por defecto, que imprime un mensaje de error y sale.

No hay ninguna sobrecarga en el sistema por incorporar sentencias try al código. La sobrecarga se produce cuando se genera la excepción.

Hemos dicho ya que un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada, indicando a todo el mundo que es capaz de generar excepciones. Esto debe ser así para que cualquiera que escriba una llamada a ese método esté avisado de que le puede llegar una excepción, en lugar del valor de retorno normal. Esto permite al programador que llama a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas. La siguiente línea de código muestra la forma general en que un método declara excepciones que se pueden propagar fuera de él:
tipo_de_retorno(parametros) throws e1,e2,e3 { }

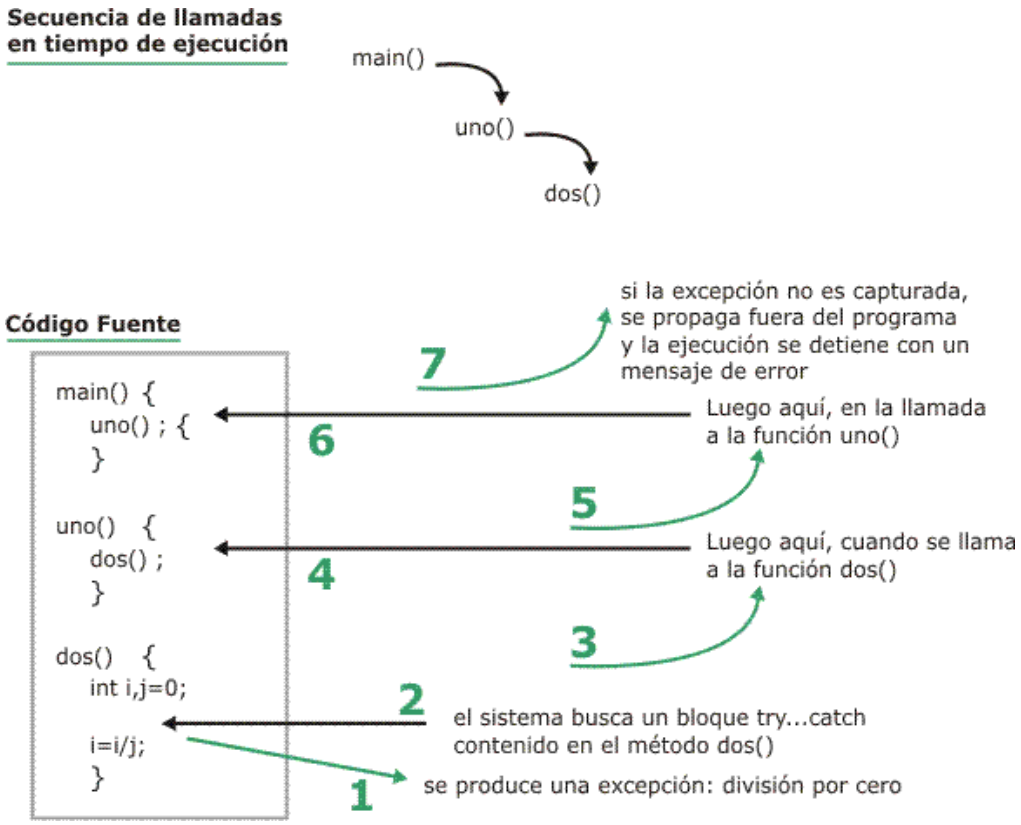
Los nombres e1,e2,... deben ser nombres de excepciones, es decir, cualquier tipo que sea asignable al tipo predefinido Throwable. Observar que, como en la llamada al método se especifica el tipo de retorno, se está especificando el tipo de excepción que puede generar (en lugar de un objeto exception).

He aquí un ejemplo, tomado del sistema Java de entrada/salida:

```
byte readByte() throws IOException;
short readShort() throws IOException;
char readChar() throws IOException;
void writeByte( int v ) throws IOException;
void writeShort( int v ) throws IOException;
void writeChar( int v ) throws IOException;
```

Algunas de las rutinas Java lanzan una excepción cuando se alcanza el fin del fichero.
En este diagrama se muestra gráficamente cómo se propaga la excepción que

se genera en el código, a través de la pila de llamadas durante la ejecución del código.



Cuando se crea una nueva excepción, derivando de una clase Exception ya existente, se puede cambiar el mensaje que lleva asociado. La cadena de texto puede ser recuperada a través de un método. Normalmente, el texto del mensaje proporcionará información para resolver el problema o sugerirá una acción alternativa.

Por ejemplo:

```
class SinGasolina extends Exception {
      SinGasolina( String s ) { // constructor
        super( s );
        }
      ....
// Cuando se use, aparecerá algo como esto
      try {
        if( j < 1 )
          throw new SinGasolina( "Usando deposito de reserva" );
      } catch( SinGasolina e ) {
        System.out.println( o.getMessage() );
        }
```

Esto, en tiempo de ejecución originaría la siguiente salida por pantalla:

> Usando deposito de reserva

Otro método que es heredado de la superclase Throwable es printStackTrace(). Invocando a este método sobre una excepción se volcará a pantalla todas las

llamadas hasta el momento en donde se generó la excepción (no donde se maneje la excepción). Por ejemplo:

```
// Capturando una excepción en un método
class testcap {
   static int slice0[] = { 0,1,2,3,4 };
   public static void main( String a[] ) {
      try {
         uno();
      } catch( Exception e ) {
         System.out.println( "Captura de la excepcion en main()" );
         e.printStackTrace();
         }
      }
static void uno() {
     try {
        slice0[-1] = 4;
     } catch( NullPointerException e ) {
        System.out.println( "Captura una excepcion diferente" );
        }
     }
}
```

Cuando se ejecute ese código, en pantalla observaremos la siguiente salida:

```
> Captura de la excepcion en main()
> java.lang.ArrayIndexOutOfBoundsException: -1
         at testcap.uno(test5p.java:19)
         at testcap.main(test5p.java:9)
```

Con todo el manejo de excepciones podemos concluir que proporciona un método más seguro para el control de errores, además de representar una excelente herramienta para organizar en sitios concretos todo el manejo de los errores y, además, que podemos proporcionar mensajes de error más decentes al usuario indicando qué es lo que ha fallado y por qué, e incluso podemos, a veces, recuperarnos de los errores.

La degradación que se produce en la ejecución de programas con manejo de excepciones está ampliamente compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.