Apuntes

de

Java

Tema 6: Excepciones

TEMA 6: EXCEPCIONES

6.1.- ¿QUÉ SON LAS EXCEPCIONES?

Definición: Una excepción es un evento excepcional que ocurre durante la ejecución del programa y que interrumpe el flujo normal de las sentencias.

Si una operación no puede completarse debido a un error, el programa deberá:

- volver a un estado estable y permitir otras operaciones.
- intentar guardar el trabajo y finalizar.

Esto es difícil debido a que generalmente, el código que detecta el error no es el que puede realizar dichas tareas. El que detecta el error debe informar al que pueda manejarlo.

La solución más habitual a esto son los códigos de error, es decir, que un método devuelva un código de error como valor de retorno. Esto presenta una serie de inconvenientes:

- No siempre queda sitio para códigos de error (p. ej.: getchar).
- No se garantiza que se vayan a consultar.
- Si se contemplan todos los errores, el código crece considerablemente.
- La lógica del programa queda oscurecida.
- No sirven en los constructores.

En definitiva, hacen que el tratamiento de errores sea complejo y que, por ello, muchas veces no se tenga en cuenta y no se traten todos los errores. Esto impide la construcción de programas robustos.

Java ofrece una solución, otra forma de tratar con los errores: **las excepciones**.

6.2.- DETECCIÓN DE ERRORES

Una condición excepcional es aquella que impide la continuación de una operación. Es decir, no se sabe cómo manejarla, pero no se puede continuar hasta que no se resuelva.

En Java, cuando ocurre algo así, se lanza una excepción (throw) para que alguien que sepa manejarla la trate en un contexto superior. Por ejemplo:

```
if( “error de CRC” )
// Si se produce un error de CRC, lanzamos una excepción de E/S
     throw new IOException();
     // Para lanzar una excepción hay que crear un objeto,
     // ya que una excepción es un objeto.
```

Con la palabra throw se lanza una excepción. Cuando ocurre esto, finaliza el método y se lanza un objeto que facilite información sobre el error ocurrido. Normalmente se utilizará una clase diferente para cada tipo de error.

Java obliga a que un método informe de las excepciones (explícitas) que puede lanzar. Un método no sólo tienen que decir qué devuelve si todo va bien, también debe indicar qué puede fallar. Un método especifica las excepciones que se pueden producir en él mediante la palabra throws en la declaración del método. Por ejemplo:

```
void f() throws EOFException, FileNotFoundException
{
     if( ... )
          throw new EOFException();
     if( ... )
          throw new FileNotFoundException();
}
```

6.3.- MANEJO DE EXCEPCIONES

Una vez que se detecta un error hace falta indicar quién se encarga de tratarlo. Para manejar excepciones hay que utilizar las palabras clave try y catch. Se ponen entre un try y un catch los métodos que pueden producir alguna excepción.

El try delimita el grupo de operaciones que pueden producir excepciones. El bloque catch es el lugar al que se transfiere el control si alguna de las operaciones produce una excepción. Es decir:

```
try{
     // Operaciones que pueden “fallar”, es decir, que
     // pueden producir alguna excepción.
}
catch( <tipoDeExcepción> ref ){
     // Tratamiento de esa excepción
}
```

Por ejemplo:

```
try{
     a.abreFichero();
     a.leeCabecera();
     a.actualizaDatos();
}
catch( IOException ref ){
     System.out.println( “Error de E/S” );
}
```

Si alguna de las operaciones del bloque produce una excepción, se interrumpe el bloque try y se ejecuta el catch. Al finalizar éste, se continúa normalmente. Si no se produce ninguna excepción el bloque catch se ignora.

Un bloque try puede tener varios catch asociados, uno para cada tipo de excepción que se pueda producir. Por ejemplo,

```
try{
     a.abreFichero();
     a.leeCabecera();
     a.actualizaDatos();
}
catch( FileNotFoundException ref ){
     System.out.println( “Error de apertura” );
}
catch( IOException ref ){
     System.out.println( “Error de E/S” );
}
```

Si se produce una excepción que no se corresponde con ningún catch indicado, la excepción se propaga hacia atrás en la secuencia de invocaciones hasta encontrar un catch adecuado. Por ejemplo:

```
void f1( int accion ) throws EOFException
{
     try{
          if( accion == 1 )
               throw new FileNotFoundException();
          else if( accion == 2 )
               throw new EOFException();
     }
     catch( FileNotFoundException e ){
          System.out.println( “Error corregido” );
     }
     System.out.println( “Finalización normal de f1” );
}

void f2( int accion )
{
     try{
          f1( accion ):
     }
     catch( EOFException e ){
          System.out.println( “Error corregido” );
     }
     System.out.println( “Finalización normal de f2” );
}
```

¿ Qué pasa si se llama a f2 pasándole los valores 1 ó 2 para el parámetro acción?
¿En qué método se producen las excepciones y en cuál se tratan?

Cuando se invoca a un método que lanza excepciones, es obligatorio:

- Que se maneje el error (en un catch).
- Que se indique mediante throws su propagación.

Es decir, o se trata el error o se avisa que se va a continuar sin haberlo corregido para que otro método lo corrija. Por ejemplo:

```
void f1() throws IOException
{
...
}

void f2()
{
     try{
          // Alternativa 1: tratar el error en cuanto se produce
          f1()
     }
     catch( IOException e ){
          // Tratamiento del error
     }
}

// Alternativa 2: se continúa propagando el error para que lo gestione
// otro método.
void f2() throws IOException
{
     f1();
}
```

Es mejor intentar tratar los errores en cuanto sea posible (alternativa 1), en vez de dejarlos para que los gestione alguien por encima (alternativa 2). De todos modos, no siempre es posible tratarlos en el mismo momento en que se producen y por tanto, a menudo hay que recurrir a la segunda alternativa.

6.4.- FINALLY

Puede haber ocasiones en que se desea realizar alguna operación tanto si se producen excepciones como si no. Dichas operaciones se pueden situar dentro de un bloque finally, de la siguiente forma:

```
try{
     // Operaciones con posibles excepciones
}
catch( <tipoDeExcepcion> ref ){
     //  Tratamiento de la excepción
}
finally{
     // Operaciones comunes
}
```

Ejemplo:

```
class Recurso
{
     void reserva(){ ... }
     void libera(){ ... }
}

class Ejemplo
{
     void prueba()
     {
          Recurso recurso = new Recurso();
          try{
               recurso.reserva();
               // Operaciones con posibles excepciones
               recurso.libera();
          }
          catch( ExceptionA a ){
               // Tratamiento del error
               recurso.libera();
          }
          catch( ExceptionB b ){
               // Tratamiento del error
               recurso.libera();
          }

          catch( ExceptionC c ){
               // Tratamiento del error
               recurso.libera();
          }
     }
}
```

En el ejemplo anterior queda suficientemente claro que independientemente de que se produzca una excepción o no, e independientemente de que ésta sea capturada o no, la liberación del recurso siempre debe llevarse a cabo (instrucción recurso.libera()). Para ello existe una solución mucho más elegante.

Solución con finally:

```
class Ejemplo
{
     void prueba()
     {
          Recurso recurso = new Recurso();
          try{
               recurso.reserva();
               // Operaciones con posibles excepciones
          }
          catch( ExceptionA a ){
               // Tratamiento del error
          }
          catch( ExceptionB b ){
               // Tratamiento del error
```

```
            }

            catch( ExceptionC c ){
                 // Tratamiento del error
            }
            finally{
                 recurso.libera();
            }
       }
  }
```

Si no se produce ninguna excepción, se ejecutarán el bloque try y el finally. En cambio, si se produce alguna excepción:

- Si es atrapada por un catch del mismo try, se ejecuta éste y luego el finally. La ejecución continúa después del finally con normalidad.
- Si no es atrapada, se ejecuta el finally y la excepción se propaga al contexto anterior.

6.5.- RELANZAMIENTO DE EXCEPCIONES

Hay situaciones en las que interesa relanzar una excepción ya atrapada, es decir, interesa dar un tratamiento determinado a una excepción pero también interesa seguir propagándola hacia arriba para que en un contexto superior se siga teniendo constancia de ella. Esto se hace así:

```
catch( Exception e ){
     // Salvar datos o cualquier otro tratamiento, pero propagar
     throw e;
}
```

La excepción pasaría a buscarse entre los catch de un contexto superior (los del mismo nivel se ignoran).

6.6.- JERARQUÍA DE EXCEPCIONES

Toda excepción debe ser una instancia de una clase derivada de la clase Throwable. De ella hereda una serie de métodos que dan información sobre cada excepción. Estos métodos son:

- getMessage
- toString
- printStackTrace (imprime en la salida estándar un volcado de la pila de llamadas).
- fillInStackTrace

La jerarquía de excepciones en Java es la siguiente:

- Throwable

- Error

  (AWTError, LinkageError, ThreadDeath, VirtualMachineError)

- Exception

  (ClassNotFoundException, CloneNotSupportedException, DataFormatException, GeneralSecurityException, IllegalAccessException, IOException, NoSuchFieldException, NoSuchMethodException, PrinterException, etc.)

- RuntimeException

  (ArithmeticException, ArrayStoreException, CannotRedoException, CannotUndoException, ClassCastException, EmptyStackException, IllegalArgumentException, IndexOutOfBoundsException, NullPointerException, etc.)

Que representada en forma gráfica da lugar a la figura 7.1.

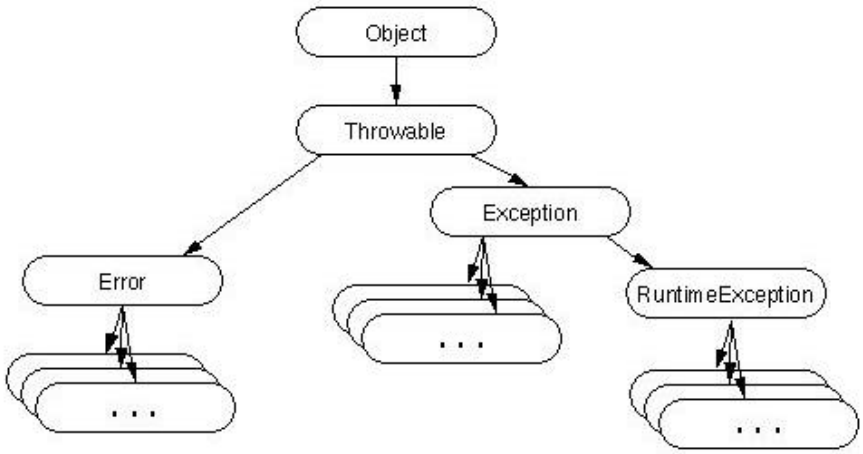

Figura 6.1: Jerarquía de clases de las excepciones Java

Todas las excepciones descienden de la clase Throwable, la cual se divide en dos subclases: Error y Exception. Las clases derivadas de Error describen errores internos de la JVM e indican errores serios que normalmente la aplicación no debería intentar gestionar. Tampoco deberían ser lanzadas por las clases del usuario. Estas excepciones rara vez ocurren, y cuando así sea, lo único que se puede hacer es intentar cerrar el programa sin perder datos. Ejemplos de este tipo de excepciones son OutOfMemoryError, StackOverflowError, etc.

Los programas en Java trabajarán con las excepciones de la rama Exception. Esta clase, a su vez, tiene una subclase llamada RuntimeException que, a su vez, tiene otras clases derivadas. Estas excepciones se clasifican en:

- **Explícitas:** las que derivan directamente de Exception.
- **Implícitas:** las que derivan de RunTimeException.

Se utilizan las RunTimeException para indicar un error de programación. Si se produce una excepción de este tipo hay que arreglar el código. Por ejemplo: un cast incorrecto, acceso a un array fuera de rango, uso de un puntero null, etc.

El resto de las Exception indican que ha ocurrido algún error debido a alguna causa ajena al programa (está correcto). Por ejemplo: un error de E/S, error de conexión, etc.

Los métodos deben declarar sólo las excepciones explícitas. Las implícitas no deben declararse (el compilador no lo exige, aunque pueden producirse igualmente). Por tanto, cuando un método declara una excepción, está avisando de que puede producirse dicho error (por causas externas al método) además de cualquier error implícito (consecuencia de un error en el código que debería ser subsanado).

6.7.- CREACIÓN DE EXCEPCIONES

Si se necesita notificar algún error no contemplado en Java se puede crear una nueva clase de Excepción. La única condición es que la clase derive de la clase Throwable o de alguna derivada.

En la práctica, normalmente derivará:

- de RuntimeException si se desea notificar un error de programación.
- de Exception en cualquier otro caso.

En un método que tiene como parámetro un entero entre 1 y 12 se recibe un 14 ¿Qué tipo de excepción se crearía para notificarlo?

Si un método para listar Clientes por impresora se encuentra con que deja de responder, ¿qué tipo de excepción debería crear?

Ejemplo de creación de una excepción:

```
class PrinterException extends Exception{}
```

Cuando se produzca una excepción de este tipo, se lanzará, como se puede ver en el siguiente ejemplo:

```
class Ejemplo
{
     void imprimirClientes( Cliente[] clientes ) throws PrinterException
     {
          for( int i=0; i<clientes.length; i++ )
          {
               // Imprimir cliente[i]
               if( “error impresión” ) throw new PrinterException();
          }
     }
}
```

Vemos ahora otro ejemplo en el que creamos y lanzamos una excepción de tipo RuntimeException:

```
class MesInvalidoException extends RuntimeException
{}

class Ejemplo
{
     void setMes( int mes )
     {
          if( mes < 1 || mes > 12 )
               throw new MesInvalidoException();
     }
}
```

En este ejemplo no se avisa que se lanza una excepción porque es de tipo RuntimeException (implícita).

6.8.- OBTENCIÓN DE INFORMACIÓN DE UNA EXCEPCIÓN

Una excepción, como cualquier otra clase, puede tener operaciones que permitan obtener más información sobre el error.

```
class PrinterException extends Exception
{
     private int numPagina;

     PrinterException( int pag )
     {
          numPagina = pag;
     }

     int getNumPagina()
     {
          return numPagina;
     }
}

class Ejemplo
{
     void imprimirClientes( Cliente[] clientes ) throws PrinterException
     {
          for( int pagina = 0; pagina<clientes.length; pagina++ )
          {
               // Imprimir cliente[ pagina ]
               if( “error impresión” ) throw new PrinterException( pagina );
          }
     }

     void Informe()
     {
          try{
               imprimirClientes( clientes );
```

```
            }
            catch( PrinterException e )
            {
                  System.out.println( “Sólo se han imprimido “ +
                        e.getNumPagina() );
            }
      }
}
```

Otro ejemplo con una excepción descendiente de RuntimeException:

```
class MesInvalidoException extends RuntimeException
{
      private int mes;

      MesInvalidoException( int m )
      {
            mes = m;
      }

      getMesInvalido()
      {
            return mes;
      }

      public String toString()
      // Método heredado de la clase Throwable, que devuelve
      // la representación String del error.
      {
            return “Mes inválido: “ + mes;
      }
}

class Fecha
{
      void setMes( int m )
      {
            if( m < 1 || m > 12 )
                  throw new MesInvalidoException( m );
      }
}

class Ejemplo
{
      public static void main( String[] args )
      {
            Fecha fecha = new Fecha();
            fecha.setMes( 14 );
      }
}
```

Java obliga a atrapar las excepciones explícitas, ¿qué pasa si no se atrapa una implícita, como en este caso?

- El compilador no da ningún error por tratarse de una excepción implícita y el programa compila correctamente.
- La excepción MesInvalidoException se produce al invocar el método setMes() con el valor 14 y al no haber sido capturada por el código que realiza la invocación (el main), el programa termina de manera anormal.
- Por defecto, se visualizará el mensaje "Mes inválido: 14 " y el programador deberá darse cuenta de que existe un error en el código que llama al método setMes().

La clase Throwable tiene una serie de métodos que tienen ya implementados todas las excepciones de Java (por herencia), y que pueden ser implementados por las nuevas excepciones que se creen. Algunos de estos métodos son:

- public String getMessage()

  Devuelve el mensaje detallado de error de este objeto (o null si el objeto no tiene mensaje)..

- public String getLocalizedMessage()

  Lo mismo, pero pensado para devolver el mensaje localizado de una forma más específica. Hay que redefinirlo, si no simplemente se comporta como getMessage()

- public String toString()

  Devuelve la representación String del error (la clase del error, básicamente).

- public void printStackTrace()

  Visualiza en el flujo de error estándar la traza de llamadas (backtrace) que se almacena automáticamente en todo error.

- public void printStackTrace(PrintStream s)

  Lo mismo, pero indicando el flujo al que se manda la información.

- public void printStackTrace(PrintWriter s)

  Idem.

- public native Throwable fillInStackTrace()

  "Rellena" la pila de llamadas; es decir, devuelve el error cambiando su traza de llamadas como si se hubiera producido ahora. Es útil cuando un error que se ha producido en un lugar queremos que se indique en otro.

6.9.- RESUMEN EXCEPCIONES EXPLÍCITAS-IMPLÍCITAS

Excepciones explícitas:

- Derivan de Exception (no de RuntimeException).
- Indican un error externo a la aplicación.
- Si se lanzan es obligatorio declararlas.
- Si se invoca un método que las lanza, es obligatorio atraparlas o declararlas.

Excepciones implícitas:

- Derivan de RuntimeException.
- Indican un error de programación.
- No se declaran: se corrigen.
- Se pueden atrapar. En caso contrario finaliza la aplicación.

6.10.- EXCEPCIONES Y HERENCIA

Al redefinir un método se está dando otra implementación para un mismo mensaje. El nuevo método sólo podrá lanzar excepciones declaradas en el método original.

```
class A {
     void f() throws EOFException { }
}

class B extends A {
     // Incorrecto
     void f() throws EOFException, FileNotFoundException {}
}
```

Es decir, un hijo puede lanzar menos excepciones que las que declara el padre, no puede lanzar una excepción que no lanzaba el padre, no puede lanzar nuevas excepciones.

Por otro lado, un método puede declarar excepciones que no lance realmente. Así un método base puede permitir que las redefiniciones puedan producir excepciones.

Los constructores, al no heredarse, sí pueden lanzar nuevas excepciones.

6.11.- NORMAS EN EL USO DE EXCEPCIONES

Norma 1:

- Si una excepción se puede manejar no debe propagarse.
- Es más cómodo usar métodos que no produzcan errores.

Norma 2:

- No utilizar las excepciones para evitar una consulta. No abusar de ellas.

```
try { stack.pop(); }
catch {EmptyStackExecution e) {
...
}

while (!stack.empty())
     stack.pop();
```

Norma 3:

- Separar el tratamiento de errores de la lógica.

```
for (int i = 0; i < len; i++) {
     try {
          obj1.mesg1();
     }
     catch(ExcA a) {
          // Tratamiento
     }
     try {
          obj2.mesg2();
     }
     catch(ExcB b) {
          // Tratamiento
     }
}


try {
     for (int i = 0; i < len; i++) {
          obj1.mesg1();
          obj2.mesg2();
     }
}
catch(ExcA a) {
     // Tratamiento
}
catch(ExcB b) {
     // Tratamiento
}
```

Norma 4:

- No ignorar una excepción

```
try {
     // operaciones;
}
catch {EmptyStackExecution e)
{ } // Para que calle el compilador
```

6.12.- EJEMPLOS

Ejemplo 1: Lectura de un número por teclado

```
class LeerTeclado {
     static BufferedReader inp = new BufferedReader( new
               InputStreamReader(System.in));

     public static String leerLinea() {
          String s = "";
          try {
               s = inp.readLine();
          } catch (java.io.IOException e) { }
          return s;
          // Esto es JDK 1.1 y 1.2. Otra forma de hacerlo
          // (con JDK 1.0) es esta:
          // char c;
          // try {
          //   while( (c = (char)System.in.read()) != '\n')
          //         s += c;
          // }
          // catch (java.io.IOException e) { }
     }

     public static String leerLinea( String mens ) {
          // con prompt
          System.out.print( mens );
          return leerLinea();
     }

     /* leerDoble() Devuelve NaN en caso de error */
     public static double leerDoble() {
          String s = leerLinea();
          double d;
          try {
               d = Double.valueOf( s ).doubleValue();
          }
          catch (java.lang.NumberFormatException e) {
               d = Double.NaN;
          }
          return d;
     }
```

```
        public static double leerDoble( String mens ) {
            // con prompt
            System.out.print( mens );
            return leerDoble();
        }

        public static void main (String[] a) {
            double d1 = leerDoble( "Introduce un número real: " );
            double d2 = leerDoble( "Introduce otro:           " );
            System.out.println( "La suma de ambos es: " + (d1+d2) );
        }
    }
```

Ejemplo 2: Incorporación de excepciones a ListaEnlazada.

```
    public class FueraDeListaException extends IndexOutOfBoundsException {
        FueraDeListaException( ) {
        }
        FueraDeListaException( String s ) {
            super( s );  // Llama al constructor del padre
        }
    }

    public class ListaEnlazada {
        . . .
        public void insertar( Object o, int posicion )
                                            throws FueraDeListaException {
            NodoLista aux = l;
            for (int i = 1; aux != null && i < posicion-1; i++ )
            {
                aux = aux.siguiente;
            }

            if (posicion == 1)
                l = new NodoLista( o, l );
            else {
                if (aux == null)
                    throw new FueraDeListaException("intentando insertar
                                    elemento más allá del fin de la lista" );
                else
                    aux.siguiente = new NodoLista( o, aux.siguiente );
                }
            }

        public void borrarPrimero() throws FueraDeListaException {
            if (l == null)
                throw new FueraDeListaException("intento de borrar primero en
                                                            lista vacía" );
            else
                l = l.siguiente;
        }

        public static void main( String[] a ) {
            ListaEnlazada l = new ListaEnlazada();
```

```
            try {
                 l.borrarPrimero();
            } catch (FueraDeListaException e) {
                 System.out.println( "Error: " + e.getMessage() +
                                                  ". Continuamos..." );
            }
            l.insertarPrimero( new Integer( 1 ) );
            l.insertarPrimero( new Integer( 2 ) );
            l.insertar( new Integer( 4 ), 1 );
            l.insertar( new Integer( 3 ), 2 );
            // no obligatorio recoger el error
            // pq es RuntimeException
            l.insertar( new Double( 2.5 ), 3 );
            try {
                 l.insertar( new Integer( 18 ), 10 );
            } catch (FueraDeListaException e) {
                 System.out.println( "Error: " + e.getMessage() +
                                                  ". Continuamos..." );
            }
            finally {
                 l.insertarPrimero( new Integer(18) );
            }
            System.out.println( l );
            l.destruir();
            l.borrarPrimero();
            // de este error no se recupera el programa
            System.out.println( "Aquí no se llega... error en ejecución" );
       }
  }
```

Salida:

La salida resultado de ejecutar este main() sería la siguiente:

```
Error: intento de borrar primero en lista vacía. Continuamos...
Error: intentando insertar elemento más allá del fin de la lista.
            Continuamos...
( 4 3 2.5 2 1 )
FueraDeListaException: intento de borrar primero en lista vacía
at ListaEnlazada.borrarPrimero(ListaEnlazada.java:42)
     at ListaEnlazada.main(ListaEnlazada.java:66)
```

6.13.- EJERCICIOS

Ejercicio 1: División por cero

Escribe una clase llamada DividePorCero que pruebe a dividir un entero entre otro recogiendo la excepción de división por cero y mostrando el mensaje "Error: división por cero" si se produce.

Ejercicio 2: Clase Fracción

Diseña e implementa una clase Fraccion que permite crear fracciones (numerador y denominador enteros), con métodos para sumar, restar, multiplicar y dividirlas.

Crea una excepción FraccionException que se lance siempre que en una operación de fracción se produzca un error, a saber:

- Creación de una fracción con denominador cero.
- División de una fracción entre otra con numerador cero.
- Cualquier operación que incluya una fracción con denominador cero.

Hacer que cada operación crítica lance esta excepción si se produce, con un mensaje indicativo del tipo de error (mensaje incluido en la excepción, no salida a pantalla!). Probar la clase.