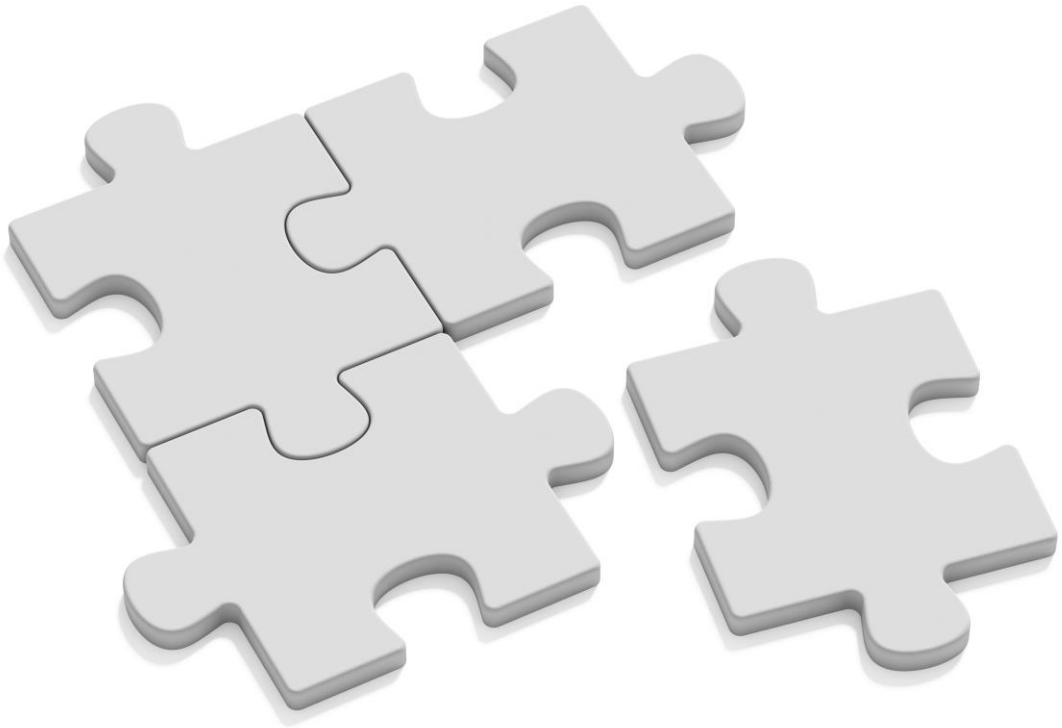
Universidad Tecnológica Nacional



2017

CÁTEDRA DE LENGUAJE DE PROGRAMACIÓN JAVA

Ings. Mario Bressano & Miguel Iwanow

ENVÍO 08/2017

Fuente:

Manejo de Errores Usando Excepciones Java
Autor: Sun
Traductor: Juan Antonio Palos
Corrección y ampliación: Mario Osvaldo Bressano

Web: http://www.programacion.net/java/tutorial

¿Por qué capturar y mostrar errores?

El lenguaje Java fue creado con la intención de ser independiente del Sistema Operativo con el cual se corre. Debido a esto se lo dotó de un manejo de errores que le confiera robustez (se dice que "Java es tolerante a las fallas") ante cualquier falla que pudiera tornar en desastrosa la ejecución de la aplicación.

Una excepción ocurre cuando no se cumple la norma establecida; su pseudocódigo sería:

Realizar tarea

    Si tarea no se realiza correctamente

        Procesar errores

El lenguaje maneja errores sincrónicos tales como:

a- Índices fuera de rango,

b- Desbordamiento aritmético,

c- División por cero,

d- Asignación fallida de memoria,

e- Interrupción de subprocesos,

f- Parámetros inválidos del método.

Por otro lado, no maneja errores asincrónicos:

a- Operaciones de E/S, de disco, de teclas, etc.

La estructura para la captura de errores es la siguiente:

```
try     {
        Contiene instrucciones que pueden causar excepciones
        }
catch (parámetros excepción)
        {
        Manejador de la instrucción
        }
finally
        {
        Se ejecuta siempre                          (Esta cláusula es opcional)
        }
```

Clases generadoras de excepciones de uso frecuente

NOMBRE DE LA CLASE	DESCRIPCIÓN
ArithmeticException	Excepción aritmética (p.e. división por cero)
ArrayStoreException	Excepción en una operación uso de un Array
NegativeArraySizeException	Excepción en una operación uso de un Array
NullPointerException	Excepción producida por una referencia a un objeto nulo
SecurityException	Excepción en el sistema de seguridad
EOFException	Excepción producida al alcanzar el final del

	fichero
IOException	Excepción en el proceso de E/S
FileNotFoundException	Excepción por no encontrar un fichero
NumberFormatException	Excepción en la conversión de una cadena de caracteres a un valor numérico

Manejo de errores en una aplicación: Ingresar dos números enteros y dividirlos. Mostrar el resultado. Manejar los errores esperados:

- División por cero
- Ingreso de números no enteros

```
//Manejo de errores- División por cero y error de formato numérico

import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

public class ManejoDeErrores extends JFrame

  implements ActionListener {

  private JTextField campoIngreso1, campoIngreso2, campoSalida;

  private int numero1, numero2, resultado;

  public static void main( String args[] )

  {

    ManejoDeErrores application = new ManejoDeErrores();

    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```


```
}

// Configuramos  GUI

public ManejoDeErrores()

{

   super( "Manejo de Excepciones" );



   // Configuramos el panel contenedor y su layout

   Container contenedor = getContentPane();

   contenedor.setLayout( new GridLayout( 3, 2 ) );

   // Configura etiqueta y campoIngreso1

   contenedor.add(

      new JLabel( "Ingresar el numerador: ", SwingConstants.RIGHT ) );

   campoIngreso1 = new JTextField();

   contenedor.add( campoIngreso1 );

   // Configura etiqueta y campoIngreso2; registro la escucha

   contenedor.add( new JLabel( "Ingresar el denominador y presionar <Enter> ",

      SwingConstants.RIGHT ) );

   campoIngreso2 = new JTextField();

   contenedor.add( campoIngreso2 );

   campoIngreso2.addActionListener( this );
```


```
    // Configura etiqueta y campoSalida
    contenedor.add( new JLabel( "Resultado ", SwingConstants.RIGHT ) );
    campoSalida = new JTextField();
    contenedor.add( campoSalida );
    setSize( 425, 100 );
    setVisible( true );
  } // fin del constructor ManejoDeErrores
  // Procesar eventos de la  GUI
  public void actionPerformed( ActionEvent event )
  {
    campoSalida.setText( "" );   // Borro campoSalida
    // Leo dos valores numéricos y calculo el cociente - Aquí comienza el bloque donde puede haber errores
    try {
      numero1 = Integer.parseInt( campoIngreso1.getText() );
      numero2 = Integer.parseInt( campoIngreso2.getText() );
      resultado = cociente( numero1, numero2 );
      campoSalida.setText( String.valueOf( resultado ) );
    }
    // Proceso el error de ingreso producido por mal formato del número
    catch ( NumberFormatException numberFormatException ) {
```


```
      JOptionPane.showMessageDialog( this,
         "Usted debe ingresar números enteros", "Formato de número no válido",
         JOptionPane.ERROR_MESSAGE );
      campoIngreso1.setText( "" );   // Borro campoIngreso1
      campoIngreso2.setText( "" );   // Borro campoIngreso2
   }
   // Procesamiento error de división por cero
   catch ( ArithmeticException arithmeticException ) {
      JOptionPane.showMessageDialog( this,
        "División por cero","Excepción Aritmética",
         JOptionPane.ERROR_MESSAGE );
      campoIngreso1.setText( "" );   // Borro campoIngreso1
      campoIngreso2.setText( "" );   // Borro campoIngreso2
      campoSalida.setText( "" );   // Borro campoSalida
   }
} // fin de método ActionPerformed
// Toma del error en la división por cero
public int cociente( int numerador, int denominador )
   throws ArithmeticException
{
```


```
    return numerador / denominador;

  }

} //
```

Cuando encontramos que un conjunto de sentencias puede lanzar un error, lo abarcamos dentro de un bloque **try**. El objeto excepción es lanzado automáticamente y es capturado por los bloques **catch**, los cuales se disponen uno a continuación del otro, de manera que el programa investiga cuál le conviene.

La declaración del método throw permite lanzar excepciones especificadas por el usuario.

Requerimientos Java para Capturar o Especificar Excepciones

Java requiere que un método o capture o especifique todas las excepciones chequeadas que se pueden lanzar dentro de su ámbito. Este requerimiento tiene varios componentes que necesitan una mayor descripción.

Capturar

Un método puede capturar una excepción proporcionando un manejador para ese tipo de excepción.

Especificar

Si un método decide no capturar una excepción, debe especificar que puede lanzar esa excepción. ¿Por qué hicieron este requerimiento los diseñadores de Java? Porque una excepción que puede ser lanzada por un método es realmente una parte del interfase de programación público del método: los llamadores de un método deben conocer las excepciones que ese método puede lanzar para poder decidir inteligente y concienzudamente qué hacer son esas excepciones. Así, en la firma del método debe especificar las excepciones que el método puede lanzar.

Excepciones Chequeadas

Java tiene diferentes tipos de excepciones, incluyendo las excepciones de I/O, las excepciones en tiempo de ejecución, y las de su propia creación. Las que nos interesan a nosotros para esta explicación son las excepciones en tiempo de ejecución, Estas excepciones son aquellas que ocurren dentro del sistema de ejecución de Java. Esto incluye las excepciones aritméticas

(como dividir por cero), excepciones de puntero (como intentar acceder a un objeto con una referencia nula), y excepciones de indexación (como intentar acceder a un elemento de un array con un índice que es muy grande o muy pequeño).
Las excepciones en tiempo de ejecución pueden ocurrir en cualquier parte de un programa y en un programa típico pueden ser muy numerosas.
Muchas veces, el costo de chequear todas las excepciones en tiempo de ejecución excede de los beneficios de capturarlas o especificarlas. Así el compilador no requiere que se capturen o especifiquen estas excepciones, pero se puede hacer.
Las excepciones chequeadas son excepciones que no son excepciones en tiempo de ejecución y que son chequeadas por el compilador (esto es, el compilador comprueba que esas excepciones son capturadas o especificadas).
Algunas veces esto se considera como un bucle cerrado en el mecanismo de manejo de excepciones de Java y los programadores se ven tentados a convertir todas las excepciones en excepciones en tiempo de ejecución. En general, esto no está recomendado.

Excepciones que pueden ser lanzadas desde el ámbito de un método

Esta sentencia podría parecer obvia a primera vista: sólo hay que fijarse en la sentencia **throw**. Sin embargo, esta sentencia incluye algo más no sólo las excepciones que pueden ser lanzadas directamente por el método: la clave esta en la frase dentro del ámbito de. Esta frase incluye cualquier excepción que pueda ser lanzada mientras el flujo de control permanezca dentro del método. Así, esta sentencia incluye.

- excepciones que son lanzadas directamente por el método con la sentencia **throw** de Java, y
- las excepciones que son lanzadas por el método indirectamente a través de llamadas a otros métodos.

Tratar con las Excepciones Java

Ahora vamos a ver cómo capturar una excepción y cómo especificar otra.

El ejemplo: ListOfNumbers

Las secciones posteriores, sobre como capturar y especificar excepciones, utilizan el mismo ejemplo. Este ejemplo define e implementa un clase llamada ListOfNumbers. Esta clase llama a dos clases de los paquetes de Java que pueden lanzar excepciones.

Capturar y Manejar Excepciones

Una vez que te has familiarizado con la clase ListOfNumbers y con las excepciones que pueden ser lanzadas, puedes aprender cómo escribir manejadores de excepción que puedan capturar y manejar esas excepciones.

Esta sección cubre los tres componentes de una manejador de excepción -- los bloques **try**, **catch**, y **finally** -- y muestra cómo utilizarlos para escribir un manejador de excepción para el método **writeList()** de la clase ListOfNumbers. Además, esta sección contiene una página que pasea a lo largo del método **writeList()** y analiza lo que ocurre dentro del método en varios escenarios.

Especificar las Excepciones que pueden ser Lanzadas por un Método

Si no es apropiado que un método capture y maneje una excepción lanzada por un método que él ha llamado, o si el método lanza su propia excepción, debe especificar en la firma del método que éste puede lanzar una excepción. Utilizando la clase ListOfNumbers, esta sección le muestra cómo especificar las excepciones lanzadas por un método.

El Ejemplo ListOfNumbers

Las dos secciones siguientes que cubren la captura y especificación de excepciones utilizan este ejemplo.

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
```


```
            victor.addElement(new Integer(i));
    }
    public void writeList() {
        PrintStream pStr = null;

        System.out.println("Entering try statement");
        int i;
        pStr = new PrintStream(
                new BufferedOutputStream(
                  new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));

        pStr.close();
    }
}
```

Este ejemplo define e implementa una clase llamada ListOfNumbers. Sobre su construcción, esta clase crea un Vector que contiene diez elementos enteros con valores secuenciales del 0 al 9.

Esta clase también define un método llamado **writeList()** que escribe los números de la lista en un fichero llamado "OutFile.txt".

El método **writeList()** llama a dos métodos que pueden lanzar excepciones. Primero la siguiente línea invoca al constructor de FileOutputStream, que lanza una excepción IOException si el fichero no puede ser abierto por cualquier razón.

```
pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));
```

Segundo, el método **elementAt()** de la clase Vector lanza una excepción ArrayIndexOutOfBoundsException si se le pasa un índice cuyo valor sea demasiado pequeño (un número negativo) o demasiado grande (mayor que el número de elementos que contiene realmente el Vector). Aquí está cómo ListOfNumbers invoca a **elementAt()**.

```
pStr.println("Value at: " + i + " = " + victor.elementAt(i));
```

Si se intenta compilar la clase ListOfNumbers, el compilador dará un mensaje de error sobre la excepción lanzada por el constructor de FileOutputStream, pero no muestra ningún error sobre la excepción lanzada por **elementAt()**.

Esto es porque la excepción lanzada por FileOutputStream, es una excepción chequeada y la lanzada por **elementAt()** es una ejecución de tiempo de ejecución. Java sólo requiere que se especifiquen o capturen las excepciones chequeadas.

Capturar y Manejar Excepciones

Ahora se verá cómo construir un manejador de excepciones para el método **writeList()** descrito en el ejemplo: ListOfNumbers. Las tres primeras páginas listadas abajo describen tres componentes diferentes de un manejador de excepciones y le muestran cómo pueden utilizarse esos componentes en el método **writeList()**. La cuarta página trata sobre el método **writeList()** resultante y analiza lo que ocurre dentro del código de ejemplo a través de varios escenarios.

El Bloque try

El primer paso en la escritura de una manejador de excepciones es poner la sentencia Java dentro de la cual se puede producir la excepción dentro de un bloque **try**. Se dice que el bloque **try gobierna** las sentencias encerradas dentro de él y define el ámbito de cualquier manejador de excepciones (establecido por el bloque **catch** subsiguiente) asociado con él.

Los bloques catch

Después se debe asociar un manejador de excepciones con un bloque **try** proporcionándole uno o más bloques **catch** directamente después del bloque **try**.

El bloque finally

El bloque **finally** de Java proporciona un mecanismo que permite a sus métodos limpiarse a si mismos sin importar lo que sucede dentro del bloque **try**. Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema.

Poniéndolo Todo Junto

Las secciones anteriores describen cómo construir los bloques de código **try**, **catch**, y **finally** para el ejemplo **writeList()**. Ahora, pasearemos sobre el código para investigar que sucede en varios escenarios.

El Bloque Try

El primer paso en la construcción de un manejador de excepciones es encerrar las sentencias que podrían lanzar una excepción dentro de un bloque **try**. En general, este bloque se parece a esto.

```
try {
   sentencias Java
```

}

El segmento de código etiquetado **sentencias java** está compuesto por una o más sentencias legales de Java que podrían lanzar una excepción.

Para construir un manejador de excepcón para el método **writeList()** de la clase ListOfNumbers, se necesita encerrar la sentencia que lanza la excepción en el método **writeList()** dentro de un bloque **try**.

Existe más de una forma de realizar esta tarea. Podríamos poner cada una de las sentencias que potencialmente pudieran lanzar una excepción dentro de su propio bloque **try**, y proporcionar manejadores de excepciones separados para cada uno de los bloques **try**. O podríamos poner todas las sentencias de **writeList()** dentro de un sólo bloque **try** y asociar varios manejadores con él. El suguiente listado utiliza un sólo bloque **try** para todo el método porque el código tiende a ser más fácil de leer.

```
PrintStream pstr;

try {
   int i;

   System.out.println("Entering try statement");
   pStr = new PrintStream(
           new BufferedOutputStream(
             new FileOutputStream("OutFile.txt")));

   for (i = 0; i < size; i++)
      pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

Se dice que el bloque **try gobierna** las sentencias encerradas dentro del él y define el ámbito de cualquier manejador de excepción (establecido por su subsiguiente bloque **catch**) asociado con él. En otras palabras, si ocurre una excepción dentro del bloque **try**, esta excepción será manejada por el manejador de excepción asociado con esta sentencia **try**.

Una sentencia **try** debe ir acompañada de al menos un bloque **catch** o un bloque **finally**.

Los Bloques catch

Como se aprendió, la sentencia **try** define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia **try** proporcionando uno o más bloques **catch** directamente después del bloque **try**.

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

No puede haber ningún código entre el final de la sentencia **try** y el principio de la primera sentencia **catch**. La forma general de una sentencia **catch** en Java es esta.

```
catch (AlgunObjetoThrowable nombreVariable) {
   Sentencias Java
}
```

Como puedes ver, la sentencia **catch** requiere un sólo argumento formal. Este argumento parece un argumento de una declaración de método. El tipo del argumento **AlgunObjetoThrowable** declara el tipo de excepción que el manejador puede manejar y debe ser el nombre de una clase heredada de la clase Throwable definida en el paquete java.lang. (Cuando los programas Java lanzan una excepción realmente están lanzado un objeto, sólo pueden lanzarse los objetos derivados de la clase Throwable..)

nombreVariable es el nombre por el que el manejador puede referirse a la excepción capturada. Por ejemplo, los manejadores de excepciones para el método **writeList()** (mostrados más adelante) llaman al método **getMessage()** de la excepción utilizando el nombre de excepción declarado **e**.

```
e.getMessage()
```

Se puede acceder a las variables y métodos de las excepciones en la misma forma que accede a los de cualquier otro objeto. **getMessage()** es un método proporcionado por la clase Throwable que imprime información adicional sobre el error ocurrido. La clase Throwable también implementa dos métodos para rellenar e imprimir el contenido de la pila de ejecución cuando ocurre la excepción. Las subclases de Throwable pueden añadir otros métodos o variables de ejemplar, Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases antecesoras.

El bloque **catch** contiene una serie de sentencias Java legales. Estas sentencias se ejecutan cuando se llama al manejador de excepción. El sistema de ejecución llama al manejador de excepción cuando el manejador es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

El método **writeList()** de la clase de ejemplo ListOfNumbers utiliza dos manejadores de excepción para su sentencia **try**, con un manejador para cada uno de los tipos de excepciones que pueden lanzarse dentro del bloque **try** -- ArrayIndexOutOfBoundsException y IOException.

```
try {
    . . .
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Ocurre una IOException

Supongamos que ocurre una excepción IOException dentro del bloque **try**. El sistema de ejecución inmediatamente toma posesión e intenta localizar el manejador de excepción adecuado. El sistema de ejecución empieza buscando al principio de la pila de llamadas. Sin embargo, el constructor de FileOutputStream no tiene un manejador de excepción apropiado por eso el sistema de ejecución comprueba el siguiente método en la pila de llamadas -- el método **writeList()**. Este método tiene dos manejadores de excepciones: uno para ArrayIndexOutOfBoundsException y otro para IOException.

El sistema de ejecución comprueba los manejadores de **writeList()** por el orden en el que aparecen después del bloque **try**. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una ArrayIndexOutOfBoundsException, pero la excepción que se ha lanzado era una IOException. Una excepción IOException no puede asignarse legalmente a una ArrayIndexOutOfBoundsException, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de **writeList()** es una IOException. La excepción lanzada por el constructor de FileOutputStream también es un una IOException y por eso puede ser asignada al argumento del manejador de excepciones de IOException. Así, este

manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

Caught IOException: OutFile.txt

El sistema de ejecución sigue un proceso similar si ocurre una excepción ArrayIndexOutOfBoundsException. Para más detalles puedes ver.

Poniéndolo todo Junto que te lleva a través de método **writeList()** después de haberlo completado (queda un paso más) e investiga lo que sucede en varios escenarios.

Capturar Varios Tipos de Excepciones con Un Manejador

Los dos manejadores de excepción utilizados por el método **writeList()** son muy especializados. Cada uno sólo maneja un tipo de excepción. El lenguaje Java permite escribir manejadores de excepciones generales que pueden manejar varios tipos de excepciones.

Como ya sabes, las excepciones Java son objetos de la clase Throwable (son ejemplares de la clase Throwable a de alguna de sus subclases). Los paquetes Java contienen numerosas clases derivadas de la clase Throwable y así construyen un árbol de clases Throwable.

El manejador de excepción puede ser escrito para manejar cualquier calse heredada de Throwable. Si se escribe un manejador para una clase 'hoja? (una clase que no tiene subclases), se habrá escrito un manejador especializado: sólo maneja excepciones de un tipo específico. Si se escribe un manejador para una clase 'nodo' (una clase que tiene subclases), se habrá escrito un manejador general: se podrá manejar cualquier excepción cuyo tipo sea el de la clase nodo o de cualquiera de sus subclases.

Módifiquemos de nuevo el método **writeList()**. Sólo esta vez, escribamoslo para que maneje las dos excepciones IOExceptions y ArrayIndexOutOfBoundsExceptions. El antecesor más cercano de estas dos excepciones es la clase Exception. Así un manejador de excepción que quisiera manejar los dos tipos se parecería a esto.

```
try {
   . . .
} catch (Exception e) {
   System.err.println("Exception caught: " + e.getMessage());
}
```

La clase Exception está bastante arriba en el árbol de herencias de la clase Throwable. Por eso, además de capturar los tipos de IOException y ArrayIndexOutOfBoundsException este manejador de excepciones, puede capturar otros muchos tipos. Generalmente hablando, los manejadores de excepción deben ser más especializados.

Los manejadores que pueden capturar la mayoría o todas las excepciones son menos utilizados para la recuperación de errores porque el manejador tiene que determinar qué tipo de excepción ha ocurrido de todas formas (para determinar la mejor estrategia de recuperación). Los manejadores de excepciones que son demasiado generales pueden hacer el código **más propenso** a errores mediante la captura y manejo de excepciones que no fueron anticipadas por el programador y para las que el manejador no está diseñado.

El Bloque finally

El paso final en la creación de un manejador de excepción es proporcionar un mecanismo que limpie el estado del método antes (posiblemente) de permitir que el control pase a otra parte diferente del programa. Se puede hacer esto encerrando el código de limpieza dentro de un bloque **finally**.

El bloque **try** del método **writeList()** ha estado trabajando con un PrintStream abierto. El programa debería cerrar ese canal antes de permitir que el control salga del método **writeList()**. Esto plantea un problema complicado, ya que el bloque **try** del **writeList()** tiene tres posibles salidas.

1. La sentencia **new FileOutputStream** falla y lanza una IOException.
2. La sentencia **victor.elementAt(i)** falla y lanza una ArrayIndexOutOfBoundsException.
3. Todo tiene éxito y el bloque **try** sale normalmente.

El sistema de ejecución siempre ejecuta las sentencias que hay dentro del bloque **finally** sin importar lo que suceda dentro del bloque **try**. Esto es, sin importar la forma de salida del bloque **try** del método **writeList()** debido a los escenarios 1, 2 ó 3 listados arriba, el código que hay dentro del bloque **finally** será ejecutado de todas formas.

Este es el bloque **finally** para el método **writeList()**. Limpia y cierra el canal PrintStream.

```
finally {
    if (pStr != null) {
        System.out.println("Closing PrintStream");
        pStr.close();
    } else {
        System.out.println("PrintStream not open");
    }
}
```

¿Es realmente necesaria la sentencia finally?

La primera necesidad de la sentencia **finally** podría no aparecer de forma inmediata.
Esta necesidad de la sentencia **finally** no aparece hasta que se considera lo siguiente: ¿cómo se podría cerrar el PrintStream en el método **writeList()** si no se proporcionara un manejador de excepción para la ArrayIndexOutOfBoundsException y ocurre una ArrayIndexOutOfBoundsException? (sería sencillo y legal omitir un manejador de excepción para ArrayIndexOutOfBoundsException porque es una excepción en tiempo de ejecución y el compilador no alerta de que **writeList()** contiene una llamada a un método que puede lanzar una).
La respuesta es que el PrintStream no se cerraría si ocurriera una excepción ArrayIndexOutOfBoundsException y **writeList()** no proporcionara u manejador para ella -- a menos que **writeList()** proporcionara una sentencia **finally**.
Existen otros beneficios de la utilización de la sentencia **finally**. En el ejemplo de **writeList()** es posible proporcionar un código de limpieza sin la intervención de una sentencia **finally**. Por ejemplo, podríamos poner el código para cerrar el PrintStream al final del bloque **try** y de nuevo dentro del manejador de excepción para ArrayIndexOutOfBoundsException, como se muestra aquí.

```
try {
   . . .
   pStr.close();      // No haga esto, duplica el código
} catch (ArrayIndexOutOfBoundsException e) {
   pStr.close();      // No haga esto, duplica el código
   System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
   System.err.println("Caught IOException: " + e.getMessage());
}
```

Sin embargo, esto duplica el código, haciéndolo díficil de leer y propenso a errores si se modifica más tarde, Por ejemplo, si se añade código al bloque **try** que pudiera lanzar otro tipo de excepción, se tendría que recordar el cerrar el PrintStream dentro del nuevo manejador de excepción (lo que se olvidará seguro si se parece a mí).

Poniéndolo todo Junto

Cuando se juntan todos los componentes, el método **writeList()** se parece a esto.

```
public void writeList() {
```


```
    PrintStream pStr = null;

    try {
        int i;

        System.out.println("Entrando en la Sentencia try");
        pStr = new PrintStream(
                new BufferedOutputStream(
                 new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (pStr != null) {
            System.out.println("Cerrando PrintStream");
            pStr.close();
        } else {
            System.out.println("PrintStream no está abierto");
        }
    }
}
```

El bloque **try** de este método tiene tres posibilidades de salida direrentes.

1. La sentencia **new FileOutputStream** falla y lanza una IOException.
2. La sentencia **victor.elementAt(i)** falla y lanza una ArrayIndexOutOfBoundsException.
3. Todo tiene éxito y la sentencia **try** sale normalmente.

Está página investiga en detalle lo que sucede en el método **writeList** durante cada una de esas posibilidades de salida.

Escenario 1:Ocurre una excepción IOException

La sentencia **new FileOutputStream("OutFile.txt")** puede fallar por varias razones: el usuario no tiene permiso de escritura sobre el fichero o el directorio, el sistema de ficheros está lleno, o no existe el directorio. Si cualquiera de estas situaciones es verdadera el constructor de FileOutputStream lanza una excepción IOException.

Cuando se lanza una excepción IOException, el sistema de ejecución para inmediatamente la ejecución del bloque **try**. Y luego intenta localizar un manejador de excepción apropiado para manejar una IOException.

El sistema de ejecución comienza su búsqueda al principio de la pila de llamadas. Cuando ocurrió la excepción, el constructor de FileOutputStream estaba al principio de la pila de llamadas. Sin embargo, este constructor no tiene un manejador de excepción apropiado por lo que el sistema comprueba el siguiente método que hay en la pila de llamadas -- el método **writeList()**. Este método tiene dos manejadores de excepciones: uno para ArrayIndexOutOfBoundsException y otro para IOException.

El sistema de ejecución comprueba los manejadores de **writeList()** por el orden en el que aparecen después del bloque **try**. El primer manejador de excepción cuyo argumento corresponda con el de la excepción lanzada es el elegido por el sistema de ejecución. (El orden de los manejadores de excepción es importante!) El argumento del primer manejador es una ArrayIndexOutOfBoundsException, pero la excepción que se ha lanzado era una IOException. Una excepción IOException no puede asignarse legalmente a una ArrayIndexOutOfBoundsException, por eso el sistema de ejecución continúa la búsqueda de un manejador de excepción apropiado.

El argumento del segundo manejador de excepción de **writeList()** es una IOException. La excepción lanzada por el constructor de FileOutputStream también es una una IOException y por eso puede ser asignada al argumento del manejador de excepciones de IOException. Así, este manejador parece el apropiado y el sistema de ejecución ejecuta el manejador, el cual imprime esta sentencia.

Caught IOException: OutFile.txt

Después de que se haya ejecutado el manejador de excepción, el sistema pasa el control al bloque **finally**. En este escenario particular, el canal PrintStream nunca se ha abierto, así el **pStr** es null y no se cierra. Después de que se haya completado la ejecución del bloque **finally**, el programa continua con la primera sentencia después de este bloque.

La salida completa que se podrá ver desde el programa ListOfNumbers cuando se lanza un excepción IOException es esta.

Entrando en la sentecia try

Caught IOException: OutFile.txt

PrintStream no está abierto

Escenario 2: Ocurre una excepción ArrayIndexOutOfBoundsException

Este escenario es el mismo que el primero excepto que ocurre un error diferente dentro del bloque **try**. En este escenario, el argumento pasado al método **elementAt()** de Vector está fuera de límites. Esto es, el argumento es menor que cero o mayor que el tamaño del array. (De la forma en que está escrito el código, esto es realmente imposible, pero supongamos que se ha introducido un error cuando alguien lo ha modificado).

Como en el escenario 1, cuando ocurre una excepción el sistema de ejecución para la ejecución del bloque **try** e intenta localizar un manejador de excepción apropiado para ArrayIndexOutOfBoundsException. El sistema busca como lo hizo anteriormente. Llega a la sentencia **catch** en el método **writeList()** que maneja excepciones del tipo ArrayIndexOutOfBoundsException. Como el tipo de la excepción corresponde con el de el manejador, el sistema ejecuta el manejador de excepción.

Después de haber ejecutado el manejador de excepción, el sistema pasa el control al bloque **finally**. En este escenario particular, el canal PrintStream si que se ha abierto, así que el bloque **finally** lo cerrará. Después de que el bloque **finally** haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida completa que dará el programa ListOfNumbers si ocurre una excepción ArrayIndexOutOfBoundsException.

Entrando en la sentencia try
Caught ArrayIndexOutOfBoundsException: 10 >= 10
Cerrando PrintStream

Escenario 3: El bloque try sale normalmente

En este escenario, todas las sentencias dentro del ámbito del bloque **try** se ejecutan de forma satisfactoria y no lanzan excepciones. La ejecución cae al final del bloque **try** y el sistema pasa el control al bloque **finally**. Como todo ha salido satisfactorio, el PrintStream abierto se cierra cuando el bloque **finally** consigue el control. De nuevo, Después de que el bloque **finally** haya completado su ejecución, el programa continúa con la primera sentencia después de este bloque.

Aquí tienes la salida cuando el programa ListOfNumbers cuando no se lanzan excepciones.

Entrando en la sentencia try
Cerrando PrintStream

Especificar las Excepciones Lanzadas por un Método

Le sección anterior mostraba como escribir un manejador de excepción para el método **writeList()** de la clase ListOfNumbers. Algunas veces, es apropiado capturar las excepciones que ocurren pero en otras ocasiones, sin embargo, es mejor dejar que un método superior en la pila de llamadas maneje la excepción. Por ejemplo, si se está utilizando la clase ListOfNumbers como parte de un paquete de clases, probablemente no se querrá anticipar las necesidades de todos los usuarios de su paquete. En este caso, es mejor **no** capturar las excepciones y permitir que alguien la capture más arriba en la pila de llamadas.

Si el método **writeList()** no captura las excepciones que pueden ocurrir dentro de él, debe especificar que puede lanzar excepciones. Modifiquemos el método **writeList()** para especificar que puede lanzar excepciones. Como recordatorio, aquí tienes la versión original del método **writeList()**.

```
public void writeList() {
    System.out.println("Entrando en la sentencia try");
    int i;
    pStr = new PrintStream(
            new BufferedOutputStream(
              new FileOutputStream("OutFile.txt")));

    for (i = 0; i < size; i++)
       pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

Como recordarás, la sentencia **new FileOutputStream("OutFile.txt")** podría lanzar un excepción IOException (que no es una excepción en tiempo de ejecución). La sentencia **victor.elementAt(i)** puede lanzar una excepción ArrayIndexOutOfBoundsException (que es una subclase de la clase RuntimeException, y es una excepción en tiempo de ejecución).

Para especificar que **writeList()** lanza estas dos excepciones, se añade la cláusula **throws** a la firma del método de **writeList()**. La clausula **throws** está compuesta por la palabra clave **throws** seguida por una lista separada por comas de todas las excepciones lanzadas por el método. Esta cláusula va después del nombre del método y antes de la llave abierta que define el ámbito del método. Aquí tienes un ejemplo.

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

Recuerda que la excepción ArrayIndexOutOfBoundsException es una excepción en tiempo de ejecución, por eso no tiene porque especificarse en la sentecia **throws** pero puede hacerse si se quiere.

La Sentencias throw

Todos los métodos Java utilizan la sentencia **throw** para lanzar una excepción.
Esta sentencia requiere un sólo argumento, un objeto Throwable. En el sistema Java, los objetos lanzables son ejemplares de la clase Throwable definida en el paquete java.lang. Aquí tienes un ejemplo de la sentencia **throw**.

throw **algunObjetoThrowable**;

Si se intenta lanzar un objeto que no es 'lanzable', el compilador rehusa la compilación del programa y muestra un mensaje de error similar a éste.

testing.java:10: Cannot throw class java.lang.Integer; it must be a subclass
of class java.lang.Throwable.
        throw new Integer(4);

Echemos un vistazo a la sentencia **throw** en su contexto. El siguiente método está tomado de una clase que implementa un objeto pila normal. El método **pop()** saca el elemento superior de la pila y lo devuelve.

```
public Object pop() throws EmptyStackException {
   Object obj;

   if (size == 0)
      throw new EmptyStackException();

   obj = objectAt(size - 1);
   setObjectAt(size - 1, null);
   size--;
   return obj;
}
```

El método **pop()** comprueba si hay algún elemento en la pila. Si la pila está vacía (su tamaño es igual a cero), ejemplariza un nuevo objeto de la clase EmptyStackException y lo lanza. Esta clase está definida en el paquete java.util. En páginas posteriores podrás ver cómo crear tus propias clases de excepciones. Por ahora, todo lo que necesitas recordar es que se pueden lanzar objetos heredados desde la clase Throwable.

La cláusula throws

Habrás observado que la declaración del método **pop()** contiene esta cláusula.

throws EmptyStackException

La cláusula **throws** especifica que el método puede lanzar una excepción EmptyStackException. Como ya sabes, el lenguaje Java requiere que los métodos capturen o especifiquen todas las excepciones chequeadas que puedan ser lanzadas dentro de su ámbito. Se puede hacer esto con la clausula **throws** de la declaración del método.

La Clase Throwable y sus Subclases

Sólo se pueden lanzar objetos que estén derivados de la clase Throwable. Esto incluye descendientes directos (esto es, objetos de la clase Throwable) y descendiente indirectos (objetos derivados de hijos o nietos de la clase Throwable).

Error

Cuando falla un enlace dinámico, y hay algún fallo "hardware" en la máquina virtual, ésta lanza un error. Típicamente los programas Java no capturan los Errores. Pero siempre lanzarán errores.

Exception

La mayoría de los programas lanzan y capturan objetos derivados de la clase Exception.
Una Excepción indica que ha ocurrido un problema pero que el problema no es demasiado serio.
La mayoría de los programas que escribirás lanzarán y capturarán excepciones.
La clase Exception tiene muchos descendiente definidos en los paquetes Java. Estos descendientes indican varios tipos de excepciones que pueden ocurrir. Por ejemplo, IllegalAccessException señala que no se puede encontrar un método particular, y NegativeArraySizeException indica que un programa intenta crear un array con tamaño negativo.
Una subclase de Exception tiene un significado especial en el lenguaje Java: RuntimeException.

Excepciones en Tiempo de Ejecución

La clase RuntimeException representa las excepciones que ocurren dentro de la máquina virtual Java (durante el tiempo de ejecución). Un ejemplo de estas excepciones es NullPointerException, que ocurre cuando un método intenta acceder a un miembro de un objeto a través de una referencia nula. Esta excepción puede ocurrir en cualquier lugar en que un programa intente desreferenciar una referencia a un objeto. Frecuentemente el coste de chequear estas excepciones sobrepasa los beneficios de capturarlas.
Como las excepciones en tiempo de ejecución están omnipresentes e intentar capturar o especificarlas todas en todo momento podría ser un ejercicio infructuoso (y un código

infructuoso, imposible de leer y de mantener), el compilador permite que estas excepciones no se capturen ni se especifiquen.
Los paquetes Java definen varias clases RuntimeException. Se pueden capturar estas excepciones al igual que las otras. Sin embargo, no se require que un método especifique que lanza excepciones en tiempo de ejecución. Además puedes crear sus propias subclases de untimeException.

Crear Clases de Excepciones

Cuando diseñes un paquete de clases java que colabore para proporcionar alguna función útil a sus usuarios, deberás trabajar duro para asegurarte de que las clases interactúan correctamente y que sus interfaces son fáciles de entender y utilizar. Deberías estar mucho tiempo pensando sobre ello y diseñar las excepciones que esas clases pueden lanzar.
Supón que estás escribiendo una clase con una lista enlazada que estás pensando en distribuir como freeware. Entre otros métodos la clase debería soportar estos.

objectAt(int n)

Devuelve el objeto en la posición **n** de la lista.

firstObject()

Devuelve el primer objeto de la lista.

indexOf(Object o)

Busca el Objeto especificado en la lista y devuelve su posición en ella.

¿Qué puede ir mal?

Como muchos programadores utilizarán tu clase de lista enlazada, puedes estar seguro de que muchos de ellos la utilizarán mal o abusarán de los métodos de la clase. También, alguna llamada legítima a los métodos de la clase podría dar algún resultado indefinido. No importa, con respecto a los errores, querrás que tu clase sea lo más robusta posible, para hacer algo razonable con los errores, y comunicar los errores al programa llamador. Sin embargo, no puedes anticipar como quiere cada usuario de sus clases enlazadas que se comporten sus objetos ante la adversidad. Por eso, lo mejor que puedes hacer cuando ocurre un error es lanzar una excepción.
Cada uno de los métodos soportados por la lista enlazada podría lanzar una excepción bajo ciertas condiciones, y cada uno podría lanzar un tipo diferente de excepción. Por ejemplo.

objectAt()

Lanzará una excepción si se pasa un entero al método que sea menor que 0 o mayor que el número de objetos que hay realmente en la lista.

firstObject()

Lanzará una excepción si la lista no contiene objetos.

indexOf()

Lanzará una excepción si el objeto pasado al método no está en la lista.

Pero ¿qué tipo de excepción debería lanzar cada método? ¿Debería ser una excepción proporcionada por el entorno de desarrollo de Java? O ¿Deberían ser excepciones propias?

Elegir el Tipo de Excepción Lanzada

Tratándose de la elección del tipo de excepción a lanzar, tienes dos opciones.

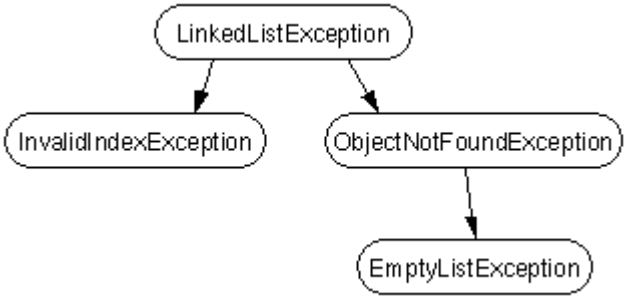
1. Utilizar una escrita por otra persona. Por ejemplo, el entorno de desarrollo de Java proporciona muchas clases de excepciones que podrías utilizar.
2. Escribirlas tu mismo.

Necesitarás escribir tus propias clases de excepciones si respondes "Si" a alguna de las siguientes preguntas. Si no es así, probablemente podrás utilizar alguna excepción ya escrita.

- ¿Necesitas un tipo de excepción que no está representada por lo existentes en el entorno de desarrollo de Java?
- ¿Ayudaría a sus usuarios si pudieran diferenciar sus excepciones de las otras lanzadas por clases escritas por por otros vendedores?
- ¿Lanza el código más una excepción relacionada?
- Si utilizas excepciones de otros, ¿Podrán sus usuarios tener acceso a estas excepciones? Una pregunta similar es "¿Debería tu paquete ser independiente y auto-contenedor?"

La clase de lista enlazada puede lanzar varias excepciones, y sería conveniente poder capturar todas las excepciones lanzadas por la lista enlaza con un manejador. Si planeas distribuir la lista enlazada en un paquete, todo el código relacionado debe empaquetarse junto. Así para la lista enlazada, deberías crear tu propio árbol de clases de excepciones.

El siguiente diagrama ilustra una posibilidad del árbol de clases para su lista enlazada.



LinkedListException es la clase padre de todas las posibles excepciones que pueden ser lanzadas por la clase de la lista enlazada, Los usuarios de esta clase pueden escribir un sólo manejador de excepciones para manejarlas todas con una sentencia **catch** como esta.

```
catch (LinkedListException) {
   . . .
}
```

O, podrñia escribir manejadores más especializados para cada una de las subclases de LinkedListException.

Elegir una Superclase

El diagrama anterior no indica la superclase de la clase LinkedListException. Como ya sabes, las excepciones de Java deben ser ejemplares de la clase Throwable o de sus subclases.
Por eso podría tentarte hacer LinkedListException como una subclase de la clase Throwable.
Sin embargo, el paquete java.lang proporciona dos clases Throwable que dividen los tipos de problemas que pueden ocurrir en un programa java: Errores y Excepción. La mayoría de los applets y de las aplicaciones que escribes lanzan objetos que son Excepciones. (Los errores están reservados para problemas más serios que pueden ocurrir en el sistema.)
Teorícamente, cualquier subclase de Exception podría ser utilizada como padre de la clase LinkedListException. Sin embargo, un rápido examen de esas clases muestra que o son demasiado especializadas o no están relacionadas con LinkedListException para ser apropiadas.
Asi que el padre de la clase LinkedListException debería ser Exception.
Como las excepciones en tiempo de ejecución no tienen por qué ser especificadas en la cláusula **throws** de un método, muchos desarrolladores de paquetes se preguntan.

Convenciones de Nombres

Es una buena práctica añadir la palabra "Exception" al final del nombre de todas las clases heredadas (directa o indirectamente) de la clase Exception. De forma similar, los nombres de las clases que se hereden desde la clase Error deberían terminar con la palabra "Error".

Excepciones en Tiempo de Ejecución - La Controversia

Como el lenguaje Java no requiere que los métodos capturen o especifiquen las excepciones en tiempo de ejecución, es una tentación para los programadores escribir código que lance sólo excepciones de tiempo de ejecución o hacer que todas sus subclases de excepciones hereden de la clase RuntimeException. Estos atajos de programación permiten a los programadores escribir código Java sin preocuparse por los consiguientes.

InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught,
or it must be declared in throws clause of this method.

fis = new FileInputStream(filename);

Errores del compilador y sin preocuparse por especificar o capturar ninguna excepción.

Mientras esta forma parece conveniente para los programadores, esquiva los requerimientos de Java de capturar o especificar y pueden causar problemas a los programadores que utilicen tus clases.

¿Por qué decidieron los diseñadores de Java forzar a un método a especificar todas las excepciones chequeadas no capturadas que pueden ser lanzadas dentro de su ámbito?

Como cualquier excepción que pueda ser lanzada por un método es realmente una parte del interfase de programación público del método: los llamadores de un método deben conocer las excepciones que el método puede lanzar para poder decidir concienzuda e inteligentemente qué hacer con estas excepciones. Las excepciones que un método puede lanzar son como una parte del interfase de programación del método como sus parámetros y devuelven un valor.

La siguiente pregunta podría ser: " Bien ¿Si es bueno documentar el API de un método incluyendo las excepciones que pueda lanzar, por qué no especificar también las excepciones de tiempo de ejecución?".

Las excepciones de tiempo de ejecución representan problemas que son detectados por el sistema de ejecución. Esto incluye excepciones aritméticas (como la división por cero), excepciones de punteros (como intentar acceder a un objeto con un referencia nula), y las excepciones de indexación (como intentar acceder a un elemento de un array a través de un índice demasiado grande o demasiado pequeño).

Las excepciones de tiempo de ejecución pueden ocurrir en cualquier lugar del programa y en un programa típico pueden ser muy numerosas. Típicamente, el coste del chequeo de las excepciones de tiempo de ejecución excede de los beneficios de capturarlas o especificarlas.

Así el compilador no requiere que se capturen o especifiquen las excepciones de tiempo de ejecución, pero se puede hacer.

Las excepciones chequeadas representan información útil sobre la operación legalmente especificada sobre la que el llamador podría no tener control y el llamador necesita estar informado sobre ella -- por ejemplo, el sistema de ficheros está lleno, o el ordenador remoto ha cerrado la conexión, o los permisos de acceso no permiten esta acción.
¿Qué se consigue si se lanza una excepción RuntimeException o se crea una subclase de RuntimeException sólo porque no se quiere especificarla? Simplemente, se obtiene la posibilidad de lanzar una excepción sin especificar lo que se está haciendo. En otras palabras, es una forma de evitar la documentación de las excepciones que puede lanzar un método.
¿Cuándo es bueno esto? Bien, ¿cuándo es bueno evitar la documentación sobre el comportamiento de los métodos? La respuesta es "NUNCA".

Reglas del Pulgar:

- Se puede detectar y lanzar una excepción de tiempo de ejecución cuando se encuentra un error en la máquina virtual, sin embargo, es más sencillo dejar que la máquina virtual lo detecte y lo lance. Normalmente, los métodos que escribas lanzarán excepciones del tipo Exception, no del tipo RuntimeException.
- De forma similar, puedes crear una subclase de RuntimeException cuando estas creando un error en la máquina virtual (que probablemente no lo hará), De otro modo utilizará la clase Exception.
- No lances una excepción en tiempo de ejecución o crees una subclase de RuntimeException simplemente porque no quieres preocuparte de especificarla.

OBSERVACIONES:
Para manejar los errores leer la documentación en línea de la API usada.

RESUMIENDO: MANEJO DE EXCEPCIONES:

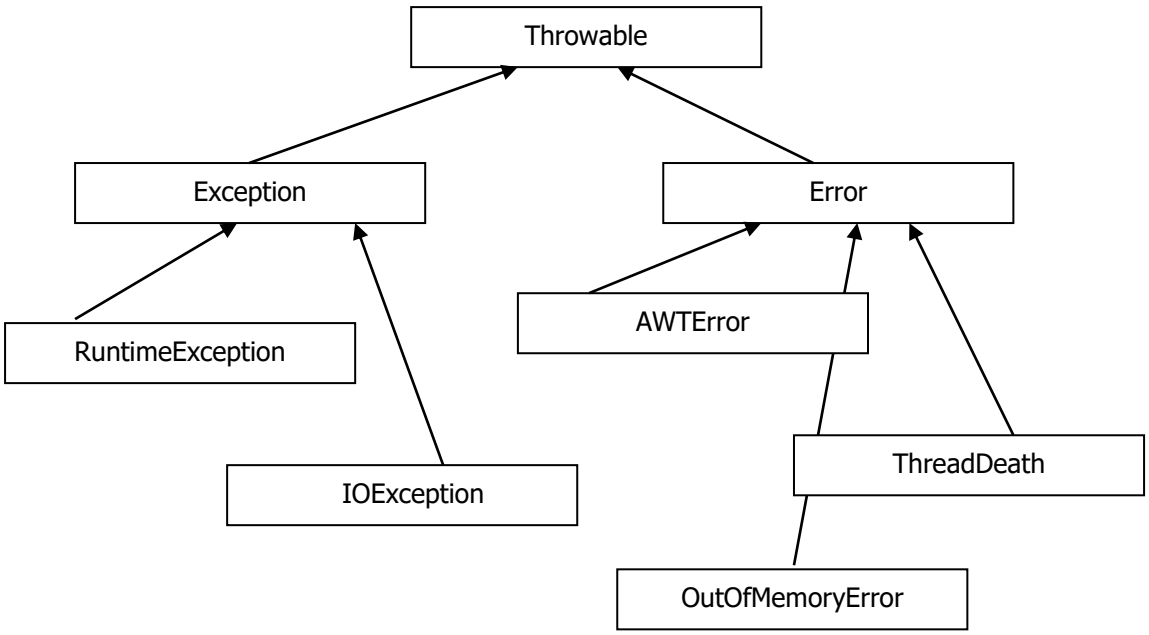
EL LENGUAJE Java, como se sabe, es independiente del Sistema Operativo, esto me implica, perder el control sobre los periféricos, los cuales son fuente, generalmente, de la mayor cantidad de errores.
Por esto, se dotó a este Lenguaje de una estructura robusta para el manejo de errores. La misma se basa en la creación de un objeto denominado "EXCEPCIÓN".
Este objeto contiene el informe sobre el acontecimiento producido y transmite esta información al método que lo llamó o al usuario mediante un mensaje.
En algunos métodos el error se captura y en otros se expulsa.

La codificación se basa en la siguiente estructura:

```
try     {
        [BLOQUE DE EJECUCIÓN NORMAL DE SENTENCIAS]
        }
catch (ClasedeExcepción1 e)     {
                                [BLOQUE DE EJECUCIÓN EXCEPCIÓN 1]
                                }
catch (ClasedeExcepción2 e)     {
                                [BLOQUE DE EJECUCIÓN EXCEPCIÓN 2]
                                }

catch (ClasedeExcepción3 e)     {
                                [BLOQUE DE EJECUCIÓN EXCEPCIÓN 3]
                                }
catch (Excepción e)     {
                                [BLOQUE DE EJECUCIÓN EXCEPCIONES NO CAPTURADAS]
                                }
finally {
        [BLOQUE DE SENTENCIAS QUE SE EJECUTAN SIEMPRE]
        }
```

La jerarquía de excepciones en Java



Este esquema es una porción de la jerarquía de herencia para la clase Throwable (subclase de Object).
Las clase Exception tiene dos subclases que están incluidas en java.lang y que contiene la mayoría de las situaciones de error que se producen. En cambio la clase Error contiene errores que se producen generalmente en tiempo de ejecución y que no deben ser capturadas por la aplicación.

Java clasifica a las excepciones en verificadas y no verificadas. Las primeras son aquellas para las cuales Java implementa un requerimiento para atrapar o declarar.
Si se intenta lanzar una excepción verificada que no esté listada en throw se produce un error de ejecución.
Las excepciones no verificadas se pueden evitar programando correctamente. (Ej. Caso ArithmeticException o NumberFormatExcpetion, que es subclase de RuntimeException, no necesita colocarse en la cláusula throw.

Ejercicio 73:
Realizar un clase que me permita ingresar dos valores enteros y me contemple los distintos errores.

```
// PruebaDivisionEntreCero.java
// Un ejemplo de manejo de excepciones que comprueba la división entre cero.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PruebaDivisionEntreCero extends JFrame
   implements ActionListener {

   private JTextField campoEntrada1, campoEntrada2, campoSalida;
   private int numero1, numero2, resultado;

   // configurar GUI
   public PruebaDivisionEntreCero()
   {
      super( "Demostración de las excepciones" );

      // obtener panel de contenido y establecer su esquema
      Container contenedor = getContentPane();
      contenedor.setLayout( new GridLayout( 3, 2 ) );

      // establecer etiqueta y campoEntrada1
      contenedor.add(
         new JLabel( "Escriba el numerador ", SwingConstants.RIGHT ) );
      campoEntrada1 = new JTextField();
      contenedor.add( campoEntrada1 );

      // establecer etiqueta y campoEntrada2; registrar componente de escucha
      contenedor.add( new JLabel( "Escriba el denominador y oprima Intro ",
         SwingConstants.RIGHT ) );
      campoEntrada2 = new JTextField();
      contenedor.add( campoEntrada2 );
      campoEntrada2.addActionListener( this );

      // establecer etiqueta y campoSalida
      contenedor.add( new JLabel( "RESULTADO ", SwingConstants.RIGHT ) );
      campoSalida = new JTextField();
      contenedor.add( campoSalida );

      setSize( 475, 100 );
      setVisible( true );
```


```
   } // fin del constructor de PruebaDivisionEntreCero

   // procesar eventos de GUI
   public void actionPerformed( ActionEvent evento )
   {
      campoSalida.setText( "" );   // borrar campoSalida

      // leer dos números y calcular el cociente
      try {
         numero1 = Integer.parseInt( campoEntrada1.getText() );
         numero2 = Integer.parseInt( campoEntrada2.getText() );

         resultado = cociente( numero1, numero2 );
         campoSalida.setText( String.valueOf( resultado ) );
      }

      // Si se ingresa un número que no representa un entero válido indica la entrada con
formato incorrecto
      catch ( NumberFormatException excepcionFormatoNumero ) {
         JOptionPane.showMessageDialog( this,
            "Debe escribir dos enteros", "Formato de número inválido",
            JOptionPane.ERROR_MESSAGE );
      }

      // procesar los intentos de dividir entre cero
      catch ( ArithmeticException excepcionAritmetica ) {
         JOptionPane.showMessageDialog( this,
            excepcionAritmetica.toString(), "Excepción aritmética",
            JOptionPane.ERROR_MESSAGE );
// Línea que se puede usar
//  JOptionPane.showMessageDialog( this,
            "Se produce error al dividir por cero", "Excepción aritmética",
            JOptionPane.ERROR_MESSAGE );
      }

   } // fin del método actionPerformed

   // demuestra cómo lanzar una excepción cuando ocurre una división entre cero
   public int cociente( int numerador, int denominador )
      throws ArithmeticException
   {
      return numerador / denominador;
   }

   public static void main( String args[] )
   {
      PruebaDivisionEntreCero aplicacion = new PruebaDivisionEntreCero();
      aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   }

} // fin de la clase PruebaDivisionEntreCero
```

La cláusula Finally

Esta cláusula generalmente es opcional y se la usa principalmente para liberar recursos (conexiones de archives, bases de datos, conexiones de red, etc.) No así de memoria ya que Java a diferencia de C y C++ junta toda la basura en forma automática.

Ejercicio 74:
Proponer resoluciones en ejercicios anteriores en los cuales se puedan manejar errores presentados en las mismas