Ciclo de Desarrollo de Aplicaciones Multiplataforma

Grado Superior

U.T.6: EXCEPCIONES

Módulo de Programación

Departamento de Informática

I.E.S. Monte Naranco

6.1. EXCEPCIONES PREDEFINIDAS

6.1.1. CONCEPTO DE EXCEPCIÓN

Un programa escrito en java puede tener errores, como el nombre de una variable incorrecto, ausencia de un punto y coma, etc. Este tipo de errores nos los indica nuestro compilador. Pero existen otros no sintácticos que se producen durante la ejecución debidos a la lógica del programa (leer un fichero que no existe, acceder a un valor N de un array que contiene menos de N elementos...). A estos errores los conocemos como excepciones.

Podemos definir por tanto una excepción como una situación anómala que puede tener lugar cuando ejecutamos un programa. La forma en la que el programador trate la misma para controlar esa situación y especificar cómo ha de responder el programa, es lo que se conoce como manejo o gestión de la excepción.

Hay que tener claro que el manejo de excepciones no sirve para "corregir" estos errores de ejecución, así si un programa no encuentra un archivo determinado por medio de las excepciones no vamos a conseguir que el archivo "aparezca".

Ejemplo: desbordamiento en el acceso a elementos de un array.

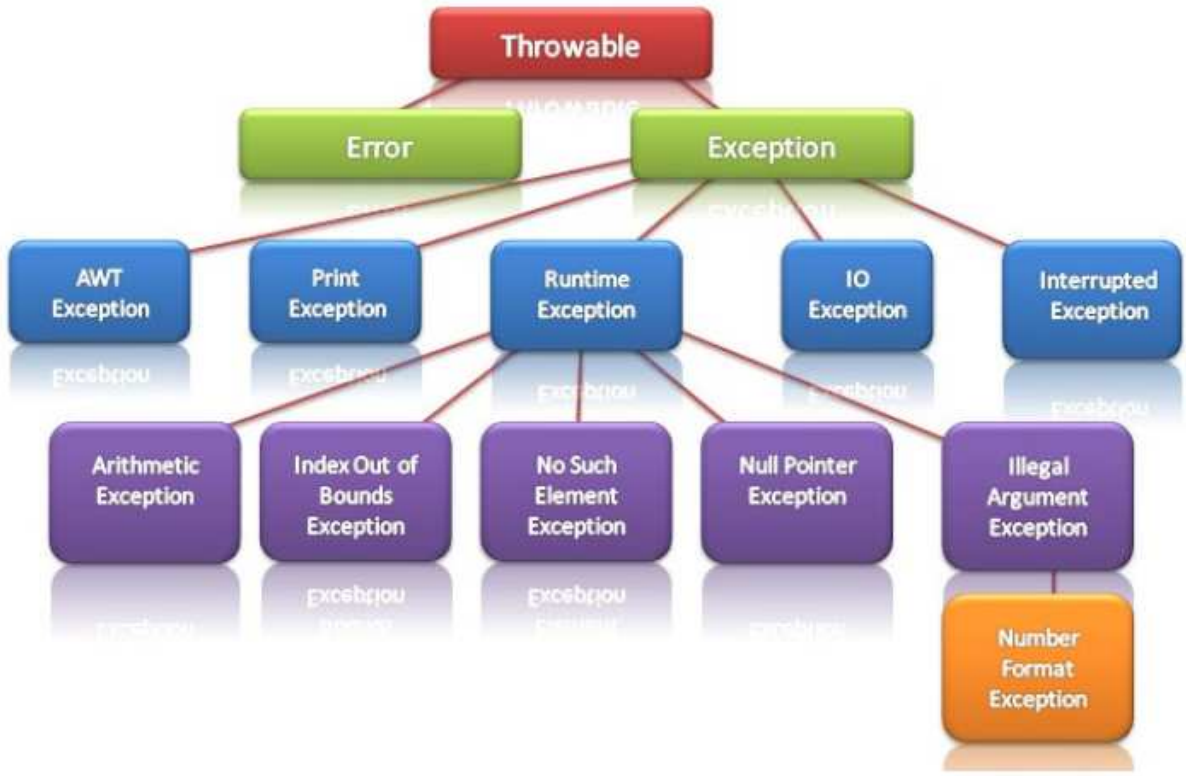
```
public class Desbordamiento {
  public static void main(String[ ] args)  {
        String mensajes[ ] = {"Primero", "Segundo", "Tercero"};
        for (int i=0; i<=3; i++)
              System.out.println(mensajes[i]);
  }
}
```

Estos errores hacen que no se ejecute nuestro programa. Con el tratamiento de excepciones debemos tratar de conseguir que la parte del programa que no genere el error se ejecute. Debemos tratar de controlar los errores que se producen en una parte del código para que el resto del programa continúe su ejecución.

En definitiva, las excepciones son objetos pertenecientes a una clase determinada que se crean sin que los veamos cuando se produce una situación extraordinaria en la ejecución de un programa. Estos objetos almacenan información acerca del tipo de situación anormal que se ha producido y el lugar donde ha ocurrido.

6.1.2. TIPOS DE EXCEPCIONES

El gráfico muestra la jerarquía de herencia de las excepciones en java, el paquete de todas ellas es el java.lang.



Como vemos, la superclase que engloba a todas las excepciones es la clase Throwable (sólo las instancias de esta clase y de sus subclases pueden ser utilizadas como excepciones).

La clase Throwable tiene 2 clases derivadas: Error y Exception.

CLASE ERROR

Sirve de superclase para una serie de clases derivadas ya definidas que nos informan de situaciones anormales relacionadas con errores de muy difícil recuperación, producidos por la máquina virtual de java no por nuestro programa y que no pueden ser controlados en el mismo (escasez de memoria, desbordamiento de buffer, etc). Este tipo de excepciones aunque pueden gestionarse no es recomendable hacerlo.

CLASE EXCEPTION

Sirve como superclase para un amplio número de clases derivadas que representan errores o excepciones provocados en el programa. Estas excepciones podemos dividirlas en:

1. Comprobadas o checked: son aquellas que no son culpa del programador, es decir, son situaciones ajenas al código del programa pero se espera que se gestionen, bien capturándolas o relanzándolas. Representan un error del cual técnicamente podemos recuperarnos, como una operación de lectura/escritura sobre un fichero que no existe, crear un programa que deba buscar una imagen en una carpeta y que no se encuentre, etc. Entre las subclases que definen este tipo de excepciones son todas las derivadas de Exception excepto RuntimeException.

2. No comprobadas o unchecked: proceden de errores de programación, como leer el contenido de la posición N de un array cuando tiene menos de N posiciones. Este tipo de excepciones no obligan al programador a gestionarlas, porque son errores que se pueden solventar con solo realizar mejor el programa para que no lance una excepción. Entre las subclases que definen este tipo de excepciones destacamos RuntimeException y todas sus derivadas.

Constructores de la clase Exception

Vamos a destacar 2:

1. Exception(): crea la excepción con un mensaje por defecto con información acerca del tipo de situación anormal que se ha producido.

2. Exception (String message): el parámetro de tipo String nos permite crear la excepción con un determinado mensaje. Este mensaje o información almacenada en el objeto Exception puede ser accedido por medio de los siguientes métodos que destacamos a continuación.

Métodos de la clase Exception

La clase Exception en sí no cuenta con métodos propios, pero sí con algunos heredados tanto de la superclase Throwable como de Object entre los que destacamos:

1) toString(): pertenece a la clase Object aunque es redefinido en la clase Throwable. Devuelve un String con la descripción de la excepción.

Ejm: `java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 4`

2) getClass(): heredado de la clase Object. Devuelve la clase (Class) del objeto sobre el que se invoca.

Ejm: `class java.lang.ArrayIndexOutOfBoundsException`

3) getMessage(): pertenece a la clase Throwable. Devuelve un String con el valor que produce la excepción.

Ejm: `Index 7 out of bounds for length 4`

4) printStackTrace(): pertenece a la clase Throwable. No devuelve nada (void). Imprime por consola el objeto Throwable desde el que se invoca, así como la traza de llamadas a métodos desde el que se ha producido la excepción. Resulta de utilidad sobre todo para depurar programas, ya que nos ayuda a saber el punto exacto de nuestro código en el que surge la excepción y el método que la ha producido. Es el tipo de excepción que sale por defecto.

```
java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds for length 4
	at tema6_excepciones.Uno.<init>(Uno.java:14)
	at tema6_excepciones.Uno.main(Uno.java:26)
```

Subclases de Exception más destacadas

Principalmente dos:

1) IOException: define excepciones que se producen cuando se encuentra un problema en cualquier operación de lectura o escritura en un medio externo, como la lectura o escritura en un fichero.

Entre sus subclases la más común es:

-**FileNotFoundException:** define una excepción que se produce cuando se intenta abrir algún fichero que no existe o no ha sido encontrado.

2) RuntimeException: define el conjunto de las excepciones que tienen la peculiaridad de que el usuario no tiene por qué capturarlas (todas las demás excepciones deberán ser capturadas o gestionadas de algún modo por el usuario), no es necesario bloques try-catch ni lanzarlas.

El costo de gestionar este tipo de excepciones en tiempo de ejecución excede el beneficio de capturarlas al consumir muchos recursos, por tanto depende del programador si quiere hacerlo o no.

Dentro de la clase RunTimeException podemos encontrar un numeroso grupo de subclases que definen diferentes tipos de excepciones, como:

-**ClassCastException:** tiene lugar cuando intentamos hacer un “cast” de un objeto a una clase de la que no es subclase.

Ejm:

```
Object x = new Integer(0);
System.out.println((String)x);
```

Al no ser la clase Integer una subclase de String , no podemos hacer un “cast” de forma directa.

-**IndexOutOfBoundsException:** tiene lugar cuando intentamos acceder a un índice de un array mayor que el número de elementos de dicha estructura.

Ejm:

```
int array_enteros [ ] = new int [50];
System.out.println (array_enteros [67]);
```

-**NegativeArraySizeException:** tiene lugar cuando intentamos crear un array con longitud negativa.

Ejm:

```
int array_enteros [ ] = new int [-50];
```

- **ArrayStoreException:** tiene lugar cuando se intenta almacenar un valor de tipo erróneo en una matriz de objetos.

- **IllegalArgumentException:** tiene lugar cuando se le ha pasado un argumento ilegal o inapropiado a un método.

- **NumberFormatException:** tiene lugar cuando intentamos convertir un dato de tipo String a algún tipo de dato numérico, pero el dato de tipo String no tiene el formato adecuado.

Se usa en los métodos parseDouble(): double, parseInt(): int, parseFloat(): float, ...

- **NullPointerException:** tiene lugar cuando un objeto aún no se ha creado, su valor todavía es null. Si se intenta acceder a algún miembro suyo o invocar a algún método se produce una excepción de este tipo.

Ejemplo:

```
Object ds = null;
ds.toString();
```

El objeto ha sido inicializado con el valor null y al intentar acceder al método toString(), obtenemos la excepción “NullPointerException”.

Similar excepción se obtiene también en la siguiente situación:

```
Object [ ] array_ob = new Object [25];
array_ob [1]. toString();
```

Declaramos y creamos un array de objetos de la clase Object pero no lo inicializamos, esto quiere decir que cada componente del array contiene el valor null. Al intentar acceder mediante el método toString() a una determinada posición, obtenemos una excepción de tipo NullPointerException.

- Arithmetic Exception: tiene lugar cuando ocurre un operación aritmética errónea, por ejemplo división entre 0. Con los valores de tipo real no se produce esta excepción.

6.1.3. MANEJO DE EXCEPCIONES

Hay dos formas de tratar los errores en java: capturarlos y gestionarlos (try-catch-finally) o lanzarlos. Las excepciones que derivan de la clase RuntimeExcepcion es opcional tratarlas, depende del programador.

a) CAPTURA Y GESTIÓN DE EXCEPCIONES (TRY....CATCH....FINALLY)

Cuando se prevé que en una situación extraordinaria una instrucción o un bloque de instrucciones pudiera llegar a provocar algún tipo de error, se tendría que capturar ese posible error con el fin de poder controlarlo. El objetivo es desviar con ello la ejecución del programa y llevar a cabo las operaciones que sean necesarias ante tal situación.

Esta captura se hace insertando las instrucciones que pueden provocar algún error en un bloque try (intentar). Si ocurre un error dentro del bloque se lanza una excepción que se puede capturar por medio del bloque catch. Es dentro de este tipo de bloques donde se hace el manejo de las excepciones.

Las excepciones que derivan de la clase RuntimeException no es obligatorio capturarlas, depende del programador. Las que derivan de la clase Exception que no pertenecen a la clase RuntimeException nos obligan a capturarlas o lanzarlas.

Sintaxis y funcionamiento con una sola excepción

```
try {
// Instrucciones que pueden causar cierto tipo de excepciones
}
catch (TipoDeExcepción identificador) {
//Instrucciones que se deben ejecutar si ocurre la excepción de tipo: TipoDeExcepción
}
```

- try: se le denomina bloque de "intento". Aquí se escribe el código del programa susceptible de causar cierto tipo de excepciones (siempre lleva llaves). Siempre tiene que ir seguido al menos de una cláusula **catch** o una cláusula **finally.**

Puede haber más de una sentencia que genere excepciones, por tanto, deberíamos proporcionar un bloque try si es necesario para cada una de ellas.

Los bloques try pueden estar anidados, es decir, un try puede estar dentro de otro try. Si el bloque try interno no tiene un catch cuyo parámetro cuadre con la excepción producida, se va retrocediendo a bloques try jerárquicamente superiores hasta encontrar uno que lo trate, o en último extremo llegar al gestor por defecto que interrumpirá la ejecución.

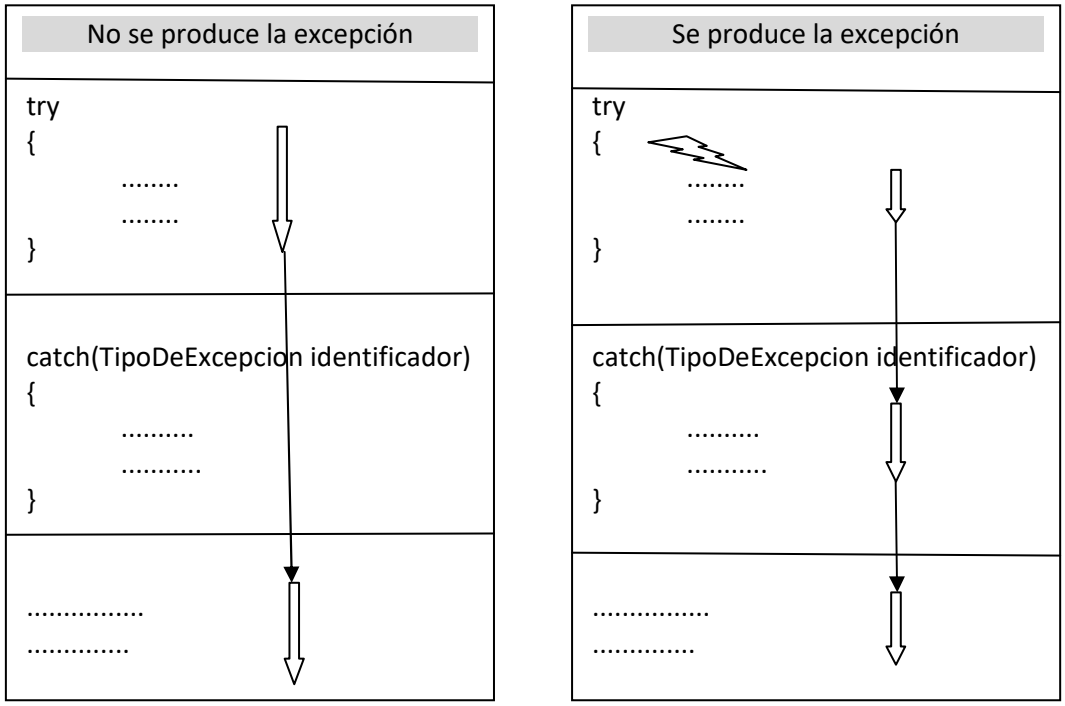
Ejemplo:

```
try
{
  ------
  if (-----) {
    --------
   try
  {
    ------
  }
 catch(----)// Si este catch no captura la excepción correspondiente a su try, vamos al grupo
                catch del try anterior a él para ver si puede capturar la excepción.
 {
    ---------
  }
}//if
}
catch(----) {-----}
catch(----) {-----}
```

- **catch (TipoDeExcepción identificador)**: Es el código que se ejecuta cuando se produce la excepción que cuadre con su parámetro (va siempre entre llaves). Debe colocarse inmediatamente después del bloque try no pudiendo haber ningún tipo de instrucciones entre un bloque try y un catch, ni entre bloques catch.
En estos bloques debemos asegurarnos no colocar código que genere excepciones.

Funcionamiento:

Se ejecutan las instrucciones del bloque **try.** Si se produce una excepción del tipo indicado (TipoDeExcepción) o descendiente de este, Java omite la ejecución del resto del código del bloque try y ejecuta el código situado en el bloque **catch**, y posteriormente las instrucciones siguientes al bloque catch. Si no se produce la excepción no se ejecutan las instrucciones del bloque catch.



Ejemplo de error de ejecución: desbordamiento en el acceso a elementos de un array.

```
public class Desbordamiento {
        public static void main(String [ ] args)   {
                String [ ] mensajes = {"Primero", "Segundo", "Tercero"};
                 try {
                        for (int i = 0; i <= 3; i++)
                                System.out.println(mensajes[i]);
                }
                catch (ArrayIndexOutOfBoundsException e) {
                        System.out.println ("Estas accediendo a una posición fuera del array");
                }
        }
}
```

Para visualizar la excepción capturada

```
…
catch (Exception e) {
   System.out.println("La excepcion es: "+ e.getMessage());
   // También podríamos poner: System.out.println(e.toString()); o simplemente System.out.println(e);
   System.err.println(e.getMessage()); //visualiza en rojo el valor que hace saltar la excepción
}
```

Ejemplo: Detectar division por 0

```
public class ExcepcionDividirPorCero {
    public static void main(String[ ] args) {
        try{
            int a = args.length;
            System.out.println(“a = “ + a);
            int b= 3/a;
         }
        catch (ArithmeticException e) {
            System.out.println (“No dividas entre 0 (“ + e +”)");
        }
        System.out.println (“La ejecución del programa sigue …..");
    }
}
```

Ejemplo de excepción tipo checked o comprobada:

```
import java.io.*;
public class Principal {
        public static void main (String[ ] args) {
                FileWriter fichero;
                try {
                        //Las siguientes líneas pueden lanzar una excepción de tipo IOException
                        fichero=new FileWriter ("ruta");
                        fichero.Write ("Esto se graba en el fichero");
                }
                catch (IOException e) {
                        //Capturamos cualquier excepción IOException que se lance
                        e.printStackTrace();
                }
         }
}
```

Sintaxis y funcionamiento con más de una excepción

El código contenido en un bloque try puede provocar más de una excepción, por lo que java permite asociar varios bloques catch a un mismo bloque try. Cada **catch** tiene entre paréntesis la declaración de una excepción (nombre de una clase derivada de Exception o la propia Exception seguida del nombre de una variable), es decir, cada catch puede capturar un tipo de excepción diferente.

```
try {
// instrucciones que pueden causar cierto tipo de excepciones
}
catch (TipoExcepción1 identificador) {
//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepción1
}
catch (TipoExcepción2 identificador) {
//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepción2
}
.....
catch (TipoExcepciónN identificador) {
//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepciónN
}
```

Se deben colocar primero los catch más específicos (capturan errores concretos) y luego los más generales (agrupan un conjunto de errores), ya que de no ser así nos encontramos con el siguiente error de compilación: código no va a ser nunca alcanzado.

Si en el primer catch ponemos:

```
catch (Exception e) {
}
```

La clase Exception incluye todos los tipos de errores controlables, por tanto, cuando se produzca un error se ejecutará este catch y ya no se ejecutarán los otros que haya por debajo.

Funcionamiento:

Cuando ocurre (se levanta) una excepción en las instrucciones del bloque try se interrumpe la ejecución de dicho bloque y se inspeccionan en orden los bloques catch hasta que se encuentra uno que coincide con el tipo de excepción que se ha producido, o con un tipo que sea superclase de la excepción que se ha producido que será el que se ejecute.

Después de ejecutar el catch correspondiente no se continúa con la siguiente sentencia en el try, sino que se continúa la ejecución en las sentencias posteriores a los bloques catch.

De esta manera sería correcto implementar la secuencia:

```
try {
// instrucciones que pueden causar cierto tipo de excepciones
}
catch (NullPointerException e) {
//instrucciones que se deben ejecutar si ocurre la excepción de tipo NullPointerException
}
catch (RuntimeException e) {
/*instrucciones que se deben ejecutar si ocurre la excepción de tipo RuntimeException o de una clase derivada de ella distinta a NullPointerException*/
}
catch (Excepción e) {
/*instrucciones que se deben ejecutar si ocurre la excepción de tipo Excepción o de una clase derivada distinta de RuntimeException*/
}
```

Resulta habitual utilizar bloques catch correspondientes a excepciones con un mismo nivel de herencia, por ejemplo, con subclases de RuntimeException.

Bloque finally

Es un bloque opcional, sin embargo, un bloque try necesita al menos una sentencia catch o finally.

Cuando se produce una excepción dentro de un bloque try las instrucciones siguientes a la que causa la excepción no se ejecutan. Esto puede dar lugar a que dejemos el sistema en un estado no deseado (ficheros abiertos, recursos bloqueados, etc). Para evitar este tipo de situaciones java proporciona la posibilidad de incluir un bloque cuyas instrucciones siempre se ejecuten después de un try, haya o no excepción, es el bloque finally.

Si en el bloque try existe una sentencia return las instrucciones del bloque finally también se ejecutan antes que el return.

Si hay varios bloques try/catch, finally se ejecutará después de un bloque try/catch y antes del siguiente bloque try/catch.

El finally siempre tiene que estar al final de todos los catch, e incluso si se lanza una excepción y no hay ninguna sentencia catch que la capture, el bloque finally se ejecutará.

```
try {
  .........
}
finally {
    ........
}
```

Sintaxis:

try {

// instrucciones que pueden causar cierto tipo de excepciones

}

catch (TipoExcepción1 identificador){

//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepción1

}

catch (TipoExcepción2 identificador){

//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepción2

}

.....

catch (TipoExcepciónN identificador) {

//instrucciones que se deben ejecutar si ocurre la excepción de tipo TipoExcepciónN

}

finally{

// instrucciones que se ejecutarán siempre

}

b) LANZAR EXCEPCIONES

Se pueden lanzar excepciones en dos situaciones distintas:

1. Cuando se produce una excepción comprobada dentro de un método (las que derivan de la clase Exception que no pertenecen a la clase RuntimeException y que java obliga a controlar), por ejemplo IOException, si no se quiere controlar dicha excepción dentro del método capturándola (try-catch), bien porque no se tiene previsto tratarla o porque el método es tan general que no puede establecer un tratamiento que se ajuste a todos los métodos que lo invocan, se puede lanzar. Cuando se lanza no se sigue ejecutando el método y la recibe el método que invocó al que lanzó dicho error. Este método puede capturar la excepción o puede volver a lanzarla. Así hasta que se capture o hasta que la lance el main al sistema operativo.

Para que un método pueda lanzar una excepción debemos añadir en su encabezado la cláusula throws, y tras ella indicar el tipo de excepción que puede provocar el código del método. Por tanto el método tendrá la siguiente cabecera:

tipo_devuelto nombre_método ([parámetros]) **throws** tipo_excepción_que se puede producir

Ejm:

public String leer (FileImputStream archivo) **throws** IOException { ……..

Un método puede lanzar más de una excepción, para ello la cabecera de dicho método contendrá un listado de las excepciones que se pueden lanzar separadas por comas.

tipo_devuelto nombre_método ([parámetros]) **throws** tipo_excepción1, tipo_excepción2.....

Ejm:

public Image cargar (String s) **throws** EOFException, MalfomaedURLException { .....

Ejm:

```
import java.io.*;
public class Principal {
        public static void main (String[ ] args) throws IOException {
                FileWriter fichero=new FileWriter ("ruta");
                fichero.Write ("Esto se graba en el fichero");
        }
}
```

En este caso como no se captura la excepción y no hay más métodos que la puedan capturar, es la máquina virtual de java quien finaliza la ejecución lanzando un error por consola.

La cláusula throws es necesaria para todas las excepciones excepto para las RuntimeException. Las excepciones que derivan de RuntimeException aunque se permite no hace falta y no se aconseja lanzarlas, ya que en el caso de que se produzca dicha excepción se propagan automáticamente si no se captura dentro del método.

2. En muchas ocasiones después de tratar una excepción a un nivel resulta conveniente que también sea tratada a niveles superiores, es decir, dentro de un método podemos querer controlar un error pero puede suceder que también queramos controlarlo fuera de dicho método (en java puede considerarse una excepción como un segundo tipo de valor que puede devolver un método). Por ejemplo, si estamos leyendo de un fichero origen y escribiendo su contenido en un fichero destino nos podemos encontrar con un error de lectura.

Además de tratar la excepción a ese nivel (cerrar ficheros, etc) sería conveniente propagarla al programa llamante para que informe al usuario y le permita hacer un nuevo intento si lo desea. Para ello es necesario capturar dicho error dentro del propio método (con try-catch) y dentro de dicho catch lanzar dicho error de forma explícita (sin que lo genere el intérprete de java), con el fin de que lo pueda recibir el método que lo llamó.

Para propagar o devolver de forma explícita una excepción, un método debe crear un objeto de la clase Exception (bien utilizando un parámetro de la cláusula catch o creándolo con el operador new) y usar la palabra reservada throw (parecido al return), seguida del objeto excepción.

throw nombre_objeto_tipo_Exception;
throw creamos el objeto de la clase Exception con new;

La instrucción throw provoca que se abandone el flujo de ejecución de dicho método cediendo el control al método que lo llama y pasando por el catch (del método que lo llama) que captura dicha excepción.

Si el método lanza una excepción no perteneciente a la clase RuntimeException en la cabecera de dicho método hay que poner:

tipo_devuelto nombre_método ([parámetros]) **throws** tipo_excepción_que devuelve el throw

Sin embargo si la excepción pertenece a la clase RuntimeException no es necesario poner dicha cabecera.

Ejm:

```
try {
   CopiarFichero(Destino);
}
catch(IOException e) {
        System.out.println ( "Error de lectura. ¿Desea intentarlo de nuevo?");
}

public void CopiarFichero (TipoFichero Destino) throws IOException {
   try  {
        //código
    }
    catch (IOException elem) {
        //cerramos ficheros
        throw elem;
    }
}
```

El método CopiarFichero después de tratar la excepción la lanza de una forma explícita (throw) al método que le llama, que a su vez hace otro tratamiento de la excepción.

Ejm:

```
public class Clase {

    public static void main (String[ ] args) {
        try  {
            metodo2();
        }
        catch (IllegalAccessException e) {
            System.out.println ("Error"+e);
        }
    }

public static void metodo2() throws IllegalAccessException {
    System.out.println ("Estamos en metodo2");
    throw new IllegalAccessException ("error"); /*Indicamos que se lanza una excepción de
    tipo IllegalAccessException desde método2, para que main que es el método que le
    invoca la lance de nuevo o la capture como es el caso.*/
}
```

No hay recetas universales sobre cómo trabajar con excepciones.

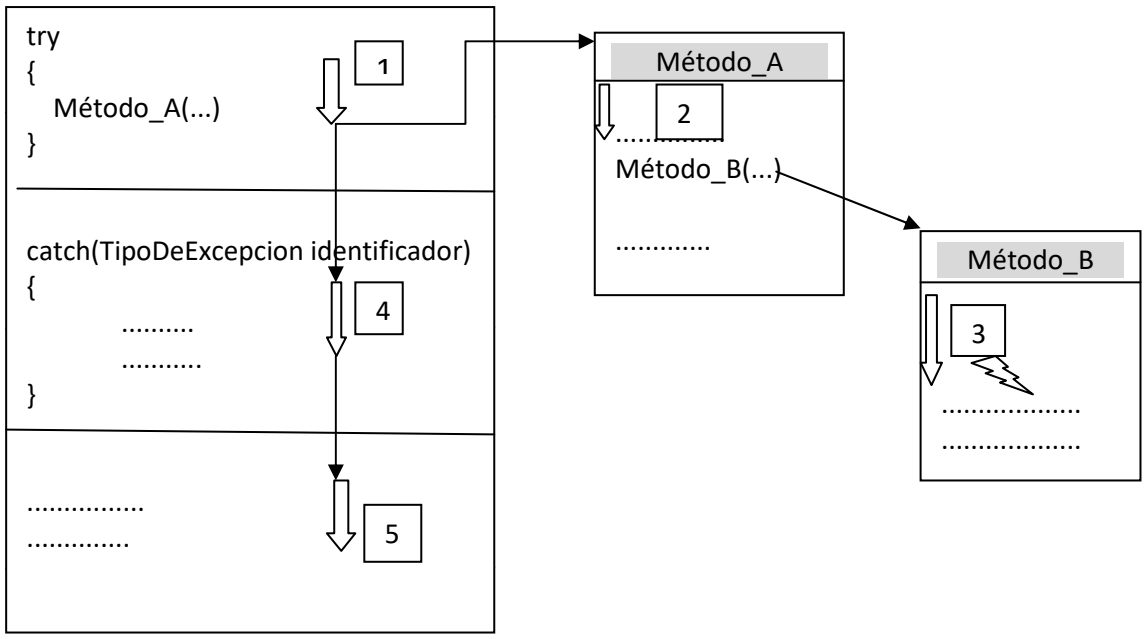
c) PROPAGACIÓN DE EXCEPCIONES

Solo se propagan las excepciones que derivan de la clase RuntimeException. Las excepciones definidas por la clase Exception que no pertenecen a RuntimeException no se propagan y nos obligan a capturarlas o lanzarlas.

Como se propagan las excepciones que derivan de RuntimeException

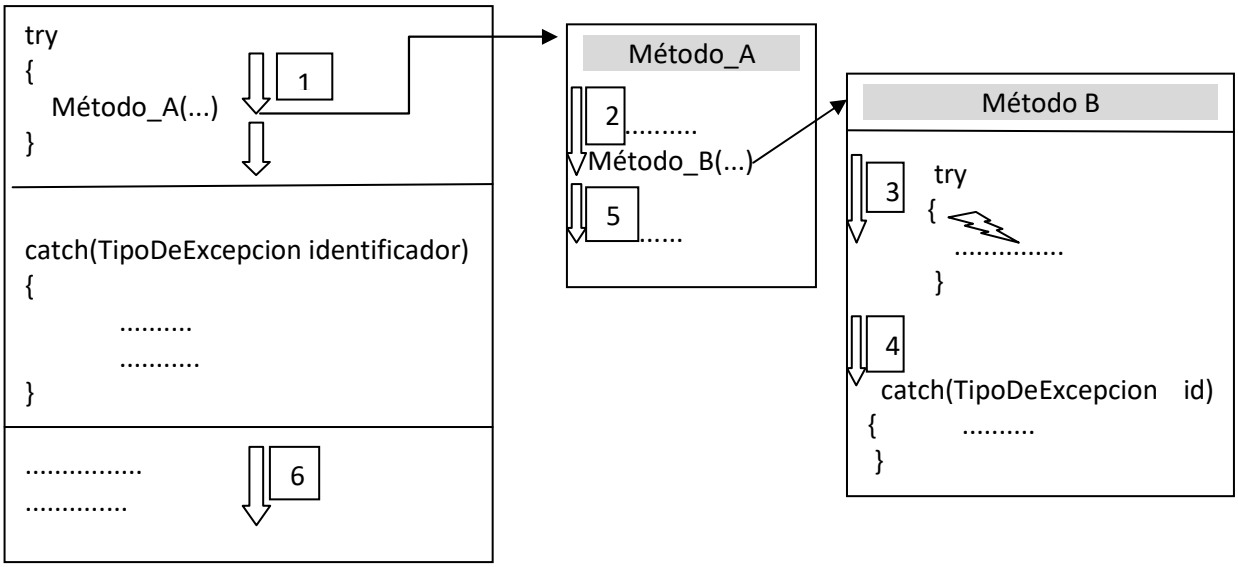
Cuando se produce una excepción dentro de un método se puede capturar dentro de él y tratarla desde ahí mismo. Si no se captura java automáticamente la propaga al método que lo llamó para que dicho método la capture. Si no fuera así se volvería a propagar automáticamente al método que llamó a este último. Así, sucesivamente hasta llegar al main.

El main a su vez lo propagaría al sistema operativo que lo gestionaría lanzando un error por consola.



En el primer bloque try hay una llamada al Método_A que a su vez hace una llamada al Método_B donde se produce una excepción. Como la excepción no es tratada en el propio Método_B se propaga al bloque que le llama (Método_A). La excepción no es tratada en el Método_A por lo que se propaga al bloque try donde sí que es capturada por un catch.

En el caso de que un método capture y trate una excepción el mecanismo de propagación se termina y la ejecución de los métodos llamados y el bloque try del método que llama continúan como si la excepción no se hubiera producido, como vemos en el siguiente gráfico.



6.2. EXCEPCIONES DEFINIDAS POR EL PROGRAMADOR

Aunque las excepciones predefinidas cubren las situaciones de error más habituales con las que nos podemos encontrar, podemos crear nuestras propias clases de excepciones si las que propone la API de java no nos sirven para lo que queremos hacer.

Tenemos que tener en cuenta que el mecanismo de excepciones es muy lento en ejecución comparado con otro tipo de instrucciones como las condicionales, por lo que debemos utilizarlo únicamente en situaciones que realmente son excepcionales, es decir, dependiendo de la frecuencia con la que se nos presente una situación determinada. Si ocurre con poca frecuencia podemos usar excepciones, en caso contrario deberíamos pensar en usar instrucciones condicionales.

Definición de una excepción definida por el programador

Como las excepciones son objetos debemos crear una clase que herede de Exception (tendremos la obligación de capturarla o lanzarla), de este modo habremos creado una clase de objetos que pueden ser lanzados (throw) y por tanto pueden ser utilizados para señalar una situación anómala de cualquier tipo.

En esta clase incluiremos al menos el constructor vacío y otro que contenga un String como argumento (la clase Exception tiene al menos dos constructores como vimos en un apartado anterior). Este String se inicializa automáticamente con el nombre de la clase y el texto que pongamos al hacer uso del segundo constructor.

A la hora de declarar una excepción debemos buscar un nombre que resulte descriptivo de la misma.

Ejm:

```
public class ExPropia extends Exception {
  public ExPropia()  {
     super(); /* También puedo poner: super ("Esta es mi propia excepción"); para añadir este
                 texto al nombre de la clase*/
  }
  public ExPropia (String mensaje) { //Nos permite añadir este mensaje al instanciar la clase
    super(mensaje); // Invoca al constructor de Exception
  }
}
```

A partir de ahora podremos hacer uso de la clase anterior como si fuese una excepción más del sistema.

```
public class UsoExPropia {
    public void metodo() throws ExPropia {  /*Nuestro método es susceptible de lanzar la
                                                excepción */

        Scanner tecla=new Scanner(System.in);
        int num=tecla.nextInt();
        if(num==0)
            throw new ExPropia(); // Se lanza la excepción cuando se lee un cero
    }
 }

public class Principal {
    public static void main(String[ ] args {
        System.out.println ("Vamos a comenzar");
        try {
            UsoExPropia instancia=new UsoExPropia();
            instancia.metodo();
        }
        catch (ExPropia e) { // En la clase de más alto nivel se captura la excepción lanzada
            System.out.println (e);
        }
    }
}
```