

UT 5: Ficheros

**1. Excepciones..... 2**

1.1. Excepciones u errores .....2

1.2. Capturar excepciones.....4

1.2.1. Bloque try.....4

1.2.2. Lanzamiento de excepciones .....5

1.2.3. Creación de excepciones .....6

## 1. Excepciones

Una de las tareas más complicadas en programación es la gestión de errores. Un error mal controlado puede detener la ejecución de un programa sin que lleve a cabo su cometido.

La tarea del programador es conocer los puntos de su programa en los que se pueden producir errores y tener cierto control sobre ellos.

Aunque muchas veces es imposible evitar un error (por ejemplo si se quiere leer un fichero y está dañado o el motor de la BBDD no está funcionando, etc.) pero sí se deben gestionar de forma lógica.

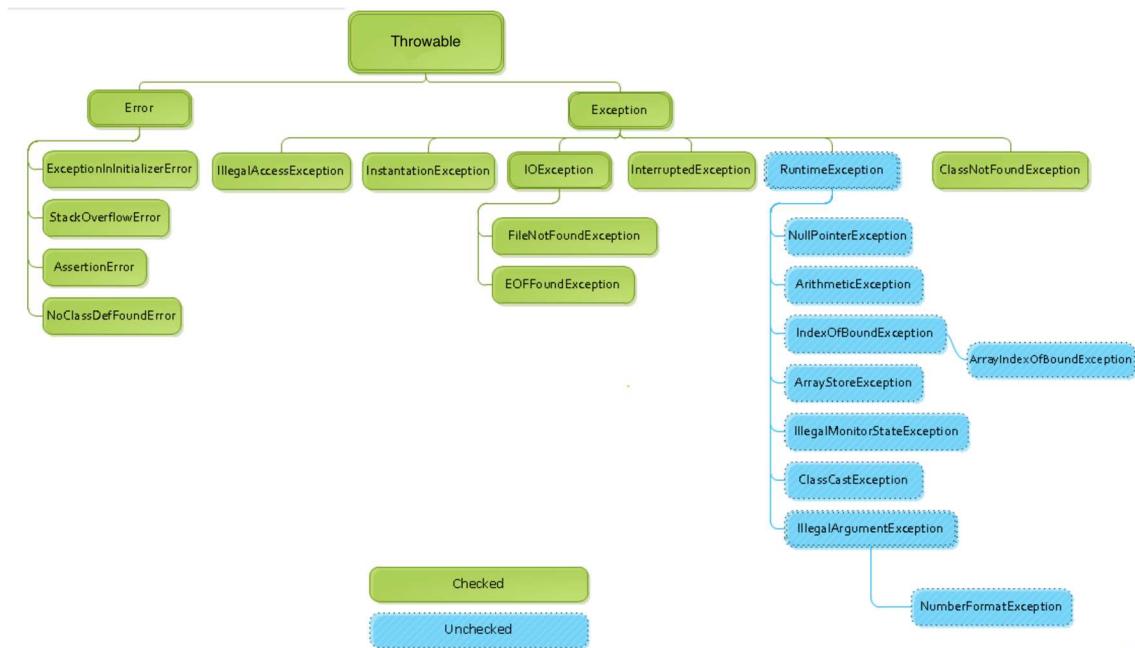
Para poder manejar los errores en Java se utilizan las excepciones, que son errores que se producen en tiempo de ejecución.

### 1.1. Excepciones u errores

En Java se distinguen dos tipos de errores, los que descenden de *Error* y la excepciones, que descenden de *Exception*:

- **Error**: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores (no se deben capturar.)
- **Exception**: Representa errores que no son críticos y por lo tanto pueden ser tratados y continuar la ejecución de la aplicación. La mayoría de los programas Java utilizan estas excepciones para el tratamiento de los errores que puedan ocurrir durante la ejecución del código.

Estas dos clases, descenden a su vez de *Throwable* que engloba cualquier tipo de error. También hay que destacar *RuntimeException*, que es hija de *Exception*, puesto que su descendencia es muy importante:



Algunas de las más destacadas son las siguientes:

- ***ArrayIndexOutOfBoundsException***: se produce cuando se intenta acceder a una posición que esta fuera de los límites de un *array* o colección.
- ***ClassCashException***: se produce cuando se intenta hacer una conversión explícita entre objetos que no está permitida (no verifica el IS-A).
- ***IllegalArgumentExpection***: se produce cuando un método recibe un argumento no permitido en la llamada. Por ejemplo, el método *setPriority*(int num) de un *Thread* espera recibir un entero de 1 a 10.

```
new Thread().setPriority(11);
```

```
Exception in thread "main" java.lang.IllegalArgumentException
    at java.lang.Thread.setPriority(Unknown Source)
    at scjp.exception_.PruebaExcepciones3.main(PruebaExcepciones3.java:10)
```

- ***NullPointerException***: se produce cuando se intenta acceder a un objeto a través de una referencia que contiene un valor nulo (*null*), es decir, se intenta acceder a un atributo o un método y el objeto es nulo.
- ***NumberFormatException***: se produce cuando un método que convierte un *String* en un número recibe una cadena que NO puede ser formateada como tal.

```
String miCadena = "JC";
int miEntero = Integer.parseInt(miCadena);
System.out.println(miEntero);
```

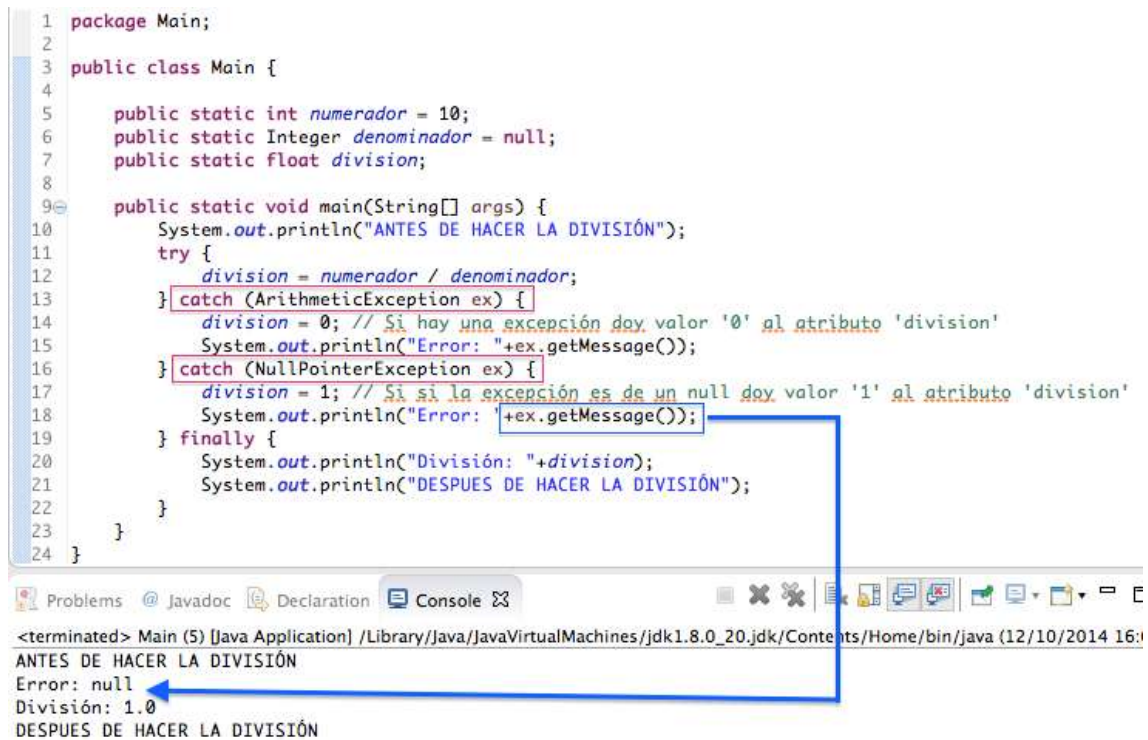
```
Exception in thread "main" java.lang.NumberFormatException: For input string: "JC"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
```

## 1.2. Capturar excepciones

El manejo de excepciones Java se gestiona a través de cinco palabras clave: **try**, **catch**, **throw**, **throws**, y **finally**.

### 1.2.1. Bloque try

Permite gestionar los errores que se producen dentro del **try**:



```
1 package Main;
2
3 public class Main {
4
5     public static int numerador = 10;
6     public static Integer denominador = null;
7     public static float division;
8
9     public static void main(String[] args) {
10         System.out.println("ANTES DE HACER LA DIVISIÓN");
11         try {
12             division = numerador / denominador;
13         } catch (ArithmeticException ex) {
14             division = 0; // Si hay una excepción doy valor '0' al atributo 'division'
15             System.out.println("Error: "+ex.getMessage());
16         } catch (NullPointerException ex) {
17             division = 1; // Si la excepción es de un null doy valor '1' al atributo 'division'
18             System.out.println("Error: "+ex.getMessage());
19         } finally {
20             System.out.println("División: "+division);
21             System.out.println("DESPUES DE HACER LA DIVISIÓN");
22         }
23     }
24 }
```

The screenshot shows the output in the console:   
<terminated> Main (5) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_20.jdk/Contents/Home/bin/java (12/10/2014 16:00:00)  
ANTES DE HACER LA DIVISIÓN  
Error: null  
División: 1.0  
DESPUES DE HACER LA DIVISIÓN

El funcionamiento es el siguiente:

- Dentro del **try** pondríamos todo el código que puede generar una excepción o error.
- Tendríamos un **catch** por cada excepción que capturemos.
- El bloque **finally** es opcional y permitiría ejecutar siempre un código tanto si se producen o no excepciones (incluso después de un **return**, **break**, etc.). Muy útil para cerrar conexiones de BBDD o cerrar **streams** de ficheros, etc.

```
try {  
    // Instrucciones cuando no hay una excepción  
} catch (TypeException ex) {  
    // Instrucciones cuando se produce una excepción  
} finally {  
    // Instrucciones que se ejecutan, tanto si hay como si no hay excepciones  
}
```

**Muy importante**, se pueden obviar el *catch* o el *finally* pero no los dos a la vez.

### 1.2.2. Lanzamiento de excepciones

Es posible que en ciertas condiciones no merezca la pena capturar una excepción, sino volver a lanzarla. Estos casos se dan cuando en la vuelta de alguna llamada a un método se dispone de una gestión bastante buena de excepciones (en la clase en la que se invoca al método).

Para poder lanzar excepciones se dispone de dos mecanismos:

- **Throws**: en la signatura del método se indica si una o varias excepciones se van a lanzar:

```
public void leeFichero() throws IOException, FileNotFoundException {  
    // Cuerpo del método  
}
```

- **Throw**: permite lanzar una excepción. Se suele utilizar en las siguientes situaciones:
  - Lanzamiento de excepciones propias o utilización de las que existen.
  - Relanzar una excepción capturada en un *catch*, pero dejando información en la consola o en los *logs*.
  - Encapsular la información de una excepción capturada en un *catch*, dentro de otra, que será la que se lance de nuevo. **MUY IMPORTANTE**, no hay que caer en la tentación de encapsular todas las excepciones en *Exception*, porque de esa manera se perdería mucha información.

```
public void leeFichero() throws IOException, FileNotFoundException {  
    // Código...  
    throw new IOException("Se ha producido un error al leer un fichero");  
    // Código...  
}
```

Dentro de estos mecanismos hay una distinción según el tipo de excepción:

- **Excepciones *checked***: son aquellas que deben tratarse.

- La primera forma sería declararlas en el método mediante la palabra *throws*.

```
public class Main {  
    public static void main(String[] args) throws Exception {  
        Matematicas matematicas=new Matematicas();  
  
        double c=matematicas.dividir(-1.6, 0);  
    }  
}
```

- También sería posible capturarlas con un *catch* para tratarlas:

```
public class Main {  
    public static void main(String[] args) {  
        Matematicas matematicas=new Matematicas();  
        try {  
            double c=matematicas.dividir(-1.6, 0);  
        } catch (Exception ex) {  
            //Tratar la excepción  
        }  
    }  
}
```

- **Excepciones *unchecked*:** son aquellas que no es necesario declararlas en un método y por tanto, no es necesaria su gestión, aunque en muchos casos sería recomendable:

```
public class Teoria {  
    public static void main(String[] args) {  
  
        int a = 0;  
        int b = 0;  
  
        if(b == 0) {  
            throw new RuntimeException("El argumento b no puede ser 0");  
        }  
  
        System.out.println("Llego hasta aquí");  
  
        double resultado = a/b;  
        System.out.println("Resultado: " + resultado);  
    }  
}
```

La excepción *RuntimeException* es *unchecked*, por tanto no es necesario su gestión, aunque el programa no acabaría su ejecución:

```
Exception in thread "main" java.lang.RuntimeException: El argumento b no puede ser 0  
at com.venancio.dam.tema8.Teoría.main(Teoria.java:13)
```

### 1.2.3. Creación de excepciones

Para poder crear excepciones propias se deben conocer algunos métodos y constructores:

**Constructores:**

- `public Exception()`
- `public Exception(String message)`
- `public Exception(String message, Throwable cause)`

### Métodos importantes de *Exception*:

- `String getMessage();` Muestra el mensaje de la excepción
- `Void printStackTrace();` Muestra la traza completa de error por consola.

```
7 public static void main(String[] args) {
8     IOException e = new IOException("Se ha producido un error al leer un fichero");
9     e.printStackTrace();
10    System.out.println("Mensaje de error: " + e.getMessage());
11 }
12
```

Terminated Teoria (5) [Java Application] [C:\Users\paz\AppData\Local\Programs\Java\jdk-11.0.2\bin\java.exe]

**java.io.IOException: Se ha producido un error al leer un fichero**  
**at com.venancio.dam.tema8.Teoría.main(Teoría.java:8)**  
Mensaje de error: Se ha producido un error al leer un fichero

**Creación de una excepción propia:** simplemente se debe extender de *Exception* y usar sus constructores, en los de la nueva excepción:

```
public class NoExisteException extends Exception{
    private static final long serialVersionUID = 1L;

    public NoExisteException() {
        super();
    }
    public NoExisteException(String msg) {
        super(msg);
    }
    public NoExisteException(String msg, Throwable e) {
        super(msg, e);
    }
}
```