

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

Excepciones

© ADR Infor SL

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

viewnext.adrformacion.com © ADR Infor SL
VICTOR TENA PALOMARES

Indice

Competencias y Resultados de Aprendizaje desarrollados en esta unidad	3
Excepciones.	4
Objetivos.	4
Definición.	4
Causas de excepciones	6
Bloque try ... catch ... finally.	7
try	7
catch	8
Objeto Exception.	9
finally	10
throws	11
Sintaxis.	11
Cómo tratar las excepciones.	13
Excepción no controlada.	13
Excepción controlada con el bloque try.	13
Controlar IOException con bloque try.	14
Combinar el uso de throws con try.	15
Uso del método printStackTrace().	17
Ejercicios	19
Ejercicio 1. Excepciones trabajando con números.	19
Recomendaciones.	19
Datos a mostrar por consola.	19
Por consola cuando se lanza la ArithmeticException	19
Por consola cuando se lanza la NumberFormatException	20
Ejercicio 2. Sumar números.	20
Recomendaciones.	20
Datos a mostrar por consola.	21
Si todo va bien	21
Si se introduce texto en lugar de números	21
Lo necesario para comenzar.	21
Recursos	23
Enlaces de Interés	23

Competencias y Resultados de Aprendizaje desarrollados en esta unidad

Competencia:

Conocer el concepto de Excepción en un programa Java y aprender su manejo

Resultados de Aprendizaje:

- Conocer y entender el significado de las excepciones en Java
- Usar de forma correcta las clases `java.lang.Throwable` y `java.lang.Exception`.
- Conocer las clases de Excepción más comunes en Java.
- Aprender el manejo de excepciones en Java.

Excepciones.

Objetivos.

- Conocer y entender el significado de las excepciones en Java.
- Conocer las clases `java.lang.Throwable` y `java.lang.Exception`.
- Conocer las clases de Excepción más comunes en Java.
- Aprender el manejo de excepciones en Java.

Definición.

El control de flujo de un programa Java sabemos que se lleva a cabo con sentencias del tipo `if`, `while`, `for`, `return`, `break`, etc... Estas sentencias forman un conjunto de palabras reservadas que determinan cierta funcionalidad.

Ninguna de estas sentencias tiene en cuenta que se puedan producir errores en tiempo de ejecución de un programa y por tanto Java necesita de un conjunto de palabras nuevas para tener en cuenta que cualquier código puede fallar o ser mal interpretado en tiempo de ejecución.

Vocabulario: Excepción

Una excepción es un suceso anómalo relacionado con determinadas líneas del código de un programa que puede provocar un fallo en su ejecución o compilación.

A nivel de programación, las excepciones **son objetos de clases, forman parte de la API** y cuentan con sus correspondientes constructores y métodos.

Aparte de las excepciones que se encuentran en la API, el programador puede definir excepciones propias.

Ejemplo: Excepciones asociadas a errores de ejecución:

- Un programa intenta acceder a un fichero que no existe. Se produce o lanza una `FileNotFoundException`.
- Un programa trata de realizar una operación matemática cuyo resultado está indeterminado o es infinito. Se lanza una `ArithmeticException`.
- Un programa intenta la conexión con un servidor cuyo nombre de dominio es incorrecto. Se lanza una `UnknownHostException`.
- Un programa necesita un valor numérico y se le introduce una letra. Se lanza una `NumberFormatException`.

Ejemplo: Excepciones asociadas a errores de compilación:

- Lectura de un dato introducido a través del teclado con el método de `java.io.BufferedReader` **`String readLine()`**. El compilador muestra un mensaje indicando que el código puede provocar una excepción, en este caso una **`IOException`**. A efectos de programación, se traduce en que el compilador obliga al programador a gestionar la excepción, a dar una alternativa de ejecución en el caso de que se produzca la excepción. Si no se hace esto, no se permite la compilación.
- Aplicación de un retardo en la ejecución de código mediante el método estático de `java.lang.Thread` **`void sleep(long retardo)`**. Si no se gestiona la **`InterruptedException`** que puede provocar la utilización de este método, no se permite la compilación.

Hay varias formas de saber qué métodos de una API lanzan una excepción de gestión obligatoria a nivel de compilación del código:

- Si la excepción no es del tipo `RuntimeException` y aparece la cláusula **`throws`** **NombreExcepcion** en la explicación del método proporcionada por la API, la gestión es obligatoria.
Se las suele llamar excepciones comprobadas o **`checked`**.
Ejemplo: `String readLine()` con `IOException` o `void sleep(long retardo)` con `InterruptedException`. Consultar la API.
- Si la excepción es de tipo `RuntimeException`, no es de gestión obligatoria. No importa que aparezca la cláusula `throws NombreExcepcion` en la explicación del método.
Se las suele llamar excepciones no comprobadas o **`unchecked`**.
Ejemplo: `static int parseInt(String num)` de `java.lang.Integer` con `NumberFormatException` (es hija de `IllegalArgumentException`, que a su vez lo es de `RuntimeException`)

Existen dos mecanismos empleados para la gestión de excepciones en Java: capturarlos o lanzarlos.

De forma introductoria diremos que hay dos formas de tratar errores en Java: **capturarlos o lanzarlos**.

El uso de **`try – catch – finally`** corresponde a la captura de errores.

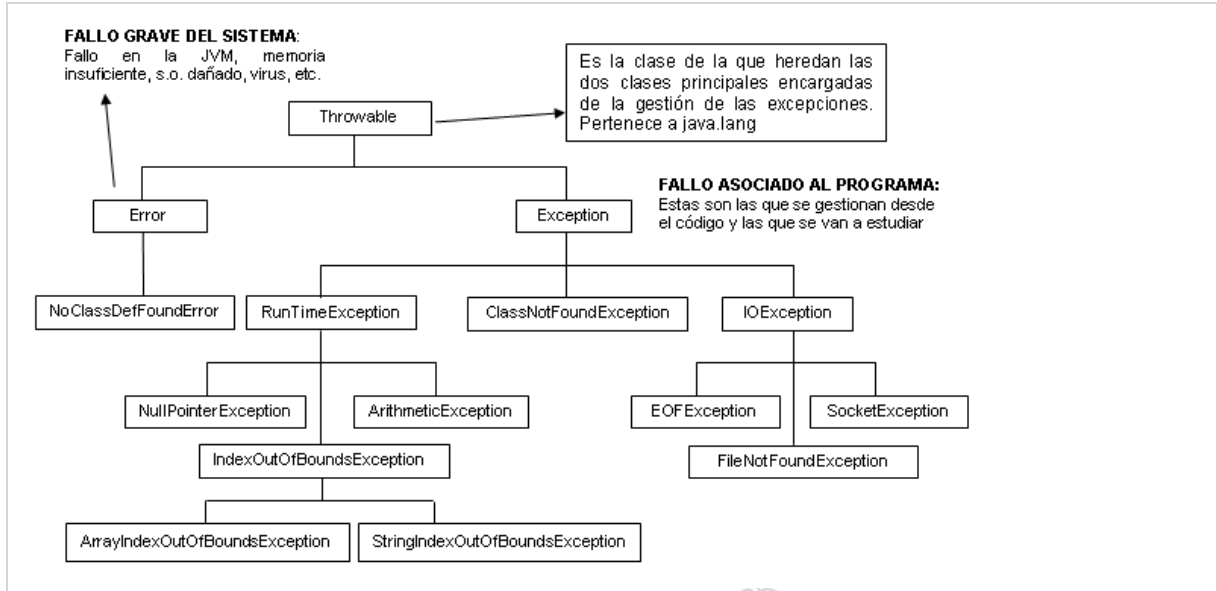
Vamos a poner un símil sencillo:

Un error es algo inesperado, como encontrarte un ladrón dentro de tu casa.

Cuando nos encontramos con un error podemos capturarlo (equivaldría a capturar el ladrón) o **lanzarlo** (equivaldría a tratar de hacer huir al ladrón, de hacer que salga fuera del lugar donde se encuentra).

- **Bloque `try ... catch ... finally`**. Corresponde a la captura de errores. Este bloque se ejecuta cuando ocurre el error de la clase especificada en la cláusula `catch`.
- **Cláusula `throws`**. Corresponde a lanzar el error. Esto supone que el error no ha sido controlado todavía y será delegado para que lo trate el elemento que lo ha invocado.

Alguna de las clases de la API vinculadas a excepciones más o menos habituales se muestran en el siguiente diagrama: consultar la API para ampliar información.



Causas de excepciones

Hay muchas causas de excepciones y la mayoría llevan asociada una clase Java que las ejemplariza. Vamos a ver ahora una lista con las clases más importantes:

- **IOException: InputOutputException.** Asociada a errores genéricos de introducción de datos a un programa o de generación de salida desde el mismo. Muy importante pues de ella heredan otras muchas que se emplean con mucha frecuencia. No suele producirse casi nunca. No obstante, es de gestión obligatoria.
- **FileNotFoundException:** asociada al hecho de no encontrar un fichero.
- **EOFException: EndOfFileException:** indica que se ha llegado al final de un fichero.
- **NoClassDefFoundError:** no se encuentra el class de la clase principal. Es un error o fallo grave, no una excepción.
- **ClassNotFoundException:** carga incorrecta de una clase externa que necesita un programa para funcionar correctamente.
- **ArithmeticException:** engloba errores aritméticos como
 - División por cero.
 - Cálculo de razones trigonométricas con argumentos incorrectos.
 - Cálculo de logaritmos de números negativos, etc.

- **NullPointerException:** se produce al intentar acceder a una variable referenciada que no ha sido inicializada.
- **ArrayIndexOutOfBoundsException:** se produce al intentar acceder a un elemento de un array con un índice que es igual o mayor que su tamaño. Los arrays se estudian en el tema siguiente.
- **StringIndexOutOfBoundsException:** se produce al aplicar un método de String, por ejemplo “char charAt(int indice)” pasándole al argumento un índice que es igual o mayor que su tamaño.
- **NumberFormatException:** se produce cuando se introduce un tipo de dato incorrecto en los argumentos de algunos métodos. Ejemplo: el método estático “int parseInt(String num)” espera recibir un número almacenado en forma de String. Si recibe texto, se lanza una NumberFormatException. Hereda de IllegalArgumentException y ésta de RuntimeException.

Bloque try ... catch ... finally.

Una forma de tratar errores es a través del bloque **try ... catch ... finally**.

Este bloque se encarga de capturar los errores sucedidos en la ejecución de un código, tratarlos y realizar acciones tras haberse producido el error.

La sintaxis de este bloque es la siguiente:

```
try{
    /* Código que se ejecuta en condiciones normales. Está vigilado
    * porque se encuentra entre un try y un catch. En el momento en
    * que la ejecución de alguna línea de este código provoque una
    * excepción, dejan de ejecutarse las líneas de código restantes y
    * pasa a ejecutarse el código del bloque catch correspondiente.
    */
} catch (Exception1 e1){
    /* Código que se ejecuta si se produce una excepción de tipo Excepcion1*/
} catch (Exception2 e2){
} catch (...){
    .
    .
    .
} catch (Exception e){
    /* Código que se ejecuta si se produce una excepción no capturada con los catch
    * previos
    */
} finally{
    /* Código que se ejecuta siempre, independientemente de si se producen o no
    * excepciones o no. Suele emplearse para cerrar conexiones con bases de datos
    * o flujos de lectura y escritura.
    */
}
```

try

El código que se **encuentra entre la sentencia try y la primera sentencia catch** se ejecutará normalmente en la aplicación, pero **si se produce un error se abortará el proceso y se pasará a ejecutar la línea del catch correspondiente.**

catch

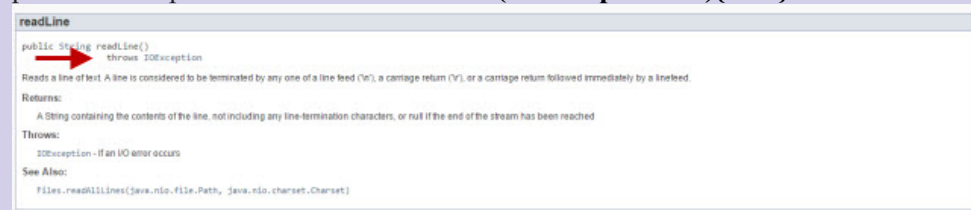
La sentencia catch se encarga de **tratar un tipo de excepción cuando esta sucede**, esto es, si durante la ejecución del código 'vigilado' se produce un error de tipo NullPointerException el proceso pasará a ejecutar el código que se encuentre en la sentencia

```
catch(NullPointerException npe){
//codigo que se ejecutará
}
```

El **número** de catch que se pueden utilizar en un bloque es **ilimitado**, el único requisito es que la excepción tratada por el catch pueda ser lanzada por el código 'vigilado'.

Anotación: Comprobar excepciones lanzadas.

Para saber que excepciones puede lanzar nuestro código debemos revisar el API de los métodos que utilizamos, por ejemplo, si usamos el método readLine() de la clase BufferedReader podemos mirar en el API <http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html> y comprobar que lanza una excepción de tipo IOException que podríamos capturar con la sentencia **catch(IOException ioe){ }**



La herencia de Java permite que una sentencia catch pueda tratar las excepciones del tipo que tiene declarado y de todo aquel tipo que herede de él. En Java todas las excepciones heredan obligatoriamente de la clase Exception, por lo que es habitual el uso de una sentencia del tipo

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

que se encargaría de tratar todas las posibles excepciones del código.

Habitualmente se emplea un catch que contiene una excepción genérica asociada a la clase Exception que contiene a todas y, mediante el método **String getMessage()** de java.lang.Throwable o **String toString()** de java.lang.Object se obtiene información sobre el tipo de excepción que ha ocurrido en el código.

Con esto se **evitan muchas líneas de código** y la necesidad de conocer el nombre de todas las excepciones, pero se **pierde flexibilidad** ya que no se puede programar en función del tipo de excepción producido. No se podría conseguir que el código hiciera una cosa si se produce la IOException e1 e hiciera otra si se produce la FileNotFoundException e2.

Si, durante la ejecución de un programa, se lanza una excepción y es capturada por un bloque catch, se ejecuta el código asociado al mismo y luego no se evalúan el resto de catch, el proceso continuará ejecutando el bloque finally si es que existe.

Recuerda: finally

El bloque finally no es obligatorio.
Se ejecuta siempre, haya ocurrido o no un error.

Si no existe bloque finally se continúa la ejecución en la siguiente línea de código después del bloque **try{ ... }catch{ ... }**.

Objeto Exception.

Asociado a cada tipo de excepción se tiene un objeto de una clase de la excepción tratada que permite su gestión.

En la sentencia
catch(IOException ioe){ ... }
ioe es un objeto de tipo **IOException** que nos aportará información sobre el error que ha sucedido.

Estos objetos son los argumentos de los bloques catch y se envían al mismo en el momento en que se lanza la excepción.

Todos los tipos de excepciones **herendan**, entre otros, los métodos **getMessage()** y **toString()** de la clase padre Exception que nos permiten recoger información de qué línea de código es la que ha causado el error y porque se ha originado. Ambos métodos devuelven un mensaje en forma de String que detalla el tipo de excepción producido siendo un poco más completo el detalle del segundo método.

Para mostrar la causa del error en la consola podemos utilizar el siguiente código:

```
try{
.....
br.readLine();
.....
}catch(IOException ioe){
System.out.println(ioe.getMessage());
}
```

donde si el método `readLine()` que utilizamos sufre un error de tipo `IOException` durante su ejecución el bloque `catch` correspondiente tomará el control del programa y mostrará en consola el detalle del error.

Truco: `printStackTrace`

Además de estos dos métodos se emplea bastante el **`void printStackTrace()`** de `java.lang.Throwable` que muestra, además de información sobre la excepción lanzada, los **nombres de los métodos que han sido ejecutados** hasta llegar al método en el que se ha producido la excepción (pila de métodos ejecutados o `method call stack`).

Una vez que un `catch` ha cogido el control de la ejecución ya no se evalúan el resto de los definidos para ese bloque `try`.

finally

El bloque `finally` es **opcional**, puede aparecer o no en un bloque `try`.

En caso de existir, este bloque aparece siempre tras la definición de todos los `catch` necesarios.

Contiene código generado por el programador que se ejecutará siempre, se lance o no alguna excepción durante la ejecución del programa.

Anotación: Uso más común.

Se suele utilizar para cerrar recursos que hayamos abierto, por ejemplo cerrar los buffer de lectura, cerrar las conexiones a bases de datos, etc.

Tanto las sentencias `finally` como las sentencias `catch` pueden encadenar sucesivos bloques `try` en su interior si el código que incluyen es susceptible de lanzar excepciones.

finally con try encadenado.

```

...
try{
    ...
    br.readLine();
    ....
}catch(IOException ioe){
    ....
}finally{
    try{
        br.close();
    }catch(IOException ioe2){
        ...
    }
}
}

```

Hay que recordar que si no se tratan los errores que lanzan los métodos que utilizamos se producirán errores de compilación que no permitirán que el código se ejecute. Una forma de tratarlos es el bloque try.



Manejar excepción con Try-Catch

throws

Es la base del otro mecanismo para gestionar excepciones. La diferencia con try .. catch es que no se ejecuta código en respuesta a la excepción sino que se delega la posibilidad de hacerlo al método invocante de aquel que ha provocado la excepción.

La cláusula **throws** permite crear métodos susceptibles de producir errores sin necesidad de tratarlos en dichos métodos. Estos errores serán lanzados al nivel superior de ejecución y deberán ser tratados por el código que utilice ese método o bien vueltos a lanzar a otro nivel superior.

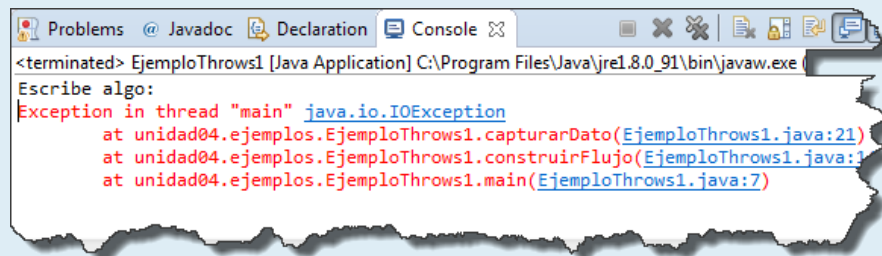
La clase `java.lang.Exception` hereda de la clase `java.lang.Throwable` lo que permite que todas las excepciones puedan ser 'lanzadas' al nivel superior de ejecución y no ser manejadas en el método en el que se producen.

Sintaxis.

La cláusula `throws` debe aparecer siempre en la definición del método, a continuación de los parámetros:

```
public String readLine() throws IOException{
...
}
```

```
import java.io.*;
public class EjemploThrows1 {
    // Recoge la excepción lanzada por la línea et1.construirFlujo(); y la vuelve a lanzar.
    public static void main(String args[]) throws IOException{
        EjemploThrows1 et1=new EjemploThrows1();
        et1.construirFlujo();
        System.out.println("FIN DE PROGRAMA");
    }
    // Recoge la excepción lanzada por la línea capturarDato(br); y la vuelve a lanzar.
    void construirFlujo() throws IOException{
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        capturarDato(br);
        System.out.println("Fin del metodo construirFlujo");
    }
    // Recoge la excepción lanzada por la línea String teclado=filtro.readLine(); y la vuelve a l
    anzar.
    void capturarDato(BufferedReader filtro) throws IOException{
        System.out.print("Escribe algo: ");
        String teclado=filtro.readLine();
        System.out.println("Lo introducido por teclado es "+teclado);
    }
}
```



En el caso del ejemplo anterior **ningún método se encarga de tratar la excepción**, todos la lanzan al nivel superior, por lo que en caso de que durante la ejecución de la aplicación la sentencia `filtro.readLine()` genere una `IOException`, **la ejecución de la aplicación se abortará y se mostrará en consola el error** que se ha producido. Esta es la forma en la que maneja Java las excepciones no controladas en el nivel más alto.



Manejar Excepciones con Throws

Cómo tratar las excepciones.

Vamos a ver ahora con ejemplos varias formas de tratar excepciones en Java.

Todos los códigos de este tema se guardarán en **c:\cursojava\unidad4** o en el **workspace correspondiente** si se emplea Eclipse.

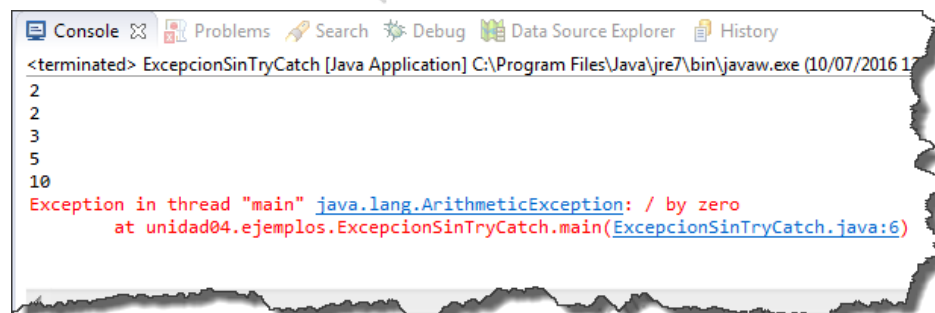
Excepción no controlada.

En este caso vamos a ver que sucede si nuestro código genera una excepción que no hemos controlado.

Se producirá un error al intentar dividir un número entre 0 y esto hará que se lance una excepción de tipo [ArithmeticException](#).

```
public class ExcepcionSinTryCatch{
    public static void main(String args[]){
        int x=10;
        for(int i=5;i>=0;i--){
            System.out.println(x/i);
        }
        System.out.println("FIN DE PROGRAMA");
    }
}
```

El resultado que aparecerá en la consola será el siguiente:



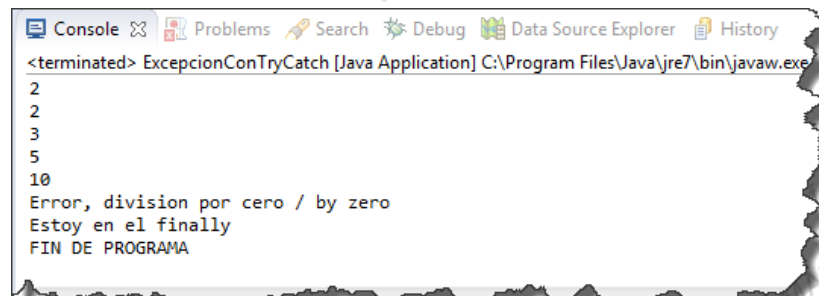
Al no tener tratada la excepción esta se lanza hasta el nivel más alto y es Java el que se encarga de abortar la ejecución y mostrar la causa del error.

Excepción controlada con el bloque try.

Vamos a repetir el ejemplo anterior pero controlando el error que se produce con un bloque try.

```
public class ExcepcionConTryCatch{
    public static void main(String args[]){
        int x=10;
        try{
            for(int i=5;i>=1;i--) {
                System.out.println(x/i);
            }
        } catch(ArithmeticException ae){
            System.out.println("Error, division por cero " + ae.getMessage());
        } finally {
            System.out.println("Estoy en el finally");
        }
        System.out.println("FIN DE PROGRAMA");
    }
}
```

El resultado que aparecerá en la consola será el siguiente:



En este caso no se aborta la ejecución del programa, se ejecuta el bloque del catch y se continúa mostrando la línea de FIN DE PROGRAMA.

No se muestra la traza del error.

Controlar IOException con bloque try.

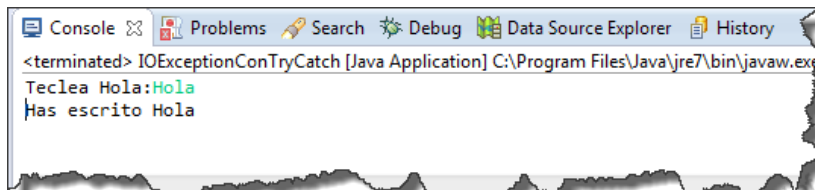
Nuestro código se va a encargar de realizar la lectura de datos desde el teclado utilizando para ello el método `readLine()` de la clase `BufferedReader`. Este método lanza una excepción de tipo `IOException` que controlaremos con un bloque try.

```

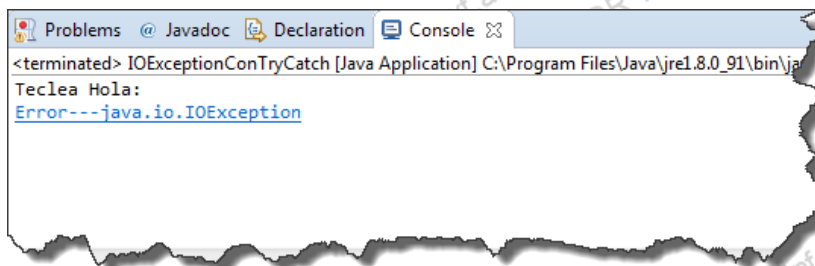
import java.io.*;
public class IOExceptionConTryCatch{
    public static void main(String args[]){
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        System.out.print("Teclea Hola:");
        try{
            String teclado=br.readLine();
            if(teclado.equals("Hola")){
                System.out.println("Has escrito Hola");
            }else{
                System.out.println("No has escrito el texto correcto. Debes escribir Hola");
            }
        }catch(IOException e){
            System.out.println("Error---" + e.toString());
        }
    }
}

```

El resultado que aparecerá en la consola será el siguiente:



Si todo va bien se ejecutará el código sin problemas, pero en el caso de que se produzca un error en la lectura del teclado que genere una IOException se mostrará por pantalla:



Combinar el uso de throws con try.

En este caso vamos a ver el ejemplo que utilizamos en el apartado de la cláusula throws y lo vamos a combinar con el uso del bloque try.

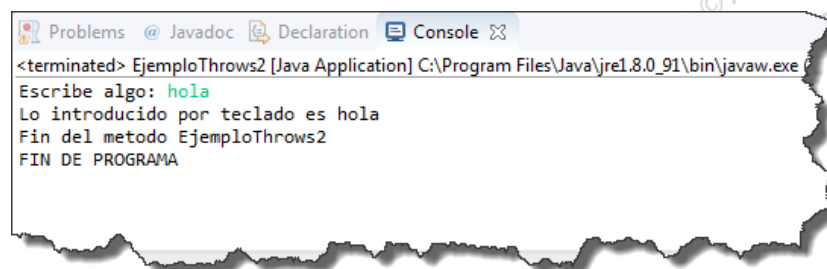
```

import java.io.*;
public class EjemploThrows2{
    public static void main(String args[]){
        EjemploThrows2 et2=new EjemploThrows2();
        et2.construirFlujo();
        System.out.println("FIN DE PROGRAMA");
    }
    void construirFlujo(){
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        try{
            capturarDato(br);
        }catch(IOException e){
            System.out.println("Error---" + e.toString());
        }
        System.out.println("Fin del metodo EjemploThrows2");
    }
    void capturarDato(BufferedReader filtro) throws IOException{
        System.out.print("Escribe algo: ");
        String teclado=filtro.readLine();
        System.out.println("Lo introducido por teclado es " + teclado);
    }
}

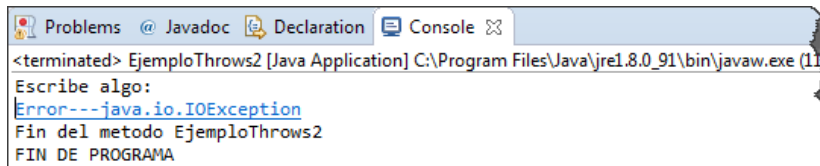
```

Ahora vemos que no es necesario utilizar la cláusula throws en el método construirFlujo ni tratar la excepción en el método main ya que se trata en el bloque try del método construirFlujo.

El resultado que aparecerá en la consola será el siguiente:



Si todo va bien se ejecutará el código sin problemas, pero en el caso de que se produzca un error en la lectura del teclado que genere una IOException se mostrará por pantalla:



Problems @ Javadoc Declaration Console

<terminated> EjemploThrows2 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (11)

Escribe algo:

Error---java.io.IOException

Fin del metodo EjemploThrows2

FIN DE PROGRAMA

Uso del método printStackTrace().

El método void printStackTrace() se utiliza para mostrar en consola información sobre el error acontecido.

En este caso el programa convertirá lo introducido por teclado en un objeto de tipo Integer. Esto hace que si el dato a convertir no es un número se genere una excepción de tipo NumberFormatException que controlaremos en el try.

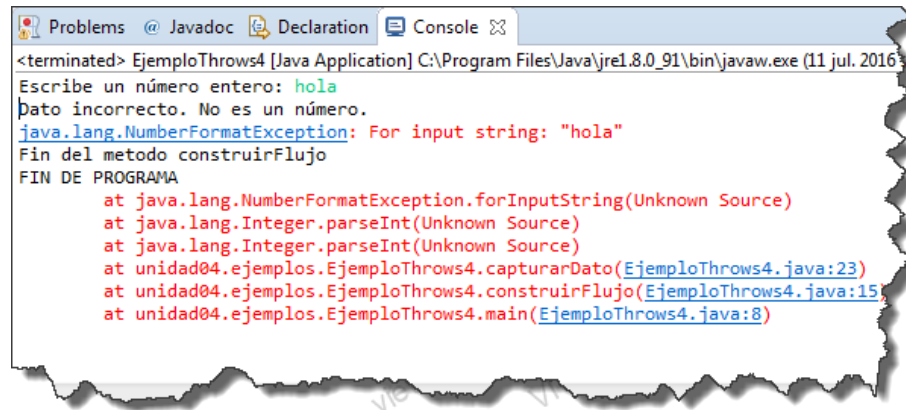
```
import java.io.*;

public class EjemploThrows4 {
    public static void main(String args[]) {
        EjemploThrows4 et4 = new EjemploThrows4();
        et4.construirFlujo();
        System.out.println("FIN DE PROGRAMA");
    }

    void construirFlujo() {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        capturarDato(br);
        System.out.println("Fin del metodo construirFlujo");
    }

    void capturarDato(BufferedReader filtro) {
        System.out.print("Escribe un número entero: ");
        try {
            String teclado = filtro.readLine();
            int num = Integer.parseInt(teclado);
            System.out.println("Número entero introducido " + teclado);
        } catch (IOException e) {
            System.out.println("Error ---" + e.toString());
        } catch (NumberFormatException nfe) {
            System.out.println("Dato incorrecto. No es un número.");
            nfe.printStackTrace();
        }
    }
}
```

Si se introduce un texto en lugar de un número entero el resultado que aparecerá en la consola será:



The screenshot shows a Java IDE console window with the following text:

```
<terminated> EjemploThrows4 [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (11 jul. 2016)
Escribe un número entero: hola
Dato incorrecto. No es un número.
java.lang.NumberFormatException: For input string: "hola"
Fin del metodo construirFlujo
FIN DE PROGRAMA
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at unidad04.ejemplos.EjemploThrows4.capturarDato(EjemploThrows4.java:23)
    at unidad04.ejemplos.EjemploThrows4.construirFlujo(EjemploThrows4.java:15)
    at unidad04.ejemplos.EjemploThrows4.main(EjemploThrows4.java:8)
```



Vamos a organizar nuestro código

Ejercicios

Todos los códigos en `c:\cursojava\unidad4` o en `eclipse_home\MyProjects\unidad4` si se emplea Eclipse

Ejercicio 1. Excepciones trabajando con números.

20

Crear una clase pública de nombre `EjercicioExcepciones` que contenga sólo al método `main` y que haga lo siguiente:

- pedirá por teclado dos números enteros que almacenará en dos variables `int` de nombres `inicio` y `fin`.
- el bucle empezará en `inicio` e irá disminuyendo de uno en uno hasta llegar a `fin`.
- el código obtendrá la parte entera de la división entre 10 y el valor numérico de cada paso del bucle tal y como se ha hecho en el Ejemplo 1. Si el `inicio` del bucle es menor que el `final`, el código no hará nada.

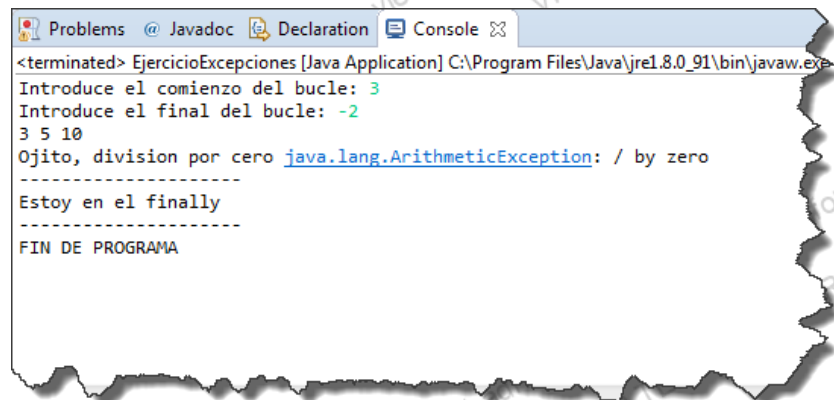
Recomendaciones.

Se considerarán tres posibles excepciones con tres bloques `catch`:

- Una `ArithmeticException` asociada a la división por cero de gestión no obligatoria.
- Una `NumberFormatException` asociada a la introducción por teclado de un dato no numérico de gestión no obligatoria.
- Una `IOException` asociada al empleo del método `String readLine()` de gestión obligatoria.

Datos a mostrar por consola.

Por consola cuando se lanza la `ArithmeticException`

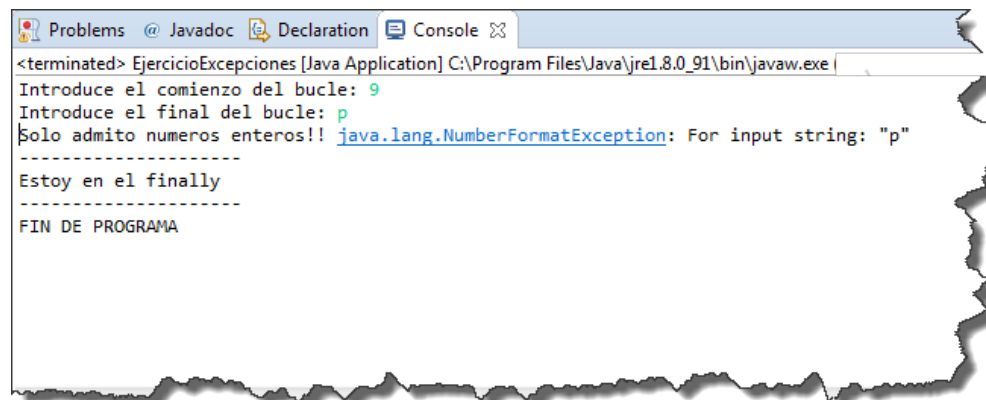


```

Problems @ Javadoc Declaration Console
<terminated> EjercicioExcepciones [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Introduce el comienzo del bucle: 3
Introduce el final del bucle: -2
3 5 10
Ojito, division por cero java.lang.ArithmeticException: / by zero
-----
Estoy en el finally
-----
FIN DE PROGRAMA

```

Por consola cuando se lanza la NumberFormatException



```

Problems @ Javadoc Declaration Console
<terminated> EjercicioExcepciones [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Introduce el comienzo del bucle: 9
Introduce el final del bucle: p
Solo admito numeros enteros!! java.lang.NumberFormatException: For input string: "p"
-----
Estoy en el finally
-----
FIN DE PROGRAMA

```

Ejercicio 2. Sumar números.

20

Realizar un programa que sume una serie de números introducidos por consola y que además obtenga la suma de los positivos y de los negativos.

El programa, antes de que el usuario introduzca los números, le **preguntará cuántos números desea sumar**.

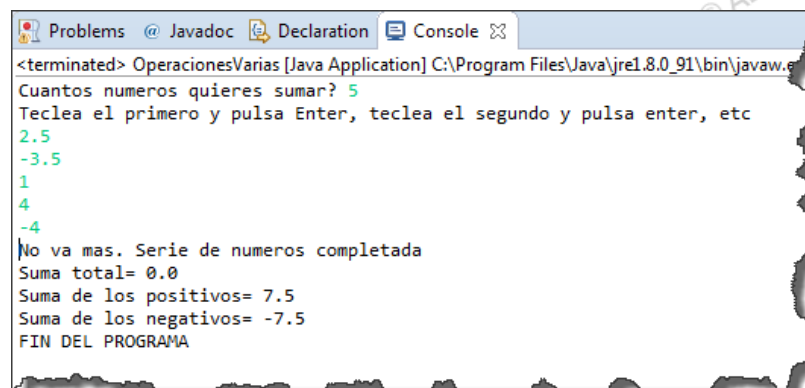
Recomendaciones.

Considerar mediante **bloques try ... catch** adecuados, las **NumberFormatException** que pueden producirse, para que el programa, en caso de introducción de algo distinto de lo que solicita, reconduzca la actitud del usuario hasta lograr que introduzca lo que necesita.

La **IOException** de gestión obligatoria se tendrá en cuenta con la cláusula **throws** en la declaración del método main

Datos a mostrar por consola.

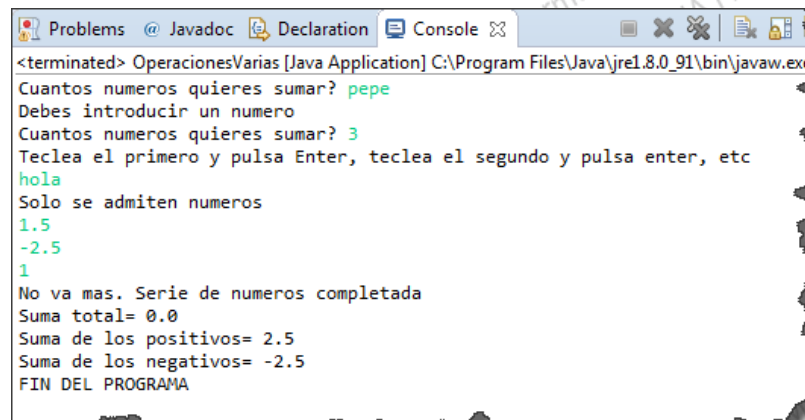
Si todo va bien



```

Problems @ Javadoc Declaration Console
<terminated> OperacionesVarias [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Cuantos numeros quieres sumar? 5
Teclea el primero y pulsa Enter, teclea el segundo y pulsa enter, etc
2.5
-3.5
1
4
-4
No va mas. Serie de numeros completada
Suma total= 0.0
Suma de los positivos= 7.5
Suma de los negativos= -7.5
FIN DEL PROGRAMA
  
```

Si se introduce texto en lugar de números



```

Problems @ Javadoc Declaration Console
<terminated> OperacionesVarias [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe
Cuantos numeros quieres sumar? pepe
Debes introducir un numero
Cuantos numeros quieres sumar? 3
Teclea el primero y pulsa Enter, teclea el segundo y pulsa enter, etc
hola
Solo se admiten numeros
1.5
-2.5
1
No va mas. Serie de numeros completada
Suma total= 0.0
Suma de los positivos= 2.5
Suma de los negativos= -2.5
FIN DEL PROGRAMA
  
```

Lo necesario para comenzar.

Se **partirá del siguiente esqueleto de código**: sólo tendrá método main. Pueden agregarse las variables locales que el programador considere oportunas.

Excepciones

```
import java.io.*;
public class OperacionesVarias {
    public static void main(String args[]) throws IOException {
        double total=0;
        double totalPositivos=0;
        double totalNegativos=0;
    }
}
```

Recursos

Enlaces de Interés



<http://programacion.com/java/cursos.htm>

<http://programacion.com/java/cursos.htm>

Manejo de Errores Utilizando Excepciones



<http://docs.oracle.com/javase/tutorial/essential/exceptions/>

<http://docs.oracle.com/javase/tutorial/essential/exceptions/>

Gestión de excepciones



<http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>

API BufferedReader



<http://docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html>

API ArithmeticException



<http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html>

API IOException