

8.A. Excepciones.

Sitio: [Formación Profesional a Distancia](#)

Curso: Programación

Libro: 8.A. Excepciones.

Imprimido por: Iván Jiménez Utiel

Día: lunes, 10 de febrero de 2020, 15:58

Tabla de contenidos

[1. Excepciones.](#)

[1.1. Capturar una excepción.](#)

[1.2. El manejo de excepciones.](#)

[1.3. Delegación de excepciones con throws.](#)

1. Excepciones.

A lo largo de nuestro aprendizaje de Java nos hemos topado en alguna ocasión con **errores**, pero éstos suelen ser los que nos ha indicado el [compilador](#). Un punto y coma por aquí, un nombre de [variable](#) incorrecto por allá, pueden hacer que nuestro [compilador](#) nos avise de estos descuidos. Cuando los vemos, se corrigen y obtenemos nuestra [clase](#) compilada correctamente.

Pero, ¿Sólo existen este tipo de Errores? ¿Podrían existir Errores no sintácticos en nuestros programas? Está claro que sí, un programa perfectamente compilado en el que no existen Errores de [sintaxis](#), puede generar otros tipos de Errores que quizá aparezcan en tiempo de ejecución. A estos Errores se les conoce como **excepciones**.

Aprenderemos a gestionar de manera adecuada estas excepciones y tendremos la oportunidad de utilizar el potente sistema de manejo de Errores que Java incorpora. La potencia de este sistema de manejo de Errores radica en:

1. Que el código que se encarga de manejar los Errores, es perfectamente identificable en los programas. Este código puede estar separado del código que maneja la aplicación.
2. Que Java tiene una gran cantidad de Errores estándar asociados a multitud de fallos comunes, como por ejemplo divisiones por cero, fallos de entrada de datos, etc. Al tener tantas excepciones localizadas, podemos gestionar de manera específica cada uno de los Errores que se produzcan.

En Java se pueden preparar los fragmentos de código que pueden provocar Errores de ejecución para que si se produce una [excepción](#), el flujo del programa es lanzado (**throw**) hacia ciertas zonas o rutinas que han sido creadas previamente por el programador y cuya finalidad será el tratamiento efectivo de dichas excepciones. Si no se captura la [excepción](#), el programa se detendrá con toda probabilidad.

En Java, las excepciones están representadas por clases. El paquete **java.lang.Exception** y sus subpaquetes contienen todos los tipos de excepciones. Todas las excepciones derivarán de la [clase](#) **Throwable**, existiendo clases más específicas. Por debajo de la [clase](#) **Throwable** existen las clases **Error** y **Exception**. **Error** es una [clase](#) que se encargará de los Errores que se produzcan en la máquina virtual, no en nuestros programas. Y la [clase](#) **Exception** será la que a nosotros nos interese conocer, pues gestiona los Errores provocados en los programas.

Java lanzará una [excepción](#) en respuesta a una situación poco usual. Cuando se produce un **Error** se genera un [objeto](#) asociado a esa [excepción](#). Este [objeto](#) es de la [clase](#) **Exception** o de alguna de sus herederas. Este [objeto](#) se pasa al código que se ha definido para manejar la [excepción](#). Dicho código puede manipular las propiedades del [objeto](#) **Exception**.

El programador también puede lanzar sus propias excepciones. Las excepciones en Java serán objetos de clases derivadas de la [clase](#) base **Exception**. Existe toda una jerarquía de clases derivada de la [clase](#) base **Exception**. Estas clases derivadas se ubican en dos grupos principales:

- Las excepciones en tiempo de ejecución, que ocurren cuando el programador no ha tenido cuidado al escribir su código.

- Las excepciones que indican que ha sucedido algo inesperado o fuera de [control](#).

En la siguiente imagen te ofrecemos una aproximación a la jerarquía de las excepciones en Java.



Imagen procedente de curso de Programación del MECD

1.1. Capturar una excepción.

Para poder capturar excepciones, emplearemos la estructura de captura de excepciones **try-catch-finally**.

Básicamente, para capturar una [excepción](#) lo que haremos será declarar bloques de código donde es posible que ocurra una [excepción](#). Esto lo haremos mediante un bloque try (intentar). Si ocurre una [excepción](#) dentro de estos bloques, se lanza una [excepción](#). Estas excepciones lanzadas se pueden capturar por medio de bloques **catch**. Será dentro de este tipo de bloques donde se hará el manejo de las excepciones.

Su **sintaxis** es:

```
try {  
    código que puede generar excepciones;  
} catch (Tipo_excepcion_1 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_1;  
} catch (Tipo_excepcion_2 objeto_excepcion) {  
    Manejo de excepción de Tipo_excepcion_2;  
}  
...  
finally {  
    instrucciones que se ejecutan siempre  
}
```

En esta estructura, la parte **catch** puede repetirse tantas veces como excepciones diferentes se deseen capturar. La parte **finally** es opcional y, si aparece, solo podrá hacerlo una sola vez.

Cada **catch** maneja un tipo de [excepción](#). Cuando se produce una [excepción](#), se busca el **catch** que posea el manejador de [excepción](#) adecuado, será el que utilice el mismo tipo de [excepción](#) que se ha producido. Esto puede causar problemas si no se tiene cuidado, ya que la [clase](#) `Exception` es la [superclase](#) de todas las demás. Por lo que si se produjo, por ejemplo, una [excepción](#) de tipo `ArithmeticException` y el primer **catch** captura el tipo genérico `Exception`, será ese **catch** el que se ejecute y no los demás.

Por eso el último **catch** debe ser el que capture excepciones genéricas y los primeros deben ser los más específicos. Lógicamente si vamos a tratar a todas las excepciones (sean del tipo que sean) igual, entonces basta con un solo **catch** que capture objetos `Exception`.

Ejercicio resuelto

Realiza un programa en Java en el que se solicite al usuario la introducción de un número por teclado comprendido entre el 0 y el 100. Utilizando manejo de excepciones, debes controlar la entrada de dicho número y volver a solicitarlo en caso de que ésta sea incorrecta.

Solución:

```
/*
```

```
* Ejercicio resuelto sobre manejo de excepciones.

* El programa solicita que el usuario introduzca por teclado

* un número entre 0 y 100, debiendo gestionarse la entrada

* por medio de excepciones.

*/

import java.io.*;

public class ejercicio_resuelto_excepciones {

    public static void main(String[] args){

        int numero=-1;

        int intentos=0;

        String linea;

        BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));

        do{

            try{

                System.out.print("Introduzca un número entre 0 y 100: ");

                linea = teclado.readLine();

                numero = Integer.parseInt(linea);

            }catch(IOException e){

                System.out.println("Error al leer del teclado.");

            }catch(NumberFormatException e){

                System.out.println("Debe introducir un número entre 0 y 100.");

            }finally{

                intentos++;

            }

        }
```

```
}while (numero < 0 || numero > 100);  
  
System.out.println("El número introducido es: " + numero);  
System.out.println("Número de intentos: " + intentos);  
  
}  
}
```

Se han utilizado estructuras try-catch-finally. En este programa se solicita repetidamente un número utilizando una estructura do-while, mientras el número introducido sea menor que 0 y mayor que 100. Como al solicitar el número pueden producirse los errores siguientes:

- De entrada de información a través de la [excepción](#) IOException generada por el [método](#) readLine() de la [clase](#) BufferedReader.
- De conversión de tipos a través de la [excepción](#) NumberFormatException generada por el [método](#) parseInt().

Entonces se hace necesaria la utilización de bloques catch que gestionen cada una de las excepciones que puedan producirse. Cuando se produce una [excepción](#), se compara si coincide con la [excepción](#) del primer catch. Si no coincide, se compara con la del segundo catch y así sucesivamente. Si se encuentra un catch que coincide con la [excepción](#) a gestionar, se ejecutará el bloque de sentencias asociado a éste.

Si ningún bloque catch coincide con la [excepción](#) lanzada, dicha [excepción](#) se lanzará fuera de la estructura try-catch-finally.

El bloque finally, se ejecutará tanto si try terminó correctamente, como si se capturó una [excepción](#) en algún bloque catch. Por tanto, si existe bloque finally éste se ejecutará siempre.

Autoevaluación

Si en un programa no capturamos una [excepción](#), será la máquina virtual de Java la que lo hará por nosotros, pero inmediatamente detendrá la ejecución del programa y mostrará una traza y un mensaje de error. Siendo una traza, la forma de localizar dónde se han producido errores. ¿Verdadero o Falso?

- ☐ Verdadero
☐ Falso

1.2. El manejo de excepciones.

Como hemos comentado, siempre debemos controlar las excepciones que se puedan producir o de lo contrario nuestro software quedará expuesto a fallos. Las excepciones pueden tratarse de dos formas:

- **Interrupción.** En este caso se asume que el programa ha encontrado un error irrecuperable. La operación que dio lugar a la [excepción](#) se anula y se entiende que no hay manera de regresar al código que provocó la [excepción](#). Es decir, la operación que originó el error, se anula.
- **Reanudación.** Se puede manejar el error y regresar de nuevo al código que provocó el error.

Java emplea la primera forma, pero puede simularse la segunda mediante la utilización de un bloque `try` en el interior de un `while`, que se repetirá hasta que el error deje de existir. En la siguiente imagen tienes un ejemplo de cómo llevar a cabo esta simulación.

```
7 public static void main(String[] args){
8     boolean fueradelimites=true;
9     int i; //Entero que tomará valores aleatorios de 0 a 9
10    String texto[] = {"uno","dos","tre","cuatro","cinco"}; //String que representa la moneda
11
12    while(fueradelimites){
13        try{
14            i= (int) Math.round(Math.random()*9); //Generamos un indice aleatorio
15            System.out.println(texto[i]);
16            fueradelimites=false;
17        }catch (ArrayIndexOutOfBoundsException exc){
18            System.out.println("Fallo en el indice");
19        }
20    }
21 }
22 }
23 }
```

En este ejemplo, a través de la función de generación de números aleatorios se obtiene el valor del índice `i`. Con dicho valor se accede a una posición del array que contiene cinco cadenas de caracteres. Este acceso, a veces puede generar un error del tipo `ArrayIndexOutOfBoundsException`, que debemos gestionar a través de un `catch`. Al estar el bloque `catch` dentro de un `while`, se seguirá intentando el acceso hasta que no haya error.

1.3. Delegación de excepciones con throws.

¿Puede haber problemas con las excepciones al usar llamadas a métodos en nuestros programas?

Efectivamente, si se produjese una [excepción](#) es necesario saber quién será el encargado de solucionarla.

Puede ser que sea el propio [método](#) llamado o el código que hizo la llamada a dicho [método](#).

Quizá pudiéramos pensar que debería ser el propio [método](#) el que se encargue de sus excepciones, aunque es posible hacer que la [excepción](#) sea resuelta por el código que hizo la llamada. Cuando un [método](#) utiliza una sentencia que puede generar una [excepción](#), pero dicha [excepción](#) no es capturada y tratada por él, sino que se encarga su gestión a quién llamó al [método](#), decimos que se ha producido **delegación de excepciones**.

Para establecer esta delegación, en la cabecera del [método](#) se declara el tipo de excepciones que puede generar y que deberán ser gestionadas por quien invoque a dicho [método](#). Utilizaremos para ello la sentencia **throws** y tras esa palabra se indica qué excepciones puede provocar el código del [método](#). Si ocurre una [excepción](#) en el [método](#), el código abandona ese [método](#) y regresa al código desde el que se llamó al [método](#). Allí se posará en el **catch** apropiado para esa [excepción](#). Su [sintaxis](#) es la siguiente:

```
public class delegacion_excepciones {  
  
    ...  
  
    public int leeañõ(BufferedReader lector) throws IOException,  
    NumberFormatException{  
  
        String linea = teclado.readLine();  
  
        Return Integer.parseInt(linea);  
  
    }  
  
    ...  
  
}
```

Donde **IOException** y **NumberFormatException**, serían dos posibles excepciones que el [método](#) **leeañõ** podría generar, pero que no gestiona. Por tanto, un [método](#) puede incluir en su cabecera un listado de excepciones que puede lanzar, separadas por comas.

Para saber más

Si deseas saber algo más sobre la delegación de excepciones, te proponemos el siguiente enlace:

[Excepciones y delegación de éstas.](#)

Además te volvemos a remitir al vídeo demostrativo sobre manejo de excepciones en Java que se incluyó en el epígrafe anterior, titulado "capturar una [excepción](#)".