

UNIDAD 6: EXCEPCIONES

ÍNDICE

1. INTRODUCCIÓN	2
2. ¿QUÉ ES UNA EXCEPCIÓN?	4
3. MANEJO DE EXCEPCIONES.....	8
1. ATRAPAR UNA EXCEPCIÓN.....	8
2. LANZAR UNA EXCEPCIÓN	12
3. TIPOS DE EXCEPCIONES	13
4. MÉTODOS DE INFORMACIÓN.....	15
4. EXCEPCIONES IOEXCEPTION CON CLASE SCANNER	17



1. INTRODUCCIÓN

Existe una regla de oro en el mundo de la programación:

EN LOS PROGRAMAS, OCURREN ERRORES

Esto es sabido. Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Cómo se maneja el error? ¿Quién lo maneja? ¿Puede recuperarlo el programa?

El lenguaje Java utiliza excepciones para proporcionar capacidades de manejo de errores. En esta lección aprenderás qué es una excepción, cómo lanzar y capturar excepciones, qué hacer con una excepción una vez capturada, y cómo hacer un mejor uso de las excepciones heredadas de las clases proporcionadas por el entorno de desarrollo de Java.

Por desgracia, ya nos hemos encontrado con excepciones a lo largo de estos meses como, por ejemplo:

- **ArrayIndexOutOfBoundsException** cuando se intenta acceder a una posición de un array que está fuera de los límites de este.
- **NullPointerException** cada vez que se utiliza una referencia null en donde se espera un objeto y este todavía no ha sido instanciado.
- **NumberFormatException** cuando se lee un número que no tiene el formato esperado.
- ...

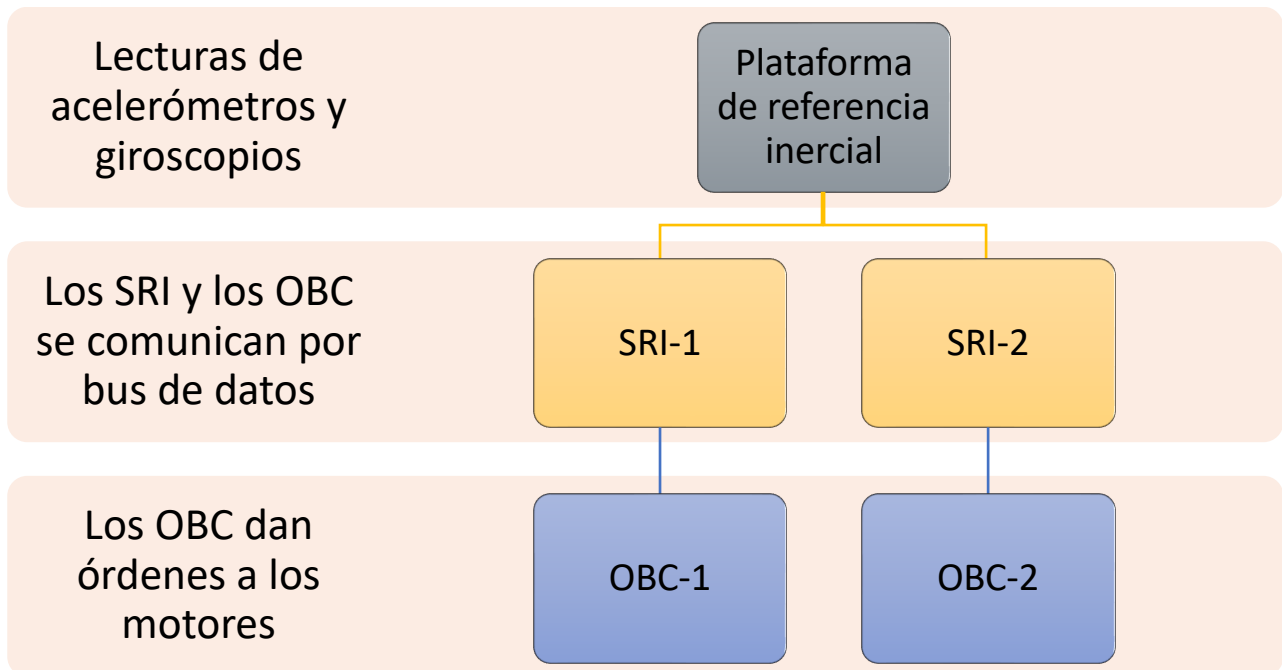
¿Es importante tratar las excepciones? Veámoslo con un ejemplo real:

https://www.youtube.com/watch?v=PK_yguLapgA

Caso del Ariane 5: Vuelo 501, 4 de junio de 1996. Coste: 500 millones de dólares.



SISTEMA DE CONTROL DE VUELO



El informe del equipo de investigación fue el siguiente:

Aproximadamente a los 39 segundos, el Ariane 5 comenzó a desintegrarse debido a los esfuerzos aerodinámicos excesivos ocasionados por un ángulo de ataque inapropiado. Esto condujo a la separación de los propulsores auxiliares respecto del cuerpo principal de la nave, por lo que se disparó el mecanismo de **AUTODESTRUCCIÓN** del cohete.

El ángulo de ataque resultó ser consecuencia de una completa desalineación entre las toberas de los propulsores auxiliares y la del motor principal. La desalineación de las toberas fue ordenada por el ordenador de abordo a partir de información transmitida por el sistema de referencia inercial (SRI-2). Parte de estos datos no contenían información de vuelo sino un patrón del SRI, el cual fue interpretado como información de vuelo.

La razón por la cual el SRI-2 no envió datos reales es que la unidad se había declarado en falla por una **excepción de software**.

El ordenador de a bordo, no pudo acceder al SRI-1 porque este se había desactivado algunos milisegundos antes por la misma razón que el SRI-2.

La excepción interna en el software del SRI fue ocasionada durante la ejecución de una instrucción de conversión de un valor en formato punto flotante (Double) de 64 bits a entero con signo de 16 bits, en circunstancias que el valor excedía la capacidad del tipo entero. La operación de conversión no estaba protegida aun cuando asignaciones de variables similares sí lo estaban...

La operación inválida generaba correctamente una excepción, sin embargo, el mecanismo de gestión de excepciones no fue diseñado para responder adecuadamente.

El mecanismo de tolerancia a fallas del Ariane 5 estaba concebido para tratar fallas aleatorias (por ejemplo, mal funcionamiento del hardware de uno de los ordenadores) pero no errores de diseño (ambas SRI experimentaron el mismo comportamiento al momento de producirse la falla).

El equipo de investigación sugirió cambiar el **enfoque de desarrollo** utilizado hasta entonces:

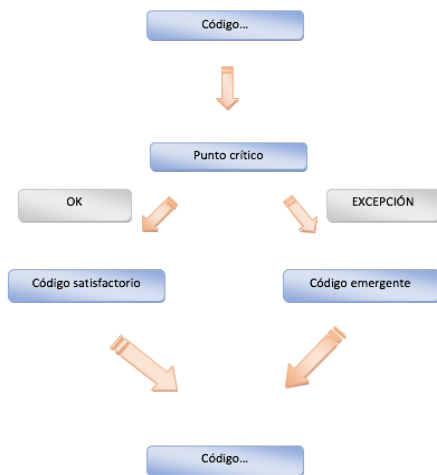
“El software se considera correcto hasta que se demuestra que falla” por el enfoque “se asume que el software es defectuoso hasta que, tras aplicar métodos aceptados como buenas prácticas, se logre demostrar que está correcto”.

2. ¿QUÉ ES UNA EXCEPCIÓN?

Una **excepción** es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre “excepción” implica que el problema ocurre con poca frecuencia; si la “regla” es que una instrucción generalmente se ejecuta en forma correcta, entonces la “excepción a la regla” es cuando ocurre un problema. El manejo de excepciones permite crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Los aspectos que estudiaremos en esta unidad, permiten a los programadores escribir **programas tolerantes a fallas** y **robustos** (es decir, programas que traten con los problemas que puedan surgir sin dejar de ejecutarse).



Todo programador debe, por tanto, contemplar el caso ideal pero también las contingencias que puedan ocurrir y cómo corregir el rumbo.



Una excepción en Java es **cualquier objeto** de clase `java.lang.Exception` o subclase de la misma (si el programador desea crear **sus propias excepciones** deberá entonces crear subclases de la clase mencionada).

```

public class MiExcepcion extends Exception{

    public MiExcepcion ( ) {
        super ("Esta es una excepción provocada");
    }

}
  
```

¿Qué ocurre con una excepción?

- Cuando se genera una excepción el intérprete crea un objeto para representar la excepción.
- Envía (lanza) el objeto (la excepción) al método que ha provocado la excepción.
- Si el método no captura la excepción debe lanzarla hasta que alguno la capture y trate. Si al final, ningún método la captura, entonces el intérprete la captura y realiza las acciones pertinentes (detención del programa y avisos o mensajes por pantalla).

Java tiene un sistema para que el programador defina la captura y gestión de las excepciones.

Cuando un error ocurre dentro de un método Java, el método crea un objeto 'exception' y lo maneja fuera, en el sistema de ejecución. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología java, crear un objeto exception y manejarlo por el sistema de ejecución se llama **lanzar una excepción**.

Después de que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la excepción. El conjunto de "algunos" métodos posibles para manejar la excepción es el conjunto de métodos de la pila de llamadas del método donde ocurrió el error. **El sistema de ejecución busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el "manejador de excepción" adecuado (catch).**

Un manejador de excepción es considerado adecuado si el tipo de la excepción lanzada es el mismo que el de la excepción manejada por el manejador. Así la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido **captura la excepción**.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado, el sistema de ejecución finaliza (y consecuentemente y el programa Java también).

Mediante el uso de excepciones para manejar errores, los programas Java tienen las siguientes ventajas:

- ✓ Separar el Manejo de Errores del Código "Normal".
- ✓ Propagar los Errores sobre la Pila de Llamadas.
- ✓ Agrupar los Tipos de Errores y la Diferenciación de éstos.

Si no gestionas excepciones te pueden pasar cosas como esta:

```
public class EjemExcepciones {
    public static void main(String[] args) {
        int a[] = new int[3];
        for ( inti = 0; i<= 3; i++){
            a [i] = i;
        }
        metodo( a );
    }

    static void método (int b[ ]) {
        b [2] = 1 / b[0];
    }
}
```

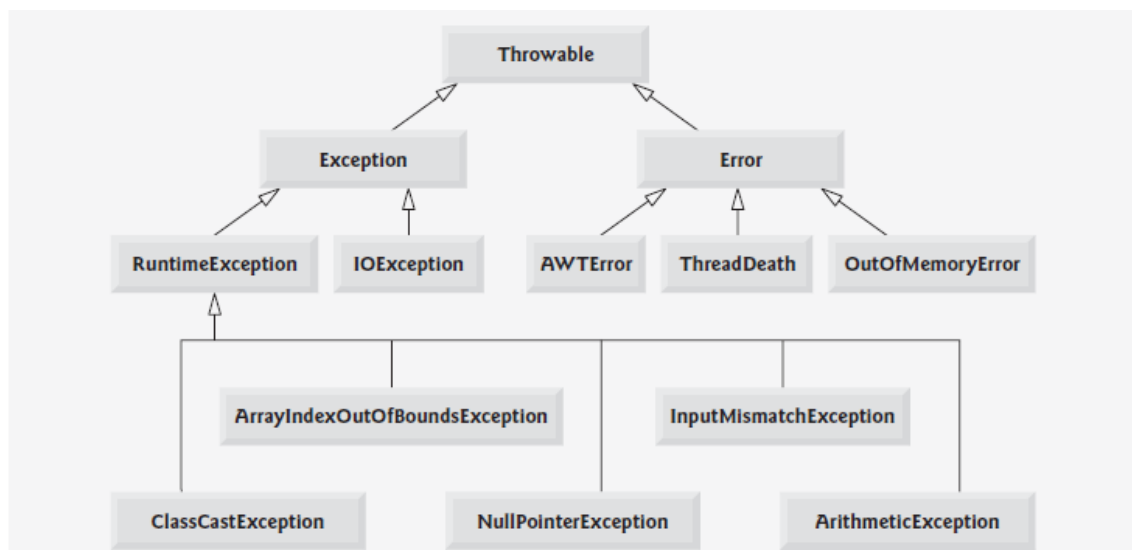
java.lang.ArrayIndexOutOfBoundsException: 3at
EjemExcepciones.EjemExcepciones.main(EjemExcepciones.java:7)
Exception in thread "main"

El intérprete lanza una excepción del tipo (clase) `ArrayIndexOutOfBoundsException`, es decir, el error se ha producido por intentar acceder a una posición externa al array. Además el intérprete nos informa de la posición o índice incorrecto dentro de la matriz (3), así como del método que ha provocado la excepción.

java.lang.ArithmeticException: / by zero
atEjemExcepciones.EjemExcepciones.fun(EjemExcepciones.java:11)
atEjemExcepciones.EjemExcepciones.main(EjemExcepciones.java:7)

El intérprete lanza una excepción del tipo (clase) `ArithmeticException`, es decir, el error se produce al dividir uno por cero. El intérprete nos informa además del método que produjo la excepción

Jerarquía de excepciones en Java.



En la figura muestra una pequeña porción de la jerarquía de herencia para la clase **Throwable** (una subclase de **Object**), que es la superclase de la clase **Exception**. Sólo pueden usarse objetos **Throwable** con el mecanismo para manejar excepciones. La clase **Throwable** tiene dos subclases: **Exception** y **Error**. La clase **Exception** y sus subclases (por ejemplo, **RuntimeException**, del paquete **java.lang**, e **IOException**, del paquete **java.io**) representan situaciones excepcionales que pueden ocurrir en un programa en Java, y que pueden ser atrapadas por la aplicación. La clase **Error** y sus subclases (por ejemplo, **OutOfMemoryError**) representan situaciones anormales que podrían ocurrir en la JVM. Los errores tipo **Error** ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones; por lo general, no es posible que las aplicaciones se recuperen de los errores tipo **Error**.

[Nota: la jerarquía de excepciones de Java contiene cientos de clases. En la API de Java se puedes encontrar información acerca de las clases de excepciones de Java]

Java clasifica a las excepciones en dos categorías: **excepciones verificadas (o comprobadas)** y **excepciones no verificadas (o no comprobadas)**. Esta distinción es importante, ya que el compilador de Java implementa un **requerimiento de atrapar o declarar** para las excepciones verificadas.

El tipo de una excepción determina si es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase **RuntimeException** (paquete **java.lang**) son excepciones **no verificadas**. Esto incluye a las excepciones que ya hemos visto, como las excepciones **ArrayIndexOutOfBoundsException** y **ArithmeticException**.

Todas las clases que heredan de la clase **Exception** pero no de la clase **RuntimeException** se consideran como excepciones **verificadas**; y las que heredan de la clase **Error** se consideran como no verificadas. El compilador *verifica* cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza excepciones verificadas. De ser así, el compilador asegura que la excepción verificada sea atrapada o declarada en una cláusula **throws**.

La cláusula throws especifica las excepciones que lanza un método. Dichas excepciones no se atrapan en el cuerpo del método.

- Para satisfacer la parte relacionada con *atrapar* del requerimiento de atrapar o declarar, el código que genera la excepción debe envolverse en un bloque **try**, y debe proporcionar un manejador **catch** para el tipo de excepción verificada (o uno de los tipos de su superclase).
- Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula **throws** que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo.
- Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error, indicando que la excepción debe ser atrapada o declarada. Esto obliga a los programadores a **pensar** acerca de los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza excepciones verificadas. Las clases de excepciones se definen para verificarse cuando se consideran lo bastante importantes como para atraparlas o declararlas.

Mis neuronas eligiendo el nombre de una variable



Aah pero cuando eligen el nombre de una variable para iterar



$$\begin{aligned}
 0^0 &= x \\
 \sqrt[0]{0^0} &= \sqrt[0]{x} \\
 0 &= \sqrt[0]{x} \quad | : \sqrt[0]{x} \\
 0 &= \frac{\sqrt[0]{x}}{\sqrt[0]{x}} \\
 0 &= 1
 \end{aligned}$$

QUANTUM, OFFICER?



A diferencia de las excepciones verificadas, el compilador de Java no verifica el código para determinar si una excepción no verificada es atrapada o declarada. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada. Por ejemplo, la excepción `ArithmeticException` no verificada que lanza el método cociente puede evitarse si el método se asegura que el denominador no sea cero antes de tratar de realizar la división. No es obligatorio que se listen las excepciones no verificadas en la cláusula `throws` de un método; aun si se listan, no es obligatorio que una aplicación atrape dichas excepciones.

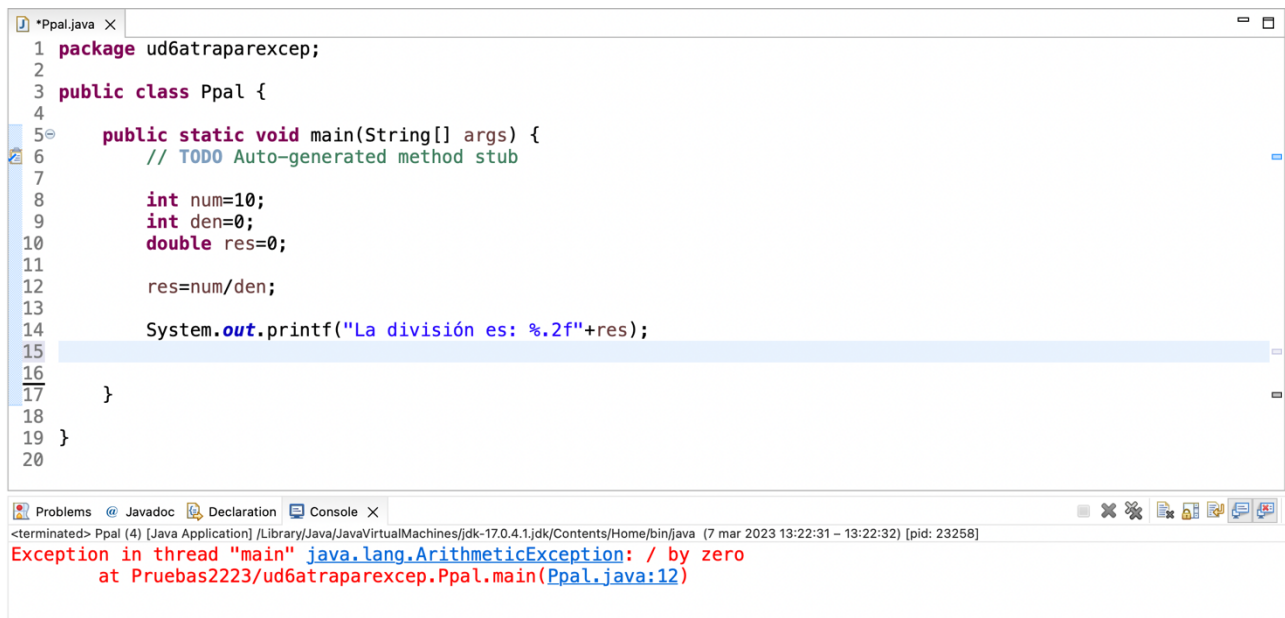
3. MANEJO DE EXCEPCIONES

1. ATRAPAR UNA EXCEPCIÓN

Cuando un método se encuentra con una anomalía que no puede resolver, lo lógico es que lance (throw) una excepción, esperando que quien lo llamó directa o indirectamente la atrape (catch) y maneje la anomalía. Incluso él mismo podría atrapar y manipular dicha excepción. Si la excepción no se atrapa, el programa finalizará automáticamente.

Veamos un ejemplo concreto:

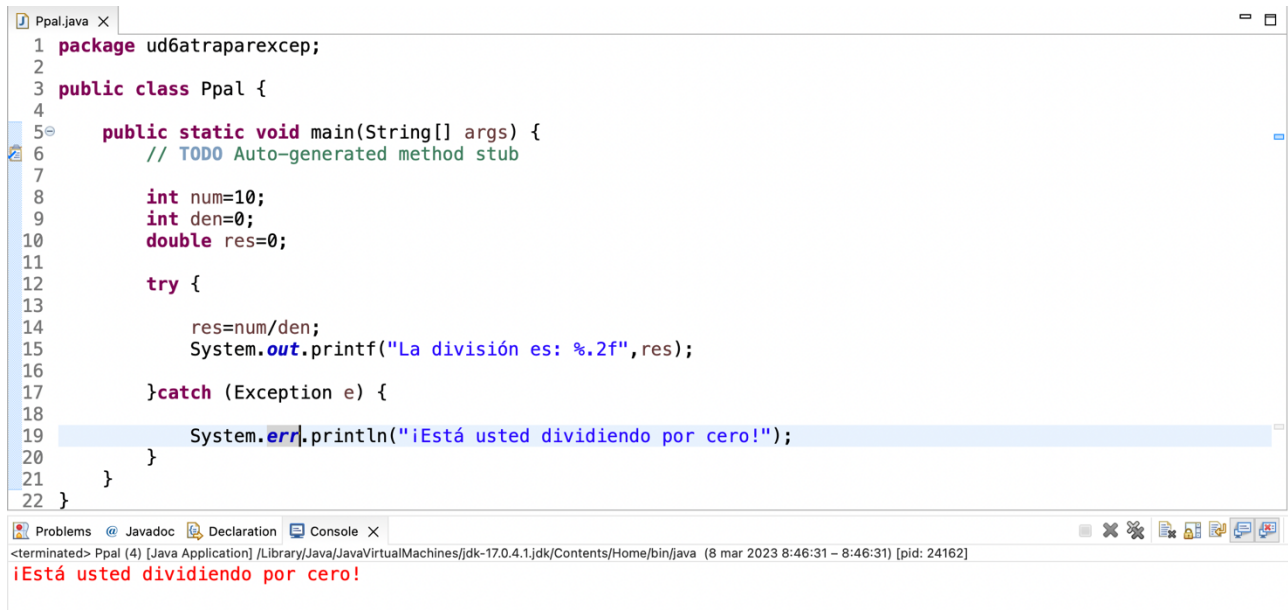
a) Primero sin capturar la excepción



```
1 package ud6atraparexcep;
2
3 public class Ppal {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         int num=10;
9         int den=0;
10        double res=0;
11
12        res=num/den;
13
14        System.out.printf("La división es: %.2f"+res);
15    }
16 }
17
18 }
19 }
20 }
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Pruebas2223/ud6atraparexcep.Ppal.main(Ppal.java:12)

b) Atrapando la excepción



```
1 package ud6atraparexcep;
2
3 public class Ppal {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         int num=10;
9         int den=0;
10        double res=0;
11
12        try {
13            res=num/den;
14            System.out.printf("La división es: %.2f", res);
15        } catch (Exception e) {
16            System.err.println("¡Está usted dividiendo por cero!");
17        }
18    }
19 }
20
21
22 }
```

Problems Javadoc Declaration Console X

<terminated> Ppal (4) [Java Application] /Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java (8 mar 2023 8:46:31 - 8:46:31) [pid: 24162]

¡Está usted dividiendo por cero!

1) Bloque try:

En la parte try debe escribirse la parte de código que es susceptible de producir una excepción, es decir, parte del código donde se puede dar un error, no solo la línea en cuestión, se suele poner todo el código relacionado para no tener que escribir muchos bloques try, cosa que hace menos fácil de leer el código.

Detrás del bloque try siempre debe ir un bloque catch o un bloque finally.

En este bloque, cuando se genera una excepción, automáticamente el flujo del programa pasa a la llave final del try y comienza a buscar que bloque catch puede atrapar la excepción.

2) Bloque catch:

Cuando se genera una excepción, la ejecución del programa pasa al bloque catch cuyo parámetro coincida con el tipo de excepción generada.

Obligatoriamente deben ir con el bloque try. Aquí va el trozo de código que debe atrapar la excepción lanzada en el bloque try.

Después de la palabra catch se abren paréntesis y dentro de estos se debe escribir el tipo de excepción que atrapa ese bloque.

OJO: Se pueden escribir varios bloques catch uno detrás de otro, tantos como tipos de excepciones queramos atrapar, pero siempre se debe seguir un orden determinado, **el primer catch debe ser el dedicado a la excepción MÁS CONCRETA** (más abajo en la jerarquía de excepciones) y así hacia abajo hasta llegar a la última

que debe ser el tipo de excepción más genérica, por ejemplo, la última debería ser la Exception. Si, por ejemplo, la más genérica fuera la primera (más arriba en el código) este catch atraparía todas las excepciones.

Dentro de cada bloque irá el código o mensaje que se mostrará cuando tenga lugar ese tipo de excepción o se lanzará una nueva excepción.

Cada catch solo puede atrapar un tipo de excepción, no se pueden escribir más de un tipo dentro de los paréntesis.

Tampoco es posible escribir código entre los bloques try-catch.

NOTA: En nuevas versiones de Java se pueden usar los bloques multicatch. Se deja esto para que lo investigues por tu cuenta.

3) Bloque Finally:

Este bloque se ejecuta SIEMPRE, se genere la excepción o no, con lo que este bloque se utiliza para realizar alguna acción obligatoria, por ejemplo, cerrar un fichero o liberar recursos. **Va siempre al final**, es decir, detrás de los bloques catch.

Otro ejemplo de lectura:

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            throwException(); // call method throwException
        } // end try
        catch ( Exception exception ) // exception thrown by throwException
        {
            System.err.println( "Exception handled in main" );
        } // end catch
        doesNotThrowException();
    } // end main
    // demonstrate try...catch...finally
    public static void throwException() throws Exception
    {
        try // throw an exception and immediately catch it
        {
            System.out.println( "Method throwException" );
            throw new Exception(); // generate exception
        } // end try
    }
}
```

```
catch ( Exception exception ) // catch exception thrown in try
{
    System.err.println(
        "Exception handled in method throwException" );
    throw exception; // rethrow for further processing

    // any code here would not be reached, exception rethrown in catch
} // end catch

finally // executes regardless of what occurs in try...catch
{
    System.err.println( "Finally executed in throwException" );
} // end finally

// any code here would not be reached
} // end method throwException

// demonstrate finally when no exception occurs
public static void doesNotThrowException()
{
    try // try block does not throw an exception
    {
        System.out.println( "Method doesNotThrowException" );
    } // end try

    catch (Exception exception ) // does not execute
    {
        System.err.println( exception );
    } // end catch

    finally // executes regardless of what occurs in try...catch
    {
        System.err.println(
            "Finally executed in doesNotThrowException" );
    } // end finally

    System.out.println( "End of method doesNotThrowException" );
} // end method doesNotThrowException
} // end class UsingExceptions
```

2. LANZAR UNA EXCEPCIÓN

Aquellos métodos que puedan lanzar una excepción deben declararla. Ya hemos visto cómo se atrapan.

Para lanzarla al método que lo llamó basta con añadir la palabra “throws” después del paréntesis donde ponemos los parámetros del método y el tipo de excepción lanzada. Si el método lanza más de una se pueden añadir separadas por comas antes del cuerpo del método.

Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o mediante métodos que se llamen desde el cuerpo. Un método puede lanzar excepciones de las clases que se listen en su cláusula throws, o en la de sus subclases.

Para lanzar la excepción dentro del código se usará la palabra throw seguida de una referencia al objeto excepción.

Por ejemplo, si no deseamos que el método escribirDatos atrape las excepciones que se produzcan dentro de él, deberíamos escribir (la excepción IOException se puede dar al abrir o cerrar un fichero):

```
public static void escribirDatos(CBanco banco, String fich) throws IOException
{
    PrintWriter fcli = null;
    CCuenta cliente;
    CListaClientes lista = new CListaClientes(banco.longitud());
    for (int i = 0; i < banco.longitud(); i++)
    {
        cliente = banco.clienteEn(i);
        lista.añadir(cliente.obtenerNombre(), i);
    }
    // Abrir el fichero para escribir. Se crea el flujo fcli;
    fcli = new PrintWriter(new FileWriter(fich));
    lista.escribir(fcli);

    // Cerrar el fichero
    if (fcli != null) fcli.close();
}
```

Vemos que no hay bloques catch, que atrape la excepción IOException. Esta forma de proceder obligará a cualquier método que invoque a escribirDatos a atrapar la excepción, por ejemplo, en el main:

```
public static void main(String[] args)
{
    ...
    case 8: // escribir
        try
        {
            flujoS.print("Fichero: "); nombre = Leer.dato();
            escribirDatos(banco, nombre);
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
        break;
    ...
}
```

3. TIPOS PROPIOS DE EXCEPCIONES

Una de las cosas más importantes es conocer los distintos tipos de excepciones que pueden ocurrir. Java proporciona muchas clases en el API para la gestión de excepciones, pero también, el programador puede crear sus propias excepciones, sobre esto, se puede leer más en el apartado 13.11 del libro Deitel.

El programador puede crear sus propias excepciones y utilizarlas para indicar errores, pueden extender de Exception (llamadas comprobadas) o de RuntimeException (llamadas No Comprobadas).

```
public static class MiExcepción extends Exception {}
```

Si queremos asociar mensajes con información adicional a una excepción nuestra, debemos escribir los constructores:

```
public static class MiExcepción extends Exception {
    // constructor sin argumentos
    public MiExcepción() {
        super();
    }
    // constructor con un string como argumento
    public MiExcepción(String infoAdicional) {
        super(infoAdicional);
    }
}
```

```
}
```

```
}
```

Un método donde se lanza una excepción comprobada, deberá:

- Tratarla con un bloque try-catch
- O declarar que la lanza con una cláusula throws

Sintaxis de la cláusula **throws**:

```
public tipo nombreMétodo(parámetros) throws ClaseExcepción1, ClaseExcepción2
{
    declaraciones;
    instrucciones; // lanzan las excepciones
}
```

Cláusula throws en métodos anidados

```
public void metodo3() throws MiExcepción {
    ...
    throw new MiExcepción();
}
```

Lanza la excepción, por eso lo indica

```
public void metodo2() throws MiExcepción {
    ...
    metodo3();
}
```

No trata la excepción, por eso indica que puede lanzarla

```
public void metodo1() {
    ...
    try {
        metodo2();
    } catch (MiExcepción e) {
        ...
    }
}
```

Trata la excepción, por eso no tiene la cláusula throws

Otro ejemplo de propagación de excepciones propias muy sencillo:

```
class SinGasolina extends Exception {  
    SinGasolina( String s ) {    // constructor  
        super( s );  
    }  
    ....  
}  
// Cuando se use, aparecerá algo como esto  
try {  
    if( j < 3 )//j es el número de litros que quedan  
        throw new SinGasolina( "Usando depósito de reserva" );  
}  
catch( SinGasolina e ) {  
    System.out.println( o.getMessage() );  
}
```

4. MÉTODOS DE INFORMACIÓN

Ejemplo:

```
public class UsingExceptions  
{  
    public static void main( String args[] )  
    {  
        try  
        {  
            method1(); // call method1  
        } // end try  
        catch ( Exception exception ) // catch exception thrown in method1  
        {  
            System.err.printf( "%s\n\n", exception.getMessage() );  
            exception.printStackTrace(); // Imprime el rastreo de la pila  
            // Obtiene la información del rastreo de la pila  
            StackTraceElement[] traceElements = exception.getStackTrace();  
            System.out.println( "\nStack trace from getStackTrace:" );  
            System.out.println( "Class\t\tFile\t\tLine\tMethod" );  
            // itera a través de elementosRastreo para obtener la descripción de la excepción  
            for ( StackTraceElement element : traceElements )  
            {
```

```
        System.out.printf( "%s\t", element.getClassName() );
        System.out.printf( "%s\t", element.getFileName() );
        System.out.printf( "%s\t", element.getLineNumber() );
        System.out.printf( "%s\n", element.getMethodName() );
    } // end for
} // end catch
} // end main
// Llama al método 2. Lanza las excepciones de vuelta al main.
public static void method1() throws Exception
{
    method2();
} // end method method1
// Llama al método e. Lanza las excepciones de vuelta al método 1
public static void method2() throws Exception
{
    method3();
} // end method method2
// Lanza la excepción Exception de vuelta al método 2
public static void method3() throws Exception
{
    throw new Exception( "Exception thrown in method3" );
} // end method method3
} // end class UsingExceptions
```

Salida:

La excepción se lanzó en método3

java.lang.Exception: La excepción se lanzó en método3
at UsoDeExcepciones.metodo3(UsoDeExcepciones.java:49)
at UsoDeExcepciones.metodo2(UsoDeExcepciones.java:43)
at UsoDeExcepciones.metodo1(UsoDeExcepciones.java:37)
at UsoDeExcepciones.main(UsoDeExcepciones.java:10)

Rastreo de la pila de getStackTrace:

Clase Archivo Línea Método

UsoDeExcepciones UsoDeExcepciones.java 49 metodo3
UsoDeExcepciones UsoDeExcepciones.java 43 metodo2
UsoDeExcepciones UsoDeExcepciones.java 37 metodo1
UsoDeExcepciones UsoDeExcepciones.java 10 main

4. EXCEPCIONES IOEXCEPTION CON CLASE SCANNER

Imaginad que estoy haciendo un programa que lee números int del teclado y quiero tratar los errores:

- cuando introducen un char;
- cuando introducen un float;
- cuando introducen un entero más grande que int;

```
public static void Menu()
{
    leer= new Scanner (System.in);
}

public static int lee()
{
    int opciones=0;
    try
    {
        opciones = leer.nextInt();
    }
    Catch (NumberFormatException e) {
        System.err.println("Error");
    }

    return opciones;
}

public static void main(String args[])
{
    opcion= lee();
}
```

¡NO FUNCIONA! ¿Qué ocurre?

RESPUESTA:

El método scanner.nextInt () no lanza excepciones, por lo que nunca te saltarán, escribas lo que escribas.

Hay métodos que lanzan excepciones (y en la declaración del método se pone un **throws**). Por ejemplo, el constructor de FileWriter, lleva un throws IOException. Normalmente, si el método no pone throws alguna excepción, no lanzará nada y no tiene sentido poner el try-catch (no sirve para nada).

En otras ocasiones pueden saltar excepciones que no están controladas. Los métodos no dicen que las lanzan, pero ocurren normalmente por algún error de programación. Por ejemplo, un `NullPointerException` (Objeto sin instanciar)

En nuestro caso, para resolver esto, en vez de leer un `int` con `scanner`, que no lanza excepciones, intenta leer un `String` y luego convertirlo tú a entero con un `Integer.parseInt()`, que sí lanza excepciones (eso es lo que solemos hacer nosotros).

Código:

```
try {  
    String cadena = leer.nextLine();  
    int opciones = Integer.parseInt(cadena); // Este sí lanza excepciones  
} catch (NumberFormatException e) {  
    ...  
}
```

