Unidad 5. Excepciones

Contenido

[1 Manejo de errores con excepciones 2](#_Toc125663039)

[1.1 ¿Qué es un Excepción y Por Qué Debo Tener Cuidado? 2](#_Toc125663040)

[1.1.1 Ventaja 1: Separar el Manejo de Errores del Código "Normal" 4](#_Toc125663041)

[1.1.2 Ventaja 2: Propagar los Errores sobre la Pila de Llamadas 6](#_Toc125663042)

[1.1.3 Ventaja 3: Agrupar Errores y Diferenciación 9](#_Toc125663043)

[2 Empezando a entender las Excepciones Java 10](#_Toc125663044)

[2.1 Api de Excepciones de Java 10](#_Toc125663045)

[2.2 Capturar y Manejar Excepciones 13](#_Toc125663046)

[2.2.1 El Bloque Try 13](#_Toc125663047)

[2.2.2 Los Bloques catch 15](#_Toc125663048)

[2.2.3 Capturar Varios Tipos de Excepciones con un Manejador 18](#_Toc125663049)

[2.2.4 El stackTrace o traza de pila de ejecución 22](#_Toc125663050)

[2.2.5 El Bloque finally 23](#_Toc125663051)

[2.2.6 Poniéndolo todo Junto 25](#_Toc125663052)

[3 Creación y propagación de excepciones con un ejemplo práctico. 26](#_Toc125663053)

[3.1 Jerarquía de excepciones 26](#_Toc125663054)

[3.2 Las excepciones 30](#_Toc125663055)

[3.2.1 Supresión de excepciones 32](#_Toc125663056)

[3.3 Throw throwable. La clase Impuestos 33](#_Toc125663057)

[3.4 La clase Empleado 36](#_Toc125663058)

[3.5 La clase EntradaDeDatos 37](#_Toc125663059)

[3.6 La clase Principal App. 40](#_Toc125663060)

[3.7 Ejercicio 42](#_Toc125663061)

[4 Anexo I. Excepciones en Java 42](#_Toc125663062)

# Manejo de errores con excepciones

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 errore**s. 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.

## ¿Qué es un Excepción y Por Qué Debo Tener Cuidado?

El término **excepción** es una forma corta da la frase "suceso excepcional" y puede definirse de la siguiente forma.

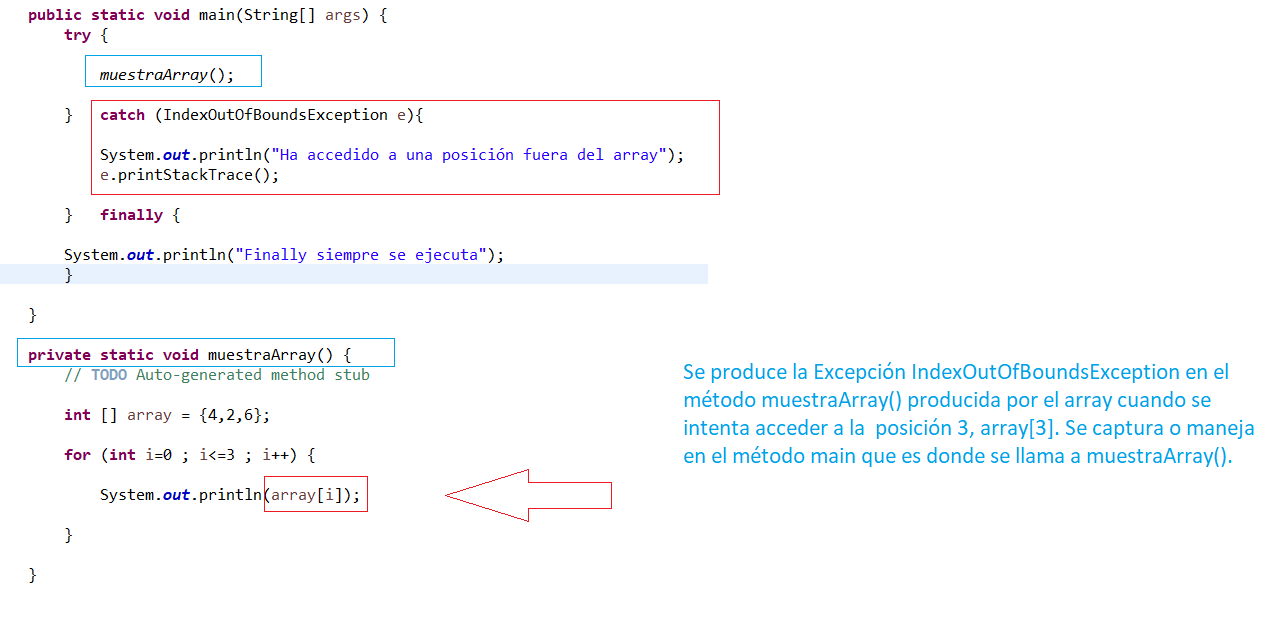
**Definición**: Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.

**Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro**, a los simples errores de programación**, como tratar de acceder a un elemento de un array** fuera de sus límites. Cuando **dicho 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 de tipo 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.

Un **manejador de excepción es considerado adecuado** para **manejar una excepción** si el tipo de la excepción lanzada es el **mismo que el de la excepción controlada 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**.

En el siguiente ejemplo se muestra claramente como del **método muestraarray** salta la **excepción hasta el manejador en la función main porque es adecuado,** controla el **outofbounds del array**, el **salirse de los límites del array**. En la pila **de llamadas estamos en la función muestraArray()** pero **el manejador de la excepción está en la función main()** y la JVM, máquina virtual, **va buscando la primera llamada que puede manejar esa excepción de arrays**. El **programa java continuará su ejecución ahí,** en **el bloque catch de manejo de la excepción** en el método **main()** y la **función se da por terminada**.



En la **pila de ejecución tenemos en ejecución el método muestraArray().** Cuando se produce la **excepción la JVM va a intentar buscar el primer manejador** de **excepción valido.** Si **no lo encuentra mandará un error y el programa se acaba**. Si **lo encuentra la JVM pasa inmediatamente el flujo** de ejecución **al método que tiene el manejador de excepciones**, en este **caso main**, sin terminar el otro método, pues tiene **algún tipo de problema o error**. El **flujo de ejecución es pasado directamente al bloque catch** que maneja **esa excepción**.

Pila de ejecución del Java Run Time

|  |  |  |
| --- | --- | --- |
| **función en ejecución** | **Excepción producida** | **Manejador de excepciones** |
| muestraArray() | ArrayIndexOutOfBounds | Se propaga la excepción |
| Main() |  | catch (ArrayIndexOutOfbounds e) |

Además, el **ejemplo es un buen ejemplo porque es interesante manejar** las **excepciones a nivel de hilo o programa principal**. Como no se abordan los hilos hasta segundo curso, dejémoslo en intentar manejar las excepciones lo más cerca del programa principal o hilo principal. Cuando veáis **controladores en el modelo vista-controlador**, se indicará que las excepciones **también se pueden abordar ahí**.

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** frente a las técnicas de manejo de errores tradicionales.

### Ventaja 1: Separar el Manejo de Errores del Código "Normal"

En la **programación tradicional, la detección, el informe y el manejo de errores** se **convierte en un código muy lioso**. Por ejemplo, supongamos que tenemos una función que lee un fichero completo dentro de la memoria. **En pseudo-código**, la función se podría parecer a la siguiente.

leerFichero {

abrir el fichero;

determinar su tamaño;

asignar suficiente memoria;

leer el fichero a la memoria;

cerrar el fichero;

}

A **primera vista esta función parece bastante sencill**a, pero ignora **todos aquello errores potenciales.**

* ¿Qué sucede si no se puede abrir el fichero?
* ¿Qué sucede si no se puede determinar la longitud del fichero?
* ¿Qué sucede si no hay suficiente memoria libre?
* ¿Qué sucede si la lectura falla?
* ¿Qué sucede si no se puede cerrar el fichero?

Para **responder a estas cuestiones dentro de la función**, tendríamos que **añadir mucho código para la detección y el manejo de errores**. El aspecto final de la función se parecería al siguiente. Tiene demasiadas sentencias condicionales anidadas, no sería un código limpio

codigodeError **leerFichero {**

inicializar codigodeError = 0;

**abrir el fichero;**

if (ficheroAbierto) {

**determinar la longitud del fichero;**

if (obtenerLongitudDelFichero) {

**asignar suficiente memoria;**

if (obtenerSuficienteMemoria) {

**leer el fichero a memoria;**

if (falloDeLectura) {

codigodeError = -1;

}

} else {

codigodeError = -2;

}

} else {

codigodeError = -3;

}

**cerrar el fichero;**

if (ficheroNoCerrado && codigodeError == 0) {

codigodeError = -4;

} else {

codigodeError = codigodeError and -4;

}

} else {

codigodeError = -5;

}

return codigodeError;

**}**

**Con la detección de errores, las 7 líneas originales (en negrita**) se **han convertido en 29 líneas de código-- a aumentado casi un 400 %.** Lo peor, **existe tanta detección y manejo de errores y de retorno que en las 7 líneas originales** y el código está totalmente plagado de este control de errores. Y aún peor, **el flujo lógico del código también se pierde**, haciendo difícil poder decir si el código hace lo correcto (si ¿se cierra el fichero realmente si falla la asignación de memoria?). Incluso **es difícil asegurar que el código continue haciendo las cosas correctas** cuando **se modifique la función tres meses después de haberla escrito**. Muchos programadores "resuelven" este problema ignorádlo-- se informa de los errores cuando el programa no funciona.

Java proporciona una solución elegante al problema del tratamiento de errores: las **excepciones**. Las excepciones **te permiten escribir el flujo principal de su código y tratar los casos excepcionales en otro lugar**. Si la función **leerFichero** utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto.

**leerFichero {**

try {

abrir el fichero;

determinar su tamaño;

asignar suficiente memoria;

leer el fichero a la memoria;

cerrar el fichero;

} catch (falloAbrirFichero) {

hacerAlgo;

} catch (falloDeterminacionTamaño) {

hacerAlgo;

} catch (falloAsignaciondeMemoria) {

hacerAlgo;

} catch (falloLectura) {

hacerAlgo;

} catch (falloCerrarFichero) {

hacerAlgo;

}

**}**

**Observa que las excepciones no evitan el esfuerzo de hacer el trabajo de detectar, informar y manejar errores**. Lo que proporcionan las excepciones es la posibilidad de separar los detalles oscuros de qué hacer cuando ocurre algo fuera de la normal.

Además, **el factor de aumento de codigo de este es programa es de un 250%** -- comparado con el 400% del ejemplo anterior. Además el código es mucho más limpio y legible.

### Ventaja 2: Propagar los Errores sobre la Pila de Llamadas

Una segunda ventaja de las excepciones es la posibilidad del propagar el error encontrado **sobre la pila de llamadas a métodos**. Supongamos que el método **leerFichero** es el cuarto método en **una serie de llamadas a métodos anidadas realizadas** por un **programa principal**: **metodo1** llama a **metodo2**, que llama a **metodo3**, que finalmente llama a **leerFichero**.

metodo1 {

call metodo2;

}

metodo2 {

call metodo3;

}

metodo3 {

call leerFichero;

}

Supongamos también que **metodo1** es el **único método interesado en el error** que ocurre dentro de **leerFichero**. Tradicionalmente **las técnicas de notificación del error forzarían** a **metodo2** y **metodo3** a **propagar el código de error devuelto** ( que devuelvan la cadena “Error” por ejemplo” en vez de la línea del fichero) por **leerFichero** sobre **la pila de llamadas** hasta que **el código de error llegue** finalmente a **metodo1** -- el único método que está interesado en él.

**metodo1 {**

codigodeErrorType error;

error = **call metodo2;**

if (error)

procesodelError;

else

proceder;

**}**

codigodeErrorType **metodo2 {**

codigodeErrorType error;

error = **call metodo3;**

if (error)

return error;

else

proceder;

**}**

codigodeErrorType **metodo3 {**

codigodeErrorType error;

error = **call leerFichero;**

if (error)

return error;

else

proceder;

**}**

**Como mencionamos anteriormente, el sistema de ejecución Java busca** hacia atrás en la **pila de llamadas para encontrar cualquier método** que esté interesado **en manejar una excepción particula**r. Un método **Java puede "esquivar" cualquier excepción lanzada dentro de él,** por lo tanto, **permite a los métodos que están por encima de él en la pila de llamadas poder capturarlo**. Sólo los métodos interesados en el error deben preocuparse de detectarlo.

**metodo1 {**

try {

**call metodo2;**

} catch (excepcion) {

procesodelError;

}

**}**

**metodo2** throws excepcion **{**

**call metodo3;**

**}**

**metodo3** throws excepcion **{**

**call leerFichero;**

**}**

**Sin embargo, como se puede ver desde este pseudo-código**, requiere **cierto esfuerzo por parte de los métodos centrales**. Cualquier **excepción chequeada que pueda ser lanzada dentro de un método forma parte del interface de programación público del método** y debe ser especificado en la cláusula **throws** del método. **Así el método informa a su llamador sobre las excepciones** que **puede lanzar**, para **que el llamador pueda decidir qué hacer** con esa excepción.

Observa **de nuevo la diferencia del factor de aumento de código y el factor de ofuscación** entre las dos técnicas de manejo de errores. El código que **utiliza excepciones es más compacto y más fácil de entender.**

### Ventaja 3: Agrupar Errores y Diferenciación

**Frecuentemente las excepciones se dividen en categorías o grupos**. Por ejemplo, podríamos **imaginar un grupo de excepciones, cada una de las cuales representara un tipo de error específico** que pudiera ocurrir durante la **manipulación de un array: el índice está fuera del rango** del tamaño del array, el elemento que se quiere insertar en el array no es del tipo correcto, o el **elemento que se está buscando no está en el array**. Además, **podemos imaginar que algunos métodos querrían manejar todas las excepciones** de esa categoria (todas las excepciones de array), y otros métodos podría manejar sólo algunas excepciones específicas (como la excepción de índice no válido).

**Como todas las excepciones lanzadas dentro de los programas Java** son **objetos de primera clase, agrupar o categorizar las excepciones** es una **salida natural de las clases y las superclases**. Las **excepciones Java deben ser instancias de la clase Throwable**, o de cualquier descendiente de ésta. Como **de las otras clases Java, se pueden crear subclases de la clase Throwable y subclases de estas subclases**. Cada clase 'hoja' (una clase sin subclases) representa un tipo específico de excepción y cada clase 'nodo' (una clase con una o más subclases) representa un grupo de excepciones relacionadas.

IndexOutOfBoundsException, ArrayStoreException, y NullPointerException son todas **clases hija de RuntimeException**. **Cada una representa un tipo específico de error** que puede ocurrir cuando se manipula un array. Un **método puede capturar una excepción basada en su tipo específico** (su clase inmediata o interface). Por ejemplo, **un manejador de excepción que sólo controle la excepción de índice no válido**, tiene una sentencia **catch** como esta.

catch (IndexOutOfBoundsException e) {

. . .

}

**ArrayException es una clase base y representa cualquier error que pueda ocurrir durante la manipulación de un objeto array**, incluyendo aquellos errores representados específicamente por una de sus subclases. Un **método puede capturar una excepción basada en este grupo o tipo general** especificando **cualquiera de las superclases de la excepción** en la sentencia **catch**. Por ejemplo, para capturar todas las excepciones de tiempo de ejecución, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento RunTimeException.

catch (RunTimeException e) {

. . .

}

**Este manejador podría capturar todas las excepciones de tiempo de ejecución**, incluyendo las de array que hemos mencionado antes **IndexOutOfBoundsException, IndexOutOfBoundsException, y NullPointerException**. Se puede descubrir el tipo de excepción preciso que ha ocurrido comprobando el parámetro del manejador **e**. Incluso **podríamos seleccionar un manejador de excepciones** que controlara **cualquier excepción que ocurra en el programa** con este manejador. Para ello usamos la clase **base Exception que es hija directa Throwable directamente**

catch (Exception e) {

. . .

}

**Los manejadores de excepciones que son demasiado generales, como el mostrado aquí, pueden hacer que el código sea propenso a errores** mediante la captura y manejo de excepciones que no se hubieran anticipado y por lo tanto no son manejadas correctamente dentro de manejador. Como regla no se recomienda escribir manejadores de excepciones generales.

Como has visto, **se pueden crear grupos de excepciones y manejarlas de una forma general, o se puede especificar un tipo de excepción específico** para diferenciar **excepciones y manejarlas de un modo exacto**.

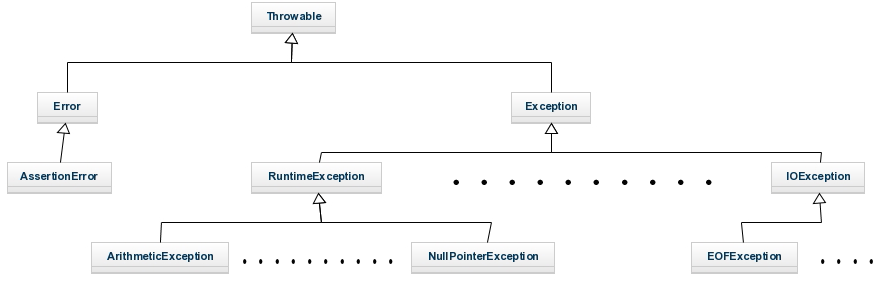
Ahora que has entendido qué son las excepciones y las ventajas de su utilización en los programas Java, **es hora de aprender cómo utilizarlas.**

# Empezando a entender las Excepciones Java

## Api de Excepciones de Java

En Java (al igual que en otros lenguajes de programación), **existen muchos tipos de excepciones** y **enumerar cada uno de ellos seria casi una labor infinita**. En lo referente a **las excepciones hay que decir que se aprenden a base experiencia**, de encontrarte con ellas y de saber solucionarlas.

Cuando en **Java se produce una excepción se crea un objeto de una determina clase** (dependiendo del tipo de error que se haya producido), **que mantendrá la información sobre el error producido** y nos **proporcionará los métodos necesarios para obtener dicha información**. Esto ya lo hemos visto. Además, existe un **modelo de clases para dar soporte a estos objetos**. Estas clases tienen como **clase padre o base es la clase Throwable**, por tanto, se mantiene una jerarquía en las excepciones. A continuación, mostramos algunas de las clases para que nos hagamos una idea de la jerarquía que siguen las excepciones, pero existen muchísimas más excepciones que las que mostramos:



**La clase Error se usa para errores irrecuperables y no la podemos capturar y manejar** **con un catch**, conforme indica la documentación oficial. **El programa acabaría con error**. La **clase Exception e hijas las podemos manejar como ya hemos visto**. Pero además, podemos distinguir entre dos tipos de excepciones en Java según si es necesario manejarlas explícitamente o no.

**Excepciones que no hace falta manejar explícitamente o “unchecked”**: todas las **excepciones que heredan de RunTimeException** no es necesario manejarlas o añadir un manejador. Es un proceso voluntario por parte del programador. Las **excepciones de Array que hemos visto antes** heredan de **RunTimeException,** son de este tipo. Si modificamos el ejemplo anterior de **muestraArray y quitamos los manejadores de excepciones** el **compilador no expone ningún problema.**

**public** **class** ExcepcionRuntime {

**public** **static** **void** main(String[] args) {

*muestraArray*();

}

**private** **static** **void** muestraArray() {

// **TODO** Auto-generated method stub

**int** [] array = {4,2,6};

**for** (**int** i=0 ; i<=3 ; i++) {

System.***out***.println(array[i]);

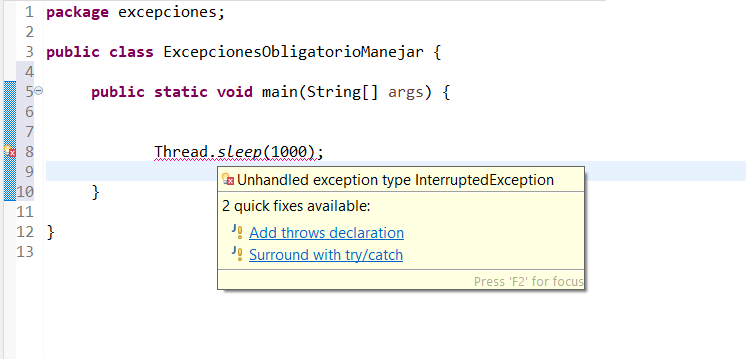
}

}

}

**Excepciones que hacen falta manejar explícitamente, o “checked”**: el compilador te **obliga a añadir una excepción.**

En el siguiente ejemplo **ExcepcionesObligatorioManejar.java** podéis comprobar como el compilador obliga a capturar la excepción **InterruptedException** para el método **Thread.sleep(1000)** que duerme el hilo principal durante un segundo. La excepción se produce si el hilo intenta ser activado por otro hilo mientras está durmiendo. Este código no lo entenderéis todavía pero aquí **lo importante es entender que es obligatorio capturar esta excepción**.



En la imagen **podéis ver que el compilador del IDE** nos indica que **debemos manejar esa excepción**. Con dos opciones: **try/(catch o añadir throws a la declaración** que veremos después. Podéis **ver la primera solución en el ejemplo posterior**. Añadimos un **catch para capturar la excepción. catch** (InterruptedException e)

Mostramos un mensaje de error con el método System.***out***.println y además **mostramos la traza de la pila**, que explicaremos más adelante con el método e.printStackTrace();

**package** excepciones;

**public** **class** ExcepcionesObligatorioManejar {

**public** **static** **void** main(String[] args) {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

System.***out***.println("Se ha producido un error al dormir el hilo principal");

e.printStackTrace();

}

}

}

## Capturar y Manejar Excepciones

[**El Bloque try**](http://programacion.com/cursos/excepciones/try.htm)

El primer paso en la escritura de una manejador de excepciones es poner la sentencia Java dentro de la cual se puede producir la excepción dentro de un bloque **try**. Se dice que el bloque **try** **gobierna** las sentencias encerradas dentro de él y define el ámbito de cualquier manejador de excepciones (establecido por el bloque **catch** subsecuente) asociado con él.

[**Los bloques catch**](http://programacion.com/cursos/excepciones/catch.htm)

Después se debe asociar un manejador de excepciones con un bloque **try** proporcionándole uno o más bloques **catch** directamente después del bloque **try**.

[**El bloque finally**](http://programacion.com/cursos/excepciones/finally.htm)

El bloque **finally** de Java proporciona un mecanismo que permite a sus métodos limpiarse a si mismos sin importar lo que sucede dentro del bloque **try**. Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema.

[**Poniéndolo Todo Junto**](http://programacion.com/cursos/excepciones/putittogether.htm)

Las secciones anteriores describen cómo construir los bloques de código **try**, **catch**, y **finally** para el **ejemplo de array con múltiples excepciones ExcepcionesEnArray**. Ahora, pasearemos sobre el código para investigar que sucede en varios escenarios.

### El Bloque Try

El **primer paso en la construcción de un manejador de excepciones** es encerrar **las sentencias que podrían lanzar una excepción** dentro de un bloque **try**. En general, este bloque se parece a esto.

try {

**sentencias Java**

}

El segmento de código etiquetado como **sentencias java** está compuesto **por una o más sentencias legales de Java** que podrían lanzar una excepción.

Existe **más de una forma de realizar esta tarea**. Podríamos **poner cada una de las sentencias que potencialmente pudieran lanzar una excepción** dentro de su propio bloque **try**, y **proporcionar manejadores de excepciones separados** para cada uno de los bloques **try**. O **podríamos poner todas las sentencias dentro** de un sólo bloque **try** y **asociar varios manejadores** con él que es lo que vamos a hacer. El s**iguiente ejemplo utiliza un sólo bloque** **try** para todo **el método porque el código tiende a ser más fácil de leer**. El ejemplo que expondremos entero al final es ExcepcionesEnArray.java.

En el **bloque try estamos forzando todas las posibles excepciones que un array puede lanzar** con un **switch case.**

**try** {

// Multiple condiciones en el switch

**switch** (i) {

**case** 0:

output = array[0]; // Genera NullPointerException.

**break**;

**case** 1:

array = **new** **int**[arrayInc]; // Genera NegativeArraySizeException.

output = array[arrayInc];

**break**;

**case** 2:

array = **new** **int**[arraySize]; // Genera ArrayIndexOutOfBoundsException.

output = array[arraySize];

**break**;

**case** 3:

array = **new** **int**[5]; // Genera IndexOutOfBoundsException.

output = array[5];

**case** 4:

Object newArray = **new** ~~Integer~~(0); // Genera ClassCastException.

System.***out***.println((String) newArray);

**case** 5:

Object X[] = **new** String[-5]; // Genera ArrayStoreException.

X[0] = **new** ~~Integer~~(0);

System.***out***.println(X);

}

}

Se dice que el bloque **try** **gobierna** las sentencias encerradas dentro del él y **define el ámbito de cualquier manejador de excepción** (establecido por su subsecuente bloque **catch**) asociado con él. En otras palabras, si **ocurre una excepción** dentro del bloque **try**, esta **excepción será manejada por el manejador de excepción asociado** con esta sentencia **try**.

Una sentencia **try** debe ir acompañada de al menos un bloque **catch** o un bloque **finally** como veremos más adelante. Los errores que estamos controlando para el array son;

1. **NullPointerException** - En Java, NullPointerException es una clase que también herada RuntimeException. Existen las siguientes condiciones donde se genera NullPointerException. Se producen cuando una aplicación intenta utilizar un objeto, pero la variable contiene el valor null.

2. **NegativeArraySizeException** - Este error se produce cuando alguien quiere crear un array con un tamaño negativo. NegativeArraySizeException es una clase en Java que hereda RuntimeException.

3. **ArrayIndexOutOfBoundsException** - Este tipo de error se genera cuando se ha accedido a un array con un índice ilegal. En otras palabras, cuando se accede a un array mediante un índice negativo o de valor mayor o igual que el tamaño del array. En Java es una clase Java que hereda de la clase IndexOutOfBoundException.

4. **IndexOutOfBoundsException** - Este tipo de excepción es lanzada por todos los tipos de datos de patrones de indexación, como una cadena o array, etc., cuando se accede fuera del índice (rango). IndexOutOfBoundException también es una clase en Java y hereda RuntimeException.

5. **ClassCastException** - ClassCastException se produce cuando el código siguiente ha intentado **convertir** o castear un objeto a una subclase de la que no es una instancia. ClassCastException también es una clase en Java y hereda RuntimeException.

### Los Bloques catch

Como se aprendió en el ejemplo anterior, la sentencia **try** define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia **try** proporcionando uno o más bloques **catch** directamente después del bloque **try**.

Podemos tener un solo bloque catch:

try {

// Protected code

} catch (ExceptionName e1) {

// Catch block

}

O múltiples bloques catch para tratar una sóla excepción.

try {

. . .

} catch ( . . . ) {

. . .

} catch ( . . . ) {

. . .

} . . .

No puede haber ningún código entre el final de la sentencia **try** y el principio de la primera sentencia **catch**. La forma general de una sentencia **catch** en Java es esta.

catch (**AlgunObjetoThrowable** **nombreVariable**) {

**Sentencias Java**

}

Como **puedes ver, la sentencia catch requiere un sólo argumento formal.** Este argumento parece **un argumento de una declaración de método**. El **tipo del argumento AlgunObjetoThrowable** declara el **tipo de excepción que el manejador puede manejar y debe ser el nombre de una clase** heredada de la **clase Throwable** definida en el paquete java.lang. (Cuando los programas **Java lanzan una excepción realmente están lanzado un objeto**, sólo pueden **lanzarse los objetos derivados de la clase Throwable**.

**El nombre nombreVariable** es el **nombre de variable por el que el manejador puede referirse a la excepción capturada**. Por ejemplo, los manejadores de excepciones para el array (mostrados más adelante) llaman al método **getMessage()** de la excepción utilizando el nombre de la variable de tipo NullPointerException declarada como **ex1**. Este código se corresponde al ejemplo **ExcepcionesEnArray.java.**

**catch** (NullPointerException ex1) {

System.***out***.println("\n Excepción Generada: "

+

ex1.getMessage());

ex1.printStackTrace();

}

En el ejemplo el **tipo del argumento es algún objeto de tipo Throwable como NullPointerException** declara el tipo **de excepción que el manejador puede manejar** y esuna **clase que hereda de la clase Throwable** .

**La llamada al método ex1.getMessage()**

Se **puede acceder a las variables y métodos de las excepciones** en la **misma forma** que accede a los de cualquier **otro objeto**. **getMessage()** es un método **proporcionado por la clase Throwable que imprime información adicional sobre el error ocurrido**. La clase **Throwable** también **implementa dos métodos para rellenar e imprimir el contenido de la pila de ejecución** cuando **ocurre la excepción**. Las subclases de **Throwable pueden añadir otros métodos o variables de ejemplar**, Para buscar qué métodos **implementar en una excepción**, se puede **comprobar la definición de la clase** y las **definiciones de las clases antecesoras. Más adelante** veremos los **métodos que proporciona Exception** y que son más específicos.

El bloque **catch** debe **contener una serie de sentencias** Java. Estas **sentencias se ejecutan cuando se llama al manejador de excepción**. El sistema de **ejecución llama al manejador de excepción** cuando **el manejador es el primero en la pila de llamadas** cuyo tipo coincide con el de **la excepción lanzada**.

En el ejemplo ***tenemos seis manejadores de excepciones para***

try {

. . .

**catch** (NullPointerException ex1) {

System.***out***.println("\n Excepción Generada: "

+

ex1.getMessage());

ex1.printStackTrace();

} **catch** (NegativeArraySizeException ex2) {

System.***out***.println("\n Excepción Generada: "

+

ex2.getMessage());

ex2.printStackTrace();

} **catch** (ArrayIndexOutOfBoundsException ex3) {

System.***out***.println("\n Excepción Generada: "

+

ex3.getMessage());

ex3.printStackTrace();

} **catch** (IndexOutOfBoundsException ex4) {

System.***out***.println("\n Excepción Generada: "

+

ex4.getMessage());

ex4.printStackTrace();

} **catch** (ClassCastException ex5) {

System.***out***.println("\n Excepción Generada: "

+

ex5.getMessage());

ex5.printStackTrace();

} **catch** (ArrayStoreException ex6) {

System.***out***.println("\n Excepción Generada: "

+

ex6.getMessage());

ex6.printStackTrace();

}

### Capturar Varios Tipos de Excepciones con un Manejador

Volvemos al ejemplo de muestra array modificado EjemploArrayOutOfBounds. Estamos **capturando por separado en esta nueva versión** **las dos excepciones**, que como indicaremos están relacionadas, pues son clase padre e hija, que son ArrayIndexOutOfBoundsException y IndexOutOfBoundsException.

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (ArrayIndexOutOfBoundsException e){

System.***out***.println("Ha accedido a una posición fuera del array");

}

**catch** (IndexOutOfBoundsException e){

System.***out***.println("Indice ilegal. Ha accedido a una posición fuera del array");

}

}

En **el resultado de la ejecución podéis ver como se entra** en la **más específica de las dos excepciones** ArrayIndexOutOfBoundsException. La excepción ArrayIndexOutOfBoundsException es clase hija de IndexOutOfBoundsException y **especializada para arrays**. Hay que tener en cuenta que IndexOutOfBoundsException **también vale para cadenas** entre otros. Como se ha **producido el error para arrays** se lanza la más específica ArrayIndexOutOfBoundsException.

4

2

6

Ha accedido a una posición fuera del array

**Ejemplo EjemploOutOfBoundsSinAgrupar.java.**

**public** **class** EjemploOutOfBoundsSinAgrupar {

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (ArrayIndexOutOfBoundsException e){

System.***out***.println("Ha accedido a una posición fuera del array");

}

**catch** (IndexOutOfBoundsException e){

System.***out***.println("Indice ilegal. Ha accedido a una posición fuera del array o cadena");

}

}

**private** **static** **void** muestraArray() {

// **TODO** Auto-generated method stub

**int** [] array = {4,2,6};

**for** (**int** i=0 ; i<=3 ; i++) {

System.***out***.println(array[i]);

}

}

}

Los **dos manejadores de excepción utilizados** por el método **muestraArray** son muy especializados. Cada **uno sólo maneja un tipo de excepción**. El **lenguaje Java permite escribir manejadores de excepciones generales** que pueden manejar **varios tipos de excepciones**.

Es habitual en casos como este en **que la clase ArrayIndexOutOfBoundsException** que **hereda** de **IndexOutOfBoundsException**, agruparlos en uno sólo, mas general que vale para Arrays y Strings.

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (IndexOutOfBoundsException e){

System.***out***.println("Indice ilegal. Ha accedido a una posición fuera del array o cadena");

}

}

Lo podéis ejecutar en **EjemploOutOfBounds.java**

**public** **class** EjemploOutOfBounds {

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (IndexOutOfBoundsException e){

System.***out***.println("Ha accedido a una posición fuera del array o cadena");

}

**finally** {

System.***out***.println("Finally siempre se ejecuta");

}

}

**private** **static** **void** muestraArray() {

// **TODO** Auto-generated method stub

**int** [] array = {4,2,6};

**for** (**int** i=0 ; i<=3 ; i++) {

System.***out***.println(array[i]);

}

}

}

**En el nuevo ejemplo de ejecución** **se captura la excepción más genérica** **porque *sólo hay un manejador*** (bloque catch) de excepciones.

4

2

6

Ha accedido a una posición fuera del array

Finally siempre se ejecuta

La **clase Exception** **está bastante arriba en el árbol de herencias de la clase Throwable**. Por eso, **además de capturar los tipos de IndexOutOfBoundException y ArrayIndexOutOfBoundsException** este manejador de excepciones, puede capturar otros muchos tipos. En resumen, los **manejadores de excepción deben ser más especializados**.

Los **manejadores que pueden capturar la mayoría o todas las excepciones** son **menos utilizados para la recuperación de errores porque el manejador tiene que determinar qué tipo de excepción ha ocurrido** de todas formas (para determinar la mejor estrategia de recuperación). Los **manejadores de excepciones que son demasiado generales** pueden hacer el código **más propenso** **a errores** mediante la captura y manejo de excepciones que no fueron anticipadas por el programador y para las que el manejador no está diseñado.

Un **caso extremo es el siguiente en el que capturamos** la excepción con la clase Exception. Es tan genérico que podemos mandar un mensaje muy general

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (Exception e){

System.***out***.println("Error en el array");

}

}

### El stackTrace o traza de pila de ejecución

Podemos **imprimir el Stack Trace o traza de la pila de ejecució**n cuando capturamos una excepción. Para ello **vamos a añadir** en el siguiente ejemplo **EjemploOutOfBounds.java** le **añadimos** la línea **siguiente línea** e.printStackTrace(); en el bloque catch.

**public** **class** EjemploOutOfBounds {

**public** **static** **void** main(String[] args) {

**try** {

*muestraArray*();

}

**catch** (IndexOutOfBoundsException e){

System.***out***.println("Ha accedido a una posición fuera del array o cadena");

e.printStackTrace();

}

**finally** {

System.***out***.println("Finally siempre se ejecuta");

}

}

**private** **static** **void** muestraArray() {

// **TODO** Auto-generated method stub

**int** [] array = {4,2,6};

**for** (**int** i=0 ; i<=3 ; i++) {

System.***out***.println(array[i]);

}

}

}

Si **lo ejecutamos además del mensaje de error** nos **escribirá la pila en consola**, es decir todos **los métodos y por tanto clases**, por los que pasa el flujo del programa desde que se produce el error**. En este caso sólo una clase**. En **futuros ejemplos donde hay un modelo** **veremos que las excepciones se pueden propagar** entre **métodos de distintas clases**. Entre otras cosas contiene el punto exacto donde se produjo el error:

muestraArray(EjemploOutOfBounds.java:37)

4

2

6

Ha accedido a una posición fuera del array

java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3

at Excepciones/excepciones.EjemploOutOfBounds.muestraArray(EjemploOutOfBounds.java:37)

at Excepciones/excepciones.EjemploOutOfBounds.main(EjemploOutOfBounds.java:9)

Finally siempre se ejecuta

### El Bloque finally

El **paso final en la creación de un manejador de excepción** es proporcionar un **mecanismo que limpie el estado del método** antes (posiblemente) de **permitir que el control pase a otra parte diferente del programa**. Se puede **hacer esto encerrando el código de limpieza dentro** de un **bloque** **finally** se usa para **una vez realizado la ejecución de las instrucciones que pueden causar una excepción**, cerrar recursos.

Aunque **no lo conozcamos aún el siguiente código abre un fichero**. Este es un ejemplo teórico para que **entendáis como funciona el bloque finally**. Si todo va bien cuando termine el bloque try, se ejecutará el **contenido del finally**, que cierra el fichero. Si el **programa va mal** y hay una **excepción de entrada salida , IOException**, se hace catch, se indica el error y **también se ejecuta el contenido del bloque finally**.

PrintStream pstr;

try {

int i;

System.out.println("Entering try statement");

pStr = new PrintStream(

new BufferedOutputStream(

new FileOutputStream("OutFile.txt")));

for (i = 0; i < size; i++)

pStr.println("Value at: " + i + " = " + victor.elementAt(i));

}

catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

}

finally {

if (pStr != null) {

System.out.println("Closing PrintStream");

pStr.close();

} else {

System.out.println("PrintStream not open");

}

}

**¿Es realmente necesaria la sentencia finally?**

Si **no usamos el bloque finally tendríamos** que hacer **algo como en el siguiente código** que no es correcto

try {

. . .

**pStr.close(); // No haga esto, duplica el código**

} catch (ArrayIndexOutOfBoundsException e) {

**pStr.close(); // No haga esto, duplica el código**

System.err.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

}

Sin embargo, **esto duplica el código, haciéndolo difícil de leer y propenso a errores** si se modifica más tarde, Por ejemplo, si se añade código al bloque **try** que pudiera lanzar otro tipo de excepción**, se tendría que recordar el cerrar el PrintStream dentro del nuevo manejador de excepción** (lo que se olvidará seguro si se parece a mí).

### Poniéndolo todo Junto

**Cuando se juntan todos los componentes** **en nuestro ejemplo** de **ExcepcionesEnArray**, nos queda el **ejemplo final**. El **bloque try , los bloques catch y el bloque finally** todos juntos para realizar un **correcto manejo de excepciones**.

**ExcepcionesEnArray.java**

**public** **class** ExcepcionesEnArray {

**public** **static** **void** main(String[] args) {

**int** array[] = **null**;

**int** arraySize = 4;

**int** arrayInc = -0;

**int** output;

**for** (**int** i = 0; i < 6; i++) {

**try** {

// Multiple condiciones en el switch

**switch** (i) {

**case** 0:

output = array[0]; // Genera NullPointerException.

**break**;

**case** 1:

array = **new** **int**[arrayInc]; // Genera NegativeArraySizeException.

output = array[arrayInc];

**break**;

**case** 2:

array = **new** **int**[arraySize]; // Genera ArrayIndexOutOfBoundsException.

output = array[arraySize];

**break**;

**case** 3:

array = **new** **int**[5]; // Genera IndexOutOfBoundsException.

output = array[5];

**case** 4:

Object newArray = **new** ~~Integer~~(0); // Genera ClassCastException.

System.***out***.println((String) newArray);

**case** 5:

Object X[] = **new** String[-5]; // Genera ArrayStoreException.

X[0] = **new** ~~Integer~~(0);

System.***out***.println(X);

}

} **catch** (NullPointerException ex1) {

System.***out***.println("\n Excepción Generada: "

+

ex1.getMessage());

ex1.printStackTrace();

} **catch** (NegativeArraySizeException ex2) {

System.***out***.println("\n Excepción Generada: "

+

ex2.getMessage());

ex2.printStackTrace();

} **catch** (ArrayIndexOutOfBoundsException ex3) {

System.***out***.println("\n Excepción Generada: "

+

ex3.getMessage());

ex3.printStackTrace();

} **catch** (IndexOutOfBoundsException ex4) {

System.***out***.println("\n Excepción Generada: "

+

ex4.getMessage());

ex4.printStackTrace();

} **catch** (ClassCastException ex5) {

System.***out***.println("\n Excepción Generada: "

+

ex5.getMessage());

ex5.printStackTrace();

} **catch** (ArrayStoreException ex6) {

System.***out***.println("\n Excepción Generada: "

+

ex6.getMessage());

ex6.printStackTrace();

} **finally** {

//Se usa para cerra los recursos abiertos

System.***out***.println("Finaliza cerrando recursos");

}

}

}

}

El bloque **try** de este método **tiene seis posibilidades a diferentes con seis catch diferentes**. Y el **finally que se ejecutará siempre**, haya excepción o no.

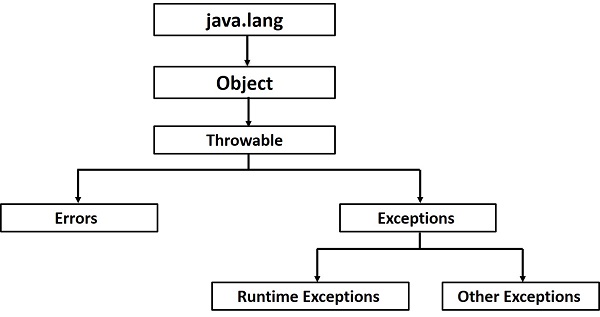
# Creación y propagación de excepciones con un ejemplo práctico.

La **API de excepciones de Java no es un sistema cerrado**. Los **programadores pueden crear sus propias excepciones** para **controlar los errores de sus programas**. Sólo **necesitan crear sus propias excepciones heredando de Exceptio**n o **de una de las clases hijas de esta.**

## Jerarquía de excepciones

Todas las **clases de excepción son subtipos de la clase java.lang.Exception**. La **clase Exception es una subclase de la clase Throwable**. Aparte de la clase de excepción, hay otra **subclase llamada Error que se deriva de la clase Throwable**.

Cuando el **Throwable** que se lanza es **de tipo Error** **son condiciones anormales que ocurren en caso de fallos graves**, estos no son manejados por los programas Java, no se pueden controlar en Java ya que no te lo permite el compilador. Los **errores se generan para indicar los errores generados por el entorno de ejecución**. Ejemplo: el **proceso de la** **JVM no tiene memoria**. Normalmente, los **programas no pueden recuperarse de errores y terminan de forma abrupta**.



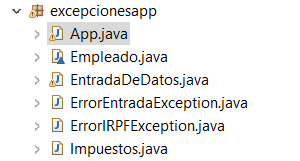
La **clase Exception tiene dos subclases principales**: **IOException y RuntimeException**.

En el **anexo I podéis encontrar una lista detallada** de las excepciones más comunes.

**Métodos de la clase Exception**

|  |  |
| --- | --- |
|  | **Constructores** |
|  | **public Exception()**  Crea una nueva excepción con null como mensaje de detalle. La causa no se inicializa y posteriormente se puede inicializar mediante una llamada a Throwable.initCause(java.lang.Throwable). |
|  | **public Exception(mensaje de cadena)**  Crea una nueva excepción con el mensaje de detalle especificado. La causa no se inicializa y posteriormente se puede inicializar mediante una llamada a Throwable.initCause(java.lang.Throwable). |
|  | **public Exception(String message,**  **Throwable cause)**  Construye una nueva excepción con el mensaje de detalle y la causa especificados.  Tened en cuenta que el mensaje de asociado con la causa no se incorpora automáticamente en el mensaje de la excepcion. Se añade uno nuevo  **Parametros:**  message - el mensaje (que se guarda para su posterior recuperación mediante el método Throwable.getMessage()).  cause - la causa (que se guarda para su posterior recuperación mediante el método Throwable.getCause()). (Se permite un valor nulo e indica que la causa es inexistente o desconocida). |
|  | **public Exception(Throwable cause)**  Construye una nueva excepción con la causa especificada y un mensaje que contiene el toString() de cause si no es nulo(cause==null ? null : cause.toString()) (que normalmente contiene la clase y el mensaje de detalle de causa). Este constructor es útil para excepciones que son poco más que contenedores para otros dispositivos desechables (por ejemplo, PrivilegedActionException).  **Parámetros:**  cause - la causa (que se guarda para su posterior recuperación mediante el método Throwable.getCause()). (Se permite un valor nulo e indica que la causa es inexistente o desconocida). |
|  | **protected Exception(String message,**  **Throwable cause,**  **boolean enableSuppression,**  **boolean writableStackTrace)**  Crea una nueva excepción con el mensaje de detalle especificado, la causa, la supresión habilitada o deshabilitada y el seguimiento de pila grabable habilitado o deshabilitado.  **Parámetros:**  Mensaje: el mensaje  causa - la causa. (Se permite un valor nulo e indica que la causa es inexistente o desconocida).  enableSuppress: si la supresión de la excepción está habilitada o deshabilitada  writableStackTrace: si el seguimiento de la pila debe escribirse o no |
|  |  |
| **Sr.No.** | **Método y descripción** |
| 1 | **public String getMessage()**  Devuelve un mensaje detallado sobre la excepción que se ha producido. Este mensaje se inicializa en el constructor Throwable. |
| 2 | **public Throwable getCause()**  Devuelve la causa de la excepción representada por un objeto Throwable. |
| 3 | **public String toString()**  Devuelve el nombre de la clase concatenada con el resultado de getMessage(). |
| 4 | **public void printStackTrace()**  Imprime el resultado de toString() junto con el el stack trace o información de la pila en System.err, la secuencia de salida del error. |
| 5 | **public StackTraceElement [] getStackTrace()**  Devuelve un array que contiene cada elemento el stack trace o información de la pila. El elemento del índice 0 representa la parte superior de la pila de llamadas y el último elemento de la matriz representa el método en la parte inferior de la pila de llamadas. |
| 6 | **public Throwable fillInStackTrace()**  Rellena el stack trace o información de la pila de este objeto Throwable con el seguimiento de pila actual, agregando a cualquier información anterior en el seguimiento de pila. |

Usaremos **el siguiente modelo para explicar** lo que **nos queda de excepciones**



* Una clase **App** que contiene un menú con el **programa principal, que pide introducir empleados y calcula su sueldo.** Se **encargará de capturar y manejar las excepciones. Intentamos** siempre la **captura de excepciones en Java** lo más cerca del **hilo principal o de los controladores en el modelo MVC**. En este caso en el método **main() del programa principal**.

El **programa principal o los controladores suelen ser los encargados** de **controlar lo que se muestra en pantalla y los mensajes de error**. Es por eso que **capturamos las excepciones en estos puntos**, **mandamos un mensaje de error** y el programa sigue funcionando. Se recupera del error

* Una clase **Empleado**, que **representa empleados con nombre, apellido, sueldo e irpf como propiedades.**
* Una clase **Impuestos que ayuda a calcular los impuestos del empleado**. En esta clase **lanzaremos la excepción de tipo checked** que vamos a crear en caso de que el **IRPF del Empleado sea mayor de 0,5 o menor que 0**.
* La clase **ErrorIRPFException que hereda de Exception** y que la lanzaremos dentro de la clase Impuestos si cuando **calcula el sueldo el IRPF del Empleado es mayor de 0,5 o menor que 0.**
* La clase **ErrorEntradaException que hereda de Exception**. Capturamos **InputMismatchException, que es una Exception de tipo RuntimeException** que no es obligatorio controlar, **y lanzamos una propia, ErrorEntradaException** que si hay que **calcular reportando del error de entrada de datos**. Es una práctica **muy habitual en programación Java.**
* La clase **EntradaDeDatos** que se **encarga de recoger los datos de pantalla**. En caso de que **los datos recogidos por la consola con la clase Scanner con tipos erróneos**, por ejemplo que **al métodi nextDouble()** de la clase Scanner el usuario le introduzca un **carácter ‘a’** se producirá una **excepción InputMismatchException de tipo RuntimeException, unchecked**. **Envolveremos esta excepción** en una propia **ErrorEntradaException,**  porque **es buena práctica** tener nuestras excepciones propias y **porque las excepciones unchecked no necesitan ser manejadas**, y por esa razón **a veces producen errores indeseados en los programas**, porque se nos olvida controlarlas**.**

## Las excepciones

**Tenemos dos excepciones en nuestro modelo**. Una para cuando **al calcular los impuestos el IRPF sea erroneo**, mayor de 0.5 o menor que 0. Para ello **creamos una clase Error IRPFException** y la hacemos **heredar de Exception**.

**package** excepcionesapp;

**public** **class** ErrorIRPFException **extends** Exception{

/\*\*

\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** ErrorIRPFException() {

**super**("Error de IRPF en el empleado. Demasiado alto. Solucionar");

}

}

La **otra Excepción es ErrorEntradaException que la lanzaremos cuando el tipo introducido** por **el usuario en la consola no coincida con el tipo de la variable** a la que se asigna. Por ejemplo, que cuando **estemos pidiendo un sueldo** y el **usuario escriba caracteres alfanuméricos.** Esta situación produce la excepción **InputMismatchException**. Lo que **haremos es crear nuestra excepción a partir de InputMismatchException** que es de **tipo unchecked**, hereda de **Runtime**.

Para ello al crear nuestra excepción recibimos como parámetro la excepción **InputMismatchException y** se la pasamos alsuper junto a un mensaje de error **, el constructor de Exception. super**("Error en la entrada de teclado",em). Podríamos hacerlo tan solo con sólo dos parámetros usando el constructor de dos parámetros. Vamos a **aprovecharnos de este ejemplo** para pasar dos parámetros más e **introducir dos conceptos extra**. Podéis **verlo en los constructores de la clase Exception**. La excepción de tipo **InputMismatchException** a la que no hace falta crearle un manejador , hereda de RuntimeException, la **hemos capturado y lanzamos una propia de nuestra aplicación** en su lugar **ErrorEntradaException** que es de tipo checked, hereda de Exception, y va a ser necesario capturar. Lo haremos en el main.

**import** java.util.InputMismatchException;

**public** **class** ErrorEntradaException **extends** Exception{

/\*\*

\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 2L;

**public** ErrorEntradaException(InputMismatchException em) {

**super**("Error en la entrada de teclado",em, **true**,**false**);

}

}

protected Exception(String message,

Throwable cause,

boolean enableSuppression,

boolean writableStackTrace)

**super**("Error en la entrada de teclado",em, **true**,**false**);

Crea una **nueva excepción con el mensaje de detalle especificado**, la **causa que es la excepción que envolvemos** en este caso **em** excepciónde tipo **InputMismatchException.** L**a supresión habilitada y el stack trace deshabilitada la escritura**.

### Supresión de excepciones

Tened en cuenta que **cuando una excepción causa otra, la primera excepción generalmente se detecta y** luego **la segunda excepción se lanza en respuesta**. En otras palabras, existe una **conexión causal entre las dos excepciones**. Por el contrario, hay **situaciones en las que se pueden lanzar dos excepciones independientes** en bloques de código, en particular e**n el bloque try una try y en el bloque finally generado por el compilador que cierra el recurso**.

En estas situaciones, **solo se puede propagar una de las excepciones lanzadas**. En la instrucción try, cuando hay dos excepciones de este tipo, **la excepción que se origina en el bloque try se propaga** y la **excepción del bloque finally se agrega** a la **lista de excepciones suprimidas** por la **excepción del bloque try**. Como una **excepción captura la ejecución en la pila**, puede **acumular varias excepciones suprimidas**.

## Throw throwable. La clase Impuestos

A la **clase Impuestos le vamos a dar la responsabilidad** de **calcular el sueldo** de un **Empleado**. Para ello **la clase va a ofrecer un método calculoImpuestos(Empleado emp)** que **recibe un empleado como parámetro**. Si el **empleado tiene un IRPF erróneo** el método **calculoImpuestos() va a arrojar una excepción**. Además de **las que arroja la máquina virtual Java** nosotros **podemos provocar nuestras propias excepciones**, lanzarlas, con la **palabra reservada throw** seguida de **una clase de tipo Throwable** o que herede de está como Exception.

Como veis en el ejemplo **throw** **new** ErrorIRPFException(); si el **IRPF es mayor que 0,5 o menor 0,0,** **lanzamos nuestra excepción** propia de tipo ErrorIRPFException.

**public** **double** calculoImpuestos(Empleado e) **throws** ErrorIRPFException {

**if** (e.getIrpf()> 0.50 || e.getIrpf() < 0.0) {

**throw** **new** ErrorIRPFException();

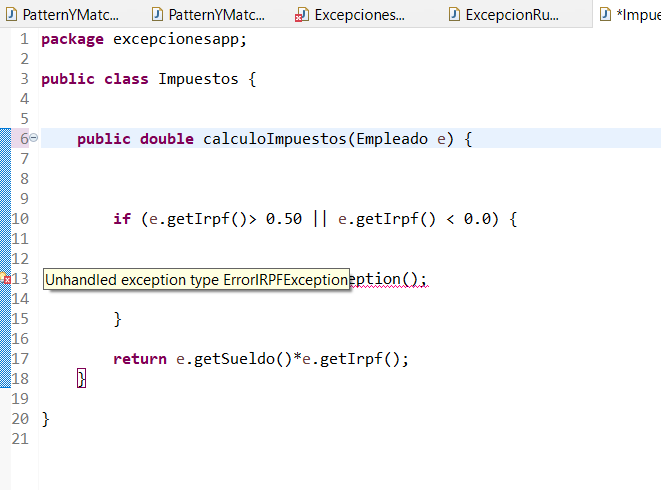
}

**return** e.getSueldo()\*e.getIrpf();

}

En la cabecera de la función **public** **double** calculoImpuestos(Empleado e), para que además **podamos propagar la excepción en el métodos,** después de los parámetros, usamos la palabra reservada **throws** seguido **del tipo de excepción**, en este caso **ErrorIRPFException**. Esta acción sirve para indicar que la excepción no se controlará ahí, sino **en el método que llame** a este método **calculoImpuestos()**.

En este caso **esta excepción es de tipo checked**, es **obligatorio controlarla**. Podéis **comprobarlo si quitamos el throws el compilador** nos **informará de un error**. Nos informa que no se está capturando, que no hay manejador en el método para la excepción **ErrorIRPFException**. O **la capturamos con un catch** o la **propagamos con throws** para que la **captura lo haga otro que llame a este**



Impuestos.java

**package** excepcionesapp;

**public** **class** Impuestos {

**public** **double** calculoImpuestos(Empleado e) **throws** ErrorIRPFException {

**if** (e.getIrpf()> 0.50 || e.getIrpf() < 0.0) {

**throw** **new** ErrorIRPFException();

}

**return** e.getSueldo()\*e.getIrpf();

}

}

Al **añadir la sentencia throws le estamos diciendo que el método encargado** de controlar la excepción es el **método que llame a** calculoImpuestos(). El **método que llama a** calculoImpuestos() es calculaSueldo() de la clase **Empleado** que **también propaga la excepción.**

**En clase Empleado**

**public** **double** calculaSueldo() throws ErrorIRPFException {

**return** **this**.sueldo - impuesto.calculoImpuestos(**this**);

}

A **su vez el método calculaSueldo**()  **es llamado por el método main()** de la **clase principal App**. Y aquí sí que **ya tenemos un manejador de excepción**. El **siguiente código está en el método main() de la clase principal** y ya se encarga de la excepción como podéis ver con un catch, **catch** (ErrorIRPFException e).

Método main() la clase App.java.

emp = **new** Empleado(id, nombre, sueldo, IRPF);

sueldoNeto = emp.calculaSueldo();

System.***out***.println("El sueldo del empleado");

} **catch** (ErrorIRPFException e) {

// **TODO** Auto-generated catch block

System.***out***.println("Excepción en el IRPF: " + .getMessage());

}

Esta **es la manera correcta de trabajar**. Las **excepciones como se apuntó en un principio se deben manejar o capturar cerca del hilo o ventana principales de la aplicación** o una **clase controladora de ventanas**, donde **controlamos los errores y podemos mandar mensajes de error**. Como veis en el ejemplo **cuando capturamos la excepción y manejamos el error,** mostramos **al usuario.** El bucle donde se encuentran el try y el catch no se para y el programa seguirá funcionando. **El programa procederá a preguntar al usuario por el siguiente empleado.**

## La clase Empleado

La **clase empleado tiene cuatro atributos básicos, id, nombre, sueldo e IRPF**. Como destacable, lo hemos comentado en el ejemplo anterior el método calculaSueldo() que hace uso del método **calculoImpuestos()** de la clase **Impuestos**. Recordar que el método calculaSueldo() **propaga** la excepción **IRPFException** hacia el método **main() de la clase App**.

**Empleado.java**

**public** **class** Empleado {

**int** id;

String nombre;

**double** sueldo=0.0;

**double** irpf=0.0;

Impuestos impuesto;

**public** Empleado () {

}

**public** Empleado (**int** id, String nombre,**double** sueldo, **double** irpf) {

**this**.id=id;

**this**.nombre=nombre;

**this**.sueldo=sueldo;

**this**.irpf=irpf;

impuesto = **new** Impuestos();

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getNombre() {

**return** nombre;

}

**public** **void** setNombre(String nombre) {

**this**.nombre = nombre;

}

**public** **double** getSueldo() {

**return** sueldo;

}

**public** **void** setSueldo(**double** sueldo) {

**this**.sueldo = sueldo;

}

**public** **double** getIrpf() {

**return** irpf;

}

**public** **void** setIrpf(**double** irpf) {

**this**.irpf = irpf;

}

**public** **double** calculaSueldo() **throws** ErrorIRPFException {

**return** **this**.sueldo - impuesto.calculoImpuestos(**this**);

}

@Override

**public** String toString() {

**return** "Empleado [id=" + id + ", nombre=" + nombre + ", sueldo=" + sueldo + ", irpf=" + irpf + "]";

}

}

**Pila de ejecución del Java Run Time en caso de error en el IRPF y que se lance IRPFException**

|  |  |  |
| --- | --- | --- |
| **función en ejecución** | **Excepción producida** | **Manejador de excepciones** |
| Clase Impuestos  calculoImpuestos() | ErrorIRPFException | Se propaga la excepción |
| Clase Empleado  calculaSueldo() | ErrorIRPFException  Se produjo  en la clase Impuestos | Se propaga la excepción |
| Clase App  Main() |  | Se controla la excepción  catch (ErrorIRPFException e) |

## La clase EntradaDeDatos

Está **clase será usada como clase envolvente de la clase Scanner** con el **fin de realizar nuestra propia recogida de datos de consola** dentro de nuestro modelo. **Ofrece tres métodos** leerString() para leer cadenas de la consola. Igualmente, leerDouble() para **leer un decimal** de la consola, y leerEntero(), para **leer un entero** de la consola.

En los **tres métodos ocurre lo mismo** y **se desarrollan igual**, así que **vamos a estudiar uno de ellos**, leerDouble(). Si al **leer un double, un decimal, con la clase Scanner** se **introduce una “a” Java va a producir una excepción** de **tipo RuntimeException**, **unchecked**, de las que no es obligatorio controlar, **InputMismatchException**. Es muy **típico que los programadores Java** controlen **las excepciones de tipo Runtime**, unchecked, **en escenarios en los que se puede producir,** como este en el que el usuario puede introducir erróneamente valores. No interesa que se puedan producir errores incontrolados en tu programa.

Además, **el programador Java suele controlar la excepción** y **envolverla con una propia**, en este caso **ErrorEntradaException** que es de **tipo checked**, hay que **manejarla con un catch**. Igual que en **el caso anterior estos métodos propagan la excepción** propia ErrorEntradaException de la que **nos encargaremos de nuevo en el programa principal** que es donde se llama.

**public** **double** leerDouble() **throws** ErrorEntradaException {

**try** {

**return** sc.nextDouble();

} **catch** (InputMismatchException ime) {

**throw** **new** ErrorEntradaException(ime);

}

}

**Propagamos la excepción en los métodos de leer** y el **programa principal** que llama al método leerDouble() en el **método main() de la clase APP** se **encargará de manejar la excepción**

Código de la clase App en el método main()

System.***out***.println("Introduce el sueldo del empleado. Numero positivo");

sueldo = entradaDatos.leerDouble();

System.***out***.println("Introduce el IRPF del empleado. Un numero entre 0 y 0,5");

IRPF = entradaDatos.leerDouble();

} **catch** (ErrorEntradaException eee) {

System.***out***.println(eee.getMessage());

}

**Capturamos las excepciones** en la **función main del programa principal** donde es más conveniente mandar un mensaje de error. Además, **nos permite controlar que el flujo** del **programa funcional no se pare** y siga **funcionando recuperándose del error limpiamente**. Si lo hiciéramos en **otra clase dentro del programa no tenemos esta certeza**

System.***out***.println(eee.getMessage());

**La clase EntradaDeDatos.java**

**import** java.util.InputMismatchException;

**import** java.util.Scanner;

**public** **class** EntradaDeDatos {

**private** Scanner sc = **null**;

**public** EntradaDeDatos() {

Scanner sc = **new** Scanner(System.***in***);

}

**public** String leerString() **throws** ErrorEntradaException {

**try** {

**return** sc.next();

} **catch** (InputMismatchException ime) {

**throw** **new** ErrorEntradaException(ime);

}

}

**public** **double** leerDouble() **throws** ErrorEntradaException {

**try** {

**return** sc.nextDouble();

} **catch** (InputMismatchException ime) {

**throw** **new** ErrorEntradaException(ime);

}

}

**public** **int** leerEntero() **throws** ErrorEntradaException {

**try** {

**return** sc.nextInt();

} **catch** (InputMismatchException ime) {

**throw** **new** ErrorEntradaException(ime);

}

}

}

## La clase Principal App.

En la **clase principal vamos a realizar toda la comunicación con el usuari**o. El **programa se dedica a pedir los datos del Empleado** **al usuario y calcular sus impuestos**. Además, **controla los errores posibles**. Como **tenemos dos tipos de Excepciones que controlar**, *ErrorEntradaException y ErrorIRPFException*, y **los métodos que las propagan se ubican en diferentes posiciones**, se ha **decidido que el try envuelva toda la sentencia switch del menú**. De esta **manera podemos controlar al final todas las excepciones juntas**, en un solo punto del programa. Es la **manera correcta de realizar el control de excepciones**.

**En verde se han señalado los métodos** que producen la **excepción** ErrorEntradaException y **en amarillo calculaSueldo()** que **propaga** *ErrorIRPFException*. Como **los métodos están dispersos** **realizamos el control de las excepciones** de esta manera, haciendo **un bloque try amplio** que **cubra todos los métodos** que pueden **propagar una excepción.** Al final los **bloques catch y el finally** en azul.

**try** {

**switch** (entrada) {

**case** 0:

id = entradaDatos.leerEntero();

System.***out***.println("Introduce el nombre del empleado");

nombre = entradaDatos.leerString();

System.***out***.println("Introduce el sueldo del empleado. Numero positivo");

sueldo = entradaDatos.leerDouble();

System.***out***.println("Introduce el IRPF del empleado. Un numero entre 0 y 0,5");

IRPF = entradaDatos.leerDouble();

**break**;

**case** 1:

System.***out***.println("Fin del programa");

**break**;

**default**:

System.***out***.println("introduce un número de menú correcto");

}

emp = **new** Empleado(id, nombre, sueldo, IRPF);

sueldoNeto = emp.calculaSueldo();

System.***out***.println("El sueldo del empleado");

} **catch** (ErrorIRPFException e) {

// **TODO** Auto-generated catch block

System.***out***.println("Excepción en el IRPF: " + .getMessage());

} **catch** (ErrorEntradaException eee) {

System.***out***.println(eee.getMessage());

}

**finally** {

// Borramos el empleado incorrecto

System.***out***.println("Procedemos a borrar el empleado");

emp = **null**;

}

Los bloques catch mandan **mensajes de error relacionados con la excepción** que controlan. El bloque finally, que se ejecuta siempre haya error o no, se **dedicará a borrar el empleado** tras hacer todos los cálculos o si se produce un error, en los dos escenarios.

Lo **importante es que el programa controla el error** con las excepciones, pero no finaliza, se recupera del error. El bucle donde se encuentran el try y el catch no se para y el programa seguirá funcionando. **El programa procederá a preguntar al usuario por el siguiente empleado. De manera elegante el programa se ha recuperado del error** con un buen manejo de las excepciones. Si lo hiciéramos en **otra clase dentro del programa** no tenemos esta certeza

## Ejercicio

**Realizar la pila de ejecución para EntradaDeDatos**. Tener en cuenta que hay **dos excepciones está vez.**

# Anexo I. Excepciones en Java

-**ClassNotFoundException** Esta **excepción tiene lugar cuando intentamos ejecutar un proyecto y, por ejemplo, la clase que contiene la función “main”** no ha sido añadida al mismo o no es encontrada (una causa muy común para lo mismo es el haber configurado mal el proyecto).

<https://docs.oracle.com/javase/8/docs/api/java/lang/ClassNotFoundException.html>

**-RuntimeException** : la clase **RuntimeException representa el conjunto de las excepciones que pueden tener lugar durante el proceso de ejecución de un programa sobre la JVM, con** la peculiaridad **de que el usuario no tiene que prestar atención al hecho de capturarlas** (todas las demás excepciones deberán ser capturadas o gestionadas en algún modo por el usuario).

<https://docs.oracle.com/8/docs/api/java/lang/RuntimeException.html>

Dentro de la **especificación de la clase RunTimeException ()** podemos encontrar un **numeroso grupo de excepciones**, con algunas de las cuales ya estamos familiarizados, y que conviene citar:

<https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html>

**-ClassCastException** : excepción que **tiene lugar cuando intentamos hacer un “cast” de un objeto a una clase de la que no es subclase**. Un ejemplo sencillo está en la propia API de Java: 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. La situación anterior produciría una “ClassCastException” en tiempo de ejecución, no antes, que no es necesario que el usuario gestione.

<https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html>

**-IndexOutOfBoundsException:** **excepción que tiene lugar cuando intentamos acceder a un índice de un “array”, “String” o “vector”** mayor que **el número de elementos de dicha estructura**. Veamos también un ejemplo sencillo: int array\_enteros [] = new int [50]; System.out.println (array\_enteros [67]); **Al intentar acceder a una posición del “array” mayor que la dimensión del mismo**, podemos **observar como Java lanza una excepción para advertirnos de que dicha dirección** de **memoria no ha sido reservada**. De nuevo, estamos ante una excepción que aparecerá en tiempo de ejecución, y que no es necesario que el usuario gestione.

<https://docs.oracle.com/javase/8/docs/api/java/lang/IndexOutOfBoundsException.html>

**-NegativeArraySizeException**): **excepción que tiene lugar cuando intentamos crear un “array” con longitud negativa**. Un ejemplo **de una situación donde aparecería tal excepción sería el siguiente: int array\_enteros [] = new int [-50];** 13 De nuevo es una excepción que el usuario no debe gestionar (pero debe ser consciente de que puede surgir en sus programas). Repetimos la pregunta que lanzábamos con la excepción anterior, ¿qué habría pasado en C++ ante esa situación?

(<https://docs.oracle.com/javase/8/docs/api/java/lang/NegativeArraySizeException.html>

**-InputMismatchException**: **excepción que lanzan varios métodos de la librería “Scanner” de Java** cuando están recorriendo una entrada en busca de un dato y éste no existe en la entrada.

Por ejemplo, si **tenemos la siguiente entrada abierta: Hola Esto es un scanner** Que sólo contiene texto Y sobre el mismo **tratamos de ejecutar el método “nextDouble():** double”, **obtendremos dicha excepción**. Una vez más, no es obligatorio gestionarla, aunque dependiendo de nuestra aplicación puede ser importante hacerlo.

<https://docs.oracle.com/javase/8/docs/api/java/util/InputMismatchException.html>

**-NumberFormatException**: esta **excepción 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. Es usada, por ejemplo, en los métodos “parseDouble(): double”, “parseInt(): int”, “parseFloat(): float”, ... .

<https://docs.oracle.com/javase/8/docs/api/java/lang/NumberFormatException.html>

**-NullPointerException**: quizá ésta sea **la excepción que más comúnmente aparece cuando se trabaja con Java**. Surge siempre que **intentamos acceder a cualquier atributo o método de un objeto al que le hemos asignado valor “null”.** Un caso sencillo donde aparecería sería el siguiente:

Object ds = null; ds.toString();

Como se puede observar, **el objeto “ds” ha sido inicializado (con el valor “null”), lo que nos previene de obtener un error de compilación en Java**, y al intentar acceder a cualquiera de sus métodos (por ejemplo, el método “toString(): String”), **al invocar a un método sobre la referencia nula,** obtenemos la excepción “**NullPointerException**”.

<https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html>

-**IOException:** ésta es **otra de las excepciones más comúnmente usadas en la librería de Java**. Los métodos de **Java la lanzan siempre que encuentren un problema en cualquier operación de lectura o escritura** a un medio externo (lectura o escritura a un fichero, por ejemplo).

<https://docs.oracle.com/javase/8/docs/api/java/io/IOException.html>

-**FileNotFoundException**: esta excepción **hereda de la clase “IOException”** que acabamos de introducir. En general, la lanzan diversos métodos de la API FILE **cuando no se encuentra un fichero que se está referenciando con un descriptor**.

<https://docs.oracle.com/javase/8/docs/api/java/io/FileNotFoundException.html>