Tema 7. Excepciones

Introducción

En cualquier tarea de programación, hasta en las más simples, se suelen cometer errores. Si hablamos de proyectos complejos, con decenas o cientos de clases diferentes, y miles o decenas de miles de líneas de código, se puede casi asegurar que se habrán cometido algunos errores.

Una técnica para lidiar con dichos errores, **antes** de que se produzcan, es lo que se conoce como *programación defensiva*. Es una forma de desarrollar código que busca garantizar el comportamiento correcto de todo elemento de una aplicación ante cualquier situación de uso, por incorrecta o imprevisible que esta pueda parecer. Un ejemplo podría ser: validar que los índices en un acceso a un array no excedan los límites del propio array, o que un objeto sobre el que se invoca un método no sea *null*.

Después de que ha aparecido el error, hay varias formas de afrontarlo:

- Notificar al usuario de la aplicación. Por ejemplo, mediante mensajes de texto en pantalla detallando el error (salida con *System.out.print*).

- Notificar al *cliente*. Esto es adecuado cuando se trata de un conjunto de clases con una estructura cliente-servidor, es decir, una clase servidor proporciona un conjunto de funcionalidades que no utiliza el usuario directamente, sino que las usa otra clase cliente, que se aprovecha de dichas funcionalidades. Clases servidor: *Collections*, *Math*, *Arrays*, etc. El servidor tiene 2 opciones para notificar a su *cliente* del error:

    * Devolviendo un valor de retorno especial, bien sea *false*, *null*, o un valor numérico conocido que indique una situación de error.

    * Lanzando una **excepción**, que deberá ser recogida y tratada por el *cliente*.

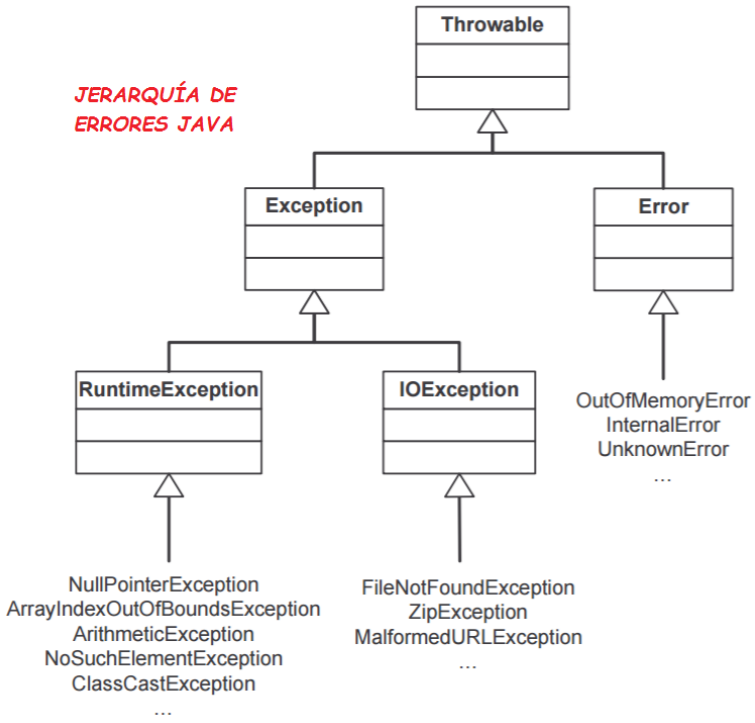
Cada una de estas formas tiene sus ventajas y desventajas, pero nos centraremos únicamente en el manejo de excepciones, siendo una técnica más potente y avanzada, que permite un mejor tratamiento de los errores.

Definición

En Java, cuando se produce un error durante la ejecución del código, generalmente se lanza un objeto *Throwable*, el cual no es mas que un tipo especial de clase en Java.

(en la siguiente página se muestra un diagrama de la **jerarquía de errores** en Java)

Las excepciones (*Exception*) no son más que subclases de esta clase raíz *Throwable*, que almacenan información sobre errores o condiciones anómalas durante la ejecución, y transmiten dicha información fuera del flujo nomal de un programa. Se propagan hacia atrás (hacia "arriba") a través de la secuencia de llamadas habitual, hasta que son *capturadas* por alguna clase. Tras capturar la excepción, el control no vuelve al método en el cual se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se capturó la excepción.



La **diferencia** principal entre los elementos *Error* y las *Exception*, es que los primeros generalmente se deben a fallos irrecuperables, que el programador no debe tratar de gestionar (desbordamiento de memoria, errores en la JVM, etc), mientras que los segundos son errores que si pueden ser gestionados en el propio código.

Los dos tipos principales de *Exception* son:

- *RuntimeException*. Se deben a errores de programación: divisiones entre 0, acceso a un array fuera de sus límites, punteros a objetos *null*, etc.
- *IOException*. Son errores no evitables por el programador, generalmente relacionados con la entrada/salida el programa. Se dice que no son evitables, porque el programador normalmente no puede controlar el contexto externo.

Aunque existen más subclases de *Exception*, al mismo nivel que las dos antreriores, como pueden ser *GeneralSecurityException*, *DataFormatException*, *SQLException*, etc, las cuales más o menos son autodescriptivas según su propio nombre.

A continuación vamos a ver algunas de las excepciones (*Runtime*) más comunes:

Excepciones (*RuntimeException*)	**Significado**
ArithmeticException	Desbordamiento o división entera entre 0
NumberFormatException	Conversión ilegal de un *String* a un tipo numérico
IndexOutOfBoundsException	Acceso a un elemento fuera de los límites
ArrayIndexOutOfBoundsException	Extiende a la anterior, aplicada exclusivamente a arrays estáticos
NegativeArraySizeException	Intento de crear un array de longitud negativa
NullPointerException	Intento de uso de un puntero (referencia) nulo
SecurityException	Violación de seguridad en tiempo de ejecución
NoSuchElementException	El elemento pedido no existe (*getters*)

Se debe distinguir entre la captura y posterior manejo de una excepción, y el hecho de lanzar una excepción en un punto del programa como condición de error.

- La **captura** de una excepción, y su posterior manejo o tratamiento, es el proceso mediante el cuál el programador puede preveer la aparición de una excepción (un error) en algún punto del código, y, por tanto, puede establecer los pasos necesarios para tratar dicho error, y mitigarlo de la forma adecuada. Usamos: t*ry – catch - finally*.

- El **lanzamiento** de una excepción, es el proceso mediante el cuál el programador puede lanzar una excepción en un punto concreto del código, como indicativo de una condición de error que crea necesario notificar. Usamos: *throw* y *throws*.

Bloques *try*, *catch*, *finally*

Captura de una excepción

Los bloques *try*, *catch* y *finally* se utilizan para controlar la captura de excepciones que se hayan podido producir o lanzar durante la ejecución del programa.

A continuación, se muestra la estructura general con la que se utilizan dichos bloques:

```
try {
   // aquí está el código bajo supervisión de errores
   // es decir, las sentencias que puedan lanzar una excepción
}
catch(TipoExceptionConcreta nombreVariableException) {
   // aquí está el código de tratamiento de la excepción capturada
}
.
.
catch(Exception e) {
   // los bloques catch van del más específico al más general
}
finally {
   // (bloque opcional) contiene instrucciones que se ejecutarán siempre
}
```

El bloque ***try*** contiene todo el código sobre el que se intentará capturar el error que se produzca, es decir, será la porción de código bajo supervisión, por así decirlo. El conjunto de sentencias susceptibles de producir una excepción.

El bloque (o bloques) ***catch*** serán los que contengan el conjunto de instrucciones necesarias para tratar el problema o error capturado. Este bloque recibe como parámetro un objeto de tipo *Throwable* (lo que inclue, entre otros, *Exception*).

Los bloques *catch*, en caso de haber más de uno, siempre irán desde el más específico hasta el más general. Esto es así, porque si pusiéramos como primer bloque *catch* aquel que fuese más general (la clase *Exception*, por ejemplo, o incluso *Throwable*), las excepciones siempre serían capturadas en el primer bloque, ignorando el resto.

En caso de que una excepción producida en el *try* no sea capturada en ningún bloque *catch*, dicha excepción se propagará hacia arriba, llegando incluso hasta el *main* del programa, lo que originaría la terminación del mismo.

El bloque ***finally*** es opcional. Siempre se ejecutará, y contiene instrucciones que nos interesa asegurar que se realicen, independientemente de cómo haya transcurrido la ejecución del bloque *try*. Es decir, si existe el bloque *finally* (puesto que es opcional), su conjunto de instrucciones siempre se ejecutará al finalizar el resto de bloques, tanto si se han producido errores (dentro del *try*) como si no. Existen 3 posibilidades:

- Si no se producen excepciones, se ejecuta después del bloque *try*.
- Si se produce una excepción, se ejecuta después del bloque *catch* que la capture.
- Si se produce una excepción, y ninguna cláusula *catch* la está capturando, el bloque *finally* se ejecuta antes de que la excepción comience a propagarse hacia ⬆.

Un ejemplo de uso de este bloque es, por ejemplo, si en el bloque *try* abrimos un fichero para su lectura o modificación, deberemos cerrarlo en algún punto, para asegurar su consistencia. Si durante la ejecución del bloque *try* se produjera alguna excepción, deberemos asegurarnos que, después de que sea tratada o manejada, el fichero se cierre adecuadamente, cosa que se podría hacer en el bloque *finally*.

Manejo genérico de excepciones

Este código es la forma más genérica que hay de tratar un error cualquiera en Java, y de hecho, es la forma predeterminada que emplea el compilador. Básicamente, se captura el objeto de error, y se imprime su traza de ejecución:

```
public void miMétodo() {
    try {
      //código }
    catch (Throwable t) {
      t.printStackTrace(); } }
```

Sentencia *throw* y palabra reservada *throws*

Antes de comenzar con este apartado, hay que definir una nueva clasificación para las excepciones, y se trata de distinguir entre excepciones comprobadas (*checked*) y excepciones no comprobadas (*unchecked*). Lo utilizaremos más adelante.

- Excepciones ***checked***. Son excepciones que deben ser controladas en alguna parte del código, porque sino, derivan en un error de compilación. Se pueden controlar capturándolas y manejándolas, o bien relanzándolas (como veremos ahora) hacia arriba en la secuencia de llamadas.

  Pueden relanzarse incluso hasta el *main*, donde pueden volver a relanzarse una vez más, y, dado que no habría más llamadas por encima, haría finalizar el programa. O bien controlarse directamente en el *main*.

  Son excepciones comprobadas todas aquellas subclases de *Exception*, excepto las correspondientes a la clase *RuntimeException* y sus clases hijas. Es decir, todas aquellas excepciones que no son en tiempo de ejecución (*runtime*).

- Excepciones ***unchecked***. Son excepciones que no tienen porqué ser necesariamente controladas en ningún punto del código, aunque puede hacerse si se desea. En general, corresponden a errores de programación no previstos.

  Son excepciones no comprobadas únicamente la clase *RuntimeException* y todas las clases que hereden de ella, es decir, sus subclases. Nada más.

Lanzamiento de una excepción

La sentenica *throw* se usa para lanzar explícitamente objetos de tipo *Throwable*. La forma general de uso es '*throw objetoTipoThrowable*'. Ejemplos:

```
throw new Exception();     throw new RuntimeException("Mensaje");
```

Cuando se lanza una excepción, ocurre lo siguiente:

1. Se sale inmediatamente del bloque de código actual.
2. Si el bloque tiene asociada una cláusula *catch* adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la claúsula *catch*.
3. Sino, se sale del super-bloque (o método) dentro del cual esté el bloque en el que se produjo la excepción, y se busca una claúsula *catch* adecuada, moviéndose de forma ascendente en la secuencia de llamadas.
4. El proceso continúa hasta llegar al *main*. Si el *main* tampoco contiene una claúsula *catch* adecuada, la máquina virtual de Java se interrumpe y finaliza su ejecución mostrando un mensaje de error.

Propagación (declaración) de una excepción

El uso de la palabra reservada *throws* tiene su base en la siguiente pregunta: ¿cómo sabemos que algún método lanza una excepción que tenemos que controlar?

De la misma forma que un método declara el número y tipo de sus argumentos, así como el valor de su retorno, también debe declarar en su signatura las excepciones que dicho método pueda llegar a lanzar. Y con excepciones que pueda lanzar el método, se debe entender tanto aquellas que lanza directamente en su cuerpo, como aquellas que le llegan de clases inferiores que las han propagado, y dicho método a su vez las vuelve a propagar, es decir, las lanza de forma indirecta (relanzando).

Esto se hace con la palabra reservada *throws*, al final de la declaración del método:

```
public void ejemplo() throws Exception { ... }
```

También puede darse el caso de que un mismo método pueda lanzar varias:

```
public void ejemplo2() throws IOException, SQLException { ... }
```

En estos casos, además de declarar esa excepción, lo que estamos haciendo es *propagarla* en la secuencia de llamadas hacia arriba. Por tanto, dado que la propagamos, no es realmente necesario que la tratemos (manejemos) en ese método concreto; delegamos esa responsabilidad a clases superiores.

Una curiosidad que merece la pena comentar, es que si se declara que se lanzará una *Exception* de forma genérica, como se hace en el siguiente código:

```
public void ejemplo3() throws Exception { ... }
```

No es necesario declarar el resto de excepciones más específicas. Es decir, aunque el método lance varias excepciones de diferentes tipos, podemos declarar solo que lanzará un tipo *Exception* genérico, y el compilador no se lo tomará como un error.

Aún así, esta forma de programar no es para nada recomendable, puesto que al declarar el lanzamiento de excepciones, lo que queremos hacer precisamente es proporcionar información sobre ese método, a aquella persona que lo vaya a usar, por lo que, cuánto más específica y detallada sea esa información, mejor.

Por último, hay que comentar lo más importante sobre la declaración de excepciones:

'Un método solo está **obligado a declarar** mediante la claúsula *throws* aquellas excepciones que sean **excepciones comprobadas**. Es decir, aunque se puede hacer, **no es obligatorio**, ni dará un error de compilación, el **declarar excepciones no comprobadas** (como las de la clase ***RuntimeException*** *y* todas su subclases).'

Creación de excepciones propias (extends *Exception*)

Cuando es necesario, se pueden definir nuevas clases de excepciones más descriptivas usando herencia. Las nuevas clases se pueden definir de dos formas:

- Si queremos crear excepciones no comprobadas, haremos que extiendan (*extends*) directamente de la clase *RuntimeException*, o de alguna de sus clases hijas.

- Si queremos crear excepciones comprobadas, podrán heredar directamente de la clase *Exception*, o bien de alguna de sus subclases: *IOException*, *SQLException*, etc, excepto, lógicamente, de la clase *RuntimeException*.

Una de las principales **razones** para crear nuevas excepciones, es la de incluir en las mismas información más detallada acerca de la causa del fallo, para proporcionar un diagnóstico de error y recuperación más sencillo a las clases que capturen dicha excepción. Por ejemplo, es bastante común que en aplicaciones un poco complejas o grandes, se suela definir una jerarquía propia de excepciones (que herede de la jerarquía de Java), con el fin de representar las situaciones excepcionales específicas de esa aplicación concreta.

La forma de definir estos nuevos tipos de excepción es como sigue:

```
public class DivideByZeroException extends ArithmeticException {
  public DivideByZeroException() {
    // mensaje estándar para el nuevo tipo de excepción
    super("Error al dividir entre 0."); }

  public DivideByZeroException(String mensaje) {
    // mensaje personalizado al crear un objeto de la excepción
    super(mensaje); } }
```

Como se puede ver, basta con extender la excepción de la cual se quiera heredar, creando una subclase de la misma, la cual podrá ser lanzada como una excepción más en cualquier parte del código:

```
throw new DivideByZeroException("Mensaje personalizado.");
```

Ventajas del mecanismo de excepciones

Las excepciones, como otros tantos mecanismos y recursos en Java, son opcionales. Sin embargo, presentan algunas ventajas respecto a otras formas de tratar errores:

- ✔ Permiten **separar el código de tratamiento de errores**, del resto de código estándar del programa, disminuyendo el acoplamiento (dependencia entre componentes) y aumentando la cohesión (funciones similares agrupadas juntas) del código. Eso crea un flujo de programa más sencillo, y evita tener que tratar con complejos códigos de error en el código fuente principal.
- ✔ Permiten la **propagación de errores a lo largo de la pila de llamadas** a métodos. Evitando así por ejemplo, la necesidad de retornos adicionales (como los retornos *boolean* para indicar algún tipo de error).

  De hecho, es un mecanismo tan potente, que no solo permite propagar los errores desde bloques o métodos más internos a otros más externos, sino también desde clases servidoras o otras clases clientes, e incluso, en sistemas distribuidos (una sola instancia de programa funcionando en varias máquinas a la vez – programación distribuída), permiten propagar los errores entre distintas máquinas que ejecuten el programa.
- ✔ Permiten un agrupamiento y definición de tipos de errores como clases. Es decir, permiten **crear una auténtica jerarquía de errores**, bien estructurada, para tratar los errores a diferentes niveles de especificidad.