Tema 7. Excepciones y su tratamiento

Contenido

1 Introducción. Manejo de Excepciones.

Se denomina **excepción** a una **situación que no se puede resolver** y que provoca la detención del programa; es decir una **condición de error en tiempo de ejecución** (ocurre cuando el programa ya ha sido compilado y se está ejecutando). Ejemplos:

- El archivo que queremos abrir no existe.
- El programa pide un número y el usuario inserta letras.
- La clase que se desea utilizar no se encuentra en ninguno de los paquetes reseñados con **import**

Los errores de sintaxis son detectados durante la compilación, pero las excepciones pueden provocar situaciones irreversibles y su control debe hacerse en tiempo de ejecución.

Hay cantidad de errores que se pueden presentar en cualquier programa. En casi todos ellos es necesario comprobar determinadas circunstancias que pueden provocar fallos. Por ejemplo, podemos pensar en la necesidad de controlar que no se realicen divisiones por cero, o evitar que se acceda a una posición de un vector que no existe, o impedir que se intente comprobar un carácter de un String más allá del último (o anterior al primero), introduciendo un valor negativo para la posición dentro del String o el índice del vector.

También tenemos que comprobar cada vez que leemos un número desde teclado que lo que se teclea se puede considerar un número, y se puede convertir en número sin que el programa aborte cuando introducimos caracteres extraños. Por ejemplo **Xt56?e4** no se podrá convertir en número, por más que nos empeñemos, así que tenemos que asegurarnos de que cuando se espere un número y se teclee esa cadena, el error no va a suponer que nuestro programa termine bruscamente, abortando.

Es importante, al escribir un programa, tener en cuenta este tipo de situaciones que se pueden llegar a dar al ejecutarlo.

Por ejemplo, si el programa pide un número y el usuario inserta letras, para evitar que se cierre el programa de forma abrupta podemos hacer una de las dos opciones que siguen:

- Cerrar el programa con un mensaje que indique que hubo un error al introducir los datos.
- Pedir al usuario que introduzca el número otra vez.

El control del flujo de instrucciones de un programa en Java se puede realizar con las estructuras `if`, `switch`, `for`, `while`, `do-while`, y también con el **control de excepciones**.

2 Excepciones básicas

En general, **por excepción entenderemos una situación anómala en la ejecución de un programa, que impide que se siga ejecutando el flujo normal del programa, sin que en el ámbito en que se produce esa situación de error tengamos información suficiente para corregirlo**, por lo que debemos pasar el control del flujo del programa a otro ámbito en el que quizás sea posible manejar ese error disponiendo de más información. Así se propaga la excepción al método desde el que se invocó el código que generó la excepción, que si tampoco la sabe tratar la pasará a su vez al método que lo invocó, y así sucesivamente, hasta en el peor de los casos llegar al método `main()`, que le pasará la excepción a la máquina virtual Java, que escribirá un mensaje de error indicando el error ocurrido, escribirá la lista de invocaciones a métodos que han producido la excepción, y terminará la ejecución del programa. Esta es la situación menos deseable, pero en cualquier caso garantiza que cualquier excepción va a ser tratada, y no va a poder ignorarse sin más.

En Java existe la clase `Throwable`, y cualquier situación anómala en la ejecución de un programa genera directa o indirectamente un objeto de la clase `Throwable`.

La clase `Throwable` tiene dos subclases definidas en Java, cada una para un tipo de error o situación anómala en la ejecución:

- `Error`. Indica que se ha producido un fallo irrecuperable, para el que es imposible recuperar y continuar la ejecución del programa. La máquina Virtual Java presenta un mensaje en el dispositivo de salida, y concluye la ejecución del programa. Para este tipo de errores, no hay nada que el programador pueda hacer. Podemos encontrar la información adicional sobre sus posibles subclases en la especificación de la API de Java.
- **`Exception`**. Dentro de ella se puede distinguir:
  1. **`RuntimeException`**: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar excepciones implícitas.
  2. **Las demás clases** derivadas de **`Exception`** son excepciones explícitas. Java obliga a tenerlas en cuenta y chequear si se producen.

En el caso de `RuntimeException` el propio Java, durante la ejecución de un programa, chequea y lanza automáticamente las excepciones que derivan de esta clase. El programador no necesita establecer los bloques **`try/catch`** para controlar este tipo de excepciones. Representan dos casos de errores de programación:

1. Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia `null` en un método.
2. Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un `ArrayIndexOutOfBoundsException` automáticamente).

Se podría comprobar estos tipos de errores, pero el código se complica excesivamente si hay que comprobar continuamente todo tipo de errores (que las referencias son distintas de **`null`**, que todos los argumentos de los métodos son correctos, etc).

Las clases derivadas de **`Exception`** pueden pertenecer a distintos packages de Java. Algunas pertenecen a `java.lang` (`Throwable`, `Exception`, `RuntimeException`,…); otras a `java.io` (`EOFException`, `FileNotFoundException`, …) o a otros packages. Por heredar de `Throwable` todos los tipos de excepciones pueden usar los métodos siguientes:

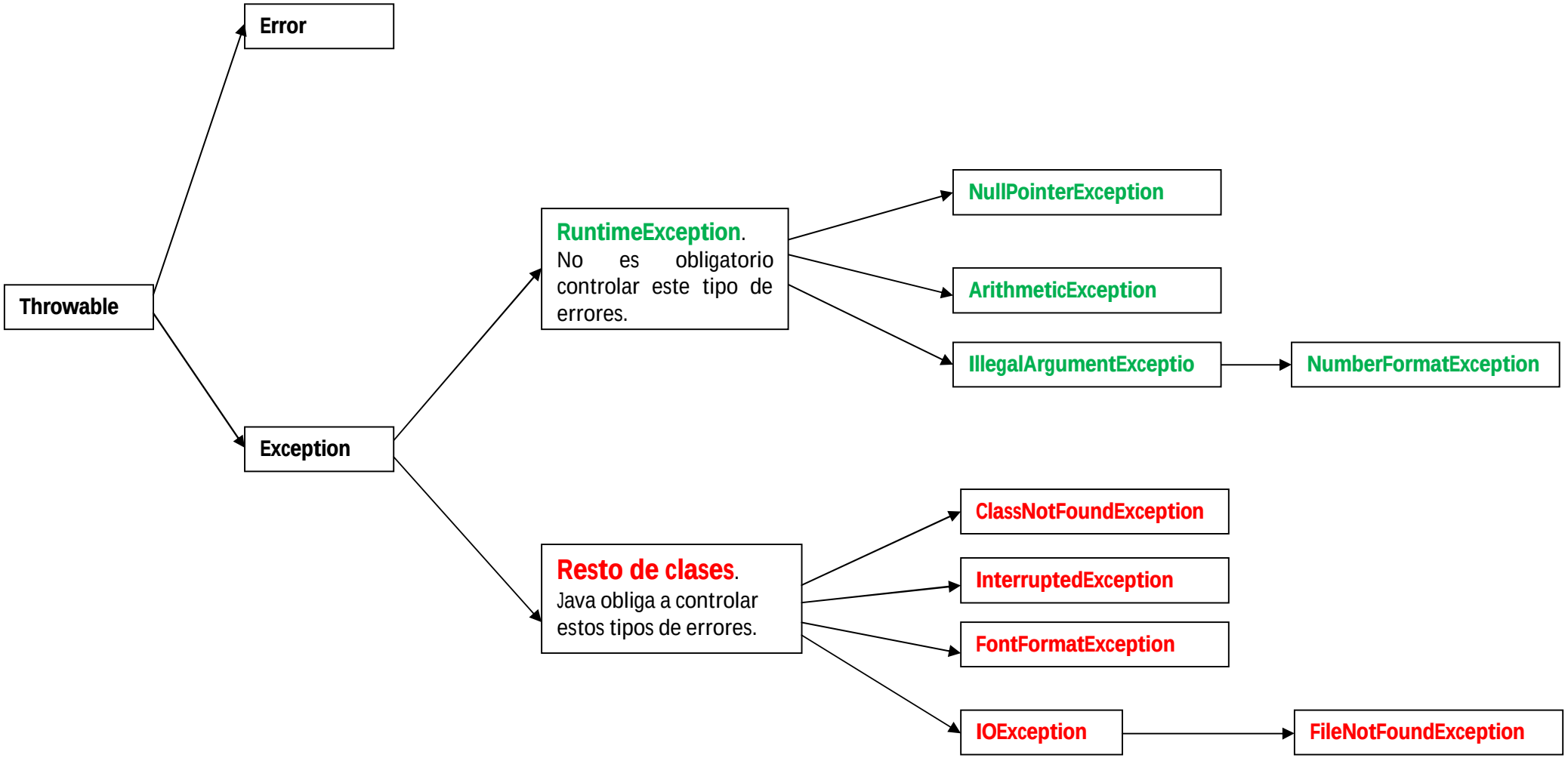
1. `String` **`getMessage`**(). Extrae el mensaje asociado con la excepción.
2. `String` **`toString`**(). Devuelve un String que describe la excepción.
3. `void` **`printStackTrace`**(). Indica el método donde se lanzó la excepción.

Así por ejemplo, una de las subclases de la clase `Exception` es `IOException`, encargada de capturar errores de Entrada/Salida. Tiene a su vez 26 subclases, entre las que se encuentra `EOFException`, que se lanza cuando se alcanza inesperadamente el final de un fichero realizando una entrada de datos desde un fichero o flujo.

Además de la inmensa cantidad de subclases de `Exception` que ya nos da definidas el lenguaje Java, y que abarcan la práctica totalidad de situaciones problemáticas que se puedan imaginar, **el programador puede definir, lanzar y usar sus propias excepciones en Java**, por lo que las posibilidades de capturar y tratar errores son infinitas.

En general, la sintaxis del manejo de excepciones en Java incluye el uso de las siguientes palabras reservadas: **`try, catch, finally, throw, throws.`**

JERARQUÍA DE EXCEPCIONES



3 Manejo de excepciones

3.1 Capturar una excepción (try - catch - finally)

¿Cómo identificar en el programa el código que puede generar excepciones?

- **De ello se encarga el bloque de código que comienza con la palabra reservada `try`**, seguida de un bloque de sentencias entre llaves, que **son el código que puede generar el error**.

¿Cómo indicar el código al que se transfiere el control en el caso de que se produzca la excepción?

- **Es la misión del bloque iniciado por la palabra reservada catch seguida de un paréntesis que contiene la declaración del objeto de tipo `Exception` (o subclase de `Exception`) que se creará si se produce el error**. La cláusula `catch` lo recibe como si fuese un parámetro. El paréntesis va seguido de un bloque de sentencias contenidas entre llaves, que **constituyen el manejador de la excepción**, indicando lo que debe hacerse en el caso de que se produzca el error. El bloque `catch` irá siempre después del bloque `try`.

  Puede haber varios bloques `catch` para cada bloque `try`, cada uno encargado de indicar lo que debe hacerse en el caso de que se presente un determinado tipo distinto de excepción dentro de las sentencias del bloque `try`. No se puede tener un bloque `try` sin ningún bloque `catch`, o viceversa. Sólo se permite un bloque `try` sin ningún bloque `catch` si se ha incluido un bloque `finally`. En definitiva, **debe haber siempre un código que se ejecute cuando se lanza la excepción, que no puede quedar sin tratamiento.**

- Por último **podemos colocar opcionalmente un bloque `finally`**, que encerrará entre llaves un grupo de sentencias que se ejecutarán siempre, tanto si se produce una excepción como si no. Siempre que se incluya una cláusula `finally`, debe colocarse detrás del último bloque `catch`. Normalmente la cláusula `finally` no es muy utilizada, pero su principal utilidad es garantizar la liberación de recursos que el bloque `try` ha acaparado con uso exclusivo, y que podrían quedar definitivamente retenidos por él si no se ejecutan las sentencias que los liberan debido a que una excepción se produce antes de las sentencias encargadas de esa liberación del recurso, transfiriendo el control al bloque `catch`.

  Por ejemplo, si en el bloque `try` se reserva un fichero para escritura, ningún otro programa podrá escribir en ese fichero hasta que el bloque `try` concluya y libere ese recurso. Pero si se produce una excepción en el bloque `try` antes de haber alcanzado la sentencia que libera el recurso, debe tenerse en cuenta e incluir en el bloque `catch` correspondiente el código necesario para liberar el fichero en cuestión. Para no tener que repetir ese código que libera el recurso tanto en el bloque `try` como en el bloque `catch`, Java crea la cláusula `finally`, que se ejecutará siempre, tanto si se produce la excepción como si no. Si colocamos las sentencias que liberan el recurso en la cláusula `finally`, estamos garantizando que pase lo que pase, el recurso se liberará correctamente.

  Otro ejemplo de recurso que debería ser liberado una vez que no es necesario puede ser una conexión a una base de datos. La mayoría de las bases de datos tendrán establecido un número máximo de usuarios que pueden acceder simultáneamente a los datos. Si nuestra aplicación establece la conexión, y debido a un error, no la cierra, puede estar impidiendo que otro usuario pueda conectarse a la base de datos, aunque nuestra aplicación ya no esté usando esa conexión.

`Ejemplo01, Ejemplo02, Ejemplo02b`

Cambia el último ejemplo añadiendo un bloque `finally` en el que se muestre un mensaje y comprueba que siempre muestra dicho mensaje, independientemente de que se produzca un error o no.

3.2 Propagar excepciones

Cuando se produce un error dentro de un método, se puede capturar dentro de él y tratarlo desde ahí mismo. Si no se captura el error, Java lo propaga automáticamente al método que lo llamó para que dicho método lo capture; y así sucesivamente hasta llegar al método `main`. Si el método `main` tampoco lo trata se propaga al sistema operativo.

Sólo se propagan las excepciones que derivan de la clase `RuntimeException`. Los errores de la clase `Exception` que no sean o deriven de la clase `RuntimeException` no se propagan y es obligatorio capturarlos o lanzarlos.

`EjemPropagar, EjemPropagar_b`

3.3 Lanzar excepciones

Se pueden dar dos situaciones distintas:

1) Dentro de un método donde se puede producir una de las excepciones que Java obliga a controlar (por ejemplo `IOException`)

   Dentro de dicho método existen dos opciones:

   a) Capturar la excepción con el `try-catch`
   b) Lanzar la excepción. Un método lanza la excepción poniendo esta cabecera al método:

      tipoQueDevuelve nombreMetodo(parámetros) `throws NombreClaseException`

   `EjemLanzar01`: si en este ejemplo no queremos controlar la excepción dentro del método, tenemos la obligación de lanzarla. Esta excepción la recibe el método que llamó al que la ha lanzado, que puede capturarla o volver a lanzarla. Y así sucesivamente hasta que se capture o sea lanzada por el método `main`.

   Un método puede lanzar más de una excepción. En ese caso la cabecera de dicho método sería:

      tipoQueDevuelve nombreMetodo(parámetros) `throws exception1, excepcion2` …

   Las excepciones que deriven de la clase `RunTimeException`, como por ejemplo `NumberFormatException`, no hace falta lanzarlas porque si no se capturan dentro del método se propagan automáticamente.

   Dentro de un método queremos controlar una excepción (hay que capturarla con el bloque `try-cath`) y también queremos propagarla al método que lo ha llamado, por lo que dentro del bloque catch lanzaremos la excepción con la siguiente orden:

      `throw (nombreObjetoTipoExcepcion)`

   Cuando se ejecuta la instrucción throw se abandona el flujo de ejecución del método en el que está y se cede el control al método llamante, en concreto al bloque catch que capture la excepción que se ha lanzado.

   Si la excepción que se ha lanzado NO pertenece a la clase `RunTimeException`, en la cabecera de dicho método hay que poner

      tipoQueDevuelve nombreMetodo(parámetros) `throws NombreClaseExceptionQueDevuelveThrow`

   Si la excepción pertenece a la clase `RunTimeException`, no es obligatorio poner la cabecera anterior.

`EjemLanzar01, EjemLanzar02`

4 Normas para el manejo de excepciones

Podemos resumir un poco lo dicho anteriormente en una serie de normas. A la hora de capturar excepciones, hay varias cosas a tener en cuenta:

- **Cuando hay una excepción que se lanza desde el bloque `try`, se transfiere el control al primer bloque `catch` cuyo parámetro coincida con el tipo de excepción que se ha lanzado**.
- **Si se coloca primero un bloque `catch` para una excepción más genérica, siempre se ejecutará ese bloque `catch`, y los que aparezcan detrás, correspondientes a subclases de excepciones de la anterior, no se alcanzarán ni se ejecutarán nunca**. Por ejemplo, si el primer bloque `catch` captura una `Exception`, poner detrás una captura de `NumberFormatException` es inútil, ya que es una subclase. Para Java una `NumberFormatException` no deja de ser un tipo particular de `Exception` (una subclase de `Exception`) por lo que se considerará que es el tipo adecuado, y se ejecutará el manejador para la clase `Exception`. Si posteriormente hay un `catch` para capturar `NumberFormatException`, no se ejecutará nunca, porque se considera que la excepción ya ha sido tratada convenientemente por el manejador de `Exception`. De hecho, el compilador avisará diciendo que la excepción `NumberFormatException` ha sido ya capturada. Sí se pueden colocar en orden contrario, como en el ejemplo del apartado anterior, de forma que si se produce `NumberFormatException`, se ejecuta el código adecuado de su manejador, y si se produce cualquier otro tipo de excepción, será el manejador genérico para `Exception` el que se ejecutará.
- Cuando se ejecutan todas las sentencias del manejador de la excepción, se continúa con la sentencia siguiente, es decir, **en el manejador de la excepción nunca se produce un salto hacia atrás, a la sentencia que produjo la excepción**.
- **Si la excepción que se lanza desde el bloque `try` no se captura por ningún `catch` adecuado, se relanza hacia el método anterior en la cadena de llamadas, hasta llegar a un lugar donde se capture y se trate adecuadamente**.
- **En el peor de los casos será la propia máquina virtual la que se encargue de tratar la excepción**, escribiendo un mensaje de error y terminando la ejecución del programa. Es lo único que puede hacer la máquina virtual de forma genérica para cualquier excepción que le llegue sin haber sido capturada adecuadamente, ya que no tiene más información que le permita intentar una recuperación del error.
- **Si se escribe el bloque opcional `finally`, irá después del último `catch`, y se ejecutará siempre,** se produzca o no alguna excepción, y con independencia del tipo de excepción de la que se trate.
- Si en las sentencias del bloque `catch` se produjera una nueva excepción, sería esta última la única que se propagaría hacia los métodos llamantes.
- Aunque se puede transferir el control desde dentro del bloque `try` hacia otras zonas del código, usando `break`, `continue` o `return`, no es demasiado aconsejable hacerlo, y se debe tener presente que **si se ha especificado una cláusula `finally`, primero se ejecutará la cláusula `finally`, y luego se transferirá el control a la zona del programa que corresponda**.
- En el bloque `catch` se recibe como parámetro el objeto `Exception` del tipo de la excepción que se ha producido en el bloque `try`, y se le pueden enviar una serie de llamadas a métodos para obtener información sobre la excepción que se ha producido. Algunos de esos métodos son:
    - `getMessage()` , que recoge el mensaje de error asociado al tipo de excepción que se ha producido.
    - `printStackTrace()`, que escribe la cadena de llamadas que han generado la excepción, es decir el método que produjo la excepción, indicando desde qué otro método fue llamado, y a su vez desde qué otro método fue llamado éste, y así sucesivamente hasta llegar al método main(),desde el que se comienza la ejecución de todo el programa.

5 Creación de excepciones propias. Cláusulas throw y throws

A pesar de que Java nos proporciona una variedad suficientemente amplia de excepciones para todo tipo de situaciones posibles de error, también es interesante poder crear excepciones propias, de forma que los mensajes de error estén personalizados, o que podamos decidir las condiciones bajo las que deben lanzarse esas excepciones. Java una vez más nos ofrece esta posibilidad, aumentando la flexibilidad, y lo hace mediante las cláusulas `throw` y `throws`.

- **Debemos definir una clase que declare la excepción que queremos crear como una subclase de alguna de las excepciones definidas por el lenguaje**. Naturalmente puede ser una subclase de `Exception`.
- **`throw` Se usa para indicar en qué lugar del código de nuestro método se lanza esa excepción, y para lanzarla, de hecho.**
- `throws` Se usa en la cabecera del método para avisar a los programadores usuarios de nuestro método de que lanza un determinado tipo de excepción, y que deberán preocuparse de capturarla y manejarla convenientemente en los programas que usen ese método.

Ejemplo *EjemCrear*.