C.F.G.S. DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MODULO DE PROGRAMACIÓN

TRATAMIENTO DE ERRORES EN JAVA CON EXCEPCIONES

Tratamiento de Errores en Java

Excepciones

1. Introducción

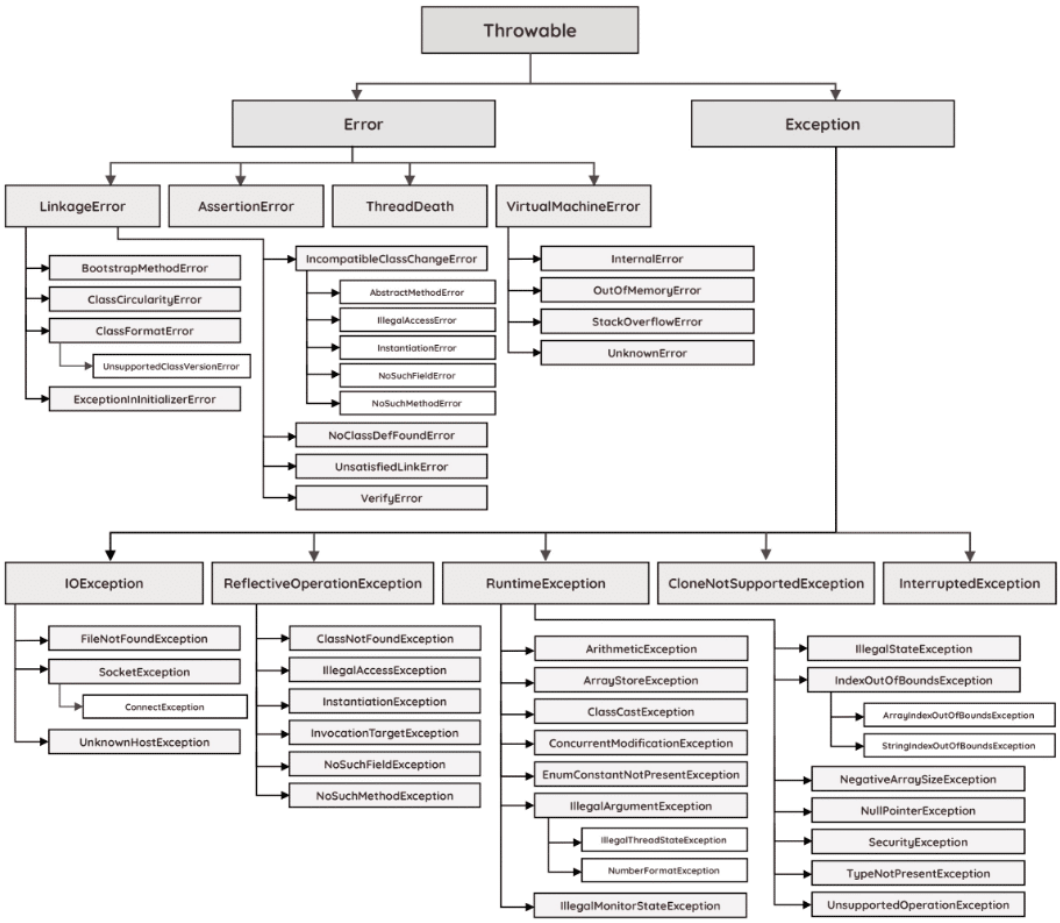
Tendemos a pensar que los programas se ejecutan en un entorno ideal exento de problemas que puedan afectar a su buen funcionamiento. Sin embargo, la realidad es bien distinta y los programas se ejecutan en entornos donde los archivos se pueden corromper, el hardware puede fallar, la red se puede caer, la memoria del sistema es finita y se puede agotar si no se gestiona de forma eficiente, se cometen errores de programación, etc. Por tanto, la salud y el buen desempeño de cualquier programa depende en buena medida de la diligencia del programador en el tratamiento de estas situaciones. Java define un sistema de tratamiento de errores mediante excepciones con el objetivo de:

- Facilitar la realización de esta tarea de forma eficiente para crear programas fiables, robustos y mantenibles.
- Facilitar la detección y depuración de errores de programación.

Se entiende por excepción la interrupción del flujo normal de ejecución de un programa para poner de manifiesto la ocurrencia de una condición de error. Este proceso involucra la creación de un objeto `Throwable` que represente la condición de error y su lanzamiento mediante la ejecución de una sentencia `throw`. Esto último, que se conoce como "lanzamiento de la excepción", es lo que provoca la interrupción del flujo normal de ejecución.

El lanzamiento de excepciones irá seguido de su tratamiento, que consiste en decidir como se va a gestionar la interrupción del flujo normal de ejecución. Las dos alternativas posibles, manejo de la excepción o propagación al contexto de llamada, se explicarán detalladamente más adelante.

La clase `Throwable` se encuentra en la raíz de una jerarquía de clases del API de java que representan una serie de errores comunes que se pueden producir durante la ejecución de un programa.



Estas clases se pueden extender para definir nuevas excepciones que representen errores personalizados.

2. Tipos de excepciones

En función de la naturaleza del error que representan, se distinguen dos tipos de objetos `Throwable`:

- **Clase `Error` y sus subclases**: representan errores severos relacionados con fallos de hardware o fallos de programación de los que normalmente resulta imposible que el programa se recupere. Por tanto, en la mayoría de los casos no tiene sentido su manejo, instanciación o extensión.

  La ejecución del ejemplo siguiente provocará el lanzamiento de `StackOverflowError`:

```
1  public class MiClase {
2          MiClase ejemplo = new MiClase();
3
4          public static void main(String[] args) {
5                  MiClase ejemplo = new MiClase();
6                  System.out.println("Esta sentencia no se ejecutará");
7          }
8  }
```

- **Clase `Exception` y sus subclases**: representan errores que tienen su origen en:
  - Fallos de programación que se van a manifestar en tiempo de ejecución, como por ejemplo utilización de índices fuera de rango en el acceso a los elementos de un array, utilización de referencias nulas o división entera por cero.
  - Situaciones ajenas a la lógica del programa. Son las derivadas de la interacción del usuario con el programa (por ejemplo, el usuario no introduce correctamente los datos que se le piden o se intenta leer un archivo que no existe) o las relacionadas con un mal funcionamiento del hardware (por ejemplo, lectura de archivos en dispositivos corruptos o averiados).

En función de cómo se debe llevar a cabo su tratamiento, se distinguen dos tipos de excepciones:

- **Checked**: dentro de esta categoría se encuentran la clase `Exception` y sus subclases, a excepción de la clase `RuntimeException` y sus subclases.

  En cualquier método en el que se lance una excepción checked, o se invoque a otro método que declare el lanzamiento de una excepción checked, el usuario estará obligado a capturarla o a declarar su lanzamiento en la cabecera del método usando la cláusula `throws`. De no hacerlo, se producirá un error de compilación.

- **Uncheked**: dentro de esta categoría se encuentran la clase `Error` y sus subclases, y la clase `RuntimeException` y sus subclases.

  No existe la obligación de capturarlas ni de declarar su lanzamiento (esto último se hace de forma implícita si no se capturan) debido a que se su manejo resulta de poca utilidad en la mayoría de los casos. En el caso de la clase `Error` ya se han expuesto las razones por lo que esto es así. En el caso de excepciones `RuntimeException`, habitualmente están relacionadas con errores de programación y su lanzamiento facilita su detección y depuración, por lo que no tiene sentido capturarlas (aunque no siempre tiene porque ser así).

En función del contexto en el que tiene lugar su lanzamiento, podemos distinguir dos tipos de excepciones:

- **Excepciones de la JVM**: la máquina virtual de Java es la responsable del lanzamiento de este tipo de excepciones.
- **Excepciones programáticas**: el lanzamiento de este tipo de excepciones es responsabilidad de los constructores y métodos que se definen en las clases del API de Java, en las clases de las librerías de terceros o en las clases de la propia aplicación en desarrollo.

3. Lanzamiento de excepciones

Cuando se detecta una condición de error en un programa Java y se desea tratar con excepciones, se lanza la excepción que representa el error con la sentencia `throw`. De forma genérica:

```
tipo_retorno unMetodo(parámetros_formales){
    ⋮
    if (condición_de_error)
        throw excepción;
    ⋮
}
```

Si la excepción es de tipo checked, es obligatorio declarar su lanzamiento en la cabecera del método o constructor usando una cláusula `throws`:

```
tipo_retorno unMetodo(parámetros_formales) throws excepción_de_tipo_checked {
    ⋮
    if (condición_de_error)
        throw excepción_de_tipo_checked;
    ⋮
}
```

El lanzamiento de una excepción con una sentencia `throw` está sujeto a las consideraciones siguientes:

- Provoca que se interrumpa la ejecución del método propagando la excepción al contexto de llamada donde se llevará a cabo su tratamiento.
- Si se hace en un constructor se abortará la construcción del objeto y, por ende, cualquier posible asignación del valor retornado por el operador `new`.
- Es posible lanzarla y capturarla en el mismo método, aunque por razones obvias no es una práctica habitual.
- No se permite su uso dentro de un bloque de inicialización, a no ser que en el mismo bloque exista un manejador que capture la excepción.

3.1. Constructores y métodos de las clases Throwable

En la clase `Throwable`, situada en la raíz de la jerarquía de excepciones Java, se definen varios constructores públicos que van a tener sus equivalentes en todas sus descendientes:

`Throwable()`

Construye un objeto **`Throwable`** sin asociar ningún un mensaje.

`Throwable(String message)`

Construye un objeto **`Throwable`** con un mensaje asociado.

`Throwable(String message, Throwable cause)`

Construye un objeto **`Throwable`** con un mensaje y una causa. Normalmente la causa es una referencia a una excepción lanzada y capturada antes de la creación de este objeto.

`Throwable(Throwable cause)`

Construye un objeto **`Throwable`** con un una causa y un mensaje que se obtiene con la expresión:

```
(cause == null ? null : cause.toString())
```

La clase `Throwable` también define varios métodos públicos que pueden resultar de utilidad durante el manejo de una excepción, entre los que se encuentran los siguientes:

Throwable	**getCause**() Retorna la causa que fue asociada a esta excepción o `null` si no tiene asociada ninguna causa.
String	**getMessage**() Retorna el mensaje asociado a la excepción.
void	**printStackTrace**() Muestra en la salida estándar de error información de la excepción. La primera línea contiene el resultado de invocar a su método `toString`, el resto de las líneas muestran una traza sobre la pila de llamadas con información recopilada durante su recorrido hacia atrás.
void	**printStackTrace**(**PrintStream** s) Lo mismo que la anterior, pero enviando el texto al stream de salida suministrado. Se usa, por ejemplo, para guardar la traza en un fichero o enviarla a través de una conexión de red.
void	**printStackTrace**(**PrintWriter** s) Lo mismo que la anterior, pero usando un writer en lugar de un stream.

4. Tratamiento de excepciones

El lanzamiento de una excepción y su tratamiento puede tener lugar en dos contextos posibles, dentro del cuerpo de un método o dentro de un bloque de inicialización.

En el primer caso, el tratamiento de una excepción consiste en realizar una de las dos acciones siguientes:

- Capturar la excepción para su manejo. Esta será la opción lógica cuando se tiene información acerca de las causas que provocaron su lanzamiento y se sabe qué hacer para que, en la medida de lo posible, el programa se recupere de esa situación y continue su ejecución.
- Dejar que se propague al contexto de llamada, donde se espera que se tenga la información necesaria para su tratamiento. Si es una excepción checked, es obligatorio declarar su lanzamiento en la cabecera del método actual usando una cláusula `throws` o de lo contrario se producirá un error de compilación.

  Si la excepción se propaga a lo largo de toda la pila de llamadas, llegará a la JVM y será esta la que se encargue de su manejo, que consiste en finalizar de forma abrupta del programa mostrando en la consola una traza de la pila de llamadas acompañada de los mensajes de error pertinentes.

En los bloques de inicialización es obligatorio capturar las excepciones checked, pero no las unchecked. Si estas últimas no se capturan, el programa no se ejecuta, tal y como se muestra en el ejemplo siguiente:

```
1  public class MiClase {
2      static int[] array;
3
4      static {
5          array = new int[10];
6          for (int i=0; i<=array.length; i++)
7              array[i] = i + 1;
8      }
9
10     public static void main(String[] args) {
11         System.out.println(Arrays.toString(array));
12     }
13 }
```

La ejecución de la sentencia de la línea 7 en la última iteración del bucle provocará el lanzamiento de `ArrayIndexOutOfBoundsException`. Como consecuencia el método `main` no se ejecutará y se mostrará la salida siguiente en la consola:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 10
        at MiClase.<clinit>(MiClase.java:14)
```

Por razones obvias, no tiene sentido pensar en capturar excepciones relacionadas con errores de programación. De ahí que los diseñadores del lenguaje Java hayan decidido representar este tipo de errores con excepciones unchecked. No obstante, el lanzamiento de excepciones de este tipo no siempre tiene por qué estar relacionado con errores de programación. Por ejemplo, el método `parseInt` de la clase `Integer` lanza la excepción `NumberFormatException` si recibe una secuencia de caracteres que no se pueda interpretar como un número entero. Esta situación se puede dar, por ejemplo, si se le pasa directamente a este método una entrada de datos por teclado y el usuario introduce los datos de forma incorrecta. Mas allá de considerar esto como un error de programación, se puede aprovechar el manejo de esta excepción para implementar un método de validación de la entrada de datos por teclado. Por tanto, es conveniente que el programador esté prevenido ante la posibilidad de que un método lance excepciones, especialmente cuando son de tipo unchecked. La mejor forma de hacerlo, sobre todo cuando se usan clases del API de java o de librerías de terceros, consiste en acudir a su documentación, donde se especificará para cada método qué excepciones puede lanzar y sus causas.

Enlace de interés: Unchecked Exceptions — The Controversy

4.1. Captura y manejo de excepciones

La captura de una excepción se lleva a cabo usando bloques `try-catch`:

```
try {
    /*
     * Una o varias sentencias, una de las cuales
     * puede provocar el lanzamiento de la excepción.
     */
} catch (excepción identificador) {

    /* "identificador" es una referencia al objeto Throwable que se ha capturado */

    /* Sentencias encargadas del manejo de la excepción: */

     ⋮
}
```

En el bloque catch se capturará el tipo de excepción especificado y cualquiera de sus subclases.

Es posible capturar más de una excepción en un bloque `catch` en el caso de dentro del bloque `try` se pueda lanzar más de una excepción:

```
try {

    ⋮

} catch (excepción1 | excepción2 | … identificador) {

    ⋮

}
```

En la lista de excepciones no se pueden incluir subclases si ya se va a incluir su superclase.

Una alternativa al caso anterior consiste en usar varios bloques catch para capturar y manejar cada excepción por separado:

```
try {
    ⋮
} catch (excepción1 identificador) {
    ⋮
} catch (excepción2 identificador) {
    ⋮
}
⋮
```

Si es necesario capturar una superclase y cualquiera de sus subclases por separado, el bloque `catch` que captura la subclase debe de aparecer antes que el bloque `catch` que captura la superclase.

La construcción `try-catch` funciona de la forma siguiente:

- Si se completa la ejecución del bloque `try` sin que se lance la excepción, esta continuará después del último bloque `catch`.
- Si alguna de las sentencias del bloque `try` provoca el lanzamiento de la excepción, este finaliza su ejecución inmediatamente sin ejecutar el resto de las sentencias y se transfiere la ejecución al bloque `catch` que captura la excepción. Una vez completada su ejecución, esta continuará en la primera sentencia que se encuentre después del último bloque `catch`.

El ejemplo siguiente muestra como capturar la excepción `NumberFormatExeception` para validar una entrada de datos por teclado:

```
1  public class RaizCuadrada {
2
3      public static void main(String[] args) {
4          Scanner in = new Scanner(System.in);
5          Double n;
6          boolean correcto;
7          do {
8              System.out.print("Numero: ");
9              try {
10                 n = Double.parseDouble(in.nextLine());
11                 in.close();
12                 correcto = true;
13             } catch (NumberFormatException e) {
14                 System.out.println("El dato introducido no es un número.");
15                 System.out.println("Introdúzcalo de nuevo.");
16                 correcto = false;
17             }
18         } while (!correcto);
19         System.out.println("Resultado: " + Math.sqrt(n));
20     }

}
```

En el bloque `try` se invoca al método `parseDouble` que, como se indica en su documentación, lanza la excepción `NumberFormatException` si el dato recibido en forma de cadena no se puede representar como número de tipo `double`. Este método recibe el valor retornado por el método `nextLine` del `Scanner` cuando el usuario introduce un dato por teclado. En este contexto pueden ocurrir dos cosas:

- El usuario introduce un dato correcto, no se lanza la excepción y se ejecutan el resto de las sentencias del bloque `try`. En la línea 12 se actualiza la variable que provoca que el bucle finalice y se muestre el resultado.
- El usuario introduce un dato incorrecto, la ejecución de la línea 10 provoca el lanzamiento de la excepción que será capturada en el bloque `catch`. El manejo de esta consiste en informar al usuario del error y evitar que el bucle finalice para que se repita la entrada de datos.

Una versión simplificada de este ejemplo, pero no estructurada, sería la siguiente:

```
public class RaizCuadrada {


    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Double n;
        do {
            System.out.print("Numero: ");
            try {
                n = Double.parseDouble(in.nextLine());
                in.close();
                break;
            } catch (NumberFormatException e) {
                System.out.println("El dato introducido no es un número.");
                System.out.println("Introdúzcalo de nuevo.");
            }
        } while (true);
        System.out.println("Resultado: " + Math.sqrt(n));
    }

}
```

A continuación, se muestra un ejemplo de la salida por pantalla que se produce cuando se ejecuta este programa y se captura la excepción:

```
Numero: abc
El dato introducido no es un número.
Introdúzcalo de nuevo.
Numero:
```

4.2. El bloque finally

Java también define la construcción `try-catch-finally` para el tratamiento de excepciones. El bloque `finally` se usa al final de todos los bloques catch para asegurarse de que un conjunto de sentencias se ejecute independientemente de que en el bloque `try` se lance o no una excepción, o de si se captura o no en caso de que se lance alguna. El ejemplo siguiente permite comprobar su funcionamiento:

```
public class UnaClase {

        public static void main(String[] args) {
                Scanner s = new Scanner(System.in);
                System.out.print("Introduce un número entero positivo mayor que cero: ");
                try {
                        int n = Integer.parseInt(s.nextLine());
                        if (n <= 0)
                                throw new RuntimeException();
                } catch (NumberFormatException e) {
                        System.out.println("en el bloque catch");
                } finally {
                        System.out.println("en el bloque finally");
                }
                System.out.println("fin");
        }

}
```

Cuando el usuario introduce algo que no es un número, en el bloque `try` se lanza la excepción `NumberFormatException`, que se captura en el bloque `catch`. La salida muestra que se ejecuta el bloque `catch`, pero también el bloque `finally` y las sentencias posteriores al `try-catch-finally`:

```
Introduce un número entero positivo mayor que cero: abc
en el bloque catch
en el bloque finally
fin
```

Si el usuario introduce un número entero menor o igual que cero, se lanza la excepción `RuntimeException`, que no se captura. Por tanto, el método `main` finaliza sin ejecutar las sentencias posteriores al `try-catch-finally`. Sin embargo, antes de dicho retorno se ejecutará nuevamente el bloque `finally`:

```
Introduce un número entero positivo mayor que cero: 0
en el bloque finally
Exception in thread "main" java.lang.RuntimeException
        at UnaClase.main(UnaClase.java:11)
```

Por último, se muestra la salida cuando no se lanza ninguna excepción, donde nuevamente se puede comprobar que también se ejecuta el bloque `finally` y las sentencias posteriores al bloque `try-catch-finally`:

```
Introduce un número entero positivo mayor que cero: 1
en el bloque finally
fin
```

Cuando se utiliza el bloque `finally` no es obligatorio incluir ningún bloque `catch`, es decir, las estructuras `try-finally` están permitidas. Esto permite asegurar, con posterioridad al lanzamiento de una excepción, la ejecución de un conjunto de sentencias, aunque no se haya capturado dicha excepción.

Si se lanza alguna excepción en el bloque `finally`, cualquier excepción previa que se haya lanzado en el bloque try y que no se haya capturado se suprime en favor de esta. La clase `Throwable` define los métodos `addSuppressed` y `getSuppressed` para que se pueda asociar una excepción suprimida a una excepción lanzada en el bloque `finally` y consultar posteriormente todas las excepciones suprimidas a lo largo de la pila de llamadas.

4.3. Relanzar excepciones

Existe la posibilidad de capturar una excepción y realizar un manejo parcial de la misma. En este caso parece lógico que se pueda relanzar la excepción nuevamente para que se propague al contexto de llamada donde se espera que se complete su manejo. De forma genérica:

```
try {
    ⋮
} catch (excepción identificador) {
    /* manejo parcial de la excepción */
    ⋮

    /* se relanza la excepción */
    throw identificador;
}
```

Además de relanzar una excepción, también es posible envolverla en una nueva excepción antes de relanzarla, haciendo que la primera se interprete como la causa de la segunda:

```
try {
    ⋮
} catch (excepción identificador) {
    /* manejo parcial de la excepción */
    ⋮

    /* se relanza la excepción */
    throw new excepción(identificador);
}
```