Manejo de Excepciones en Java

JAVA DESDE CERO

Manejo de Excepciones en Java

Intermedio (https://javadesdecero.es/categoria/intermedio/)

Dejar un comentario (https://javadesdecero.es/intermedio/manejo-de-excepciones/#respond) por Alex Walton Mar 9, 2019

Este artículo analiza el manejo de excepciones. **Una excepción es un error que ocurre en tiempo de ejecución**. Utilizando el subsistema de manejo de excepciones de Java, puede, de una manera estructurada y controlada, manejar los errores de tiempo de ejecución.

Aunque la mayoría de los lenguajes de programación modernos ofrecen algún tipo de manejo de excepciones, el soporte de Java es fácil de usar y flexible.

Table de Contenido

1. Manejo de Excepciones

Una ventaja principal del **manejo de excepciones** es que automatiza gran parte del código de manejo de errores que previamente debía ingresarse "a mano" en cualquier programa grande. Por ejemplo, en algunos lenguajes de computadora más antiguos, los códigos de error se devuelven cuando falla un método, y estos valores se deben verificar manualmente, cada vez que se llama al método. Este enfoque es tedioso y propenso a errores.

El manejo de excepciones agiliza el manejo de errores al permitir que tu programa defina un bloque de código, llamado ***manejador de excepción***, que se ejecuta automáticamente cuando ocurre un error. No es necesario verificar manualmente el éxito o el fracaso de cada operación específica o llamada a un método. Si se produce un error, será procesado por el manejador de excepciones.

Otra razón por la que el manejo de excepciones es importante es que Java define excepciones estándar para errores comunes del programa, como por ejemplo, dividir por cero o no encontrar el archivo. Para responder a estos errores, tu programa debe vigilar y manejar estas excepciones. Además, la biblioteca API de Java hace un uso extensivo de excepciones.

En el análisis final, ser un programador de Java exitoso significa que usted es completamente capaz de navegar por el subsistema de manejo de excepciones de Java. ¡Empezamos!

2. Jerarquía de excepciones

En Java, todas las excepciones están representadas por clases. Todas las clases de excepción se derivan de una clase llamada ***Throwable***. Por lo tanto, cuando se produce una excepción en un programa, se genera un objeto de algún tipo de clase de excepción.

Hay dos subclases directas de **Throwable**: **Exception** y **Error**:

1. Las excepciones de tipo ***Error*** están relacionadas con errores que ocurren en la Máquina Virtual de Java (https://javadesdecero.es/fundamentos/como-funciona-maquina-virtual/) y no en tu programa. Este tipo de excepciones escapan a su control y, por lo general, tu programa no se ocupará de ellas. Por lo tanto, este tipo de excepciones no se describen aquí.

2. Los errores que resultan de la actividad del programa están representados por subclases de ***Exception***. Por ejemplo, dividir por cero, límite de matriz y errores de archivo caen en esta categoría. En general, tu programa debe manejar excepciones de estos tipos. Una subclase importante de *Exception* es *RuntimeException*, que se usa para representar varios tipos comunes de errores en tiempo de ejecución.

3. Fundamentos de manejo de excepciones

El manejo de excepciones Java se gestiona a través de cinco palabras clave: **try**, **catch**, **throw**, **throws**, y **finally**. Forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro. A lo largo de este curso, cada palabra clave se examina en detalle. Sin embargo, es útil desde el principio tener una comprensión general del papel que cada uno desempeña en el manejo de excepciones. En resumen, así es como funcionan.

Las declaraciones de programa que desea supervisar para excepciones están contenidas dentro de un bloque **try**. Si se produce una excepción dentro del bloque try, se lanza. Tu código puede atrapar esta excepción usando **catch** y manejarlo de una manera racional. Las excepciones generadas por el sistema son lanzadas automáticamente por el sistema de tiempo de ejecución de Java. Para lanzar manualmente una excepción, use la palabra clave **throw**. En algunos casos, una excepción arrojada por un método debe ser especificada como tal por una cláusula **throws**. Cualquier código que debe ejecutarse al salir de un bloque **try** se coloca en un bloque **finally**.

3. Uso de try y catch

En el centro del manejo de excepciones están `try` y `catch`. Estas palabras clave trabajan juntas; no puedes ~~atrapar~~ (**catch**) sin ~~intentarlo~~ (**try**). Aquí está la forma general de los bloques de manejo de excepciones try/catch:

```
01.   try{
02.     //bloque de código para monitorear errores
03.    }
04.   catch (TipoExcepcion1 exOb){
05.     //Manejador para TipoExepción1
06.    }
07.   catch (TipoExcepcion2 exOb){
08.    //Manejador para TipoExepción2
09.    }
```

Aquí, *TipoExcepcion* es el tipo de excepción que ha ocurrido. Cuando se lanza una excepción, es atrapada por su instrucción **catch** correspondiente, que luego procesa la excepción. Como muestra la forma general, puede haber más de una declaración **catch** asociada con un **try**. El tipo de la excepción determina qué declaración de captura se ejecuta. Es decir, si el tipo de excepción especificado por una instrucción **catch** coincide con el de la excepción, entonces se ejecuta esa instrucción de **catch** (y todos los demás se anulan). Cuando se detecta una excepción, *exOb* recibirá su valor.

> Si no se lanza una excepción, entonces un bloque **try** finaliza normalmente, y todas sus declaraciones **catch** se pasan por alto. La ejecución se reanuda con la primera instrucción después del último **catch**. Por lo tanto, las declaraciones *catch* se ejecutan solo si se lanza una excepción.

3.1. Un ejemplo de excepción simple

Aquí hay un ejemplo simple que ilustra cómo observar y atrapar una excepción. Como saben, es un error intentar indexar una matriz más allá de sus límites. Cuando esto ocurre, la JVM lanza una **ArrayIndexOutOfBoundsException**. El siguiente programa genera a propósito tal excepción y luego la atrapa:

```
public class ExcDemo {
    public static void main(String[] args) {
        int nums[]=new int[4];

        try {
            System.out.println("Antes de que se genere la excepción.");
            //generar una excepción de índice fuera de límites
            nums[7]=10;
        }catch (ArrayIndexOutOfBoundsException exc){
            //Capturando la excepción
            System.out.println("Índice fuera de los límites!");
        }
        System.out.println("Después de que se genere la excepción.");
    }
}
```

Salida:

```
Antes de que se genere la excepción.
Índice fuera de los límites!
Después de que se genere la excepción.
```

Aunque es bastante breve, el programa anterior ilustra varios puntos clave sobre el manejo de excepciones:

- Primero, el código que desea monitorear para detectar errores está dentro de un bloque **try**.
- En segundo lugar, cuando se produce una excepción (en este caso, debido al intento de indexar *nums* más allá de sus límites), la excepción se emite desde el bloque **try** y es atrapada por la instrucción **catch**. En este punto, el control pasa al *catch*, y el bloque *try* finaliza.
- Es decir, no se llama a *catch*. Por el contrario, la ejecución del programa se transfiere a él. Por lo tanto, la instrucción que sigue a `nums[7]=10;` nunca se ejecutará.
- Después de que se ejecuta la instrucción **catch**, el control del programa continúa con las declaraciones que siguen el **catch**. Por lo tanto, es el trabajo de tu controlador de excepción remediar el problema que causó la excepción para que la ejecución del programa pueda continuar normalmente.

Recuerde, si no se lanza una excepción por un bloque **try**, no se ejecutarán declaraciones **catch** y el control del programa se reanudará después de la instrucción catch. Para confirmar esto, en el programa anterior, cambie la línea

```
nums[7] = 10;
```

por

```
nums[0] = 10;
```

Ahora, no se genera ninguna excepción, y el bloque *catch* no se ejecuta.

3.2. Un ejemplo de excepción con método

Es importante comprender que todo código dentro de un bloque **try** se supervisa para detectar excepciones. Esto incluye excepciones que pueden ser generadas por un método llamado desde dentro del bloque *try*.

Una excepción lanzada por un método llamado desde dentro de un bloque *try* puede ser atrapada por las declaraciones *catch* asociadas con ese bloque *try*, asumiendo, por supuesto, que el método no captó la excepción en sí misma. Por ejemplo, este es un programa válido:

```
01.  // Una excepción puede ser generada por un método
02.  // y atrapada por otro
03.  public class ExcEjemplo {
04.  //Generando una exepción
05.      static void genExcepcion(){
06.          int nums[]= new int[4];
07.
08.          System.out.println("Antes de que se genere la excepción.");
09.
10.          //generar una excepción de índice fuera de límites
11.          nums[7]=10;
12.          System.out.println("Esto no se mostrará.");
13.      }
14.  }
```

```
01. public class ExcDemo {
02.     public static void main(String[] args) {
03.         int nums[]=new int[4];
04.
05.         try {
06.            ExcEjemplo.genExcepcion();
07.         }catch (ArrayIndexOutOfBoundsException exc){
08.             //Capturando la excepción
09.             System.out.println("Índice fuera de los límites!");
10.         }
11.         System.out.println("Después de que se genere la excepción.");
12.     }
13. }
```

Salida:

```
Antes de que se genere la excepción.
Índice fuera de los límites!
Después de que se genere la excepción.
```

Como se llama a *genExcepcion()* desde un bloque *try*, la excepción que genera es capturada por *catch* en *main()*. Entender, sin embargo, que si *genExcepcion()* había atrapado la excepción en sí misma, nunca se hubiera pasado a *main()*.

3.3. Captura de excepciones de subclase

Hay un punto importante sobre declaraciones de múltiples **catch** que se relaciona con subclases. Una cláusula *catch* para una superclase también coincidirá con cualquiera de sus subclases.

Por ejemplo, dado que la superclase de todas las excepciones es *Throwable*, para atrapar todas las excepciones posibles, capture **Throwable**. Si desea capturar excepciones de un tipo de superclase y un tipo de subclase, coloque la subclase primero en la secuencia de **catch**. Si no lo hace, la captura de la superclase también atrapará todas las clases derivadas. Esta regla se autoejecuta porque poner primero la superclase hace que se cree un código inalcanzable, ya que la cláusula *catch* de la subclase nunca se puede ejecutar. En Java, el código inalcanzable es un error.

Por ejemplo, considere el siguiente programa:

```
01. // Las subclases deben preceder a las superclases
02. // en las declaraciones catch
03. public class ExcDemo {
04.     public static void main(String[] args) {
05.
06.         //Aquí, num es más grande que denom
07.         int nums[]={4,8,16,32,64,128,256,512};
08.         int denom[]={2,0,4,4,0,8};
09.
10.         for (int i=0;i< nums.length;i++){
11.             try {
12.                 System.out.println(nums[i]+" / "+
13.                                    denom[i]+" es "+nums[i]/denom[i]);;
14.             }catch (ArrayIndexOutOfBoundsException exc){
15.                 //Capturando la excepción (subclase)
16.                 System.out.println("No se encontró ningún elemento.");
17.             }
18.             catch (Throwable exc){
19.                 //Capturando la excepción (superclase)
20.             System.out.println("Alguna excepción ocurrió.");
```

```
            }
            }
        }
    }
```

Salida:

```
4 / 2 es 2
Alguna excepción ocurrió.
16 / 4 es 4
32 / 4 es 8
Alguna excepción ocurrió.
128 / 8 es 16
No se encontró ningún elemento.
No se encontró ningún elemento.
```

En este caso, *catch (Throwable)* detecta todas las excepciones excepto *ArrayIndexOutOfBoundsException*. El problema de detectar excepciones de subclase se vuelve más importante cuando crea excepciones propias.

4. Los bloques try pueden ser anidados

Un bloque ***try*** se puede anidar dentro de otro. Una excepción generada dentro del bloque *try* interno que no está atrapada por un ***catch*** asociado con este ***try***, se propaga al bloque *try* externo. Por ejemplo, aquí la ArrayIndexOutOfBoundsException no es capturada por el ***catch*** interno, sino por el ***catch*** externo:

```
// Uso de un bloque try anidado
public class TryAnidado{
    public static void main(String[] args) {
        //Aquí, num es más grande que denom
        int nums[]={4,8,16,32,64,128,256,512};
        int denom[]={2,0,4,4,0,8};

        try { //try externo
            for (int i = 0; i < nums.length; i++) {
                try { //try anidado
                    System.out.println(nums[i] + " / " +
                            denom[i] + " es " + nums[i] / denom[i]);
                } catch (ArithmeticException exc) {
                    //Capturando la excepción
                    System.out.println("No se puede dividir por cero!");
                }
            }
        }
        catch (ArrayIndexOutOfBoundsException exc) {
                    //Capturando la excepción
                    System.out.println("Alguna excepción ocurrió.");
                    System.out.println("ERROR: Programa terminado.");
                }
            }
        }
```

Salida:

```
4 / 2 es 2
No se puede dividir por cero!
16 / 4 es 4
32 / 4 es 8
No se puede dividir por cero!
128 / 8 es 16
Alguna excepción ocurrió.
ERROR: Programa terminado.
```

En este ejemplo, una excepción que puede ser manejada por el *try interno*, en este caso, un error de división por cero, permite que el programa continúe. Sin embargo, un error de límite de matriz es capturado por la *try* externo, lo que hace que el programa finalice.

Aunque ciertamente no es la única razón para las instrucciones *try* anidadas, el programa anterior hace un punto importante que se puede generalizar. A menudo, **los bloques try anidados se usan para permitir que las diferentes categorías de errores se manejen de diferentes maneras**. Algunos tipos de errores son catastróficos y no se pueden solucionar. Algunos son menores y pueden manejarse de inmediato.

> Puede utilizar un bloque *try* externo para detectar los errores más graves, permitiendo que los bloques *try* internos manejen los menos serios. ×

5. Lanzar una excepción

Los ejemplos anteriores han estado capturando excepciones generadas automáticamente por la JVM. Sin embargo, es posible lanzar manualmente una excepción utilizando la instrucción **throw**. Su forma general se muestra aquí:

```
throw excepcOb;
```

Aquí, *excepcOb* debe ser un objeto de una clase de excepción derivada de *Throwable*. Aquí hay un ejemplo que ilustra la instrucción `throw` arrojando manualmente una *ArithmeticException*:

```
01.  //Lanzar manualmente una excepción
02.  public class ThrowDemo {
03.      public static void main(String[] args) {
04.          try{
05.              System.out.println("Antes de lanzar excepción.");
06.              throw new ArithmeticException(); //Lanzar una excepción
07.          }catch (ArithmeticException exc){
08.              //Capturando la excepción
09.              System.out.println("Excepción capturada.");
10.          }
11.          System.out.println("Después del bloque try/catch");
12.      }
13.  }
```

Salida:

```
Antes de lanzar excepción.
Excepción capturada.
Después del bloque try/catch
```

Observe cómo se creó la *ArithmeticException* utilizando **new** en la instrucción **throw**. Recuerde, *throw* arroja un objeto. Por lo tanto, debe crear un objeto para "*lanzar*". Es decir, no puedes simplemente *lanzar* un tipo.

6. Re-lanzar una excepción:

Una excepción capturada por una declaración **catch** se puede volver a lanzar para que pueda ser capturada por un **catch** externo. La razón más probable para volver a lanzar de esta manera es permitir el acceso de múltiples manejadores/controladores a la excepción.

Por ejemplo, quizás un manejador de excepciones maneja un aspecto de una excepción, y un segundo manejador se enfrenta a otro aspecto. Recuerde, cuando vuelve a lanzar una excepción, no se volverá a capturar por la misma declaración *catch*. Se propagará a la siguiente declaración de *catch*. El siguiente programa ilustra el relanzamiento de una excepción:

```
01.  //Relanzando un excepción
02.  public class Rethrow {
03.      public static void genExcepcion() {
04.          //Aquí, num es más largo que denom
05.          int nums[] = {4, 8, 16, 32, 64, 128, 256, 512};
06.          int denom[] = {2, 0, 4, 4, 0, 8};
07.
08.          for (int i = 0; i < nums.length; i++) {
09.              try {
10.                  System.out.println(nums[i] + " / " +
11.                          denom[i] + " es " + nums[i] / denom[i]);
12.              } catch (ArithmeticException exc){
13.                  //Capturando la excepción
14.                  System.out.println("No se puede dividir por cero!.");
15.              }
16.              catch (ArrayIndexOutOfBoundsException exc) {
17.                  //Capturando la excepción
18.                  System.out.println("No se encontró ningún elemento.");
19.                  throw exc; //Relanzando la excepción
20.              }
21.          }
22.      }
23.  }
```

```
01.  public class RethrowDemo {
02.      public static void main(String[] args) {
03.          try{
04.              Rethrow.genExcepcion();
05.          }
06.          catch (ArrayIndexOutOfBoundsException exc){
07.              //Recapturando la excepción
08.              System.out.println("ERROR - Programa terminado");
09.          }
10.      }
11.  }
```

Salida:

```
4 / 2 es 2
No se puede dividir por cero!.
16 / 4 es 4
32 / 4 es 8
No se puede dividir por cero!.
128 / 8 es 16
No se encontró ningún elemento.
ERROR - Programa terminado
```

En este programa, los errores de división por cero se manejan localmente, mediante *genExcepcion()*, pero se vuelve a generar un error de límite de matriz. En este caso, es capturado por *main()*.

7. Una mirada más cercana a Throwable

Hasta este punto, hemos estado detectando excepciones, pero no hemos estado haciendo nada con el objeto de excepción en sí mismo. Como muestran todos los ejemplos anteriores, una cláusula **catch** especifica un tipo de excepción y un parámetro. El parámetro recibe el objeto de excepción. Como todas las excepciones son subclases de Throwable, todas las excepciones admiten los métodos definidos por Throwable. Varios de uso común se muestran en la siguiente tabla:

Tabla de métodos Throwable.

Método	**Sintaxis**	**Descripción**
getMessage	String getMessage()	Devuelve una descripción de la excepción.
getLocalizedMessage	String getLocalizedMessage()	Devuelve una descripción localizada de la excepción.
toString	String toString()	Devuelve un objeto String que contiene una descripción completa de la excepción. Este método lo llama println() cuando se imprime un objeto Throwable.
printStackTrace()	void printStackTrace()	Muestra el flujo de error estándar.
printStackTrace	void printStackTrace(PrintStreams)	Envía la traza de errores a la secuencia especificada.
printStackTrace	void printStackTrace(PrintWriters)	Envía la traza de errores a la secuencia especificada.
fillInStackTrace	Throwable fillInStackTrace()	Devuelve un objeto Throwable que contiene un seguimiento de pila completo. Este objeto se puede volver a lanzar.

De los métodos definidos por Throwable, dos de los más interesantes son *printStackTrace()* y *toString()*.

- Puede visualizar el mensaje de error estándar más un registro de las llamadas a métodos que conducen a la excepción llamando a **printStackTrace()**.
- Puede usar **toString()** para recuperar el mensaje de error estándar. El método *toString()* también se invoca cuando se usa una excepción como argumento para *println()*.

El siguiente programa demuestra estos métodos:

```
01.  public class ExcDemo {
02.     static void genExcepcion(){
03.         int nums[]=new int[4];
04.
05.         System.out.println("Antes de lanzar excepción.");
06.
07.         nums[7]=10;
08.         System.out.println("Esto no se mostrará.");
09.     }
10.  }
11.
12.  class MetodosThrowable{
13.      public static void main(String[] args) {
14.          try{
15.              ExcDemo.genExcepcion();
16.          }
17.          catch (ArrayIndexOutOfBoundsException exc){
18.              System.out.println("Mensaje estándar: ");
19.              System.out.println(exc);
20.              System.out.println("\nTraza de errores: ");
21.              exc.printStackTrace();
22.          }
23.          System.out.println("Después del bloque catch.");
24.      }
25.  }
```

Salida:

```
Antes de lanzar excepción.
Mensaje estándar:
java.lang.ArrayIndexOutOfBoundsException: 7

Traza de errores:
java.lang.ArrayIndexOutOfBoundsException: 7
at ExcDemo.genExcepcion(ExcDemo.java:8)
at MetodosThrowable.main(ExcDemo.java:16)
Después del bloque catch.
```

8. Uso de finally

Algunas veces querrá definir un bloque de código que se ejecutará cuando quede un bloque try/catch. Por ejemplo, una excepción puede causar un error que finaliza el método actual, causando su devolución prematura. Sin embargo, ese método puede haber abierto un archivo o una conexión de red que debe cerrarse.

Tales tipos de circunstancias son comunes en la programación, y Java proporciona una forma conveniente de manejarlos: **finally**. Para especificar un bloque de código a ejecutar cuando se sale de un bloque *try/catch*, incluya un bloque `finally` al final de una secuencia **try/catch**. Aquí se muestra la forma general de un try/catch que incluye **finally**.

```
try{
//bloque de código para monitorear errores
}
catch(TipoExcepcion1 exOb){
//manejador para TipoExcepcion1
}
catch(TipoExcepcion2 exOb){
//manejador para TipoExcepcion2
}
//...
finally{
//código final
}
```

El bloque *finally* se ejecutará siempre que la ejecución abandone un bloque **try/catch**, sin importar las condiciones que lo causen. Es decir, si el bloque *try* finaliza normalmente, o debido a una excepción, el último código ejecutado es el definido por *finally*. El bloque *finally* también se ejecuta si algún código dentro del bloque *try* o cualquiera de sus declaraciones *catch* devuelve del método.

Aquí hay un ejemplo de *finally*:

```
//Uso de finally
public class UsoFinally {
    public static void genExcepecion(int rec) {
        int t;
        int nums[]=new int[2];

        System.out.println("Recibiendo "+rec);
        try {
            switch (rec){
                case 0:
                    t=10 /rec;
                    break;
                case 1:
                    nums[4]=4; //Genera un error de indexación
                    break;
                case 2:
                    return; //Retorna desde el blorec try
            }
        }
        catch (ArithmeticException exc){
            //Capturando la excepción
            System.out.println("No se puede dividir por cero!");
            return; //retorna desde catch
        }
        catch (ArrayIndexOutOfBoundsException exc){
            //Capturando la excepción
            System.out.println("Elemento no encontrado");
        }
        finally {
            //esto se ejecuta al salir de los blorecs try/catch
            System.out.println("Saliendo de try.");
        }
    }
}
```

Salida:

```
class FinallyDemo{
    public static void main(String[] args) {
        for (int i=0;i<3;i++){
            UsoFinally.genExcepecion(i);
            System.out.println();
```

```
            }
        }
    }
```

Como muestra la salida, no importa cómo se salga el bloque *try*, el bloque **finally** sí se ejecuta .

9. Uso de throws

En algunos casos, si un método genera una excepción que no maneja, debe declarar esa excepción en una cláusula `throws` . Aquí está la forma general de un método que incluye una cláusula **throws**:

```
tipo-retorno nombreMetodo(lista-param) throws lista-excepc {
// Cuerpo
}
```

Aquí, *lista-excepc* es una lista de excepciones separadas por comas que el método podría arrojar fuera de sí mismo.

Quizás se pregunte por qué no necesitó especificar una cláusula **throws** para algunos de los ejemplos anteriores, que arrojó excepciones fuera de los métodos. La respuesta es que las excepciones que son subclases de **Error** o **RuntimeException** no necesitan ser especificadas en una lista de **throws**. Java simplemente asume que un método puede arrojar uno. Todos los otros tipos de excepciones deben ser declarados. De lo contrario, se produce un error en tiempo de compilación.

En realidad, usted vio un ejemplo de una cláusula **throws** anteriormente. Como recordará, al realizar la entrada del teclado, necesitaba agregar la cláusula a *main()*. Ahora puedes entender por qué. Una declaración de entrada podría generar una **IOException**, y en ese momento, no pudimos manejar esa excepción. Por lo tanto, tal excepción se descartaría *de main()* y necesitaría especificarse como tal. Ahora que conoce excepciones, puede manejar fácilmente IOException.

9.1. Ejemplo con throws

Veamos un ejemplo que maneja **IOException**. Crea un método llamado *prompt()*, que muestra un mensaje de aviso y luego lee un carácter del teclado. Como la entrada se está realizando, puede ocurrir una IOException.

Sin embargo, el método *prompt()* no maneja IOException. En cambio, usa una cláusula *throws*, lo que significa que el método de llamada debe manejarlo. En este ejemplo, el método de llamada es *main()* y trata el error.

```
//Uso de throws
class ThrowsDemo {
    public static char prompt(String args)
        throws java.io.IOException {
            System.out.println(args + " :");
            return (char) System.in.read();
   }

        public static void main (String[]args){
            char ch;
            try {
                //dado que prompt() podría arrojar una excepción,
                // una llamada debe incluirse dentro de un bloque try
                ch = prompt("Ingresar una letra");
            } catch (java.io.IOException exc) {
                System.out.println("Ocurrió una excepción de E/S");
                ch = 'X';
```

```
            }
             System.out.println("Usted presionó: " + ch);
        }
    }
```

En un punto relacionado, observe que IOException está totalmente calificado por su nombre de paquete java.io. Como aprenderá más adelante, el sistema de E/S de Java está contenido en el paquete **java.io**. Por lo tanto, **IOException** también está contenido allí. También habría sido posible importar java.io y luego referirme a IOException directamente.

10. Tres características adicionales de excepción

A partir de JDK 7, el mecanismo de **manejo de excepciones de Java** se ha ampliado con la adición de tres características.

1. El primero es compatible con la **gestión automática de recursos**, que automatiza el proceso de liberación de un recurso, como un archivo, cuando ya no es necesario. Se basa en una forma expandida de `try` , llamada declaración `try-with-resources` (try con recursos), y se describe más en Java Avanzado (https://javadesdecero.es/avanzado/), cuando se discuten los archivos.

2. La segunda característica nueva se llama **multi-catch**.

3. Y la tercera a veces se llama **final rethrow** o **more precise rethrow**. Estas dos características se describen aquí.

9.1. Característica multi-catch

El **multi-catch** permite capturar dos o más excepciones mediante la misma cláusula **catch.** Como aprendió anteriormente, es posible (de hecho, común) que un intento sea seguido por dos o más cláusulas *catch*. Aunque cada cláusula *catch* a menudo proporciona su propia secuencia de código única, no es raro tener situaciones en las que dos o más cláusulas *catch* ejecutan la misma secuencia de código aunque atrapen diferentes excepciones.

En lugar de tener que capturar cada tipo de excepción individualmente, puede usar una única cláusula de *catch* para manejar las excepciones sin duplicación de código.

Para crear un **multi-catch**, especifique una lista de excepciones dentro de una sola cláusula **catch**. Para ello, separe cada tipo de excepción en la lista con el operador **OR**. Cada parámetro **multi-catch** es implícitamente **final**. (Puede especificar explícitamente final, si lo desea, pero no es necesario.) Debido a que cada parámetro **multi-catch** es implícitamente **final**, no se le puede asignar un nuevo valor.

Aquí se explica cómo puede usar la función **multi-catch** para capturar **ArithmeticException** y **ArrayIndexOutOfBoundsException** con una única cláusula catch:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

Aquí hay un programa simple que demuestra el uso de **multi-catch**:

```
// Uso de multi-catch
// Este código requiere JDK7 o superior
class MultiCatch {
    public static void main(String[] args) {
```

```
05.             int a=28, b=0;
06.             int resultado;
07.             char chars[]={'A','B','C'};
08.
09.             for (int i=0; i<2;i++){
10.                 try {
11.                     if (i==0)
12.                         resultado=a/b; //genera un ArithmeticException
13.                     else
14.                         chars [5]='X'; //genera un ArrayIndexOutOfBoundsException
15.                 }catch (ArithmeticException | ArrayIndexOutOfBoundsException e){
16.                     System.out.println("Excepción capturada: "+e);
17.                 }
18.             }
19.             System.out.println("Después del multi-catch");
20.         }
21.     }
```

Salida:

```
Excepción capturada: java.lang.ArithmeticException: / by zero
Excepción capturada: java.lang.ArrayIndexOutOfBoundsException: 5
Después del multi-catch
```

El programa generará una **ArithmeticException** cuando se intenta la división por cero. Generará una **ArrayIndexOutOfBoundsException** cuando se intente acceder fuera de los límites de *chars*. Ambas excepciones son capturadas por la declaración única de **catch**.

La función más precisa de **rethrow** restringe el tipo de excepciones que pueden volver a lanzarse solo a aquellas excepciones marcadas que arroja el bloque **try** asociado, que no son manejadas por una cláusula **catch** anterior, y que son un subtipo o supertipo del parámetro. Si bien esta capacidad puede no ser necesaria a menudo, ahora está disponible para su uso.

Para que la característica **rethrow** esté en vigor, el parámetro **catch** debe ser efectivamente **final**. Esto significa que no se le debe asignar un nuevo valor dentro del bloque **catch**. También se puede especificar explícitamente como **final**, pero esto no es necesario.

Manejo de Excepciones	
Introducción al Manejo de Excepciones	★★★★★

Compartir

Sobre el Autor: Alex Walton (https://javadesdecero.es/author/qzal0/)

Hey hola! Yo soy Alex Walton y tengo el placer de compartir conocimientos hacía ti sobre el tema de Programación en Java, desde cero, Online y Gratis.