Rev 1.4 20170205

Programación II

Tema 3. Excepciones

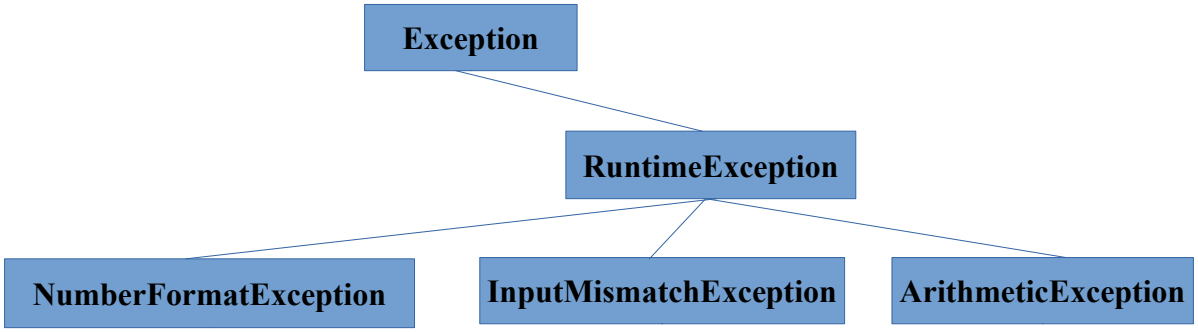
Contenido

1 Introducción

En este tema se tratan las excepciones y su gestión. Se utilizan para tratar los posibles errores que se pueden producir en un programa en Java. Mientras otros lenguajes de programación incorporan la gestión de excepciones como una forma más de manejar errores, en Java esta es la única forma que está disponible, por lo que es una parte fundamental del lenguaje de programación.

2 Excepciones básicas

Las excepciones son objetos. En el siguiente diagrama se indican cómo se relacionan las excepciones que se comentarán en este tema.



Las excepciones en la parte inferior se engloban[1] como **RuntimeException**'s, mientras que esta última se engloba dentro de **Exception**. Las excepciones que se engloban bajo **RuntimeException** se denominan excepciones *no controladas*, mientras el resto de excepciones son *controladas*. En el caso de las primeras, Java permitirá compilar un programa aún sabiendo que ciertas llamadas a métodos pueden provocar excepciones; en el caso de las segundas, si estas no se tratan, no lo hará.

En el siguiente ejemplo, se puede producir una excepción:

```
class ExcepcionNoControlada {
	public static int divide(int a, int b)
	{
		return a / b;
	}
}
```

Una vez compilada la clase, se podría intentar llamar al método *divide(a, b)* con los siguientes parámetros[2]:

1 La relación entre estas clases es de herencia, que se trata en un tema posterior. Por ahora basta entender que, a medida que se asciende en el diagrama, las clases son más generales y engloban a las inferiores.

2 Se asume que el lector está trabajando con *BueJ*, y una vez creada la clase, introduce las líneas marcadas con '>' en el *CodePad*.

```
> ExcepcionNoControlada.divide( 5, 0 );
Exception: java.lang.ArithmeticException (/ by zero)
```

Efectivamente, si en el segundo parámetro se pasa un cero, al realizar la división entre dos enteros se produce la excepción **ArithmeticException**, con el mensaje aclaratorio "*/ by zero"*. Las excepciones no solamente se producen en determinadas circunstancias, sino que también se pueden lanzar a propósito para señalar una condición de error excepcional. Por ejemplo, se podría lanzar una excepción para evitar que se produzca de forma automática la excepción **ArithmeticException**. Una nueva versión del ejemplo anterior podría ser como sigue.

```
class ExcepcionControlada {
      public static int divide(int a, int b)
      {
            if ( b == 0 ) {
                  throw new Exception( "divisor no puede ser cero" );
            }

            return a / b;
      }
}
```

Dado que las excepciones son objetos, para lanzarlas, es decir, para producirlas a propósito, es necesario utilizar el operador **new**. Sin embargo, la clase anterior no compila. Concretamente Java produce el error: *unreported exception* Exception*, must be caught or declared to be thrown*. Que significa que se está lanzando el objeto **Excepción**, sin haber especificado que el método *divide(a, b)* puede lanzarla. Esta es la diferencia entre excepciones controladas y no controladas. En el segundo caso, es necesario especificar que el método puede lanzarlas, o Java no permitirá la compilación de la clase. Para especificar esa posibilidad, debe añadírsele a la cabecera del método la aclaración[3] ***throws*** *Exception*, como sigue:

```
class ExcepcionControlada {
      public static int divide(int a, int b) throws Exception
      {
            if ( b == 0 ) {
                  throw new Exception( "divisor no puede ser cero" );
            }

            return a / b;
      }
}
```

3 Nótese que para lanzar una excepción se utiliza *throw*, mientras que el informe de posibles excepciones se hace con *throws*.

Ahora, en el caso de invocar al método *divide(a, b)*, se producirá la siguiente situación:

```
> ExcepcionNoControlada.divide( 5, 0 );
Exception: divisor no puede ser cero
```

Simplemente se ha cambiado una excepción (**ArithmeticException**) por otra (**Exception**), evitando que la primera se produzca. Sin embargo el efecto es el mismo: la detención del programa. El objetivo de una excepción es, o bien ser tratada (en el caso de que esa situación excepcional haya sido prevista), o bien parar el programa. Así, cuanto antes se detecte una situación de error, mejor, puesto que en caso de que se permita continuar el programa, puede llegar a enmascararse el método en el que en realidad se produjo el error, haciendo difícil resolver el problema.

3 Rebobinado del *Stack*

Cuando una excepción se produce, o se lanza explícitamente, recorre todo el *stack*, la pila de llamadas, hacia arriba, hasta llegar al sistema operativo y detener el programa. En el siguiente ejemplo:

```
class ExcepcionNoControlada {
    public static int divide(int a, int b)
    {
        return a / b;
    }

    public static void principal()
    {
        int x = divide( 4, 2 );
        int y = divide( 5, 0 );
        int z = divide( 8, 4 );
    }
}
```

Si se intenta ejecutar el método principal(), ocurre lo siguiente:

```
> ExcepcionNoControlada.principal();
Exception: ArithmeticException (/ by zero)
```

Es decir, la excepción se sigue comportando como anteriormente, a pesar de haber sido generada en la función *divide(a, b)*, que fue a su vez llamada por la función *principal()*. En concreto, la segunda sentencia de *principal()* provoca que se produzca la excepción, dentro de *divide(a, b)*. La excepción se pasa al método *principal()*, pues está justo antes en la pila de llamadas, y se detiene también la ejecución de este método (es decir, la tercera llamada en *principal()* nunca se ejecuta). Finalmente, se detiene el programa completo (no hay más entradas en el *stack*) y se informa de la excepción.

A esto se le llama *stack unwinding*, es decir, el rebobinado de la pila de llamadas. La excepción se produce en un método y sube por la pila de llamadas hasta que es tratada o la pila se acaba, en cuyo caso se detiene el programa. La siguiente sección habla sobre cómo tratar las excepciones.

4 Tratamiento de excepciones

Las excepciones pueden capturarse y tratarse, evitando que sigan propagándose por el *stack*. Para ello, se utilizan los bloques *try... catch*. Dado que la excepción va subiendo por el *stack*, no es necesario tratarla exactamente donde se produce, sino que debe tratarse en el método donde se está preparado para tratarla adecuadamente.

```
class ExcepcionNoControlada {
    private int x;
    private int y;
    private int z;

    public static int divide(int a, int b)
    {
        return a / b;
    }

    public ExcepcionNoControlada()
    {
        try {
            x = divide( 4, 2 );
            y = divide( 5, 0 );
            z = divide( 8, 4 );
        } catch(ArithmeticException exc)
        {
             x = y = z = -1;
        }
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getZ() {
        return z;
    }
}
```

Al intentar crear un objeto de la clase **ExcepcionNoControlada**, y producirse la excepción **ArithmeticException**, los atributos *x*, *y*, y *z* adoptan el valor -1.

```
> ExcepcionNoControlada c = new ExcepcionNoControlada();
> c.getX()
(int) -1
> c.getY()
(int) -1
> c.getZ()
(int) -1
```

Dado que el rebobinado del *stack* no ha agotado la pila de llamadas, pues había un bloque *try... catch* rodeando la llamada conflictiva a *divide()*, el usuario de la clase **ExcepcionNoControlada** no es consciente de que se haya producido una excepción: en todo caso, lo deducirá de los valores de los atributos *x*, *y*, y *z*.

En un bloque *try... catch* no es necesario que exista un solo *catch*. Pueden existir varios *catch* para varios tipos de excepciones. Para el siguiente ejemplo se utilizará el método estático *parseInt()* de la clase **Integer**, que permite convertir una cadena con contenido numérico entero, en un número entero. En caso de que no se pueda convertir, *parseInt()* lanza la excepción **NumberFormatException**. En el siguiente ejemplo, se capturan por tanto dos excepciones que se pueden producir.

```
class Excepciones {
    private int x;
    private int y;
    private int z;

    public static int divide(String a, String b)
    {
        int op1 = Integer.parseInt( a );
        int op2 = Integer.parseInt( b );

        return op1 / op2;
    }

    public Excepciones()
    {
        try {
            x = divide( "4", "f" );
            y = divide( "5", "0" );
            z = divide( "8", "4" );
        }
        catch(ArithmeticException exc)
        {
             x = y = z = -1;
        }
        catch(NumberFormatException exc)
        {
             x = y = z = -2;
        }
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getZ() {
        return z;
    }
}
```

En el método *principal()*, la primera sentencia dentro del *try... catch* produce una excepción **NumberFormatException**, que se captura en el segundo *catch* y hace que los valores de los atributos *x*, *y*, y *z,* pasen a tener el valor -2, como se puede ver a continuación.

```
Excepciones c = new Excepciones();
c.getX()
(int) -2
c.getY()
(int) -2
c.getZ()
(int) -2
```

A la hora de ordenar los *catch()*, es necesario tener en cuenta que deben siempre ponerse primero aquellos más específicos, y después los más generales. Java no permitirá compilar si no se respeta esta norma. Por ejemplo, supongamos que se desea capturar también la excepción **Exception**. Al ser la excepción más general, se capturará cualquier error que no se haya previsto.

No es buena idea confiar en **Exception** para capturar todos los errores como norma general, ya que no permite distinguir entre tipos de errores.

```
class Excepciones {
      public Excepciones()
      {
        try {
            x = divide( "4", "f" );
            y = divide( "5", "0" );
            z = divide( "8", "4" );
        } catch(ArithmeticException exc) {
             x = y = z = -1;
        } catch(NumberFormatException exc) {
             x = y = z = -2;
        } catch(Exception exc) {
            System.exit( -1 );      // Salir al sistema operativo: PANIC!
        }
      }
      //...
}
```

4.1 Recuperarse tras una excepción

Una vez que la ejecución ha llegado al código dentro de una cláusula *catch*, no es posible hacer nada para que la ejecución "vuelva atrás", y corregir aquello que hizo que la excepción se produjera. En concreto, por ejemplo, no es posible volver atrás e indicar que en la llamada *y = divide("5", "0");*, se debe sustituir el argumento "0" por otro valor. Entre otras consideraciones, sería incluso complicado saber por qué valor se podría sustituir el parámetro en un escenario más general.

Sin embargo, de esas tres llamadas que se producen en el ejemplo anterior, parece exagerado que por el hecho de que una de ellas falle, la ejecución, al fin y al cabo, se aborte completamente. Otra aproximación es la siguiente: ya que los bloques *try… catch()* se pueden anidar, y que la ejecución tras el *catch()* continúa en la siguiente línea a este, es posible realizar otras operaciones ajustando el lugar en el que se sitúa la captura de excepciones.

```
public class Divisiones {
    private int[] valores;
    private String resultados;

    public static int divide(String a, String b)
    {
        int op1 = Integer.parseInt( a );
        int op2 = Integer.parseInt( b );

        return op1 / op2;
    }

    public String getInformeResultados() {
      return this.resultados;
    }

    public Divisiones()
    {
      String[][] args = {
            { "4", "f" },
            { "5", "0" },
            { "8", "4" }
      };

      this.resultados = "";
      this.valores = new int[ args.length ];
      try {
            for(int i = 0; i < args.length; ++i) {
                  valores[ i ] = Integer.MIN_VALUE;
                  try {
                        this.resultados += '\n' + args[i][0]
                                          + " / " + args[i][1] + " = ";
                        this.valores[i] = divide( args[i][0], args[i][1] );
                        this.resultados += valores[i];

                  }
                  catch(ArithmeticException exc)
                  {
                    resultados += "ERROR calculando: " + exc.getMessage();
                  }
                  catch(NumberFormatException exc)
                  {
                    resultados += "ERROR de formato: " + exc.getMessage();
                  }
            }
      } catch(Exception exc) {
            resultados = "ERROR inesperado: " + exc.getMessage();
      }
    }

    public int[] getValores() {
        return this.valores;
    }
}
```

```
public class DivisionesApp {
    public static void main(String[] args) {
      Divisiones calculos = new Divisiones();

      System.out.println( calculos.getInformeResultados() );
    }
}
```

En el ejemplo anterior el bloque *try... catch* se ha movido dentro del bucle, de forma que, aunque se produzca una excepción, la ejecución continúa dentro del bucle hacia la siguiente iteración, y por lo tanto, el siguiente cálculo. La salida del ejemplo es la siguiente:

```
4 / f = ERROR de formato: f
5 / 0 = ERROR calculando: / by zero
8 / 4 = 2
```

De esta manera, todos los cálculos se realizan, independientemente de que alguno de ellos (o incluso todos) produzcan una excepción.

5 Limpieza de recursos

La característica final del sistema de gestión de excepciones de java es la cláusula *finally* de los bloques *try... catch*. Cuando existe, se ejecutará siempre, haya habido una excepción o no. El objetivo de las instrucciones dentro de la cláusula *finally* es siempre la de limpieza de recursos: asegurarse de que, se produzca una excepción o no, se van a cerrar todos los recursos que se hayan visto involucrados.

En el ejemplo siguiente, se aprovecha el uso de *finally* para asegurar que siempre se incluirá un salto de línea tras cada línea del informe de resultados. Si bien es un ejemplo que no tiene que ver con la limpieza de recursos (el fin último de la cláusula *finally*), ilustra perfectamente cómo funciona.

Como forma de recordar su objetivo, *finally* significa “finalmente”, es decir, se trata de instrucciones que se ejecutarán al final del método, bien por que se haya producido una excepción, o bien porque se haya llegado a la última instrucción sin problemas.

```
public Divisiones {
    // más cosas...

    public Calculos()
    {
      String[][] args = {
            { "4", "f" },
            { "5", "0" },
            { "8", "4" }
      };

      this.resultados = "";
      this.valores = new int[ args.length ];
      try {
            for(int i = 0; i < args.length; ++i) {
                  valores[ i ] = Integer.MIN_VALUE;

                  try {
                        this.resultados += args[i][0] + " / " + args[i][1] + " = ";
                        this.valores[i] = divide( args[i][0], args[i][1] );
                        this.resultados += valores[i];

                  }
                  catch(ArithmeticException exc)
                  {
                    resultados += "ERROR calculando: " + exc.getMessage();
                  }
                  catch(NumberFormatException exc)
                  {
                    resultados += "ERROR de formato: " + exc.getMessage();
                  }
                  finally {
                    resultados += '\n';
                  }
            }
      } catch(Exception exc) {
            resultados = "ERROR inesperado: " + exc.getMessage();
      }
    }

    // Más cosas...
}
```