PROGRAMACIÓN UD-10

FICHEROS DE TEXTO

ÍNDICE

[INTRODUCCIÓN 3](#_Toc128935807)

[EXCEPCIONES 3](#_Toc128935808)

[A. CONTROL DE ERRORES 3](#_Toc128935809)

[B. EXCEPCIONES DE USUARIO 5](#_Toc128935810)

[FICHEROS DE TEXTO 6](#_Toc128935811)

[A. FLUJOS 6](#_Toc128935812)

[B. BUFFERS 6](#_Toc128935813)

[C. SCANNER 7](#_Toc128935814)

[D. FLUJOS DE SALIDA 7](#_Toc128935815)

[FICHEROS XML Y JAXB 8](#_Toc128935816)

# INTRODUCCIÓN

Para comunicarnos con los dispositivos hardware de entrada y salida, en java usamos flujos. El uso de los flujos es independiente del dispositivo con el que nos comuniquemos.

Los flujos pueden ser de entrada o salida y, a su vez, pueden ser de texto (orientados a texto) o binarios (orientados a Byte) que pueden tratar cualquier tipo de datos.

# EXCEPCIONES

## CONTROL DE ERRORES

Diagrama

Descripción generada automáticamenteLas excepciones se van a encargar de capturar los errores que interrumpen el flujo normal de funcionamiento de un programa. Cada vez que se produce un error, se genera un objeto de la clase Throwable, que contiene información del error. Este objeto es el que podemos capturar para controlar los errores. Si no lo capturamos, la máquina virtual mostrará unas líneas de error por la salida estándar y muy probablemente parará la ejecución del programa.

Dentro de la clase Throwable tenemos una serie de subclases:

* **Error**: Suelen ser errores graves que no son recuperables.
* **RuntimeException**: Se ocupa de errores de código (tener algo del código mal escrito) por ejemplo una división entre 0, tipo de dato incorrecto…
* **Exception**: Son los errores que vamos a tratar. En función del tipo de objeto que usemos, ya sabremos el tipo de excepciones que pueden lanzar. Para capturarlos se usa try, catch y finally.

Si sabemos que un trozo de código es susceptible de lanzar una excepción, generaremos un bloque de código precedido de la palabra try. Las excepciones que lance este código serán capturadas por un nuevo/s bloque/s colocado/s tras el try, que llevará la palabra catch con la excepción que se quiere capturar entre paréntesis.

Se pueden poner tantos catch como se desee para capturar diferentes tipos de errores. Si se entra en un catch el resto se ignoran, por lo que es importante el orden el que se coloquen. Se podían poner varias excepciones separadas con |, para capturar en un catch varios tipos de excepciones.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

En base a los errores detectados con las pruebas, puedo depurar el código añadiendo las excepciones indicadas.

Texto, Carta

Descripción generada automáticamente

Si se hubiera capturado antes Exception, Arithmetic Eception nuca se ejecutaría.



Los tipos de excepciones también heredan. Hay que tener cuidado de no poner un bloque catch con una excepción que se superclase de otra que vaya más abajo, o esta última nunca se ejecutara. Por tanto, si capturamos Exception tendrá que ir en el último catch.

Texto

Descripción generada automáticamente

Cuando se produce una excepción en un método, podemos enviarla al método que te habrá hecho la llamada.

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteSi se genera un bloque try, se puede poner un bloque finally al final de los catch. El finally siempre se ejecutará, independientemente de si se produce o no una excepción. Aunque tenga un return, se ejecutará primero el bloque finally antes de salir. Puede ser interesante por ejemplo, para cerrar flujos con archivos.

Hay un grupo de excepciones, llamadas comprobadas que son predecibles, y el compilador nos obliga a tratarlas, bien enviando el control a quien haya hecho la llamada con un throws o bien con su captura mediante un try-cath. Si no tratamos dichas excepciones el compilador detectará un error (IOException, FileNotFoundException). Existen otras no comprobadas que se producen por errores de código y por tanto habrá que corregir el código (ArithmeticException, ArrayIndexBoundException…).

## EXCEPCIONES DE USUARIO

En Java podemos generar excepciones propias, coma y lanzar excepciones cuando nos interese.

Para generar un nuevo tipo de sección creamos una clase nueva que hereda de Exception:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

# FICHEROS DE TEXTO

## FLUJOS

Las clases que vamos a usar para trabajar con ficheros se encuentran en el paquete Java.io, y las clases que terminan en Reader las usaremos para leer y las que terminan en Writer para escribir.

El trabajo con ficheros nos puede dar excepciones del tipo IOException.

=ara abrir un fichero en modo lectura lo haremos con la sentencia FileReacer fr = new FileReacer(“C:\\ubicación\\fichero”); Si estamos ejecutando en una máquina de Linux, las barras irían en sentido contrario sin duplicar FileReader fr = new FileReader (“/home/pepe/fichero.txt”); Normalmente el duplicado no es necesario nunca.

Para leer un carácter usaremos el método read(), por ejemplo, chrCaracter = fr.read(); devuelve -1 si ha terminado el fichero.

Los flujos siempre se deben cerrar con el método close(), fr.close().

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

## BUFFERS

Podemos hacer lecturas de líneas completas asociando un buffer a nuestra entrada mediante la clase BufferedReader. BufferedReader br = new BufferedReader (new FileReader(“fichero.txt”));

Al tener un flujo podemos usar el método readLine(), que devolverá un string por cada línea del fichero br.readLine(); al llegar a final devuelve null.

Aplicación

Descripción generada automáticamente con confianza media

## SCANNER

La clase Scanner, como hemos visto, nos permite leer del flujo estándar (System.in) y analizar datos para devolver un tipo primitivo. se puede usar esta clase con cualquier otro tipo de flujos, como por ejemplo una variable String.

Texto

Descripción generada automáticamente

## FLUJOS DE SALIDA

Igual que teníamos la clase FileReader que hereda de InpusStreamReader para generar flujos de entrada, tenemos FileWriter que hereda de OutputSreamWriter para generar flujos de salida.

Para generar un nuevo flujo pondremos:

* FileWriter fichEscribir = new FileWriter(“fichero”,true); 🡪 El parámetro true es opcional, si no se indica se elimina el fichero si ya existe antes de empezar a escribir y si se pone, se escribe al final en caso de existir.
* BufferedWriter bufEscribir = new BufferedWriter(new FileWriter(“fichero”));

Los métodos de BufferedWriter son:

* write(int|String) 🡪 Escribe un carácter o una cadena en el buffer.
* newLine() 🡪 Escribe un salto de línea. Es mejor que usar ‘\n’, ya que varía según la plataforma.
* flush() 🡪 Vacía el flujo de salida, escribiendo lo que haya en el archivo.
* close() 🡪 Cierra el flujo.

al trabajar con la salida nos pueden dar excepciones del tipo IOException.

Hay que tener cuidado con el cierre de los flujos, que normalmente colocaremos en el bloque finally. Desde Java 7 se pueden cerrar recursos directamente con una estructura try-catch-resources. Tras la llave del try se podrían abrir más flujos que se cerrarían a su vez.

Imagen que contiene Texto

Descripción generada automáticamente

# FICHEROS XML Y JAXB

Como la transmisión de información por medio de ficheros XML es cada vez más frecuente, se ha creado la API JAXB (Java Architecture for XML Binding), Para enlazar ar vamos achivos XML con objetos Java.

Vamos a trasladar a objetos el siguiente ejemplo, que se suele definir como unmarshalling.

Texto

Descripción generada automáticamente con confianza media

lo normal sería basarnos en el esquema, pero en este caso lo vamos a hacer con el propio documento XML. vamos a necesitar una clase Socio, con los atributos id, nombre, dirección y alta. antes de cada clase, atributo o propiedad hay que añadir una anotación del tipo javax.xml.bind.annotation que empezarán por @Xml.

Texto

Descripción generada automáticamente