10.A. Introducción al almacenamiento de datos y flujos.

Tabla de contenidos

1 Introducción.

Cuando desarrollas programas, en la mayoría de ellos los usuarios pueden pedirle a la aplicación que realice cosas y pueda suministrarle datos con los que se quiere hacer algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos, para proporcionar una respuesta a lo solicitado.

Además, normalmente interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros, que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

Por tanto, sabemos que el almacenamiento en variables o vectores (arrays) es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o cuando el programa termina. **Las computadoras utilizan ficheros para guardar los datos**, incluso después de que el programa termine su ejecución. Se suele denominar a los datos que se guardan en ficheros **datos persistentes**, porque persisten más allá de la ejecución de la aplicación. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos cómo hacer con Java estas operaciones de crear, actualizar y procesar ficheros.

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como **Entrada/Salida (E/S).**

Distinguimos dos tipos de E/S: la **E/S estándar** que se realiza con el terminal del usuario y la **E/S a través de fichero**s, en la que se trabaja con ficheros de disco.

Todas las operaciones de E/S en Java vienen proporcionadas por el paquete estándar del API de Java denominado `java.io` que incorpora interfaces, clases y excepciones para acceder a todo tipo de ficheros.

El contenido de un archivo puede interpretarse como **campos** y **registros** (grupos de campos), dándole un significado al conjunto de bits que en realidad posee.

Para saber más

A continuación puedes ampliar tus conocimientos sobre Entrada y Salida en general, en el mundo de la informática. Verás que es un basto tema lo que abarca.

Entrada y Salida.

Además, se ha desarrollado posteriormente las APIs NIO y NIO.2 que proporcionan un sistema de E/S intensivo y con código más compacto.
Referencia a la API de java.nio

En el siguiente vídeo podemos ver ejemplo de uso de lectura de ficheros con NIO.2:
Video de ejemplo de lectura de ficheros con NIO.2

2 Excepciones.

Cuando se trabaja con archivos, es normal que pueda haber errores, por ejemplo: podríamos intentar leer un archivo que no existe, o podríamos intentar escribir en un archivo para el que no tenemos permisos de escritura. Para manejar todos estos errores debemos utilizar excepciones. Las dos excepciones más comunes al manejar archivos son:

- `FileNotFoundException`: Si no se puede encontrar el archivo.
- `IOException`: Si no se tienen permisos de lectura o escritura o si el archivo está dañado.

Un esquema básico de uso de la captura y tratamiento de excepciones en un programa, podría ser este, importando el paquete `java.io.IOException`:

```
public static void main(String args[]) {
 try {
     // Se ejecuta algo que puede producir una excepción
     } catch (FileNotFoundException e) {
     // manejo de una excepción por no encontrar un archivo

     } catch (IOException e) {
     // manejo de una excepción de entrada/salida

     } catch (Exception e) {
     // manejo de una excepción cualquiera

     } finally {
     // código a ejecutar haya o no excepción
     }

}
```

Aquí tienes el mismo código copiable:

```
import java.io.IOException;
```

```

```

```
// ...
```

```

```

```
public static void main(String[] args) {
```

```
    try {
```

```
  // Se ejecuta algo que puede producir una excepción
```

```
    } catch (FileNotFoundException e) {
```

```
  // manejo de una excepción por no encontrar un archivo
```

```
    } catch (IOException e) {
```

```
  // manejo de una excepción de entrada/salida
```

```
    } catch (Exception e) {
```

```
  // manejo de una excepción cualquiera
```

```
    } finally {
```

```
  // código a ejecutar haya o no excepción
```

```
    }
```

```
}
```

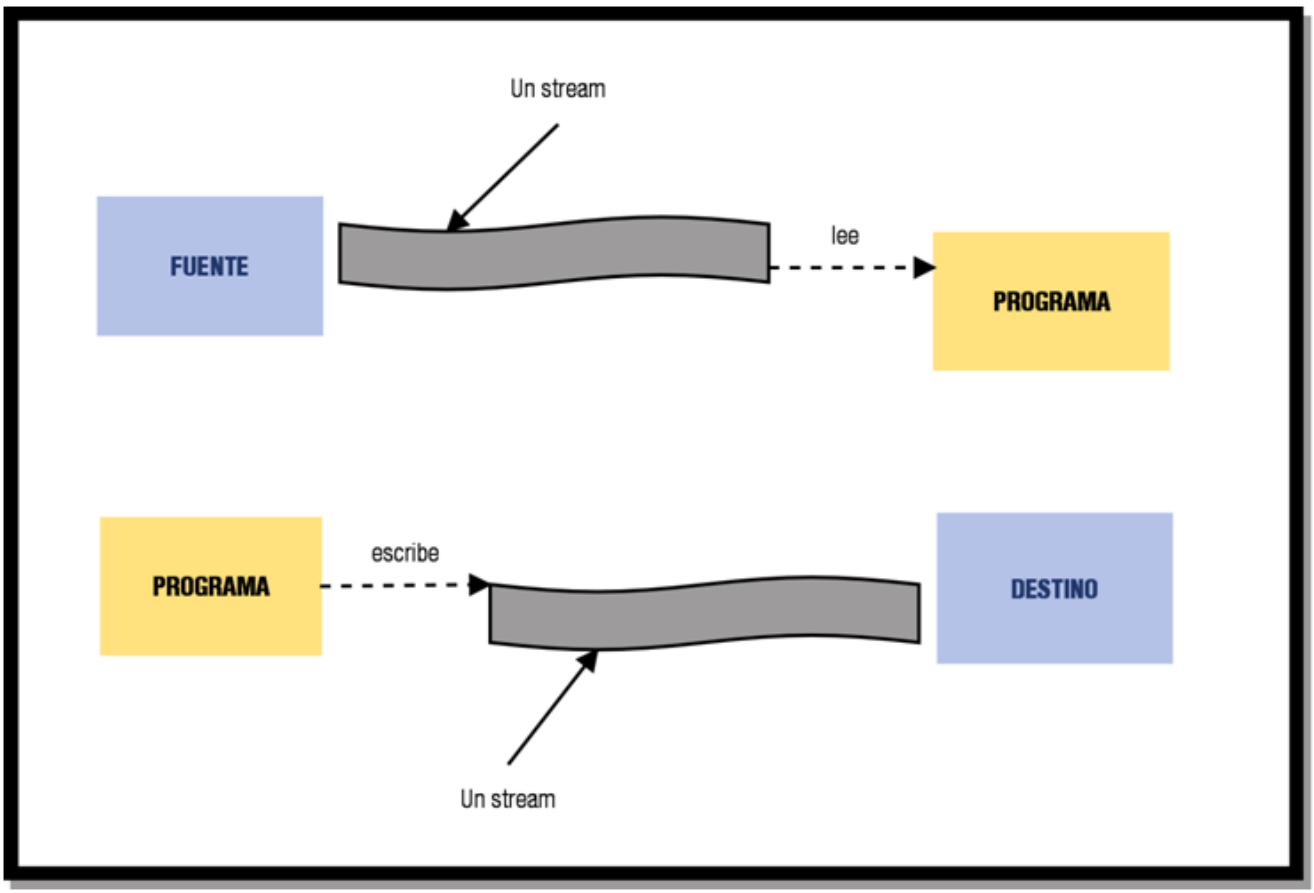
Autoevaluación

Señala la opción correcta:

- Java no ofrece soporte para excepciones.
- Un campo y un archivo es lo mismo.
- Si se intenta abrir un archivo (para lectura) que no existe, entonces saltará una excepción.
- Ninguna es correcta.

3 Concepto de flujo.

La clase `Stream` representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes que pueden ir desder un archivo (guardado en un dispositivo de entrada/salida) (en adelante E/S) a la memoria, a un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc.



Cualquier programa realizado en Java que necesite llevar a cabo una operación de entrada salida lo hará a través de un **stream**.

Un flujo es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

Las clases y métodos de E/S que necesitamos emplear son las mismas **independientemente del dispositivo con el que estemos actuando**, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red liberando al programador de tener que saber con quién está interactuando.

La vinculación de un flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

En resumen, será el flujo el que tenga que comunicarse con el sistema operativo concreto y "entendérselas" con él. De esta manera**, no tenemos que cambiar absolutamente nada en nuestra aplicación**, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es primordial en un lenguaje multiplataforma y tan altamente portable como Java**.**

Autoevaluación

Señala la opción correcta:

- ○ La clase Stream puede representar, al instanciarse, a un archivo.
- ○ Si programamos en Java, hay que tener en cuenta el sistema operativo cuando tratemos con flujos, pues varía su tratamiento debido a la diferencia de plataformas.
- ○ La clase keyboard es la clase a utilizar al leer flujos de teclado.
- ○ La vinculación de un flujo al dispositivo físico la hace el hardware de la máquina.

4 Clases relativas a flujos.

Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (character streams).

- Los **flujos de caracteres** (16 bits) se usan para manipular datos legibles por humanos (por ejemplo un fichero de texto). Vienen determinados por dos clases abstractas: `Reader` y `Writer`. Dichas clases manejan flujos de caracteres Unicode. De ellas derivan subclases concretas que implementan los métodos definidos destacados los métodos `read()` y `write()` que, en este caso, leen y escriben **caracteres** de datos respectivamente.

- Los **flujos de bytes** (8 bits) se usan para manipular datos binarios, legibles solo por la maquina (por ejemplo un fichero de programa). Su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son `InputStream` y `OutputStream`. Estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan `read()` y `write()` que leen y escriben bytes de datos respectivamente.

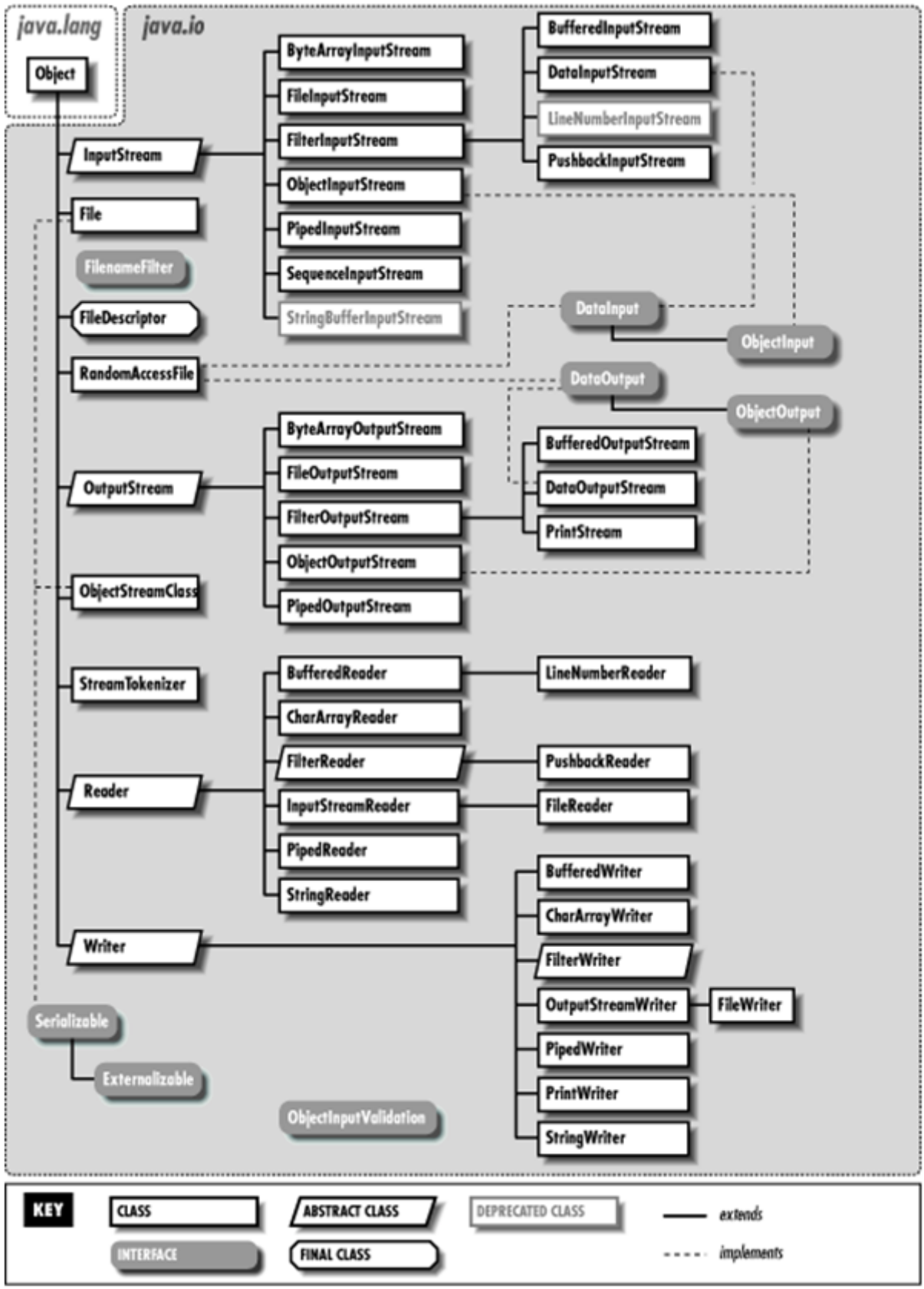


Imagen extraída de curso Programación del MECD.

Las clases del paquete `java.io` se pueden ver en la ilustración. Destacamos las clases relativas a flujos:

- `BufferedInputStream`: permite leer datos a través de un flujo con un buffer intermedio.

- `BufferedOutputStream`: implementa los métodos para escribir en un flujo a través de un buffer.

- `FileInputStream`: permite leer bytes de un fichero.

- `FileOutputStream`: permite escribir bytes en un fichero o descriptor.

- `StreamTokenizer`: esta clase recibe un flujo de entrada, lo analiza (parse) y divide en diversos pedazos (tokens), permitiendo leer uno en cada momento.

- `StringReader`: es un flujo de caracteres cuya fuente es una cadena de caracteres o string.

- `StringWriter`: es un flujo de caracteres cuya salida es un buffer de cadena de caracteres, que puede utilizarse para construir un string.

Destacar que hay clases que se **"montan" sobre otros flujos para modificar la forma de trabajar con ellos.** Por ejemplo, con `BufferedInputStream` podemos añadir un buffer a un flujo `FileInputStream`, de manera que se mejore la eficiencia de los accesos a los dispositivos en los que se almacena el fichero con el que conecta el flujo.

4.1 Ejemplo comentado de una clase con flujos.

Vamos a ver un ejemplo con una de las clases comentadas, en concreto, con `StreamTokenizer`.

La clase `StreamTokenizer` obtiene un flujo de entrada y lo divide en "tokens". El flujo tokenizer puede reconocer identificadores, números y otras cadenas.

El ejemplo que puedes ver a continuación, muestra cómo utilizar la clase `StreamTokenizer` para contar números y palabras de un fichero de texto. Se abre el flujo con ayuda de la clase `FileReader`, y puedes ver cómo se "monta" el flujo `StreamTokenizer` sobre el `FileReader`, es decir, que se construye el objeto **StreamTokenizer** con el flujo FileReader como argumento, y entonces se empieza a iterar sobre él.

```

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package tokenizer;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;


public class token {

```

```
    public void contarPalabrasyNumeros(String nombre_fichero) {

        StreamTokenizer sTokenizer = null;
        int contapal = 0, numNumeros = 0;

        try {

            sTokenizer = new StreamTokenizer(new FileReader(nombre_fichero));

            while (sTokenizer.nextToken() != StreamTokenizer.TT_EOF) {

                if (sTokenizer.ttype == StreamTokenizer.TT_WORD)
                    contapal++;
                else if (sTokenizer.ttype == StreamTokenizer.TT_NUMBER)
                    numNumeros++;
            }

            System.out.println("Número de palabras en el fichero: " + contapal);
            System.out.println("Número de números en el fichero: " + numNumeros);

        } catch (FileNotFoundException ex) {
```

```
            System.out.println(ex.getMessage()) ;
        } catch (IOException ex) {
            System.out.println(ex.getMessage()) ;
        }
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        new Main().countWordsAndNumbers("c:\\datos.txt");
    }
}
```

El método `nextToken` devuelve un int que indica el tipo de token leído. Hay una serie de constantes definidas para determinar el tipo de token:

- `TT_WORD` indica que el token es una palabra.
- `TT_NUMBER` indica que el token es un número.
- `TT_EOL` indica que se ha leído el fin de línea.
- `TT_EOF` indica que se ha llegado al fin del flujo de entrada.

En el código de la clase, apreciamos que se iterará hasta llegar al fin del fichero. Para cada token, se mira su tipo, y según el tipo se incrementa el contador de palabras o de números.

Autoevaluación

Indica si la siguiente afirmación es verdadera o falsa.

Según el sistema operativo que utilicemos, habrá que utilizar un flujo u otro. ¿Verdadero o Falso?

- Verdadero.
- Falso.