



9. Almacenando datos

Autor	Xerach Casanova
Clase	Programación
Fecha	@Mar 29, 2021 3:26 PM

1. [Introducción](#)
 - 1.1. [Excepciones](#)
2. [Concepto de flujo](#)
3. [Clases relativas a flujos](#)
 - 3.1. [Ejemplo comentado de una clase con flujos](#)
4. [Flujos](#)
 - 4.1. [Flujos predefinidos. Entrada y salida estándar.](#)
 - 4.2. [Flujos predefinidos. Entrada y salida estándar. Ejemplo.](#)
 - 4.3. [Flujos basados en bytes](#)
 - 4.4. [Flujos basados en caracteres](#)
 - 4.5. [Rutas de los ficheros](#)
5. [Trabajando con ficheros](#)
 - 5.1. [Escritura y lectura de información de ficheros.](#)
 - 5.2. [Ficheros binarios y ficheros de texto \(I\)](#)
 - 5.2.1. [Ficheros binarios y ficheros de texto \(II\)](#)
 - 5.3. [Modos de acceso. Registros](#)
 - 5.4. [Acceso secuencial](#)
 - 5.5. [Acceso aleatorio](#)
6. [Aplicaciones del almacenamiento de información en ficheros](#)
7. [Utilización de los sistemas de ficheros](#)
 - 7.1. [Clase File](#)
 - 7.2. [Interface FilenameFilter](#)
 - 7.3. [Creación y eliminación de ficheros y directorios](#)
8. [Almacenamiento de objetos en ficheros. Persistencia. Serialización](#)
 - 8.1. [Serialización: utilidad](#)
9. [Mapa conceptual](#)

1. Introducción

El almacenamiento en variables o vectores es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o si el programa termina. Los

ordenadores utilizan ficheros para guardar los datos. Solemos llamar a los datos que se guardan en ficheros "datos persistentes".

A las operaciones que constituyen un flujo de información del programa con el exterior se les conoce como Entrada/salida (E/S) y se distinguen dos tipos: E/S estándar que se realiza con el terminal del usuario y E/S a través de ficheros, que trabaja con ficheros de disco.

Todas estas operaciones vienen proporcionadas por la API de java en el paquete `java.io` e incorpora interfaces, clases y excepciones.

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

1.1. Excepciones

Trabajar con archivos puede generar errores y para manejarlos debemos usar excepciones. Las dos más comunes son:

- **FileNotFoundException** cuando no encuentra el archivo.
- **IOException** si no se tienen permisos de lectura o escritura, o si el archivo está dañado.

El esquema básico de uso de la captura y tratamiento de excepciones de un programa podría ser este

```
public static void main(String args[]){
    try{
        //Se ejecuta algo que permite producir una excepción.

    } catch (FileNotFoundException e) {
        //manejo de una excepción cuando no encuentra un archivo.

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

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

2. Concepto de flujo

La clase `Stream` representa un flujo de corriente de datos o un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de E/S, memoria, conector TCP/IP, etc...

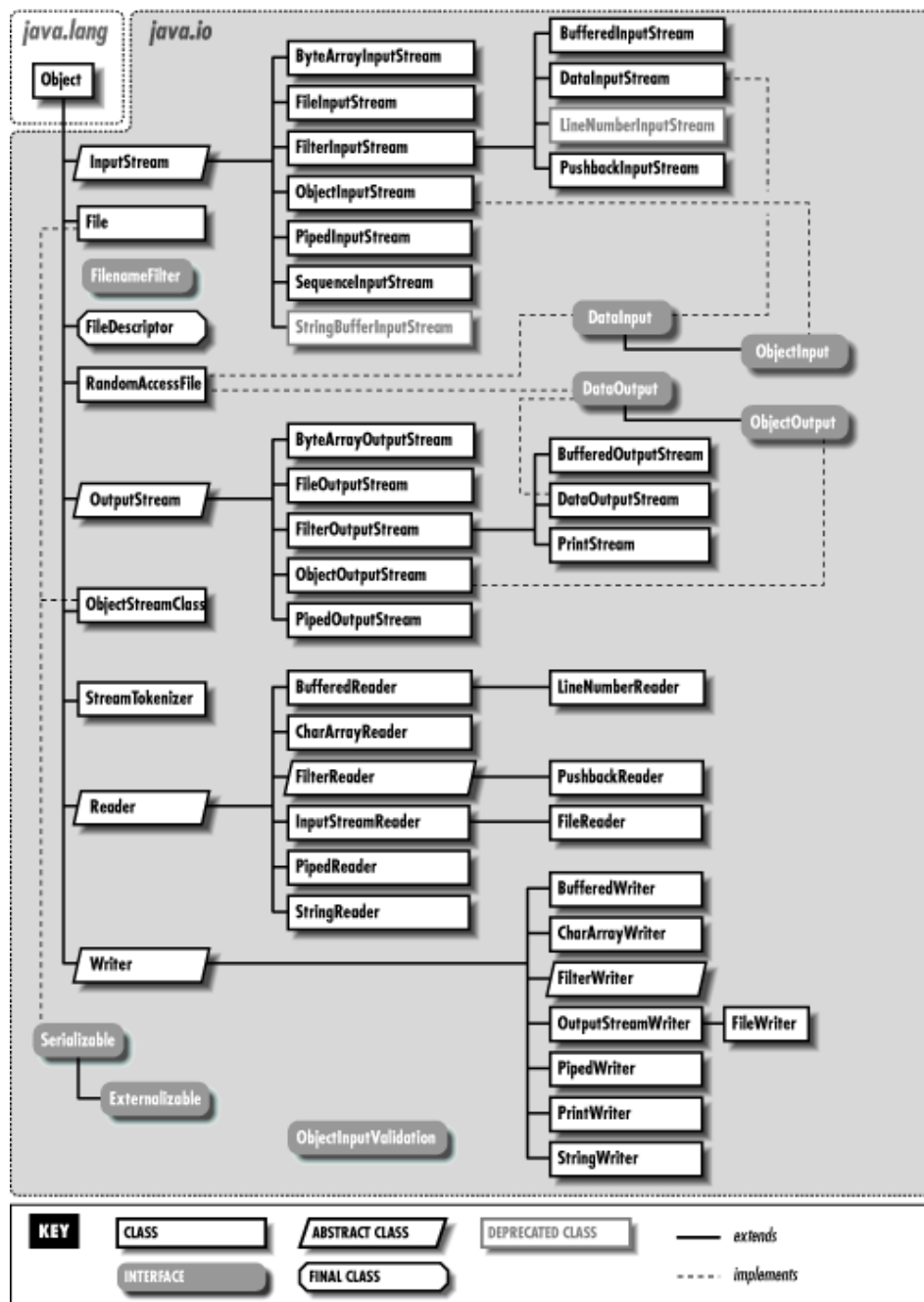
Cualquier programa realizado en Java que necesita llevar operaciones E/S lo hace a través de stream. Un flujo o stream es una abstracción de aquello que produzca o consuma información.

Las clases y métodos de E/S que necesitamos son las mismas en todos los dispositivos que usemos, el núcleo de Java es el encargado de determinar si trata con el teclado, el monitor, un sistema de archivos o un socket de red, de esta manera no tenemos que cambiar nada en nuestra aplicación, ya que ésta será independiente de los dispositivos físicos de almacenamiento y del Sistema Operativo sobre el que se ejecuta.

3. Clases relativas a flujos

Existen dos tipos de flujos:

- **Flujos de caracteres (16 bits)** se usan para manipular datos legibles por humanos, como por ejemplo un fichero de texto. Usan dos clases abstractas: Reader y Writer, las cuales manejan flujos de caracteres Unicode. De ellas derivan subclases que implementan métodos definidos destacando read() y write().
- **Flujos de bytes (8 bits)**, se usan para manipular datos binarios legibles por la máquina, como el fichero de un programa. El tratamiento del flujo de bytes viene dado por las dos clases abstractas InputStream y OutputStream, las cuales definen los métodos que implementan sus subclases entre las que destacan read() y write().



Entre las clases del paquete `java.io` destacamos:

- **BufferedInputStream:** permite leer datos a través de un flujo con un buffer intermedio.
- **BufferedOutputStream:** implementa 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:** recibe un flujo de entrada, lo analiza (parse) y divide en pedazos (tokens), permitiendo leer uno en cada momento.
- **StringReader:** es un flujo de caracteres cuya fuente es un String.
- **StringWriter:** es un flujo de caracteres cuya salida es un buffer de cadena de caracteres que se puede utilizar para construir un String.

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`, mejorando la eficiencia de acceso a dispositivos donde se almacena el fichero.

3.1. Ejemplo comentado de una clase con flujos

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 q siguiente , 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.

En este código, se aprecia que se iterará hasta llegar al fin del archivo, para cada token, se mira su tipo y según su tipo se incrementa el contador de palabras o números.

```
package token;

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

public class Token {

    public static 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) {
    contarPalabrasyNumeros("c:\\datos.txt");
}
}

```

El método `nextToken` devuelve un `int` que indica el tipo de token leído. Hay una serie de constantes que determinan 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.

4. Flujos

4.1. Flujos predefinidos. Entrada y salida estándar.

Tradicionalmente Unix, Linux y MS-DOS han utilizado un tipo de E/S estándar. El fichero de entrada estándar (`stdin`) es típicamente el teclado y el de salida (`stdout`) es típicamente la pantalla o la ventana del terminal. El fichero de salida de error estándar (`stderr`) también se dirige normalmente a la pantalla, pero se implementa como otro fichero para distinguirlo de la salida normal.

Java tiene acceso a ellos a través de la clase `System`. Los tres ficheros que se implementan son:

- **Stdin.** Es un objeto de tipo `InputStream` y está definido en la clase `System` como flujo de entrada estándar. Por defecto es el teclado, pero se puede redirigir para cada host o usuario.

- **Stdout. System.out.** Implementa stdout como una instancia de la clase `PrintStream`. Se pueden utilizar `print()` y `println()` con cualquier tipo básico Java como argumento.
- **Stderr.** Es un objeto de tipo `PrintStream`. Es un flujo de salida definido por la clase `System` y representa la salida de error estándar, pero es posible redirigirlo a otro dispositivo de salida.

Para la entrada, se usa el método `read` para leer de la entrada estándar:

- **`int System.in.read();`**
 - Lee el siguiente byte (`char`) de la entrada estándar.
- **`int System.in.read(byte[] b);`**
 - Lee un conjunto de bytes de la entrada estándar y lo almacena en un vector.

Para leer valores numéricos se toma el valor de la entrada estándar en forma de cadena y usamos métodos para transformar el texto a números.

- **`byte Byte.parseByte(String);`** Convierte una cadena en un número entero de un byte.
- **`short Short.parseShort(String);`** Convierte una cadena en un número entero corto.
- **`int Integer.parseInt(String);`** Convierte una cadena en un número entero.
- **`long Long.parseLong(String);`** Convierte una cadena en un número entero largo.
- **`float Float.parseFloat(String);`** Convierte una cadena en un número real simple.
- **`double Double.parseDouble(String);`** Convierte una cadena en un número real doble.
- **`boolean Boolean.parseBoolean(String);`** Convierte una cadena en un valor lógico.

4.2. Flujos predefinidos. Entrada y salida estándar. Ejemplo.

En este ejemplo, se lee por teclado hasta pulsar la tecla retorno, en ese momento el programa acaba imprimiendo por la salida estándar la cadena leída.

Para construir la cadena con caracteres leídos usamos la clase `StringBuffer`, que permite almacenar cadenas que cambiarán la ejecución del programa, o `StringBuilder`, similar pero no es síncrona. Para la mayoría de aplicaciones donde

solo se ejecuta un hilo, supone una mejora de rendimiento sobre StringBuffer. Todo el proceso debe estar en un bloque try catch

```
import java.io.IOException;

public class leeEstandar {
    public static void main(String[] args) {
        // Cadena donde iremos almacenando los caracteres que se escriban
        StringBuilder str = new StringBuilder();
        char c;
        // Por si ocurre una excepción ponemos el bloque try-cath
        try{
            // Mientras la entrada de teclado no sea Intro
            while ((c=(char)System.in.read())!='\n'){
                // Añadir el character leído a la cadena str
                str.append(c);
            }
        }catch(IOException ex){
            System.out.println(ex.getMessage());
        }

        // Escribir la cadena que se ha ido tecleando
        System.out.println("Cadena introducida: " + str);
    }
}
```

4.3. Flujos basados en bytes

Este tipo de flujos es idóneo para el manejo de entradas y salidas de bytes, su uso está orientado a lectura y escritura de datos binarios.

Para el tratamiento de flujos de bytes se usan las clases abstractas InputStream y OutputStream. Cada una de las cuales tiene subclases que controlan las diferencias entre distintos dispositivos E/S que se pueden usar.

```
class FileInputStream extends InputStream {
    FileInputStream (String fichero) throws FileNotFoundException;
    FileInputStream (File fichero) throws FileNotFoundException;
    ... ..
}
class FileOutputStream extends OutputStream {
    FileOutputStream (String fichero) throws FileNotFoundException;
    FileOutputStream (File fichero) throws FileNotFoundException;
    ... ..
}
```

OutputStream e InputStream y todas sus subclases reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de entrada o salida.

Por ejemplo, podemos copiar el contenido de un fichero en otro:

```
void copia (String origen, String destino) throws IOException {
    try{
        // Obtener los nombres de los ficheros de origen y destino
        // y abrir la conexión a los ficheros.
        InputStream fentrada = new FileInputStream(origen);
        OutputStream fsalida = new FileOutputStream(destino);
        // Crear una variable para leer el flujo de bytes del origen
        byte[] buffer= new byte[256];
        while (true) {
            // Leer el flujo de bytes
            int n = fentrada.read(buffer);
            // Si no queda nada por leer, salir del bucle
            if (n < 0)
                break;
            // Escribir el flujo de bytes leídos al fichero destino
            fsalida.write(buffer, 0, n);
        }
        // Cerrar los ficheros
        fentrada.close();
        fsalida.close();
    }catch(IOException ex){
        System.out.println(ex.getMessage()); }
}
```

4.4. Flujos basados en caracteres

Las clases orientadas al flujo de bytes ayudan a realizar cualquier tipo de operación de entrada y salida, pero no trabajan directamente con caracteres Unicode, los cuales están representados por dos bytes y por eso usamos las clases orientadas al flujo de caracteres, para lo cual Java tiene dos clases abstractas: Reader y Writer.

Reader, Writer y todas sus subclases reciben en el constructor el objeto que representa el flujo de datos para el dispositivo de E/S. El flujo de datos se debe cerrar cuando no se necesita.

Existen muchos tipos de flujos dependiendo de la utilidad de los datos que extraemos de los dispositivos.

Un flujo se puede envolver por otro para tratarlo de manera más cómodo. Así, un `bufferWriter` permite manipular flujos de datos como un buffer, pero si lo envolvemos en un `PrintWriter` podemos escribir con más funcionalidades para este tipo de datos.

```

try{
    PrintWriter out = null;
    out = new PrintWriter(new FileWriter("c:\\salida.txt", true));

    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    while (!(s = br.readLine()).equals("salir")){
        out.println(s);
    }
    out.close();
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}

```

4.5. Rutas de los ficheros

Cuando se opera con rutas de ficheros, el carácter separador entre directorios cambia dependiendo del S.O.

Para evitar problemas, se recomienda usar `File.separator` y se puede hacer una función que al pasarle la ruta nos devuelva la adecuada según el S.O. actual.

```

static String substFileSeparator(String ruta){
    String separador = "\\";

    try{
        // Si estamos en Windows
        if ( File.separator.equals(separador) )
            separador = "/" ;
        // Reemplaza todas las cadenas que coinciden con la expresión
        // regular dada oldSep por la cadena File.separator
        return ruta.replaceAll(separador, File.separator);
    }catch(Exception e){
        // Por si ocurre una java.util.regex.PatternSyntaxException
        return ruta.replaceAll(separador + separador, File.separator);
    }
}

```

5. Trabajando con ficheros

Se debe tener en cuenta que la manera de proceder con los ficheros es:

- Abrir o crear el fichero.
- Hacer operaciones.

- Cerrar el fichero.

Se debe tener control de las excepciones para evitar fallos en tiempo de ejecución.

5.1. Escritura y lectura de información de ficheros.

Debemos tener en cuenta las clases que vamos a emplear y determinar si es más eficiente emplear clases de Reader Writer o las de InputStream y OutputStream. También debemos considerar el acceso al fichero. Ya que podemos acceder de manera secuencial y recorrer el fichero entero hasta llegar a la posición, o de manera aleatoria para posicionarnos en una posición del fichero.

La idea del uso de las clases que usan buffer es que cuando una aplicación necesita leer datos de un fichero, debe esperar a que el disco en el que está el fichero le proporcione la información, consiguiendo reducir el número de accesos al fichero y mejorando la eficiencia de la aplicación, ya que un dispositivo de memoria masiva es más lento que la CPU del ordenador.

Para ello asociamos al fichero una memoria intermedia o buffer y cuando se necesita leer el byte del archivo, se trae hasta el buffer asociado al flujo.

Cualquier operación E/S puede generar una IOException y deben estar dentro de un bloque try/catch.

5.2. Ficheros binarios y ficheros de texto (I)

Los ficheros se utilizan para guardar información un soporte y distinguimos entre ficheros de texto y binarios. Los ficheros de texto están codificados en Unicode o en ASCII.

Podemos averiguar la codificación de un fichero con el método `getEncoding()`

```
FileInputStream fichero;
try {
    // Elegimos fichero para leer flujos de bytes "crudos"
    fichero = new FileInputStream("c:\\texto.txt");
    // InputStreamReader sirve de puente de flujos de byte a caracteres
    InputStreamReader unReader = new InputStreamReader(fichero);
    // Vemos la codificación actual
    System.out.println(unReader.getEncoding());
} catch (FileNotFoundException ex) {
}
}
```

Los archivos de texto se pueden abrir usando la clase `FileReader`, que nos proporciona métodos para leer caracteres y, cuando nos interese leer líneas

completas podemos usar la clase `BufferedReader` a partir de `FileReader`.

```
File arch = new File ("C:\\fich.txt");
FileReader fr = new FileReader (arch);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Para escribir en archivos de texto lo hacemos teniendo en cuenta

```
FileWriter fich = null;
PrintWriter pw = null;
fich = new FileWriter("/fich2.txt");pw = new PrintWriter(fichero);
pw.println("Linea de texto");
```

Si el fichero que queremos escribir existe y solo queremos añadir información pasamos el segundo parámetro como `true`.

```
FileWriter("/fich2.txt",true)
```

5.2.1. Ficheros binarios y ficheros de texto (II)

Los ficheros binarios almacenan la información en bytes codificada en binario, pueden contener fotografías, números, letras, archivos ejecutables, etc.

Los archivos binarios guardan una representación de los datos en el fichero en código UTF-8.

Para leer un fichero binario se usa la clase `FileInputStream`, que trabaja con bytes y se leen desde el flujo asociado a un fichero.

Ejemplo de lectura de fichero de bytes:

```
package leerconbuffer;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

/**
 *
 * @author
 */
public class LeerConBuffer {

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

```

try{
    // Creamos un nuevo objeto File, que es la ruta hasta el fichero desde
    File f = new File("C:\\apuntes\\test.bin");

    // Construimos un flujo de tipo FileInputStream (un flujo de entrada desde
    // fichero) sobre el objeto File. Estamos conectando nuestra aplicaci3n
    // a un extremo del flujo, por donde van a salir los datos, y "pidiendo"
    // al Sistema Operativo que conecte el otro extremo al fichero que indica
    // la ruta establecida por el objeto File f que hab3amos creado antes. De
    FileInputStream flujoEntrada = new FileInputStream(f);
    BufferedInputStream fEntradaConBuffer = new BufferedInputStream(flujoEntrada);

    // Escribimos el tama3o del fichero en bytes.
    tama = fEntradaConBuffer.available();
    System.out.println("Bytes disponibles: " + tama);

    // Indicamos que vamos a intentar leer 50 bytes del fichero.
    System.out.println("Leyendo 50 bytes...");

    // Creamos un array de 50 bytes para llenarlo con los 50 bytes
    // que leamos del flujo (realmente del fichero)*/
    byte bytearray[] = new byte[50];

    // El m3todo read() de la clase FileInputStream recibe como par3metro un
    // array de byte, y lo llena leyendo bytes desde el flujo.
    // Devuelve un n3mero entero, que es el n3mero de bytes que realmente se
    // han le3do desde el flujo. Si el fichero tiene menos de 50 bytes, no
    // podr3 leer los 50 bytes, y escribir3 un mensaje indic3ndolo.
    if (fEntradaConBuffer.read(bytearray) != 50)
        System.out.println("No se pudieron leer 50 bytes");

    // Usamos un constructor adecuado de la clase String, que crea un nuevo
    // String a partir de los bytes le3dos desde el flujo, que se almacenaron
    // en el array bytearray, y escribimos ese String.
    System.out.println(new String(bytearray, 0, 50));

    // Finalmente cerramos el flujo. Es importante cerrar los flujos
    // para liberar ese recurso. Al cerrar el flujo, se comprueba que no
    // haya quedado ning3n dato en el flujo sin que se haya le3do por la aplicaci3n. */
    fEntradaConBuffer.close();

    // Capturamos la excepci3n de Entrada/Salida. El error que puede
    // producirse en este caso es que el fichero no est3 accesible, y
    // es el mensaje que enviamos en tal caso.
} catch (IOException e){
    System.err.println("No se encuentra el fichero");
}
}
}

```

Para escribir datos a un fichero binario la clase `FileOutputStream` nos permite usar el fichero para escritura de bytes en 3l, La filosof3a es la misma pero en direcci3n contraria, desde la aplicaci3n que hace de fuente de datos, hasta el fichero que los consumen.

En la siguiente presentación puedes ver un esquema de cómo utilizar buffer para optimizar la lectura de teclado desde consola, por medio de las envolturas, podemos usar métodos como `readline()`, de la clase `BufferedReader`, que envuelve a un objeto de la clase `InputStreamReader`.



5.3. Modos de acceso. Registros

En java no se impone una estructura en un fichero y no existe el concepto de registro en los ficheros creados con Java.

Los programadores deben definir su registro con el número de bytes que le interesen, moviéndose por el fichero teniendo en cuenta ese tamaño definido.

Un fichero es de acceso directo u organización directo cuando para acceder a un registro n cualquiera, no se tiene que pasar por los $n-1$ registros anteriores.

Podemos trabajar con ficheros secuenciales y con ficheros de acceso aleatorio. En los secuenciales, para recuperar la información se hace en el mismo orden en que se introdujo, por lo que para acceder a un registro determinado se deben leer los anteriores.

Si se trata de un fichero de acceso aleatorio se puede acceder directamente.

5.4. Acceso secuencial

En el siguiente ejemplo vemos cómo se escriben datos en un fichero secuencial: el nombre y apellidos de una persona utilizando el método `writeUTF()` que proporciona `DataOutputStream`, seguido de su edad que la escribimos con el método `writeInt()` de la misma clase. A continuación escribimos lo mismo para una segunda persona y de nuevo para una tercera. Después cerramos el fichero. Y ahora lo abrimos de nuevo para ir

leyendo de manera secuencial los datos almacenados en el fichero, y escribiéndolos a consola.

```
package escylee;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 *
 * @author JJBH
 */
public class EscyLee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declarar un objeto de tipo archivo
        DataOutputStream archivo = null ;
        DataInputStream fich = null ;
        String nombre = null ;
        int edad = 0 ;
        try {
            // Creando o abriendo para añadir el archivo
            archivo = new DataOutputStream( new FileOutputStream("c:\\secuencial.dat",true) );

            // Escribir el nombre y los apellidos
            archivo.writeUTF( "Antonio López Pérez " );
            archivo.writeInt(33) ;
            archivo.writeUTF( "Pedro Piqueras Peñaranda" );
            archivo.writeInt(45) ;
            archivo.writeUTF( "José Antonio Ruiz Pérez " );
            archivo.writeInt(51) ;
            // Cerrar fichero
            archivo.close();

            // Abrir para leer
            fich = new DataInputStream( new FileInputStream("c:\\secuencial.dat") );
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            nombre = fich.readUTF() ;
            System.out.println(nombre) ;
            edad = fich.readInt() ;
            System.out.println(edad) ;
            fich.close();

        } catch(FileNotFoundException fnfe) { /* Archivo no encontrado */ }
```

```

        catch (IOException ioe) { /* Error al escribir */ }
        catch (Exception e) { /* Error de otro tipo*/
            System.out.println(e.getMessage());}

    }

}

```

Al ver el código escribimos las cadenas de caracteres del mismo tamaño para saber luego el tamaño de registro que hay que leer.

Para buscar información en un fichero secuencial se debe abrir el fichero e ir leyendo registros hasta encontrar el que buscamos, Si se quiere eliminar un registro de un fichero secuencial no se puede quitar el registro y reordenar. Una manera de hacerlo sería crear un nuevo fichero copiando cada uno de los registros excepto el que queremos borrar.

5.5. Acceso aleatorio

Podemos acceder al fichero como si fuera una base de datos, donde se accede a un registro concreto del fichero, con la clase `RandomAccessFile`, que permite utilizar un fichero de acceso aleatorio en el que el programador define el formato de los registros.

```
RandomAccessFile objFile = new RandomAccessFile( ruta, modo );
```

modo es la dirección física en el sistema de archivos y puede ser r para sólo lectura y rw para lectura y escritura.

```
RandomAccessFile in = new RandomAccessFile("input.dat", "r");
RandomAccessFile in = new RandomAccessFile("input.dat", "rw");
```

Esta clase implementa las interfaces `DataInput` y `DataOutput`.

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases distintas. Hay que especificar el modo de acceso al construir un objeto. Dispone de métodos específicos como `seek` y `skipBytes` para moverse de un registro a otro o posicionarse en una posición concreta.

En este enlace hay más información sobre el acceso aleatorio:

<http://puntocomnoesunlenguaje.blogspot.com/2013/06/java-ficheros-acceso-aleatorio.html>

6. Aplicaciones del almacenamiento de información en ficheros

La diversidad de ficheros que existe según la información que se guarda es muy extensa: fotos, música, ficheros de S.O., código fuente de lenguajes de programación, ficheros de películas, etc.

7. Utilización de los sistemas de ficheros

7.1. Clase File

La clase File proporciona una representación abstracta de ficheros y directorios y permite examinar y manipular, archivos y directorios de cualquier plataforma.

Las instancias de la clase File representan nombres de archivo y no los archivos en sí mismo.

El archivo correspondiente a un nombre dado puede no existir y hay que controlar las posibles excepciones.

Las rutas pueden ser:

- Relativas al directorio actual
- Absolutas si la ruta que pasamos empieza por:
 - Barra en Unix, Linux (/).
 - Letra de unidad en Windows (C:, D:)
 - UNC (Universal naming convention) en Windows como por ejemplo.

```
File miFile=new File("\\\\mimaquina\\download\\prueba.txt");
```

Dado un objeto file, podemos hacer las siguientes operaciones con él:

- Renombrar con el método `renameTo()`.-
- Borrar con el método `delete()` o `deleteOnExit()` (se borra cuando finaliza la ejecución de la máquina virtual).
- Crear un fichero con nombre único con el método estático `createTempFile`, crea un nuevo fichero temporal y devuelve un objeto File que apunta a él.
- Establecer fecha y hora de modificación de archivo con `setLastModified()`, por ejemplo `new File("prueba.txt").setLastModified(new Date().getTime());`

- Crear un directorio con el método mkdir() y mkdirs() si los directorios superiores no existen.
- Listar contenido de directorio con list() y listfiles().
- Listar los nombres de archivo de la raíz del sistema de archivos mediante listRoots()

7.2. Interface FilenameFilter

La interface FilenameFilter se usa para crear filtros que establezcan criterios de filtrado relativo al nombre de los ficheros y deben definir e implementar el método accept.

```
boolean accept(File dir, String nombre)
```

```
public class Filtro implements FilenameFilter {
    String extension;
    Filtro(String extension){
        this.extension=extension;
    }
    @Override
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }

    public static void main(String[] args) {
        try {
            File fichero=new File("c:\\datos\\.");
            String[] listadeArchivos;
            listadeArchivos = fichero.list(new Filtro(".odt"));
            int numarchivos = listadeArchivos.length ;

            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
            else
            {
                for (String listadeArchivo : listadeArchivos) {
                    System.out.println(listadeArchivo);
                }
            }
        } catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```

Launchpad

7.3. Creación y eliminación de ficheros y directorios

Se puede crear un fichero del siguiente modo:

- Se crea el objeto que encapsula el fichero

```
File fichero = new File("c:\\prueba\\miFichero.txt");
```

- Con el objeto File creamos el fichero físicamente, con la siguiente instrucción:

```
fichero.createNewFile()
```

Para borrar un fichero:

podemos usar la clase File, comprobando que existe previamente:

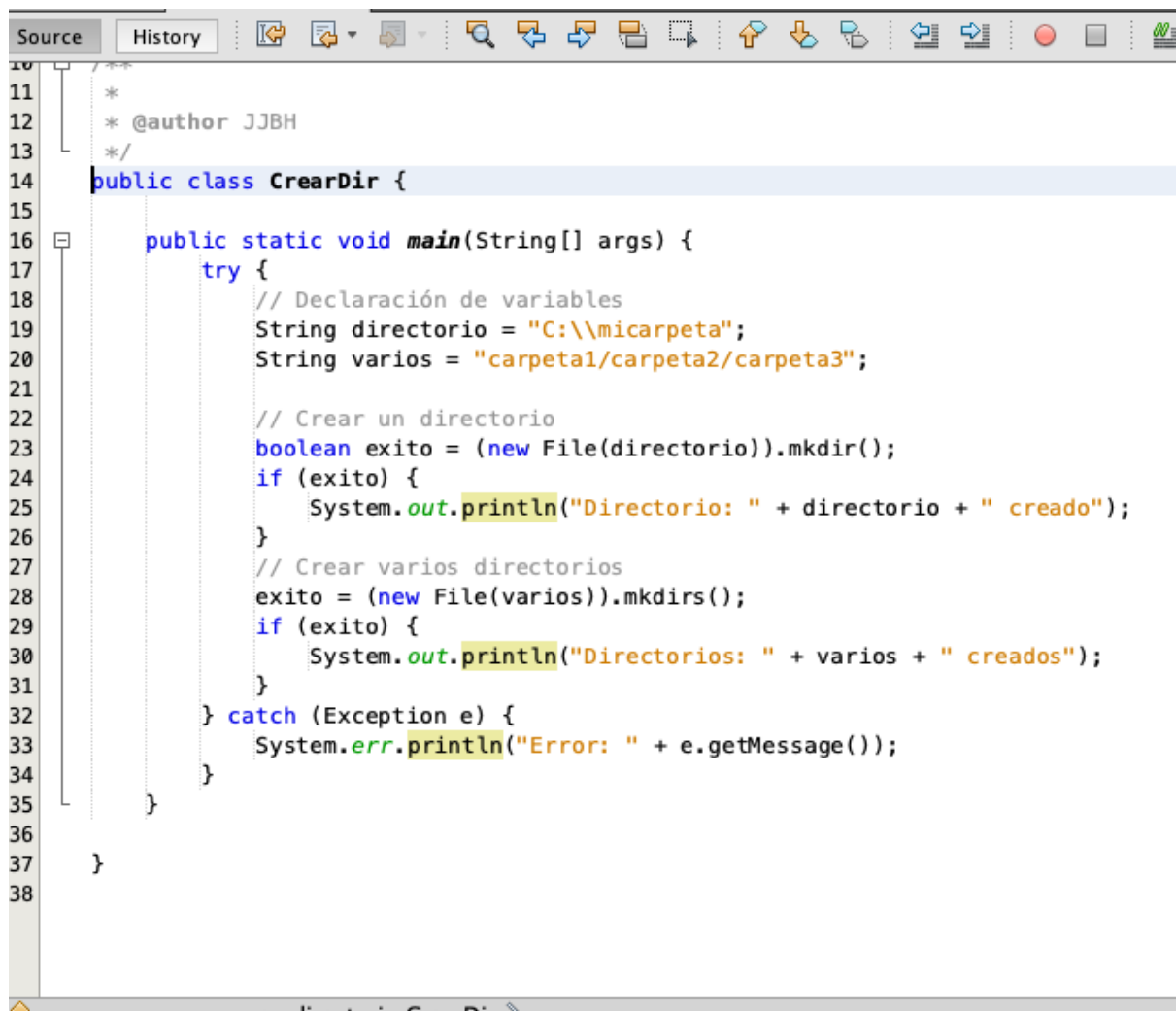
- Fijamos el nombre de la carpeta y del fichero con:

```
File fichero = new File("C:\\prueba", "agenda.txt");
```

- Comprobamos si existe con exists() y si es así lo borramos con

```
fichero.delete()
```

Para crear directorios:



```
10
11  *
12  * @author JJBH
13  */
14  public class CrearDir {
15
16      public static void main(String[] args) {
17          try {
18              // Declaración de variables
19              String directorio = "C:\\\\micarpeta";
20              String varios = "carpeta1/carpeta2/carpeta3";
21
22              // Crear un directorio
23              boolean exito = (new File(directorio)).mkdir();
24              if (exito) {
25                  System.out.println("Directorio: " + directorio + " creado");
26              }
27              // Crear varios directorios
28              exito = (new File(varios)).mkdirs();
29              if (exito) {
30                  System.out.println("Directorios: " + varios + " creados");
31              }
32          } catch (Exception e) {
33              System.err.println("Error: " + e.getMessage());
34          }
35      }
36  }
37
38
```

Para borrar un directorio

Borramos cada uno de los ficheros y directorios que contenga, se puede recorrer recursivamente el directorio y borrar los ficheros. podemos listar el directorio con:

```
File[] ficheros = directorio.listFiles();
```

y luego ir borrando. Sabemos si un elemento es directorio con el método `isDirectory`

8. Almacenamiento de objetos en ficheros. Persistencia. Serialización

La serialización es el proceso por el que un objeto se convierte en una secuencia de bytes para más tarde poder reconstruir el valor de sus variables. Permite guardar un objeto en un archivo. Para lo cual.

- El objeto debe implementar la interfaz `java.io.Serializable`, la cual no tiene métodos, solo informa al JVM que es un objeto serializable.
- Todos los objetos incluidos en él tienen que implementar dicha interfaz.

Los tipos primitivos en Java son serializables por defecto, al igual que los array y otros muchos tipos estándar.

Para leer objetos serializados:

```
FileInputStream fich = new FileInputStream("str.out");
ObjectInputStream os = new ObjectInputStream(fich);
Object o = os.readObject();
```

`readObject` lee un flujo de entrada `fich`, cuando se leen objetos desde un flujo se tiene en cuenta que tipo de objetos se esperan en el flujo y se han de leer en el mismo orden en que se guardaron.

8.1. Serialización: utilidad

Su desarrollo viene de la necesidad de convertir los parámetros necesarios a enviar a un objeto en una máquina remota o devolver valores desde ella en forma de flujos de bytes. Enviar primitivos es más fácil pero objetos complejos no.

El método `writeObject` guarda un objeto a través de un flujo de salida. El objeto pasado a `writeObject` implementa la interfaz `Serializable`.

```
FileOutputStream fisal = new FileOutputStream("cadenas.out");
ObjectOutputStream oos = new ObjectOutputStream(fisal);
oos.writeObject();
```

La serialización de objetos se emplea también con `JavaBean`. Las clases `bean` se cargan en herramientas de construcción de software visual, como `NetBeans`.

Con la paleta de diseño se puede personalizar el `bean` asignando fuentes, tamaños, texto y otras propiedades.

Una vez que se ha personalizado el `bean`, para guardarlo, se emplea la serialización: se almacena el objeto con el valor de sus campos en un fichero con extensión `.ser`, que suele emplazarse dentro de un fichero `.jar`.

9. Mapa conceptual

