Ficheros.

Las computadoras utilizan archivos para la retención a largo plazo de grandes cantidades de datos, incluso hasta después de que terminan los programas que crean esos datos.

Nos referimos a los datos que se mantienen en archivos como datos persistentes, ya que existen más allá de la duración de la ejecución del programa. Las computadoras almacenan archivos en dispositivos de almacenamiento secundario como discos duros, discos ópticos y cintas magnéticas.

El procesamiento de archivos es una de las herramientas más importantes que debe tener un lenguaje para soportar las aplicaciones comerciales, que generalmente procesan cantidades masivas de datos persistentes.

. El término "ﬂujo" se refiere a los datos ordenados que se leen de (o se escriben en) un archivo

14.2 Jerarquía de datos

Un campo es un grupo de caracteres o bytes que transmiten cierto significado. Por ejemplo, un campo que consiste de letras mayúsculas y minúsculas puede utilizarse para representar el nombre de una persona. Los campos están compuestos de caracteres o bytes

Generalmente, varios campos relacionados (pertenecen a una misma entidad) forman un registro (que se implementa como class en Java):

Nif (String).

Nombre (String).

Dirección (String).

Sueldo por hora (double).

Un fichero está formado por un conjunto de registros relacionados, representan información sobre entidades de un mismo tipo. En algunos sistemas operativos, un archivo se ve simplemente como una colección de bytes.

Existen muchas formas de organizar los registros en un archivo. La organización más común se conoce como archivo secuencial, en el cual los registros se almacenan uno junto al otro.

**14.3 Archivos y ﬂujos**

Java considera la comunicación con cada archivo como un ﬂujo (stream) secuencial de bytes que implementa mediante un objeto que hace de intermediario entre el programa y el fichero. Esto es, el programa leerá o escribirá en el flujo sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se tradu­ce en una facilidad más a la hora de escribir programas. Cada sistema operativo proporciona un mecanismo para determinar el fin de un archivo, como el marcador de fin de archivo. Un programa de Java que procesa un ﬂujo de bytes simplemente recibe una indicación del sistema operativo cuando el programa llega al fin del ﬂujo. En algunos casos, la indicación de fin de archivo ocurre como una excepción.

Los algoritmos para leer y escribir datos son siempre más o menos los mismos:

|  |  |
| --- | --- |
| Leer | Escribir |
| Abrir un flujo desde un fichero Mientras haya información Leer información Cerrar el flujo | Abrir un flujo hacia un fichero Mientras haya información Escribir información Cerrar el flujo |

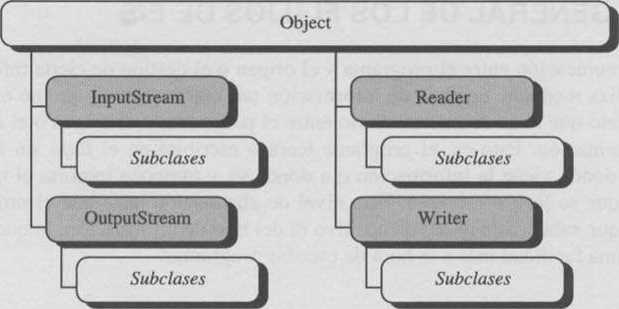
Los ﬂujos de archivos se pueden utilizar para la entrada y salida de datos, ya sea como caracteres o bytes. Los ﬂujos que reciben y envían bytes a archivos se conocen como ﬂujos basados en bytes, y almacenan datos en su formato binario. Los ﬂujos que reciben y envían caracteres de/a los archivos se conocen como ﬂujos basados en caracteres, y almacenan datos como una secuencia de caracteres.

Los archivos que se crean usando ﬂujos basados en bytes se conocen como archivos binarios, mientras que los archivos que se crean usando ﬂujos basados en caracteres se conocen como archivos de texto.

Un programa de Java abre un archivo creando un objeto y asociándole un ﬂujo de bytes o de caracteres. Java también puede asociar ﬂujos de bytes con distintos dispositivos. De hecho, Java crea tres objetos ﬂujo que se asocian con dispositivos cuando un programa de Java empieza a ejecutarse: System.in, System.out y System.err.

Los programas de Java realizan el procesamiento de archivos utilizando clases del paquete java.io.

El paquete java.io de la biblioteca estándar de Java, contiene una colección de clases que soportan estos algoritmos para leer y escribir. Estas clases se divi­den en dos grupos distintos, según se muestra en la figura siguiente. El grupo de la izquierda ha sido diseñado para trabajar con datos de tipo byte (8 bits) y el de la derecha con datos de tipo char (16 bits). Ambos grupos presentan clases análo­gas que tienen interfaces casi idénticas, por lo que se utilizan de la misma manera.



Este paquete incluye definiciones para las clases de ﬂujo como FileInputStream (para la entrada basada en bytes des- de un archivo), FileOutputStream (para la salida basada en bytes hacia un archivo), FileReader (para la entra- da basada en caracteres desde un archivo) y FileWriter (para la salida basada en caracteres hacia un archivo).

Java implementa la jerarquía de clases para la E/S sin incluir demasiadas ca­pacidades dentro de una clase porque rara vez se necesitan muchas de ellas al mismo tiempo. En cambio, sí se pueden obtener todas esas capacidades superpo­niendo una clase sobre otra. Por ejemplo, la clase InputStream no utiliza un buffer; sin embargo, la clase BufferedlnputStream añade un buffer a la clase InputStream.

Java contiene clases que permiten al programador realizar operaciones de entrada y salida con objetos o variables de tipos de datos primitivos. Los datos se siguen almacenando como bytes o caracteres tras bambalinas, lo cual permite al programador leer o escribir datos en forma de enteros, cadenas u otros tipos de datos, sin tener que preocuparse por los detalles acerca de convertir dichos valores al formato de bytes. Para realizar dichas operaciones de entrada y salida, pueden usarse objetos de las clases ObjectInputStream, y ObjectOutputStream junto con las clases de ﬂujos de archivos basadas en bytes FileInputStream y FileOutputStream.

La jerarquía completa de clases en el paquete java.io puede consultarse en la documentación en línea, en la página:

java.sun.com/javase/8/docs/api/java/io/package-tree.html

http://docs.oracle.com/javase/8/docs/api/java/io/package-tree.html

**14.4 La clase File**

Los objetos de la clase File no abren archivos ni proporcionan herramientas para procesarlos. No obstante, los objetos File se utilizan frecuentemente con objetos de otras clases de java.io para especificar los archivos o directorios que van a manipularse. Es especialmente útil para recuperar información acerca de un archivo o directorio de un disco.

Creación de objetos File

La clase File proporciona cuatro constructores.

public File( String nombre )

Ej: File nombre = new File( ruta );

especifica el nombre de un archivo o directorio que se asociará con el objeto File. El nombre puede contener información sobre la ruta, así como el nombre de un archivo o directorio. La ruta de un archivo o directorio especifica su ubicación en el disco. La ruta incluye algunos o todos los directorios que conducen a ese archivo o directorio. Una ruta absoluta contiene todos los directorios, empezando con el directorio raíz, que conducen a un archivo o directorio específico. Cada archivo o directorio en un disco duro específico tiene el mismo directo- rio raíz en su ruta. Una ruta relativa normalmente empieza desde el directorio en el que la aplicación empezó a ejecutarse y es, por lo tanto, una ruta "relativa" al directorio actual.

public File( String rutaAlNombre, String nombre )

usa el argumento rutaAlNombre (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por nombre.

public File( File directorio, String nombre )

usa un objeto File existente llamado directorio (una ruta absoluta o relativa) para localizar el archivo o directorio especificado por nombre.



|  |  |
| --- | --- |
| mkdir  mkdirs  delete  deleteOnExit  createTempFile  renamcTo  setReadOnly  toString | Crea el directorio especificado por el objeto File.  Crea el directorio especificado por el objeto File incluyen­do los directorios que no existan en la ruta especificada.  Borra el fichero o directorio especificado por el objeto File. Cuando se trate de un directorio, éste debe de estar vacío.  Igual que delete. pero cuando la máquina virtual termina.  Crea el fichero vacío especificado por los argumentos pa­sados, en el directorio temporal del sistema.  Renombra el fichero especificado por el objeto File que re­cibe este mensaje, con el nombre especificado por el objeto File pasado como argumento.  Marcar el fichero o directorio especificado por el objeto File de sólo lectura.  Devuelve la ruta especificada cuando se creó el objeto File. |

public class DemoFile {

// muestra información acerca del archivo especificado por el usuario

public void analizarRuta(String ruta) {

// crea un objeto File con base en la entrada del usuario

File nombre = new File(ruta);

if (nombre.exists()) // si existe el nombre, muestra información sobre él

{

// muestra información del archivo (o directorio)

System.out.printf(

"%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",

nombre.getName(), " existe",

(nombre.isFile() ? "es un archivo" : "no es un archivo"),

(nombre.isDirectory() ? "es un directorio"

: "no es un directorio"),

(nombre.isAbsolute() ? "es ruta absoluta"

: "no es ruta absoluta"), "Ultima modificacion: ", nombre.lastModified(), "Tamaño: ", nombre.length(),

"Ruta: ", nombre.getPath(), "Ruta absoluta: ",

nombre.getAbsolutePath(), "Padre: ", nombre.getParent());

if (nombre.isDirectory()) // muestra el listado del directorio

{

String directorio[] = nombre.list();

System.out.println("\n\nContenido del directorio:\n");

for (String nombreDirectorio : directorio) {

System.out.printf("%s\n", nombreDirectorio);

}

} // fin de else

} // fin de if exterior

else // no es archivo o directorio, muestra mensaje de error

{

System.out.printf("%s %s", ruta, "no existe.");

} // fin de else

} // fin del método analizarRuta

}

**Lectura de ficheros secuenciales de texto**

package ficherostexto1;

import java.io.\*;

/\*\*

\*

\* @author juan

\*/

public class FicherosTexto1 {

private static final String FILE\_NAME = "texto.txt";

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

// TODO code application logic here

FileReader input = null;

int nA = 0;

try {

input = new FileReader(FILE\_NAME);

int n;

char c;

while ((n = input.read()) != -1) {

c = (char) n;

System.out.println(c +", ");

if ('a' == ((char) c)) {

nA++;

}

}

System.out.printf("%d \n", n );

System.out.println("El caracter a aparece "+nA+" veces");

} catch (FileNotFoundException ex) {

System.out.println("no se ha podido abrir");

} catch (IOException ex) {

System.out.println("problema leyendo");

} finally {

try {

input.close();

} catch (IOException ex) {

System.out.println(" problema cerrando");

}

}

}

}

Otros métodos interesantes de FileReader

Están definidos en la clase InputStreamReader que es

Extendida por FileReader

int read(char[] buf, int offset, int length)

Este método lee como máximo length caracteres del archivo y los coloca

En el vector buf a partir de la posición offset.

Devuelve el número de caracteres leídos, o -1 indicando la finalización del archivo.

int read(char[] buf) Como

La anterior pero usando 0 como offset y buf.length como length

**Escritura de ficheros secuenciales de texto**

package ficherostexto2;

import java.io.\*;

import java.util.Scanner;

public class FicherosTexto2 {

private static String FILE\_NAME = "backwards.txt";

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

try {

System.out.print("Introduzca un texto: ");

String text = in.nextLine();

FileWriter output = new FileWriter(FILE\_NAME);

for (int i = text.length() - 1; i >= 0; i--) {

output.write(text.charAt(i));

}

output.close();

} catch (IOException ex) {

System.out.println("Something bad has happended");

}

}

}

**El concepto de Buffering**

*Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja.*

*Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que está lleno. Por ello has de hacer menos viajes cuando usas el carrito.*

Cualquier operación que implique acceder memoria externa es muy costosa por lo que es interesante intentar reducir al máximo las operaciones de entrada salida que realizamos sobre los ficheros, haciendo que cada operación lea o escriba muchos caracteres. Además eso también permite operaciones de más alto nivel como la de leer una línea completa y devolverla en forma de cadena.

package ficherobuffer;

import java.io.\*;

public class FicheroBuffer {

private static String grita(String text) {

return text.toUpperCase();

}

public static void main(String[] args) {

try {

BufferedReader input = new BufferedReader(new FileReader("texto.txt"));

BufferedWriter output = new BufferedWriter(new FileWriter("grito.txt"));

String line = input.readLine();

while (line != null) {

System.out.println(line);

String linea = grita(line);

// output.write(linea, 0, linea.length());

output.write(linea);

output.newLine();

line = input.readLine();

}

input.close();

output.close();

} catch (IOException ex) {

System.out.println("Algo va mal");

}

}

}

**14.6 Serialización de objetos**

. Un objeto serializado es un objeto que se representa como una secuencia de bytes, la cual incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de los datos almacenados en el mismo. Una vez que se escribe un objeto serializado en un archivo, se puede leer de ese archivo y deserializarse; es decir, la información del tipo y los bytes que representan al objeto y sus datos se puede utilizar para recrear el objeto en memoria.

Las clases ObjectInputStream y ObjectOutputStream, que implementan en forma respectiva a las interfaces ObjectInput y ObjectOutput, permiten leer/escribir objetos completos de/en un ﬂujo (posiblemente un archivo). Para utilizar la serialización con los archivos, inicializamos los objetos ObjectInputStream y ObjectOutputStream con objetos ﬂujo que pueden leer y escribir información desde/hacia los archivos; objetos de las clases FileInputStream y FileOutputStream, respectivamente. La acción de inicializar objetos ﬂujo con otros objetos ﬂujo de esta forma se conoce algunas veces como envoltura: el nuevo objeto ﬂujo que se va a crear envuelve al objeto ﬂujo especificado como un argumento del constructor. Por ejemplo, para envolver un objeto FileInputStream en un objeto ObjectInputStream, pasamos el objeto FileInputStream al constructor de ObjectInputStream.

La interfaz ObjectOutput contiene el método writeObject, el cual toma un objeto Object que implementa a la interfaz Serializable (que veremos en breve) como un argumento y escribe su información a un objeto OutputStream. De manera correspondiente, la interfaz ObjectInput contiene el método readObject, el cual lee y devuelve una referencia a un objeto Object de un objeto InputStream. Una vez que se lee un objeto, su referencia puede convertirse en el tipo actual del objeto.

La serialización de objetos se realiza mediante ﬂujos basados en bytes, de manera que los archivos secuenciales que se creen y manipulen serán archivos binarios.