COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación			CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tema 8 – Acceso a archivos

El manejo de archivos desde un programa es esencial para que la vida de los datos procesados vaya más allá del tiempo de ejecución de un programa. De poco vale tener un procesador de textos si cuando acabo de escribir no puedo guardar el texto o imprimirlo.

Un archivo es una colección de datos almacenada o disponible en alguna parte externa del sistema CPU+Memoria Principal.

Son estructuras de datos externas (no están en memoria central) y dinámicas (sin tamaño predeterminado).

Para Java un archivo es una secuencia de bytes (stream) que puede ser leída o escrita de diferentes formas dependiendo de las clases usadas: byte a byte, como caracteres Unicode, como textos, como objetos, …

> Ejemplo: Hay streams basados en byte y basados en carácter.
>
> Si se guarda 5 de la primera forma, como entero, se guarda un 5 en binario en el archivo.
>
> Si se guarda un '5' (un char) realmente se almacena un 53 en binario (UTF-8) en el archivo.
>
> El primer stream crea archivos binarios el segundo archivos de texto. Puedes ver un archivo de esta forma con un programa como HxD (https://mh-nexus.de/en/hxd/).

Por tanto es importante resaltar que un archivo, al igual que en la memoria del ordenador, sólo hay números y que dependiendo como sean interpretados esos números pueden significar una cosa u otra.

> Actividad: Entendamos lo anterior con un ejemplo más complejo. Abre un programa como GIMP o Paint y crea una imagen nueva de 1 linea y 10 pixels. Coge una herramienta que te permita pintar pixel a pixel. Usa le herramienta de cambio de colores de forma que pongas el número equivalente UTF-8 de

cada letra en cada componente en el orden BGR. Es decir, si quieres poner LOL, en la componente B y en la R pones el color 76 y en la G el valor 79. Pinta uno o varios píxeles. Graba como BMP y luego mira el contenido del archivo con un editor de texto simple o incluso con el comando cat de consola linux o el type de Windows. ¿Cómo escribirías HOLA!! ?

Esa secuencia tiene un final que es una marca "fin de archivo" y que puede variar según la plataforma (UNIX, Windows, ...). Nosotros detectaremos los finales de archivos por el valor devuelto de alguna función, por el tamaño del archivo o por el salto de una excepción.

Java accede archivo mediante clases. Existen diversas clases para el acceso a archivos, pero nos centraremos en algunas sencillas.

Nota: en sistemas de archivo antiguos se usaba el carácter 0x1A (SUB) ascii como indicador de fin de archivo (End of File), pero actualmente está en desuso siendo el sistema operativo el que indica dicha situación a la aplicación.

Diferenciaremos los archivos en las siguientes categorías:

- Texto: Se trata de manejar los archivos solo con caracteres Unicode (UTF-8, UTF-16). Su uso es muy similar al *System.out* y el Scanner que hemos visto para consola (De hecho usaremos mismas clases o con herencia común).

  Los archivos creados son legibles desde cualquier editor de texto. Son los que más usaremos.

- Streams de datos: pueden guardar cualquier tipo de dato en forma binaria. Existen varios tipos aunque nos pararemos en los de acceso aleatorio.

- Serialización: forma sencilla de guardar/leer objetos en archivos incluyendo información sobre tipos de datos.

Este curso empezaremos a trabajar con los archivos de texto y, opcionalmente, con archivos binarios (en apéndice). Los primeros son más sencillos de entender y nos da pie a comprender el manejo de estas estructuras de datos. Los archivos binarios tienen algo más de complejidad por ser de uso a bajo nivel, pero además de trabajar de forma opcional con ellos este curso, se trabajará más a fondo en varias

asignaturas de segundo curso.

Para trabajar con archivos se usa esencialmente el **paquete java.io** puedes ver la información completa de este paquete en:

http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html

Pero antes de entrar a trabajar con streams, vamos a ver como podemos obtener información general sobre archivos y directorios que haya en el disco duro de nuestro equipo.

Trabajo con archivos y directorios: clase File

Para obtener información sobre archivos o directorios que se encuentren en el disco duro Java dispone principalmente de la clase **File**.

Dicha clase dispone de varios constructores pero el que más nos interesa para empezar es el que se le pasa un **String con el path de un directorio o de un archivo** (exista o no) a partir del cual se pueden realizar diversas tareas.

A continuación se citan algunos métodos interesantes de dicha clase:

boolean **exists**() : Devuelve true si el archivo o directorio representado por el objeto existe.

boolean **isFile**() : Devuelve true si el elemento es un archivo.

boolean **isDirectory**() : Devuelve true si el elemento es un directorio.

String **getName**(): Devuelve el nombre del archivo o directorio.

String **getPath**(): Devuelve la trayectoria completa con el nombre del Archivo o directorio.

String **getParent**(): Devuelve sólo la trayectoria que contiene al elemento.

long **length**(): Tamaño del archivo en bytes.

String[] **list**(): Devuelve un array con la lista de elementos incluidos en el directorio.

File[] **listFiles**(): Devuelve un array con los elementos que hay en un directorio pero como objetos File en lugar de simplemente Strings, lo que da más información sobre cada uno.

boolean **createNewFile**(): Si el archivo no existe lo crea y devuelve true, si existe devuelve false.

boolean **delete**(): Borra el archivo o directorio. Devuelve true si lo ha borrado.

`boolean` **`mkdir`**`(): Crea un directorio nuevo. Devuelve true si lo ha creado.`

Esto es sólo un extracto, puedes ver que existen más funciones con un uso bastante claro según el nombre: renombrar, cambiar permisos de lectura, escritura y ejecución, obtener datos sobre el disco duro en cuanto a capacidad total, espacio usado y espacio libre, listar solo ficheros o unos ficheros determinados, trabajo con URLs además de ficheros, etc...

Puedes ver la información completa en:

http://docs.oracle.com/javase/7/docs/api/java/io/File.html

Existe también el **package java.nio.file** con clases como **Files** que aumenta la funcionalidad en el acceso a archivos pero no las veremos.

A continuación un ejemplo sencillo de uso que puedes ejecutar directamente en un programa de consola:

```
Scanner sc = new Scanner(System.in);
String path;

System.out.printf("Introduce path de archivo o directorio: ");
path = sc.nextLine();
File f = new File(path);

if (f.exists()) {
  System.out.printf(
       "%s existe\n%s un archivo\nTamaño %d bytes\n",
       f.getName(), f.isFile() ? "Es" : "No es", f.length());
  System.out.printf("La ruta completa es %s\n", f.getPath());
} else {
       System.out.println("El fichero/directorio no existe");

}
```

Archivos de Texto

Se entiende por un archivo de texto aquel que está compuesto sólo por caracteres en algún sistema de codificación (ASCII, UTF-8,...). Java es configurable en cuanto al sistema de codificación usado pero por defecto usa el propio de la plataforma. Esto significa que si grabo archivos en Java con las clases adecuadas para trabajar con archivos de texto, me generará archivos que podrán ser **abiertos en un editor cualquiera** como vscode, vim o notepad del propio sistema.

A la inversa también es cierto, si grabamos un archivo en dichos editores, podremos leerlos con las clases Java adecuadas sin ningún problema.

Este tipo de archivos se dice que son de **acceso secuencial** porque para llegar a cualquier punto del archivo hay que pasar por todos los puntos anteriores. Esto contrasta con los llamados archivos de acceso aleatorio (random access files) que tienen comandos para acceder a una posición determinada del archivo de forma directa.

En realidad llevamos una buena parte del curso trabajando con archivos de texto ya que **la consola realmente la tratamos como un archivo**. Lo que hacemos es obtener un stream de caracteres desde el teclado mediante la clase **Scanner** y escribimos datos en la consola mediante la clase **PrintWriter** que es similar a **PrintStream, el tipo de System.out** (La primera gestiona caracteres y la segunda bytes)**.**

Por tanto veremos que la escritura de archivos de texto es muy similar a lo que ya hemos hecho. Simplemente hay que tener claros los pasos que se han de dar cuando trabajamos con archivos, ya sea para leerlos o escribirlos.

Pasos para escritura/lectura de un archivo.

- **Apertura** del archivo: En lenguajes como Java esto se hace habitualmente en la llamada al constructor de una clase, ya que habrá un objeto que represente al archivo. Al abrir el archivo informamos al sistema operativo que lo vamos a usar y lo bloquea.

- **Procesado** del archivo: Aquí se encuentras los distintos comandos de escritura y lectura de datos del archivo. Lógicamente serán métodos del objeto que representa al archivo.

- **Cierre** del archivo: Siempre es necesario cerrar el archivo cuando se termina de usar por varios motivos: liberar recursos, liberar el archivo para que lo puedan usar otras aplicaciones u otras partes de la aplicación y volcar los datos que aún no estén guardados en el archivo (ya que el sistema gestiona el archivo en memoria). Se usará el método **close**().

Escritura de archivos de texto

Para escribir datos de texto en un archivo usaremos la clase **PrintWriter** ayudados en ocasiones por la clase **FileWriter** más genérica. Veamos un ejemplo sencillo que luego explicaremos:

```
String home=System.getProperty("user.home");
try {
	PrintWriter f = new PrintWriter(home+"/prueba.txt"); // Apertura
	f.println("hola que tal"); // Proceso
	f.close();  // Cierre (Ojo, solo ejemplo, lo haremos de otra manera)
} catch (FileNotFoundException e) {
	System.err.println("Error de acceso al archivo");
}
```

Variables de entorno

Antes de entrar en la lectura del archivo hablaremos de las variables de entorno:

La primera línea (**System.getProperty**) permite obtener el directorio de usuario actual en el sistema. Es una llamada a una función estática que gestiona las denominadas Propiedades de Sistema y nos informa de distintos aspectos de la máquina donde se está ejecutando el programa y que me puede facilitar la labor. Puedes verlo como una serie de variables de entorno como pueden ser las de Windows o Linux pero que están configuradas en Java siendo, por tanto, multiplataforma.

Se guarda en forma de lo que se denomina tabla hash, es decir, es una colección, pero tipo diccionario.En lugar de estar indexada, se accede por clave.

También se pueden añadir propiedades con **System.setProperty** para distintos usos.

Por ejemplo:

```
System.setProperty("web", "www.colegiovivas.com");
System.out.println(System.getProperty("web"));
```

Puedes probar las siguientes líneas para ver algo más de información que se puede obtener:

```
System.out.println("Sistema: " + System.getProperty("os.name"));
System.out.println("Directorio de almacenamiento temporal:"
            + System.getProperty("java.io.tmpdir"));
System.out.println("Ususario: " + System.getProperty("user.name"));
System.out.println("Directorio de usuario: "
            + System.getProperty("user.home"));
System.out.println("Directorio actual: "
            + System.getProperty("user.dir"));
System.out.println("Barra separadora: "
            + System.getProperty("file.separator"));
```

Puedes ver una lista de propiedades estándar en:

https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

Por supuesto también es posible utilizar variables de entorno del propio sistema operativo, pero hay que tener en cuenta que esto hace perder la capacidad multiplataforma. En este caso el método para realizarlo es la función **System.getenv(String variable)** como se ve en el siguiente ejemplo:

```
String user = System.getenv("HOMEDRIVE")+System.getenv("HOMEPATH");
String os = System.getenv("OS");
System.out.printf("Path de usuario: %s\n Sistema Operativo: %s\n", user, os);
```

Tal y como se ha comentado, el ejemplo previo solo funciona en un sistema Windows.

Por lo tanto siempre que sea posible es preferible usar las variables de entorno de Java.

Más info: https://www.scaler.com/topics/get-environment-variable-java/

Apertura del archivo: Creación/Sobreescritura.

Dentro del *try* tenemos los "tres pasos" de los que hablábamos: La creación del archivo mediante el constructor, la escritura del archivo y finalmente el cierre del archivo. La primera línea es la apertura:

```
PrintWriter f = new PrintWriter(home+"/prueba.txt");
```

Se realiza mediante un constructor. Se especifica el nombre del archivo con su ruta y si hay permisos y la ruta existe se crea el archivo. Si no, salta una excepción. El problema que tal como está en el ejemplo, siempre se crea un archivo nuevo y se escriben los datos en él. En caso de que el archivo ya exista, lo borra y lo vuelve a crear.

Apertura del archivo: Añadir datos (append).

Si no deseamos este comportamiento porque queremos añadir datos al final del archivo hay que realizar lo que se denomina un "**append**", es decir, abrir el archivo para **añadir datos**. Para ellos el constructor se ejecutaría de la siguiente forma (puedes añadir las siguientes líneas a continuación de las primeras):

```
try {
        PrintWriter f = new PrintWriter(new FileWriter(home
                        + "/prueba.txt", true));
        f.println("Añadimos texto al final");
        f.close();
} catch (Exception e) {
        System.err.println("Error de acceso al archivo");
}
```

En este caso a la nueva sobrecarga del constructor en lugar de pasarle el nombre de un archivo se le pasa un objeto tipo **FileWriter** que permite pasarle a su vez a su constructor un booleano que indica si **añade** datos (**true**) o **sobreescribe** (**false**). Esta clase se podría usar también directamente para escribir, pero tiene el inconveniente que solo permite manejar carácter a carácter (*Character Stream*) por eso no la vemos en profundidad.

Escritura de datos en el archivo

Se realizan con los mismos métodos que ya vimos para la salida estándar System.out: **print, println** y **printf** (se usa también **format** con la misma funcionalidad que printf). Añade las siguientes líneas antes del close.

```
int a=100;
f.printf("Formateando texto en el archivo: %5d%n",a);
f.format("Formateando texto en el archivo: %5d%n",a*a);
```

Si te fijas al final de la cadena de formato se usa %n en lugar de \n. Esto permite usar un retorno de carro adaptado al sistema operativo (UNIX \n, Windows \r\n). El *format* y por defecto *printf* permite muchas más acciones y posibilidades de las que vimos.

Si quieres profundizar echa un ojo en:

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Formatter.html

Cierre del archivo

Para liberar recursos y para volcar los datos de memoria al archivo es necesario ejecutar un **close**. El paso de volcado de memoria sin cierre se puede hacer también ejecutando el comando **flush**.

Realmente para realizar correctamente el cierre de un archivo el código debería ser algo así:

```
PrintWriter f = null;
try {
	f = new PrintWriter(new FileWriter(home
			+ "/prueba.txt", true));
	f.println("Añadimos texto al final");

} catch (Exception e) {
	System.err.println("Error");
} finally {
	if (f != null) {
		try {
			f.close();
		} catch (Exception e2) {
			System.err.println("Error");
		}
	}
}
```

Es mejor hacerlo de esta manera que de la anterior ya que independientemente que salte o no la excepción, el archivo se cierra. Una excepción puede saltar o en el

new (con lo cual el archivo no estaría abierto y sería null) o en el código que va a continuación (incluso por motivos que no tengan que ver con archivos), en ese caso si salta la excepción habría que cerrarlo.

Por supuesto esto conlleva a definir el archivo **f** fuera del *try*.

try-with-resources

Desde Java 7 existe una forma más cómoda de hacer esto último (cierre) y es usar una nueva estructura denominad **try-with-resources**. En este caso en el try se indica el recurso que debe ser cerrado (archivos, sockets,…) y el sistema lo cierra de forma automática al terminar el try si ha sido abierto haya o no algún problema con el uso de dicho recurso.

Para que un recurso pueda ser usado por esta estructura, la clase que lo define debe implementar el **interface java.lang.AutoCloseable**.

El ejemplo anterior quedaría así:

```
try (PrintWriter f = new PrintWriter(new FileWriter(home
        + "/prueba.txt", true))) {
   f.println("Añadimos texto al final");
} catch (IOException e) {
   System.err.println("Error de acceso al archivo");
}
```

Lo cual simplifica notablemente el cierre de archivos.

Los siguientes enlaces profundizan en el uso de esta estructura así como en las nuevas funcionalidades que traen desde Java 9.

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

https://dzone.com/articles/java-code-bytes-be-resourceful-with-try-with-resou

Lectura de archivos de texto

El proceso es similar al anterior pero en este caso se usa la clase **Scanner** que ya conocemos con la diferencia de que en lugar de esperar datos del teclado los esperamos de un archivo del disco duro. Un ejemplo de uso:

```
String archivo = System.getProperty("user.home") + "/prueba.txt";
String texto;
try (Scanner f = new Scanner(new File(archivo))) { //Apertura y cierre
	while (f.hasNext()) {   // Procesado
		texto = f.nextLine();
		System.out.println(texto);
	}
} catch (IOException e) {
	System.err.println("Error de acceso al archivo: " + e.getMessage());
}
```

La novedad es este caso es la realización del bucle que se queda leyendo datos mientras existe alguno más (**hasNext**). También se podría realizar un bucle infinito (while (true)) y esperar a que saltara una excepción de haber llegado al final (habitualmente **NotSuchElementException** porque no encuentra más lineas, pero en algunas clases puede dar **EOFException***:* End Of File Exception).

Al ser un Scanner se permite la lectura de cualquier tipo de dato realizando una conversión inmediata.

Observa el siguiente código para entender esto:

```
import java.io.*;
import java.util.Scanner;

class Persona {
	private String nombre;
	private int edad;

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
```

```
	public int getEdad() {
		return edad;
	}

	public void setEdad(int edad) {
		this.edad = edad;
	}
}

public class ArchivosTexto {
	public static boolean guardarDato(Persona p, String archivo) {
		try (PrintWriter f = new PrintWriter(new FileWriter(archivo, true))) {

			f.println(p.getNombre());
			f.println(p.getEdad());
		} catch (Exception e) {
			return false;
		}
		return true;
	}

	public static void leerDatos(String archivo) {
		Persona p = new Persona();
		try (Scanner f = new Scanner(new File(archivo))){
			while (f.hasNext()) {
				p.setNombre(f.nextLine());
				p.setEdad(f.nextInt());
				f.nextLine();
				System.out.printf("Nombre: %12s  Edad: %4d\n", p.getNombre(),
p.getEdad());
			}
		} catch (Exception e) {
			System.out.println("Error de acceso a archivo:" + e.getMessage());
		}

	}

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		String archivo = System.getProperty("user.home") + "/personas.txt";
		int opcion;
		Persona p;
		do {
			System.out.println("Edades\n______\n");
			System.out.println("1.- Introduce datos");
			System.out.println("2.- Leer datos");
```

```
                System.out.println("3.- Salir");
                System.out.print("\nTeclee opción (1-3): ");

                opcion = Integer.parseInt(sc.nextLine());
                switch (opcion) {
                case 1:
                        p = new Persona();
                        System.out.println("Introduce nombre");
                        p.setNombre(sc.nextLine());
                        System.out.println("Introduce Edad");
                        p.setEdad(Integer.parseInt(sc.nextLine()));
                        guardarDato(p, archivo);
                        break;
                case 2:
                        leerDatos(archivo);
                        break;
                case 3:
                        System.out.println("Hasta otra");
                        break;
                default:
                        System.out.println("Opcion no válida");
                }
            } while (opcion != 3);
        }
}
```

No es necesario que lo ejecutes ahora, simplemente comprueba la estructura. Queda el código fuente en moodle para que hagas pruebas.

Más información:

http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

Swing: JFileChooser

Al igual que existe los cuadros de diálogo JOptionPane, los sistemas suelen tener otros cuadros más o menos estándar para distintas tareas. Uno de ellos es el **JFileChooser** que es el típico formulario de selección de fichero para distintas tareas. Tenemos tres métodos para invocarlo aunque usaremos esencialmente los dos primeros:

> **showOpenDialog**(padre): Usado habitualmente para abrir archivos. El padre es el formulario desde el que se lanza para que aparezca centrado en este. Si se le pasa null, se centra en la pantalla.
>
> **showSaveDialog**(padre): Usado para guardar archivos.
>
> **showDialog**(padre, texto): Permite especificar lo que queremos que aparezca en el botón "aceptar" como texto para otros usos.

De esta forma, para lanzar un selector se hace de la siguiente forma:

```
JFileChooser fc=new JFileChooser();
fc.showOpenDialog(this);
```

Pero lo anterior no vale de nada ya que no recogemos la respuesta del usuario (el botón pulsado). Además suele ser interesante introducir algunos elementos de configuración de la caja como los siguientes:

> **setFileSelectionMode**: Modo de selección, permite decidir si dejamos que el usuario escoja sólo archivos, archivos y directorios, etc... Se usan constantes de JFileChooser.
>
> **addChoosableFileFilter**: Permite indicar qué tipos de archivos pueden ser seleccionados según su extensión. Para ello se usa un filtro creado con la clase **FileNameExtensionFilter** cuyo constructor tiene un primer parámetro que es la descripción del filtro y a continuación las extensiones que queremos incluir en dicho filtro.
>
> Si se desea que alguno de los filtros sea el filtro por defecto, entonces ese se debe añadir con **setFileFilter**, y el resto con addChoosableFileFilter.

La respuesta del diálogo es un número entero que está indicado habitualmente por

las constantes de JFileChooser **APPROVE_OPTION** (Aceptar) y **CANCEL_OPTION** (Cancelar).

Si se ha aceptado, el archivo lo podemos obtener mediante el método **getSelectedFile**.

Vemos todo esto con un ejemplo de funcionamiento:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.filechooser.*;

public class SelectorDeArchivos extends JFrame implements ActionListener{
	JButton btnCargar;
	JLabel lblImagen;

	public SelectorDeArchivos(){
		super("Prueba de JFileChooser");
		this.setLayout(new FlowLayout());

		btnCargar=new JButton("<html><b>Cargar</b><br><font
color=#aaaa00>Imagen</font></html>");
		btnCargar.addActionListener(this);
		add(btnCargar);

		lblImagen=new JLabel();
		add(lblImagen);
	}
	@Override
	public void actionPerformed(ActionEvent e) {
		int respuesta;
		FileNameExtensionFilter filtro=new FileNameExtensionFilter("Imágenes",
"jpg","jpeg","gif","png");
		JFileChooser fc=new JFileChooser();
		fc.addChoosableFileFilter(filtro);
		fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
		respuesta=fc.showOpenDialog(this);

		if (respuesta==JFileChooser.APPROVE_OPTION){
			lblImagen.setIcon(new ImageIcon(fc.getSelectedFile().getPath()));
			lblImagen.setSize(lblImagen.getPreferredSize());
			this.setSize(lblImagen.getWidth()+20, lblImagen.getHeight()+80);
		}
	}
}
```

En este ejemplo mediante **FileNameExtensionFilter** se establece un filtro cuya descripción es "Imágenes" y que permite la selección de los archivos indicados por las extensiones siguientes. Es decir, primer parámetro descripción y resto extensiones asociadas a dicha descripción.

Se agrega ese filtro a los existentes en el JFileChoose con **addChoosableFileFilter**.

Si se desea **desactivar el filtro de "Todos los archivos",** se debe usar el método:

```
fc.setAcceptAllFileFilterUsed(false);
```

A continuación se establece el modo de selección a sólo archivos. Puede establecerse a solo directorios o ambas.

Es entonces cuando se lanza el método showOpenDialog. Aquí se podría lanzar cualquiera de los otros dos según sea nuestro interés. El programa pasa el control al cuadro de diálogo parándose en esta línea a la espera que el usuario acepte o cancele el cuadro de selección.

Finalmente, como se ha recogido la respuesta de diálogo se realiza cierta acción si se ha pulsado el botón que corresponde a Aceptar.

También como añadido se ve cómo establecer las propiedades de un botón mediante HTML al igual que hacíamos con etiquetas.

Esto ha sido una breve introducción a este componente. Realmente es más amplio permitiendo una configuración mayor y otras posibilidades como la selección de múltiples archivos. Si necesitas saber más mira en la documentación siguiente:

https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html

http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

Apéndice I: Ficheros binarios de acceso aleatorio

Los archivos de acceso aleatorio implican que se puede colocar un "cursor" en cualquier posición del archivo para leer o escribir un dato determinado en dicha posición. Pueden verse como vectores con un tamaño indeterminado en los cuales se permite acceder mediante un índice a una posición.

En estos archivos voy a poder escribir cualquier tipo de dato y se guarda en forma binaria: char, integer, String, etc... También puedo leer cualquier tipo de dato. La limitación estriba en que el posicionamiento es por bytes empezando en la posición 0.

Si escribo más allá del archivo no hay problema. El archivo crece hasta el punto necesario (siempre que haya espacio en el soporte de escritura).

El manejo de estos archivo es similar a los de texto aunque teniendo en cuenta que se guarda datos no solo textuales si no de cualquier tipo.

> Ejemplo: Si guardo el número 90 que se encuentra en una variable integer en un archivo de texto, en el archivo voy a ver los caracteres unicode '9' y '0' y por tanto los datos 0x39 y 0x30 que son sus códigos Unicode.
>
> Si embargo si guardo la misma variable con el número 90 como **entero** en un archivo binario grabará 0x5A ocupando 4 bytes que es lo que ocupa un entero.

Además, como ya se comentó, la otra diferencia es que podremos situarnos en cualquier posición del archivo.

En Java manejaremos archivos de acceso aleatorio con la clase *RandomAccessFile*.

Constructor: Usaremos esencialmente una única sobrecarga que admite dos parámetros. El primero es un String con el nombre del archivo. El segundo es también un String en el que indicamos el modo de apertura del archivo. Usaremos dos modos:

"r" Sólo lectura

"rw" Escritura y/o lectura.

Escritura y lectura: Se disponen de varios métodos tanto para leer como para escribir dependiendo del dato con el que estemos trabajado.

Algunos métodos de lectura: *readByte, readChar, readDouble,* … cualquiera de estos métodos lee los bytes a partir de la posición actual del cursos en el archivo para devolverlos como el tipo de dato indicado; *readByte* leerá un byte y lo devolverá, *readInteger* leerá 4 bytes y devolverá un *int*.

Algunos métodos de escritura: *writeChar, writeByte,* … trabajan de forma inversa a los anteriores. En este caso el parámetro es el dato a guardar en el archivo.

Se debe tener en cuenta que al abrir el archivo se empieza en la posición 0, y cada vez que se realiza una operación de escritura o lectura se hace avanzar el cursor interno tantas posiciones como haya sido necesario para realizar la operación.

Posicionamiento: Dos métodos importantes:

> *seek(posición):* Sitúa el cursor interno del archivo en la posición indicada. Se cuenta desde el índice 0.
>
> *getFilePointer():* Devuelve la posición actual del cursor (También llamado puntero de archivo).

Cierre del archivo: Como en el caso de los archivos de texto se hará mediante el método *close* o con try-with-resources.

A continuación un ejemplo con lo que se debería entender todo lo anterior. Se recomienda crear la estructura *try/catch* y rellenarla poco a poco con las distintas acciones. Los resultados se deben ir visualizando en un editor hexadecimal (como el gHex de Linux) En los comentarios aparecen breves explicaciones de lo que se realiza.

```java
import java.io.*;
public class ArchivosBinarios {
	public static void main(String[] a) {
		String archivo = System.getProperty("user.home") + "/binario.dat";
		int n = 0;

		try (RandomAccessFile f= new RandomAccessFile(archivo, "rw")) {
			// Una forma cómoda de sobreescribir si ya existe o reinicializar.
			f.setLength(0);

			// Creo un archivo con números aleatorios
			for (int i = 0; i < 40; i++) {
				n = (int) (Math.random() * 16);
				f.writeByte(n);
			}
			System.out.printf("Tamaño antes del seek: %d\n", f.length());

			// Pongo el nº 255 en las posiciones múltiplos de cinco
			for (int i = 0; i < 20; i++) {
				f.seek(i * 5);
				f.writeByte(255);
			}
			System.out.printf("Tamaño tras el seek: %d\n", f.length());

			f.seek(2);
			//guarda los caracteres en UTF8 y la cantidad escrita
			f.writeUTF("ABC");
			//Guarda solo los caracteres en unicode
			f.writeChars("ABC");

			//Leemos los caracteres escritos en la posición 2
			f.seek(2);
			System.out.println(f.readUTF());

			//Leemos los mismos bytes pero tomándolos como entero
			f.seek(2);
			System.out.println(f.readInt());

		} catch (FileNotFoundException e){
			// Error en el constructor
			System.out.println("Error de apertura: " + e.getMessage());
		} catch (IOException e) {
			// Error al escribir datos
			System.out.println("Error de escritura: " + e.getMessage());
		}
	}
}
```

Más información en: http://docs.oracle.com/javase/tutorial/essential/io/rafs.html

Como ya se mencionó anteriormente, en Java hay múltiples clases que permiten trabajar con archivo. Para el caso de los binarios, si se desea trabajar con otras clases y objetos sencillos para guardar bytes se puede leer más información sobre ByteStreams en:

http://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html

O si se desea trabajar con streams de datos como los vistos en este apartado pero de forma secuencial se puede obtener más información en:

http://docs.oracle.com/javase/tutorial/essential/io/datastreams.html

Apéndice II: Serialización de objetos

Método para almacenar tanto los datos de los objetos como sus tipos. De esta forma no tenemos que preocuparnos en establecer directamente los tipos de datos a la hora de leer, ya que estos tipos están también guardados en el archivo de objetos serializados.

Por tanto un objeto serializado es una secuencia de bytes que representa al objeto, tanto a sus valores como a sus tipos.

Para realizar estas tareas se usan las clases *ObjectInputStream* y *ObjectOutputStream.* Estas clases implementan respectivamente los interfaces *ObjectInput* y *ObjectOutput* que obligan a sobreescribir los métodos *readObject* y *writeObject* respectivamente.

Además para realizar la serialización tenemos que indicarlo en el objeto. Se usa el interface Serializable que es un "tagging interface" lo que significa que NO tiene métodos, simplemente es par marcar (tag) la clase como serializable. Esto es necesario ya que si el objeto no está marcado de esta forma el *ObjectOutputStream* no funcionará.

```
import java.io.*;

public class Persona implements Serializable {
	//El resto igual que estaba
}
```

Hay que tener cuidado con los tipos usados dentro de la clase, ya que si queremos que todo funcione, esos tipos y las superclases deben ser a su vez serializables. Por defecto los tipos primitivos y String son serializables.

Una vez que se comprende la estructura, la forma de uso es muy similar a lo ya visto en otros apartados. Veamos un ejemplo y comentamos luego algunas puntualizaciones que se deben hacer.

```
import java.io.*;
import java.util.Scanner;

public class Serializacion {
	public static void guardarDato(Persona p, String archivo) {
		ObjectOutputStream f = null;

		try {
			f = new ObjectOutputStream(new FileOutputStream(archivo));
			f.writeObject(p);
			f.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}


	public static void leerDatos(String archivo) {
		Persona p = null;
		ObjectInputStream f = null;

		try {
			f = new ObjectInputStream(new FileInputStream(archivo));

			p = (Persona) f.readObject();
			System.out.printf("Nombre: %-12s  Edad: %-4d\n", p.getNombre(),
					p.getEdad());
			f.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		String archivo = System.getProperty("user.home")
				+ "/temp/personas.serial";
		int opcion;
		Persona p;

		do {
			System.out.println("Edades\n______\n");
			System.out.println("1.- Introduce datos");
			System.out.println("2.- Leer datos");
			System.out.println("3.- Salir");
			System.out.print("\nTeclee opción (1-3): ");
			opcion = Integer.parseInt(sc.nextLine());
```

```
                switch (opcion) {
                case 1:
                        p = new Persona();
                        System.out.println("Introduce nombre");
                        p.setNombre(sc.nextLine());
                        System.out.println("Introduce Edad");
                        p.setEdad(Integer.parseInt(sc.nextLine()));
                        guardarDato(p, archivo);
                        break;
                case 2:
                        leerDatos(archivo);
                        break;
                case 3:
                        System.out.println("Hasta otra");
                        break;
                default:
                        System.out.println("Opcion no válida");
                }
            } while (opcion != 3);
        }
}
```

En el ejemplo para hacerlo más breve se han obviado los distintos tipos de excepciones resumiéndolos a Exception, es, en un programa real, no se debe hacer.

Tal y como está el programa solo permite guardar un dato en el archivo. Esto es porque un objeto ObjectOutputStream una vez cerrado no permite añadir nada, si no se corrompe la serialización. Habría que ejecutar el constructor, añadir todos los datos que se desean (por ejemplo de una colección) y finalmente cerrarlo. Esto se debe a que cada vez que se ejecuta un constructor se mete una cabecera en el archivo que debe ser única, por lo que si añadimos, estamos añadiendo nuevas cabeceras.

Otra solución interesante al problema anterior tomada de www.stackoverflow.com es la de crear un objeto ObjectOutputStream al que se le pueden añadir datos simplemente sobreescribiendo la escritura de la cabecera:

```
public class AppendableObjectOutputStream extends ObjectOutputStream {

        public AppendableObjectOutputStream(OutputStream out) {
          super(out);
        }

        @Override
        protected void writeStreamHeader() throws IOException {
          // do not write a header
        }

}
```

De esta forma hay que comprobar si el archivo existe e interesa añadir, se usa esta clase nueva, si el archivo no existe o no interesa añadir se usa el clásico ObjectOutputStream.

Apéndice III: Lista de variables de entorno.

Aunque no hallamos visto aún tablas hash en Java, no debería ser difícil entender el siguiente código que nos da todas (no sólo las vistas anteriormente) las variables de entorno de Java existentes en el sistema que se está ejecutando:

```
Properties p = System.getProperties();
Enumeration<Object> e = p.keys();

while (e.hasMoreElements()) {
	String elemento = e.nextElement().toString();
	System.out.println(elemento + " : " + p.get(elemento));
}
```

Si se quieren ver las variables de entorno propias del sistema operativo la forma sería la siguiente (Similar al set de consola):

```
Map<String, String> variablesSO = System.getenv();

for (String nombre : variablesSO.keySet()) {
	System.out.format("%s = %s", nombre, variablesSO.get(nombre));
	System.out.println();
}
```

Bibliografía

Libros:

Java for programmers. Second Edition. Deitel Developers Series. Prentice Hall 2012.

Otros recursos Web:

Uso básico del JFileChooser:

http://chuwiki.chuidiang.org/index.php?title=JFileChooser

Web de Oracle: http://docs.oracle.com/