09.- Entrada y salida con ficheros

1. Ficheros

En ocasiones necesitamos que los datos que introduce el usuario o que produce un programa **persistan** cuando éste finaliza, es decir, que se conserven cuando el programa termina su ejecución. Para ello es necesario el uso de una base de datos o de **ficheros**, que permitan guardar los datos en un almacenamiento secundario como un pendrive, disco duro, DVD, etc.

En esta unidad se abordan distintos aspectos relacionados con el almacenamiento en ficheros:

- Introducción a conceptos básicos como los de registro y campo
- Clasificación de los ficheros según el contenido y forma de acceso.
- Operaciones básicas con ficheros de distinto tipo.

2. Campos y registros

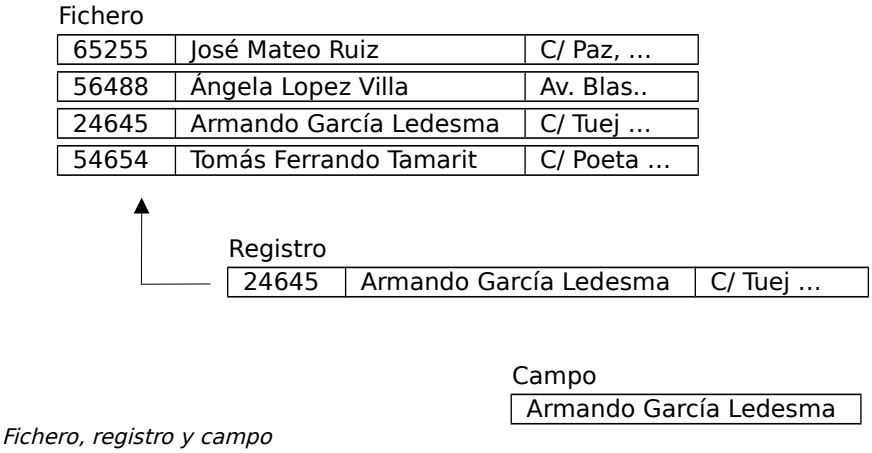
Llamamos ***campo*** a un dato en particular almacenado en una base de datos o en un fichero. Un campo puede ser en nombre de un cliente, la fecha de nacimiento de un alumno, el número de teléfono de un comercio. Los campos pueden ser de distintos tipos: alfanuméricos, numéricos, fechas, etc.

La agrupación de uno o más campos forman un ***registro.*** Un registro de alumno podría consistir, por ejemplo, de los siguientes campos

1. Número de expediente.
2. Nombre y apellidos
3. Domicilio
4. Grupo al que pertenece.

Un ***fichero*** puede estar formado por registros, lo cual dotaría al archivo de cierta estructura. En un fichero de alumnos, por ejemplo, tendríamos un registro por cada alumno. Los campos del registro serían cada uno de los datos que se almacena del alumno: Nº expediente, nombre, etc …

Para poder leer o escribir información en dicho fichero de alumnos imprescindible que sepamos qué estructura tiene el fichero. De lo contrario No sabremos cómo interpretar la información que extraigamos del fichero.



Fichero, registro y campo

3. Ficheros de texto vs ficheros binarios.

Desde un punto de vista a muy bajo nivel, un fichero es un conjunto de bits almacenados en memoria secundaria, accesibles a través de una ruta y un nombre de archivo.

Este punto de vista a bajo nivel es demasiado simple, pues cuando se recupera y trata la información que contiene el fichero, esos bits se agrupan en unidades mayores que las dotan de significado. Así, dependiendo de cuál es el contenido del fichero (de cómo se interpretan los bits que contiene el fichero), podemos distinguir dos tipos de ficheros:

- Ficheros de texto (o de caracteres)
- Ficheros binarios (o de bytes)

Un ***fichero de texto*** está formado únicamente por caracteres. Los bits que contiene se interpretan atendiendo a una tabla de caracteres, ya sea ASCII o Unicode. Este tipo de ficheros se pueden abrir con un editor de texto plano y son, en general, legibles. Por ejemplo, los ficheros .java que contienen los programas que elaboramos, son ficheros de texto.

Por otro lado, los ***ficheros binarios*** contienen secuencias de bytes que se agrupan para representar otro tipo de información: números, sonidos, imágenes, etc. Un fichero binario se puede abrir también con un editor de texto plano pero, en este caso, el contenido será ininteligible. Existen muchos ejemplos de ficheros binarios: el archivo .exe que contiene la versión ejecutable de un programa, un archivo .jpg que contiene una imagen, un documento .pdf, etc.

Las operaciones de lectura/escritura que utilizamos al acceder desde un programa a un fichero de texto están orientadas al carácter: leer o escribir un carácter, una secuencia de caracteres, una línea de texto, etc. En cambio las operaciones de lectura/escritura en ficheros binarios están orientadas a byte: se leen o escriben datos binarios, como enteros, bytes, int, double, etc.

4. Acceso secuencial vs Acceso directo.

Existen dos maneras de acceder a la información que contiene un fichero:

- Acceso secuencial
- Acceso directo (o aleatorio)

Con ***acceso secuencial,*** para poder leer el byte que se encuentra en determinada posición del archivo es necesario leer, previamente, todos los bytes anteriores. Al escribir, los datos se sitúan en el archivo uno a continuación del otro, en el mismo orden en que se introducen. Es decir, la nueva información se coloca en el archivo a continuación de la que ya hay. No es posible realizar modificaciones de los datos existentes, tan solo añadir al final.

Sin embargo, con el **acceso directo**, es posible acceder a determinada posición (dirección) del fichero de manera directa y, posteriormente, hacer la operación de lectura o escritura deseada.

No siempre es necesario realizar un acceso directo a un archivo. En muchas ocasiones el procesamiento que realizamos de sus datos consiste en la escritura o lectura de todo el archivo siguiendo el orden en que se encuentran. Para ello basta con un acceso secuencial.

En este tema vamos a trabajar únicamente con acceso secuencial a ficheros, tanto para leerlos como para escribirlos

5. Clases a utilizar y ejemplos.

Vamos a utilizar una clase distinta dependiendo de si queremos leer o escribir en un fichero y dependiendo de si la información a leer/escribir es texto o binaria. La siguiente tabla resume las clases que vamos a utilizar (aunque existen otras opciones).

	Ficheros binarios	Ficheros de texto
Para **lectura**	**Scanner**(File)	**DataInputStream**(FileInputStream(File))
Para **escritura**	**PrintWriter**(File)	**DataOutputStream**(FileOutputStream(File))

Consulta en la documentación los distintos constructores disponibles para estas clases.

5.1. Escribir un fichero de texto con PrintWriter.

PrintWriter es la clase a la que pertenece el objeto System.out.

System.out es un objeto de la clase PrintWriter que permite escribir en la salida estándar del ordenador (asociada por defecto a la pantalla). Sin embargo, podemos crear objetos de la clase PrintWriter que estén asociados a un fichero , es decir, que escriban en un fichero.

Los métodos de la clase PrintWriter son los ya conocidos:

- print(String): Escribe el String indicado.
- println(String): Escribe el String indicado haciendo un salto de línea posteriormente.

En el siguiente ejemplo vemos como crear un fichero de texto y escribir en el los números del 1 a 100, cada número en una línea.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class _01EscribirTexto1A100 {
    public static void main(String[] args) {
        //Abrimos el fichero numeros1a100.txt para escribir texto en el
        //Si el fichero no existe, lo crea
        //Si el fichero existe, LO SOBREESCRIBE
        PrintWriter f = null;
        try {
            f = new PrintWriter(
                            new File("numeros1a100.txt"));
            for(int i = 1; i <= 100; i++){
                f.println(i);
            }

        } catch (FileNotFoundException e) {
            //ERROR:
            //- El fichero no existe y no se puede crear, o
            //- El fichero existe y no se puede abrir
        System.out.println("Error de apertura");
        } finally {
            if(f!=null) {
                f.close();
            }
        }
    }
}
```

Es muy importante tener en cuenta que cuando se abre un fichero y se escribe en él …

- … si el fichero no existe se crea
- … si el fichero existe, **su contenido se reemplaza** por el nuevo. El contenido previo que tuviera el fichero se pierde.

Observa el ejemplo: Se estructura de la siguiente forma

- Se abre el fichero (creación del objeto PrintStream):
  - **Es muy importante** tener en cuenta que cuando se abre un fichero y se escribe en el…
    - … si el fichero no existe se crea
    - … si el fichero existe, **su contenido se reemplaza** por el nuevo. El contenido previo que tuviera el fichero se pierde.
  - Si el fichero no se puede crear o no se puede abrir, se produce la excepción.
- Escribir en el fichero: Utilizamos los mismos métodos que para escribir en pantalla.
- Cerrar el fichero:
  - Siempre hay que cerrar los ficheros después usarlos.
  - El lugar más conveniente es el bloque finally, ya que se ejecuta siempre, tanto si hay error como si no.

Es posible escribir en un fichero indicando que la información se añada a la que ya hay y no se reescriba el fichero. Para ello, creamos el objeto PrintWriter de otra forma: Usando un FileOutputStream. Esta clase permite en su creación un parámetro (llamado append) para indicar si queremos que la información se añada al final del fichero o que el fichero se sobreescriba como en el ejemplo anterior:

El siguiente ejemplo se añaden al fichero numeros1a100.txt los números del 101 al 200

```
public class _02AnyadirTexto101a200 {
   public static void main(String[] args) {
      PrintWriter f = null;
      try {
         //Usamos
         boolean anyadir = true;
         f = new PrintWriter(
                        new FileOutputStream(
                              new File("numeros1a100.txt"), anyadir));
         for(int i = 101; i <= 200; i++){
            f.println(i);
         }

      } catch (FileNotFoundException e) {
         System.out.println("Error de apertura");
      } finally {
         if(f!=null) {
            f.close();
         }
      }
   }
}
```

5.2. Leer un fichero de texto con Scanner.

Si observamos la documentación de las clases FileInputStream y FileOutputStream veremos que las operaciones de lectura y escritura son

muy básicas y permiten únicamente leer o escribir uno o varios bytes. Es decir, son operaciones de muy bajo nivel. Si lo que queremos es escribir información binaria más compleja, como por ejemplo un dato de tipo double o boolean o int, tendríamos que hacerlo a través de un stream que permitiese ese tipo de operaciones y asociarlo al FileOutputStream o FileOutputStream. Podríamos, por ejemplo, asociar un DataInputStream a un FileInputStream para leer del fichero un dato de tipo int.

En ejemplos posteriores se ilustrará cómo asociar un stream a un File... Stream.

5.3. Operaciones con ficheros de acceso secuencial.

Como hemos comentado anteriormente el acceso secuencial a un fichero supone que para acceder a un byte es necesario leer previamente los anteriores. Suele utilizarse este tipo de acceso cuando es necesario leer un archivo de principio a fin.

Vamos a ver una serie de ejemplos que muestren cómo leer y escribir secuencialmente un fichero.

5.4. Lectura de un fichero secuencial de texto.

Leer un fichero de texto y mostrar el número de vocales que contiene.

```
package _03Ejemplos;
import java.io.*;
public class _04ContarVocales {
    final static String VOCALES = "AEIOUaeiou";
    public static void main(String[] args) {
        try (FileReader f = new FileReader(new File("texto.txt"));) {
            int contadorVocales = 0;
            int caracter;
            while((caracter=f.read())!=-1){
                char letra = (char) caracter;
                if(VOCALES.indexOf(letra)!=-1) contadorVocales++;
            }
            System.out.println("Numero de vocales: " + contadorVocales);
        } catch (FileNotFoundException e) {
            System.out.println("Probrema al abrir el fichero");
        } catch (IOException e){
            System.out.println("Problema al leer");

        }
    }
}
```

----- *Ejemplo 3* -----

Observa que:

- Para leer el fichero de texto usamos un InputReader.
- Al crear el stream (InputReader) es posible indicar un objeto de tipo File
- La operación read() devuelve un entero. Para obtener el carácter correspondiente tenemos que hacer una conversión explícita de tipos.

- La operación read() devuelve -1 cuando no queda información que leer del stream.
- La guarda del bucle while combina una asignación con una comparación. En primer lugar se realiza la asignación y luego se compara carácter con -1.
- *FileNotFoundException* sucede cuando el fichero no se puede abrir (no existe, permiso denegado, etc), mientras que *IOException* se lanzará si falla la operación read()

5.5. Escritura de un fichero secuencial de texto.

Dada una cadena escribirla en un fichero en orden inverso

```
package _02Ejemplos;
import java.io.*;
public class _04EscribirCadenaInvertida {
	final static String CADENA = "En un lugar de la mancha...";
	public static void main(String[] args) {
		try (FileWriter f = new FileWriter(new File("texto.txt"));) {
			for(int i=CADENA.length()-1; i>=0; i--){
				f.write(CADENA.charAt(i));
			}
			System.out.println("FIN");
		} catch (FileNotFoundException e) {
			System.out.println("Probrema al abrir el fichero");
		} catch (IOException e){
			System.out.println("Problema al escribir");

		}
	}
}
```

----- Ejemplo 4 -----

Observa que:

- Para escribir el fichero de texto usamos un FileWriter.
- Tal y como se ha creado el stream, el fichero (si ya existe) se sobreescribirá.
- El manejo de excepciones es como el del caso previo.

5.6. Ficheros con buffering.

Cualquier operación que implique acceder a memoria externa es muy costosa, por lo que es interesante intentar reducir al máximo las operaciones de lectura/escritura 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.

En el libro Head First Java, describe los buffers de la siguiente forma: *"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."*

Las clases BufferedReader, BufferedWritter, BufferedInputStream y BufferedOutputStream permiten realizar buffering. Situadas "por delante"

de un stream de fichero acumulan las operaciones de lectura y escritura y cuando hay suficiente información se llevan finalmente al fichero.

En el siguiente codigo se usan buffers para leer líneas de un fichero y escribirlas en otro convertidas a mayúsculas

```
package _02Ejemplos;
import java.io.*;
public class _05DuplicadoEnMayusculas {
	final static String ENTRADA = "texto.txt";
	final static String SALIDA = "textoMayusculas.txt";
	public static void main(String[] args) {
		try (
				BufferedReader fe = new BufferedReader(new FileReader(ENTRADA));
				BufferedWriter fs = new BufferedWriter(new FileWriter(SALIDA))
		){
			String linea;
			while ((linea = fe.readLine())!=null){
				fs.write(linea.toUpperCase());
				fs.newLine();
			}
			System.out.println("FIN");
		} catch (FileNotFoundException e) {
			System.out.println("Probrema al abrir el fichero");
		} catch (IOException e){
			System.out.println("Problema al leer o escribir");
		}
	}
}
```

----- *Ejemplo 5* -----

Observa que:

- Usamos buffers tanto para leer como para escribir. Esto permite minimizar los accesos a disco.
- Los buffers quedan asociados a un FileReader y FileWriter respectivamente. Realizamos las operaciones de lectura/escritura sobre las clases Buffered… y cuando es necesario la clase accede internamente al stream que maneja el fichero.
- Es necesario escribir explícitamente los saltos de línea. Esto se hace mediante el método newLine(). newLine() permite añadir un salto de línea sin preocuparnos de cuál es el carácter de salto de línea. El salto de línea es distinto en distintos sistemas: en unos es /n, en otros /r, en otros /n/r, …
- BufferedReader dispone de un método para leer líneas completas (readLine()). Cuando se llega al final del fichero este método devuelve null.
- Fíjate como el bloque *try with resources* creamos varios objetos. Si la creación de cualquiera de ellos falla, se cerrarán todos los stream que se han abierto.

5.7. Escritura de un fichero binario secuencial.

Ya hemos visto que con FileInputStream y FileOutputStream se puede leer y escribir bytes de información de/a un archivo.

Sin embargo esto puede no ser suficiente cuando la información que tenemos que leer o escribir es más compleja y los bytes se agrupan para representar distintos tipos de datos.

Imaginemos por ejemplo que queremos guardar en un fichero "jugadores.dat", el año de nacimiento y la estatura de cinco jugadores de baloncesto:

```
package _02Ejemplos;
import java.io.*;
public class _06JugadoresBaloncesto {
	public static void main(String[] args) {
		Scanner tec = new Scanner(System.in);
		try (
				DataOutputStream fs = new DataOutputStream(
							new BufferedOutputStream(
								new FileOutputStream("jugadores.dat")));
		){
			for(int i = 1 ; i<=5 ; i++){
				//Pedimos datos al usuario
				System.out.println(" ---- Jugador " + i + " -----");
				System.out.print("Nombre: ");
				String nombre = tec.nextLine();

				System.out.print("Nacimiento: ");
				int anyo= tec.nextInt();

				System.out.print("Estatura: ");
				double est= tec.nextDouble();
				//Vaciar salto linea
				tec.nextLine();

				//Volcamos información al fichero
				fs.writeUTF(nombre);
				fs.writeInt(anyo);
				fs.writeDouble(est);
			}
		} catch (FileNotFoundException e) {
			System.out.println("Probrema al abrir el fichero");
		} catch (IOException e){
			System.out.println("Problema al leer o escribir");
		}
	}}
```

----- *Ejemplo 6* -----

Observa que:

- Para escribir información binaria usamos un DataInputStream asociado al stream. La clase tiene métodos para escribir int, byte, double, boolean, etc, etc.
- Además, como hemos hecho en ejemplos previos, usamos un buffer. Fíjate como en el constructor se enlazan unas clases con otras.
- A pesar de que en Java los ficheros son secuencias de bytes, estamos dotando al fichero de cierta estructura: primero aparece el nombre, luego el año y finalmente la estatura. Cada uno de estos tres datos constituirían un registro de formado por tres campos. Para poder recuperar información de un fichero

binario es necesario conocer cómo se estructura ésta dentro del fichero.

5.8. Lectura de un fichero binario secuencial.

```
package _02Ejemplos;
import java.io.*;
public class _07LeerNombresJugadores {
    public static void main(String[] args) {
        Scanner tec = new Scanner(System.in);
        try (
                DataInputStream fe = new DataInputStream(
                                        new BufferedInputStream(
                                                new FileInputStream("jugadores.dat")));
        ){
            while(true){
                //Leemos nombre
                System.out.println(fe.readUTF());

                //leemos y desechamos resto de datos
                fe.readInt();
                fe.readDouble();
            }
        } catch(EOFException e){
            //Se lanzará cuando se llegue al final del fichero
        } catch (FileNotFoundException e) {
            System.out.println("Probrema al abrir el fichero");
        } catch (IOException e){
            System.out.println("Problema al leer o escribir");
        }
    }
}
```

----- Ejemplo 7 -----

Observa que:

- A pesar de que necesitamos solamente el nombre de cada jugador, es necesario leer también el año y la estatura. No es posible acceder al nombre del primer jugador sin leer previamente todos los datos del primer jugador.
- La lectura se hace a través de un bucle infinito (while (true)), que finalizará cuando se llegue el final del fichero y al leer de nuevo se produzca la excepción EOFException

6. Serialización

7. Manejo de ficheros y carpetas

7.1. La clase File.

La clase File es una representación abstracta ficheros y carpetas. Cuando creamos en Java un objeto de la clase File en representación de un fichero o carpeta concretos, no creamos el fichero al que se representa. Es decir, el objeto File representa al archivo o carpeta de disco, pero no es el archivo o carpeta de disco.

La clase File dispone de métodos que permiten realizar determinadas operaciones sobre los ficheros. Podríamos, por ejemplo, crear un objeto de tipo File que represente a *c:\datos\libros.txt* y, a través de ese objeto File, realizar consultas relativas al fichero libros.txt, como su tamaño, atributos, etc, o realizar operaciones sobre él: borrarlo, renombrarlo,, ...

7.2. Constructores:

La clase File tiene varios constructores, que permiten referirse, de varias formas, al archivo que queremos representar:

public File (String ruta)	Crea el objeto File a partir de la ruta indicada. Si se trata de un archivo tendrá que indicar la ruta y el nombre.
public File (String ruta, String nombre)	Permite indicar de forma separada la ruta del archivo y su nombre
public File (File ruta, String nombre)	Permite indicar de forma separada la ruta del archivo y su nombre. En este caso la ruta está representada por otro objeto File.
public File (URI uri)	Crea el objeto File a partir de un objeto URI (UniformResourceIdentifier). Un URI permite representar un elemento siguiendo una sintaxis concreta, un estándar.

7.3. Métodos.

Aquí exponemos algunos métodos interesantes. Hay otros que puedes consultar en la documentación de Java

Relacionados con el nombre del fichero	
String getName()	Devuelve el nombre del fichero o directorio al que representa el objeto. (Solo el nombre, sin la ruta)
String getPath()	Devuelve la ruta del fichero o directorio. La ruta obtenida es dependiente del sistema, es decir, contendrá el carácter de separación de directorios que esté establecido por defecto. Este separador está definido en public static final String separator
String getAbsolutePath()	Devuelve la ruta absoluta del fichero o directorio.
String getParent()	Devuelve la ruta del directorio en que se encuentra el fichero o directorio representado. Devuelve null si no hay directorio padre.

Para hacer comprobaciones	
boolean exists() Boolean canWrite() Boolean canRead() Boolean isFile() Boolean isDirectory()	Permiten averiguar, respectivamente, si el fichero existe, si se puede escribir en el, si se puede leer de él, si se trata de un fichero o si se trata de un directorio
Obtener información de un fichero	
long length	Devuelve el tamaño en bytes del archivo. El resultado es indefinido si se consulta sobre un directorio o una unidad.
long lastModified	Devuelve la fecha de la última modificación del archivo. Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970
Para trabajar con directorios	
Boolean mkdir()	Crea el directorio al cual representa el objeto File.
Boolean mkdirs()	Crea el directorio al cual representa el objeto File, incluyendo todos aquellos que sean necesarios y no existan.
String[] list()	Devuelve un array de Strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File.
String[] list(FileNameFilter filtro)	Devuelve un array de Strings con los nombres de los ficheros y directorios que contiene el directorio al que representa el objeto File y que cumplen con determinado filtro.
public File[] listFiles()	Devuelve un array de objetos File que representan a los archivos y carpetas contenidos en el directorio al que se refiere el objeto File.
Para hacer cambios	
Boolean renameTo(File nuevoNombre)	Permite renombrar un archivo. Hay que tener en cuenta que la operación puede fracasar por muchas razones, y que será dependiente del sistema: Que no se pueda mover el fichero de un lugar a otro, que ya exista un fichero que coincide con el nuevo, etc. El método devuelve true solo si la operación se ha realizado con éxito. Existe un método move en la clase Files para mover archivos de una forma independiente del sistema.
Boolean delete()	Elimina el archivo o la carpeta a la que representa el objeto File. Si se trata de una carpeta tendrá que estar vacía. Devuelve true si la operación tiene éxito.
Boolean createNewFile()	Crea un archivo vacío. Devuelve true si la operación se realiza con éxito.
File createTempFile(String prefijo, String sufijo)	Crea un archivo vacío en la carpeta de archivos temporales. El nombre llevará el prefijo y sufijo indicados. Devuelve el objeto File que representa al nuevo archivo.

Ejemplo: Mostrar información y contenido de una carpeta

```
package _02Ejemplos;
import java.io.*;
importjava.util.*;

publicclass _01InformacionCarpeta {
	publicstaticvoid main(String[] args) {

		Scanner tec = new Scanner(System.in);
		System.out.println("Introduce ruta absoluta de una carpeta");
		StringnombreCarpeta = tec.nextLine();
		//Creamosobjeto File pararepresentar a lacarpeta
		File car = new File(nombreCarpeta);
		//Comprobamossiexiste
		if (car.exists()){
			//¿Esunacarpeta?
			if(car.isDirectory()){
				if(car.canRead()) System.out.println("Lecturapermitida");
				else System.out.println("Lectura no permitida");

				if(car.canWrite()) System.out.println("Escriturapermitida");
				else System.out.println("Escritura no permitida");

				if(car.isHidden()) System.out.println("Carpetaoculta");
				else System.out.println("Carpeta visible");

				System.out.println("---- Contenido de la carpeta ----");
				File[] contenido = car.listFiles();
				for(File f: contenido){
					System.out.println(f.getName());
				}
			}
			else System.out.println(car.getAbsolutePath() + "No esunacarpeta" );
		} else System.out.println("No existe la carpeta" + car.getAbsolutePath());
	}
}
```