

Lectura y escritura de información

1. Introducción.

Los ficheros pueden ser de texto (que contienen líneas de texto representadas en código ASCII), binarios, de datos, de aplicaciones o multimedia.

Las operaciones más comunes que se pueden realizar sobre ficheros son: crear, borrar, abrir, cerrar, leer y escribir.

Cuando tratamos este tipo de operaciones y utilizamos estructuras de tipo stream para realizar lecturas y escrituras, los pasos habituales son el ir apoyándonos en clases proporcionadas Java para este propósito.

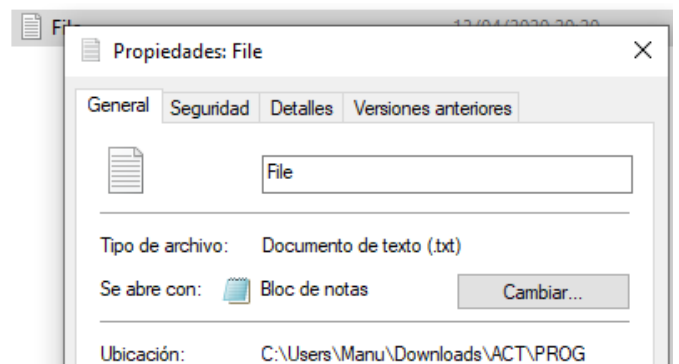
En primer lugar, veremos cómo podemos leer un fichero y su lectura/escrito como cadena de caracteres. Y posteriormente, veremos como guardar el estado actual de un objeto convirtiéndolo en un array de bytes que podremos almacenar en un fichero.

2. SISTEMA DE FICHEROS:

Dependiendo del entorno de trabajo, en la definición de un directorio o un archivo habrá que indicar de forma apropiada la estructura de directorios:

Windows: "C:\base\trabajo".
Linux: "/home/user/base/trabajo".

```
steve@gary-Lenovo-ideapad-Y700-15ISK:~$ pwd  
/home/accounts/steve
```



3. CLASE File:

La clase File permite recuperar información acerca de un archivo o directorio. Los objetos de la clase File no abren archivos ni proporcionan herramientas para procesar archivos.

Nos proporciona algunos métodos que pueden ser útiles de cara a analizar un fichero o directorio de destino/origen.

```

// Define fichero.
File file = new File("C:\\Users\\Manu\\Downloads\\ACT\\PROG\\File.txt");
// Define directorio.
File dir = new File("C:\\Users\\Manu\\Downloads\\ACT\\PROG");

// METODOS:
// Listar: Devuelve array con todos los elementos del directorio.
System.out.println(dir.list());

// Existe? Permite saber si un fichero o directorio existe.
System.out.println(file.exists());
System.out.println(dir.exists());

// Saber si estamos ante un fichero o directorio.
System.out.println(dir.isFile());
System.out.println(dir.isDirectory());

// Obtener ruta absoluta
System.out.println(file.getAbsolutePath());

// Retorna tamaño del fichero o directorio.
System.out.println(file.length());
System.out.println(dir.length());

```

3.1. CLASE FileReader Y FileWriter

Las clases `FileReader` y `FileWriter` permiten leer y escribir, respectivamente, en un fichero.

```

import java.io.*;
//Importamos todas las clases de java.io.<br />public class FicheroTextoApp {
    public static void main(String[] args) {
        try{
            //Abro stream, crea el fichero si no existe
            FileWriter fw=new FileWriter("D:\\fichero1.txt");
            //Escribimos en el fichero un String y un caracter 97 (a)
            fw.write("Esto es una prueb");
            fw.write(97);
            //Cierro el stream
            fw.close();
            //Abro el stream, el fichero debe existir
            FileReader fr=new FileReader("D:\\fichero1.txt");
            //Leemos el fichero y lo mostramos por pantalla
            int valor=fr.read();
            while(valor!=-1){
                System.out.print((char)valor);
                valor=fr.read();
            }
            //Cerramos el stream
            fr.close();
        }catch(IOException e){
            System.out.println("Error E/S: "+e);
        }
    }
}

```

4. CLASE BufferedReader & BufferedWriter

Hay diversas implementaciones para leer/escribir en un fichero. En el siguiente enlace se encuentra hasta 5 opciones posibles: <https://stackabuse.com/reading-and-writing-files-in-java/>

1. Usando las clases `FileReader` y `FileWriter`.
2. Usando las clases `BufferedReader` y `BufferedWriter`.
3. Usando las clases `FileInputStream` y `FileOutputStream`.
4. Usando las clases `BufferedInputStream` y `BufferedOutputStream`.
5. Usando las clases `java.nio`.

Entre estas opciones vamos a destacar una de las opciones más aconsejables que es usando las clases `BufferedReader` y `BufferedWriter`.

Su implementación es la siguiente:

0. NECESITAMOS LA RUTA (PATH) DEL FICHERO A LEER/ESCRIBIR.

```
String directory = System.getProperty("user.home");
String fileName = "sample.txt";
String absolutePath = directory + File.separator + fileName;
```

2. ESCRIBIR

```
// write the content in file
try(BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(absolutePath))) {
    String fileContent = "This is a sample text.";
    bufferedWriter.write(fileContent);
} catch (IOException e) {
    // exception handling
}
```

3. LEER

```
// read the content from file
try(BufferedReader bufferedReader = new BufferedReader(new FileReader(absolutePath))) {
    String line = bufferedReader.readLine();
    while(line != null) {
        System.out.println(line);
        line = bufferedReader.readLine();
    }
} catch (FileNotFoundException e) {
    // exception handling
} catch (IOException e) {
    // exception handling
}
```

La mayor ventaja de los buffered es con el `BufferedReader` que nos permite leer una línea completa, en lugar de carácter a carácter como hacia `FileReader`. Finalmente, cuando el fichero termina este devuelve `null`, no un `-1` como en `FileReader`.

* EXTRA: AYUDA CONSTRUIR LA RUTA, INDEPENDIENTEMENTE DEL EQUIPO.

```
System.out.println(System.getProperty("user.home"));
```

C:\Users\Manu

Siendo Manu el nombre del equipo, por lo tanto, como el nombre del equipo es diferente para cada uno, una forma de obtener y salvar ese inconveniente es como este pequeño “truco”.

5. SERIALIZACIÓN.

Hasta ahora hemos leído y/o escrito caracteres ASCII de un fichero.

Ahora vamos a aplicar la serialización, que consiste en convertir un objeto en un conjunto de bytes, facilitando su almacenamiento y/o transmisión.

Uso:

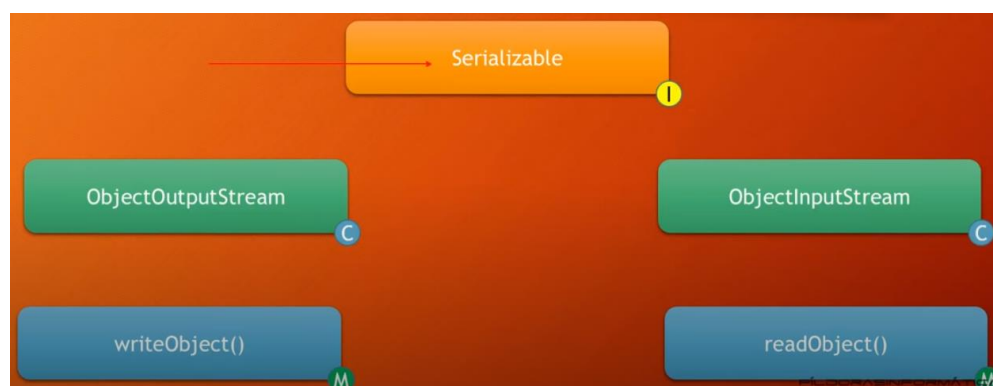
- Poder almacenar ese objeto en un medio de almacenamiento.
- Restaurar o recomponer ese objeto, en el estado en que se dejó previa a la serialización.

De este modo se obtiene un formato manejable para el almacenamiento del estado actual de un objeto (con los valores de los atributos en un instante determinado y los métodos codificados en la misma).

5.1. ¿Qué clases son necesarias?

Para manejar esta prestación Java proporciona la interfaz “**Serializable**” y una serie de clases y métodos para su utilización para este propósito.

Estructura jerárquica:



Características:

1. La clase a serializar debe implementar obligatoriamente la interfaz Serializable:
<https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>
2. Todos los objetos creados de dicha clase que implementa la clase Serializable son susceptible de ser convertidos en bytes para transferirlos a través de la red o simplemente guardarlo en un fichero en el disco duro.

Además de implementar “Serializable” vamos a tener que manejar dos clases:

1. **ObjectOutputStream**
(<https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html>).
Construye un flujo de datos a través de los cuales es posible transferir el estado actual de objeto desde nuestro programa hacia fuera (otro equipo, un fichero...).
- a. Método **writeObject()**. Crear esa sucesión de bytes correspondientes al estado de un objeto.
2. **ObjectInputStream**
(<https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html>).
Crea un flujo de datos a través del cual es posible interpretar un objeto desde fuera hacia dentro y dentro del programa recomponer esos bytes en el objeto deseado a través de casting de objetos.
- a. Método **readObject()**. Leer e interpretar esa sucesión de bytes.

5.2. EJEMPLO PRÁCTICO:

0. PREPARAR ENTORNO DE PRUEBA.

0.1. CREAMOS UNA CLASE

```
public class CuentaBancaria {  
  
    private String nombre;  
    private String DNI;  
    private int cuantia;  
  
    public CuentaBancaria(String nombre, String dni, int cuantia) {  
        this.nombre = nombre;  
        DNI = dni;  
        this.cuantia = cuantia;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getDNI() {  
        return DNI;  
    }  
  
    public void setDNI(String dni) {  
        DNI = dni;  
    }  
  
    public int getCuantia() {  
        return cuantia;  
    }  
  
    public void setCuantia(int cuantia) {  
        this.cuantia = cuantia;  
    }  
  
    @Override  
    public String toString() {  
        return "CuentaBancaria [nombre=" + nombre + ", DNI=" + DNI + ", cuantia=" + cuantia + "];"  
    }  
}
```

0.2. CREAMOS UN OBJETO DE LA CLASE

```
public static void main(String[] args) {  
  
    CuentaBancaria cuenta1 = new CuentaBancaria("Julia", "12345677U", 5000);  
    CuentaBancaria cuenta2 = new CuentaBancaria("Pepe", "12345678X", 2000);  
  
    CuentaBancaria[] banco = new CuentaBancaria[2];  
    banco[0] = cuenta1;  
    banco[1] = cuenta2;  
  
}
```

0.3. ADECUAMOS LA CLASE PARA QUE ESTA SEA SERIALIZABLE.

```
import java.io.Serializable;  
  
public class CuentaBancaria implements Serializable{  
  
    private String nombre;  
    private String DNI;  
    private int cuantia;  
  
}
```

1 CONVERTIR UN OBJETO EN UNA SUCESIÓN DE BYTES Y GUARDARLO EN UN FICHERO .DAT – UTILIZAMOS ObjectOutputStream

```
public static void main(String[] args) {  
  
    CuentaBancaria cuenta1 = new CuentaBancaria("Julia", "12345677U", 5000);  
    CuentaBancaria cuenta2 = new CuentaBancaria("Pepe", "12345678X", 2000);  
  
    CuentaBancaria[] banco = new CuentaBancaria[2];  
    banco[0] = cuenta1;  
    banco[1] = cuenta2;  
  
    try {  
        ObjectOutputStream escribiendoFichero = new ObjectOutputStream(  
            new FileOutputStream("C:\\Users\\Manu\\Downloads\\ACT\\PROG\\file.dat"));  
  
        escribiendoFichero.writeObject(banco);  
        escribiendoFichero.close();  
    } catch (IOException e) {  
        // TODO: handle exception  
    } catch (Exception e) {  
        // TODO: handle exception  
    }  
}
```

Fijarse como la clase ObjectOutputStream lanza una serie de excepciones, una de ellas es obligatorio capturarla (IOException). La capturamos y luego usamos Exception para el resto.


Throws:

IOException - if an I/O error occurs while writing stream header

SecurityException - if untrusted subclass illegally overrides security-sensitive methods

NullPointerException - if out is null

RESULTADO:

Nombre	Fecha de modificación	Tipo	Tamaño
 file.dat	13/04/2020 21:08	Archivo DAT	1 KB

2. INTRODUCIREMOS EN NUESTRO ALGORITMO EL ESTADO ACTUAL DEL OBJETO REGISTRADO EN EL FICHERO .DAT– UTILIZAMOS ObjectInputStream

```
try {  
    // Creamos un objeto de la clase ObjectInputStream apuntando a nuestro fichero .dat  
    ObjectInputStream leyendoFichero = new ObjectInputStream(  
        new FileInputStream("C:\\Users\\Manu\\Downloads\\ACT\\PROG\\file.dat"));  
  
    // Leemos su contenido extrayendo un Object. Este lo convertimos a la clase correspondiente.  
    CuentaBancaria[] objBancoRecuperado = (CuentaBancaria[])leyendoFichero.readObject();  
    // Cerramos el flujo de lectura.  
    leyendoFichero.close();  
  
    // Ya tendemos objBancoRecuperado con los datos del fichero .dat.  
    // Vamos a recorrerlo y mostrar sus datos.  
    for (CuentaBancaria cuentaBancaria : objBancoRecuperado) {  
        System.out.println(cuentaBancaria);  
    }  
  
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
} catch (ClassNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```