UNIDAD 10: LECTURA Y ESCRITURA DE FICHEROS

Profesor: José Ramón Simó Martínez

Contenido

1. Introducción 2

2. Breve repaso al concepto de ficheros 3

2.1. Gestión de ficheros en Java 3

2.2. Jerarquía del paquete *java.io* 3

3. Utilización del sistema de ficheros 5

3.1. Tabla de operaciones sobre ficheros 6

4. Ficheros de texto 7

4.1. Lectura 7

4.2. Escritura 9

5. Ficheros binarios 11

5.1. Lectura/Escritura 11

5.2. Ficheros de acceso aleatorio 14

6. Persistencia de objetos en Java 15

6.1. Guardar objetos serializables (Serialización) 15

6.2. Recuperar objetos serializados (Deserialización) 16

7. Bibliografía 19

CC BY NC SA

1. Introducción

En los programas que hemos desarrollado hasta el momento, los datos de entrada los introduce el usuario y los datos de salida los imprimimos en la consola del sistema. De cualquier forma, toda la información se pierde una vez cerramos nuestro programa. Aplicaciones como procesadores de texto, sistemas de bases de datos, aplicaciones web o videojuegos, requieren trabajar con datos almacenados de forma permanente.

Los principales componentes que permiten almacenar y recuperar datos en el sistema son:

- Los ficheros
- Las bases de datos

En esta unidad introduciremos la gestión de la información a través de ficheros donde aprenderemos las diferentes operaciones que se pueden realizar con este componente: leer y escribir datos en un fichero, copiar, borrar, mover y renombrar ficheros, etc. En Java los ficheros son tratados como objetos, lo que significa que se pueden utilizar métodos para realizar operaciones en ellos; estudiaremos diferentes las diferentes clases que del paquete *java.io* como *File*, *FileReader*, *FileWriter*, etc.

En próximas unidades estudiaremos la gestión de la información permanente de nuestro programa a través de una base de datos.

Al terminar esta unidad deberás ser capaz de:

- Identificar las clases fundamentales del paquete java.io para gestionar la entrada y salida de información.
- Reconocer los tipos de ficheros que podemos utilizar para entrada y salida de información.
- Utilizar ficheros para almacenar y recuperar información.
- Conocer y utilizar la persistencia de los objetos en Java.
- Desarrollar programas que permitan almacenar y recuperar información a través de ficheros.

Nota

El tratamiento de excepciones es fundamental en la gestión de datos en ficheros. Por tanto, antes de empezar la presente unidad, se recomienda repasar los conceptos del tratamiento de excepciones introducidos en la unidad anterior.

CC BY NC SA

2. Breve repaso al concepto de ficheros

Los ficheros los podemos en dos tipos:

- Ficheros de texto
- Ficheros binarios

Los *ficheros de texto* los podemos procesar (crear, leer o escribir) usando un editor de textos como el bloc de notas en Windows o el nano en Ubuntu. Todos los demás tipos de ficheros los consideraremos *ficheros binarios*; no podemos leer ficheros binarios con un editor de textos.

Un ejemplo sería el fichero de código fuente de Java (*.java*) que lo podemos editar y leer con un editor de textos; sin embargo, el fichero resultado de su compilación (*.class*) es un fichero binario que sólo puede ser leído y entendido por la máquina virtual de Java (JVM).

Asimismo, y de forma informal, podemos considerar a los ficheros de texto como una secuencia de caracteres y a los ficheros binarios como una secuencia de bits. Los caracteres están codificados utilizando una esquema de codificación, como por ejemplo, ASCII o Unicode.

Por ejemplo, el entero decimal 199 se guarda como una secuencia de tres caracteres (1, 9, 9) en un fichero de texto. Ahora bien, el mismo entero se guarda como valor de byte C7 en un fichero binario, ya que el 199 es igual a C7 en hexadecimal. La ventaja de los ficheros binarios es que son más eficientes de procesar que los ficheros de texto.

2.1. Gestión de ficheros en Java

El lenguaje Java proporciona una gran variedad de clases para gestionar la entrada y salida de datos en ficheros. Estas clases se pueden clasificar según el tipo de ficheros que se vaya a tratar, que son dos:

- Ficheros de texto
- Ficheros binarios

En el siguiente subapartado se presenta la jerarquía de clases para la gestión de ficheros en Java.

2.2. Jerarquía del paquete *java.io*

Es importante tener una visión global de las clases de Java implicadas en la entrada/salida de datos y sus relaciones. A continuación, el esquema de la jerarquía del paquete *java.io* (en *Java 17*):

BY NC SA

Hierarchy For Package java.io

Package Hierarchies:
All Packages

Class Hierarchy

- java.lang.**Object**
  - java.io.**Console** (implements java.io.Flushable)
  - java.io.**File** (implements java.lang.Comparable<T>, java.io.Serializable)
  - java.io.**FileDescriptor**
  - java.io.**InputStream** (implements java.io.Closeable)
    - java.io.**ByteArrayInputStream**
    - java.io.**FileInputStream**
    - java.io.**FilterInputStream**
      - java.io.**BufferedInputStream**
      - java.io.**DataInputStream** (implements java.io.DataInput)
      - java.io.**LineNumberInputStream**
      - java.io.**PushbackInputStream**
    - java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
    - java.io.**PipedInputStream**
    - java.io.**SequenceInputStream**
    - java.io.**StringBufferInputStream**
  - java.io.**ObjectInputFilter.Config**
  - java.io.**ObjectInputStream.GetField**
  - java.io.**ObjectOutputStream.PutField**
  - java.io.**ObjectStreamClass** (implements java.io.Serializable)
  - java.io.**ObjectStreamField** (implements java.lang.Comparable<T>)
  - java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)
    - java.io.**ByteArrayOutputStream**
    - java.io.**FileOutputStream**
    - java.io.**FilterOutputStream**
      - java.io.**BufferedOutputStream**
      - java.io.**DataOutputStream** (implements java.io.DataOutput)
      - java.io.**PrintStream** (implements java.lang.Appendable, java.io.Closeable)
    - java.io.**ObjectOutputStream** (implements java.io.ObjectOutput, java.io.ObjectStreamConstants)
    - java.io.**PipedOutputStream**
  - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - java.security.**BasicPermission** (implements java.io.Serializable)
      - java.io.**SerializablePermission**
    - java.io.**FilePermission** (implements java.io.Serializable)
  - java.io.**RandomAccessFile** (implements java.io.Closeable, java.io.DataInput, java.io.DataOutput)
  - java.io.**Reader** (implements java.io.Closeable, java.lang.Readable)
    - java.io.**BufferedReader**
      - java.io.**LineNumberReader**
    - java.io.**CharArrayReader**
    - java.io.**FilterReader**
      - java.io.**PushbackReader**
    - java.io.**InputStreamReader**
      - java.io.**FileReader**
    - java.io.**PipedReader**
    - java.io.**StringReader**
  - java.io.**StreamTokenizer**

Fuente: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/package-tree.html

CC BY NC SA

3. Utilización del sistema de ficheros

En este apartado haremos una primera aproximación al tratamiento de ficheros. Para ello, Java nos proporciona la clase File que forma parte del paquete java.io. En definitiva, la clase File nos ofrece la posibilidad de trabajar con ficheros y directorios del sistema de ficheros. A continuación, un ejemplo para empezar a crear ficheros en el sistema desde Java:

Ejemplo: Creación un fichero

Primero debemos instanciar un objeto de la clase File indicando la ruta (*path*) y nombre del fichero. Este objeto representa el nombre de la ruta dada:

```
File fichero = new File("miFichero.txt");
```

Luego, deberos usar el método *createNewFile()*:

```
fichero.createNewFile();
```

En este ejemplo, habrá creado el fichero el directorio raíz del proyecto (**ruta relativa**). También podemos indicar la ruta completa del destino del fichero (**ruta absoluta**):

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

Siempre que los directorios de la ruta anterior existan previamente.

En el ejemplo completo, hay que tener en cuenta el tratamiento de excepciones ya que el método *createNewFile()*, según la documentación de Java (ver aquí) , lanza una excepción de tipo *IOException*:

```
import java.io.File;
import java.io.IOException;

public class Test {
	public static void main(String[] args) {
		File fichero = new File("miFichero.txt");
		try {
			fichero.createNewFile();
		}
		catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Nota

Para repasar el concepto de ruta absoluta y relativa en informática tenéis el siguiente enlace:

https://es.wikipedia.org/wiki/Ruta_(inform%C3%A1tica)

CC BY NC SA

3.1. Tabla de operaciones sobre ficheros

A continuación, un resumen de las operaciones que se pueden realizar sobre ficheros con la clase *File*:

Método	**Descripción**	**Salida**
boolean createNewFile()	crea el fichero indicado en la ruta.	*false* si el fichero ya existía.
boolean mkdir()	crea el directorio indicado en la ruta.	*false* si el directorio ya existía.
boolean delete()	borra fichero o directorio indicado en la ruta	*true* si ha sido borrado con éxito.
String getParent()	devuelve la ruta hasta la carpeta del elemento referido por esa ruta.	La cadena de la ruta.
String getName()	devuelve el nombre del elemento que representa la ruta.	Nombre del elemento.
String getAbsolutePath()	devuelve el nombre de la ruta absoluta.	La cadena de la ruta.
boolean exists()	comprueba si la ruta existe en el sistema de ficheros.	*true* si existe.
boolean isFile()	comprueba si existe y es un fichero.	*true* si existe.
boolean isDirectory()	comprueba si existe y es un directorio.	*true* si existe.
long lenght()	devuelve el tamaño de un archivo en bytes.	Tamaño del archivo en bytes.
long lastModified()	devuelve la última fecha de edición del elemento.	Milisegundos que han pasado desde el 1 de junio de 1970.
String[] list()	lista los ficheros y directorios de la ruta indicada.	array de cadenas.
File[] listFiles()	lista todos los elementos contenidos del directorio indicado.	array de objetos tipo File

Más información sobre la API de la clase *File*:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html

CC BY NC SA

4. Ficheros de texto

En este apartado aprenderemos a leer y escribir ficheros de texto a través de las clases y métodos del paquete *java.io*.

4.1. Lectura

Para leer un fichero de texto necesitaremos combinar tres clases del paquete *java.io*:

- File
- BufferedReader
- FileReader

Ejemplo: Lectura caracter a caracter

```
public class LectorDeCaracteres {
    public static void main(String[] args) {
        BufferedReader br = null;
        FileReader fr = null;

        File fichero = new File("recursos\\datos_entrada.txt");

        try {
            // crea el fichero para lectura
            fr = new FileReader(fichero);

            // crea el buffer para optimizar la lectura
            br = new BufferedReader(fr);

            // mientras haya caracteres para leer...
            int caracter;
            while ((caracter = br.read()) != -1) {
                System.out.print((char)caracter);
            }
        } catch (FileNotFoundException e) {
            System.out.println("No se ha podido encontrar el fichero.");
        } catch (IOException e) {
            System.out.println("No se ha podido leer el fichero.");
        } finally {
            try {
                if (br != null)
                    br.close();
            } catch (IOException e) {
                    System.out.println("No se ha podido cerrar el fichero.");
            }
        }
    }
}
```

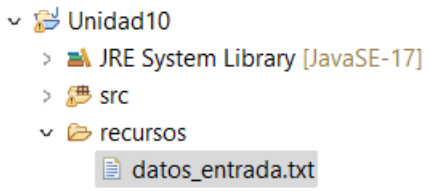
CC BY NC SA

Analicemos las partes más destacadas del ejemplo anterior:

- Creamos previamente un fichero de texto llamado “datos_entrada.txt” con el contenido siguiente:

```
cabecera fichero
línea 1
línea 2
```

  Este fichero estará guardado en la carpeta “recursos” del proyecto (se debe crear):



- Los objetos File, *BufferedReader* y *FileReader* los iniciamos a *null* ya que posteriormente necesitaremos acceder a ellos en el bloque *finally*.
- La clase *FileReader* prepara para lectura el fichero indicado en la ruta del objeto *File*; en *Windows* la ruta se marca con doble barra invertida “\\” para escapar la barra simple “\”. Con esta clase ya podríamos leer los datos del fichero; no obstante, tal y como indica la documentación oficial es recomendable crear un *buffer* de con los datos para mayor eficiencia, haciendo uso de *BufferedReader*.
- Para leer caracter a caracter utilizamos el método *read()* de la clase *BufferedReader*. Cada vez que llamamos a este método, leerá un caracter del buffer y devolverá un entero con el código *ASCII* de dicho caracter; en caso de no quedar caracteres en el buffer, devolverá el valor -1.
- El valor entero devuelto por *read()* debe ser casteado a tipo *char* para mostrar el caracter *ASCII* correspondiente.
- Atención a las excepciones:
  - El constructor de *FileReader()* lanza una excepción *FileNotFoundException* la cual es *checked*; por tanto debe ser tratada obligatoriamente en tiempo de compilación.
  - El método *read()* lanza una excepción *IOException* y también es *checked*.
  - El bloque *finally* es típicamente usado para cerrar recursos. En este caso, será suficiente con cerrar el *BufferedReader* ya que automáticamente cerrará los demás recursos. Aquí también trataremos otra excepción ya que el método *close()* lanza una excepción *IOException*.

La salida por pantalla del código anterior sería:

```
cabecera fichero
línea 1
línea 2
```

Nota

Los saltos de línea también son caracteres por lo que, aunque en el código sólo utilizamos *print()*, en la salida se han aplicado los saltos de línea.

CC BY NC SA

El proceso de apertura y cierre de fichero es el mismo que en el ejemplo anterior. Lo único que cambio es el modo en que vamos a leer los datos; para ello, utilizaremos el método *readLine()* de la clase *BufferedReader*.

Así que solamente deberemos cambiar el bloque de lectura del fichero (el bucle) por el siguiente bloque:

```
String linea;
while ((linea = br.readLine()) != null) {
    System.out.println(linea);
}
```

> **Nota**
>
> El método *readLine()* devuelve la línea de texto leída mientras queden líneas de texto por leer; en caso contrario devolverá *null*. Además, este método no tiene en cuenta los saltos de línea, por ello utilizamos *println()* en vez de *print()*.

4.2. Escritura

Para escribir en un fichero de texto necesitaremos combinar tres clases del paquete *java.io*:

- File
- FileWriter
- PrintWriter

Ejemplo: Escritura en ficheros sin buffer

```
public class EscritorSinBuffer {
    public static void main(String[] args) {
        PrintWriter pw = null;
        FileWriter fw = null;

        File fichero = new File("recursos\\datos_salida.txt");

        try {
            // crea el fichero para escritura
            fw = new FileWriter(fichero);
            // crea el PrintWriter para usar los métodos printXXX
            pw = new PrintWriter(fw);
            // escribe la cadena en el fichero
            pw.println("cadena1");
            // escribe un entero
            pw.println(3);
        } catch (IOException e) {
            System.out.println("No se ha podido escribir en el fichero.");
        } finally {
            pw.close(); // no lanza excepción, no requiere tratarla
        }
}}
```

Analicemos las partes más destacadas del ejemplo anterior:

- La clase *FileWriter* prepara para escritura el fichero indicado en la ruta del objeto *File*. En este caso, por defecto si no existe el fichero lo crea; si existe, borra el contenido del fichero. Para que esto último no suceda, es decir, para que al abrir el fichero adjunte la información nueva, deberemos llamar al constructor con un segundo parámetro booleano con el valor a *true*:

```
fw = new FileWriter(fichero, true);
```

- La clase *PrintWriter* se utiliza principalmente para poder usar sus métodos *print*, *println* o *printf*, ya que facilitan el formateo de los datos. Al igual que hacíamos en *System.out.printXXX(…)* a dichos métodos podemos pasarles como parámetros diferentes tipos de datos (*String*, *int*, *double*, etc). También podemos concatenar cadena de datos con el símbolo "+":

```
pw.println("cadena1" + "cadena2" + numero);
```

- Sólo necesitamos cerrar el recurso de *PrintWriter* y este ya se encargará de cerrar los demás recursos (en este caso los de *FileWriter*).
- En cuanto al tratamiento de excepciones, sólo la clase *FileWriter* lanza una excepción *IOExcepcion*. Por otra parte, el método *close()* de *PrintWriter* no lanza ninguna excepción y por tanto no hace falta que se trate en el bloque *finally*.

El uso de *PrintWriter* tiene la ventaja de hacer que la escritura de datos en el fichero sea muy cómoda para el programador. Sin embargo, arrastra dos inconvenientes:

- Oculta las excepciones que puedan ocurrir durante la escritura de datos; los métodos *printXXX* no lanzan ninguna excepción.
- Es menos eficiente que utilizar un *buffer*.

Para hacer escritura segura y con uso de buffer se utilizará la clase *BufferedWriter*. Para saber más sobre esta clase:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedWriter.html

BY NC SA

5. Ficheros binarios

En este apartado aprenderemos a leer y escribir ficheros binarios a través de las clases y métodos del paquete *java.io*. Además, veremos el acceso aleatorio (al contrario que secuencial) a los datos de un fichero.

En este apartado veremos un ejemplo conjunto de lectura y escritura en un fichero binario para simplificar el concepto y asegurarnos que trabajos con un fichero binario. Primero realizaremos la escritura de datos en binario para luego leernos también en formato binario. Para ello usaremos la extensión de fichero binario *.dat*

5.1. Lectura/Escritura

Para leer un fichero de texto necesitaremos combinar tres clases del paquete *java.io*:

- File
- FileInputStream
- FileOutputStream

Ejemplo: Escritura y lectura de un fichero binario

```
public class LectorDeBytesSinBuffer {
    public static void main(String[] args) {
        FileInputStream fis = null;
        FileOutputStream fos = null;

        File fichero = new File("recursos\\datos_binarios.dat");

        // Escribimos los datos en un fichero binario
        try {
            // crea el fichero para escritura
            fos = new FileOutputStream(fichero);

            // escribimos los datos en formato binario
            for(int i = 0; i < 10; i++) {
                fos.write(i);
            }
        } catch (FileNotFoundException e) {
            System.out.println("No se ha podido encontrar el fichero.");
        } catch (IOException e) {
            System.out.println("No se ha podido leer el fichero.");
        } finally {
            try {
                if (fos != null)
                    fos.close();
            } catch (IOException e) {
                System.out.println("No se ha podido cerrar el fichero.");
            }
        }
        // Hemos terminado de escribir los datos y cerramos el fichero de escritura
```

CC BY NC SA

```
        // … continúa

        // Leemos los datos del fichero binario creado anteriormente
        try {

            // crea el fichero para lectura
            fis = new FileInputStream(fichero);
            // leemos los datos en formato binario
            int dato = 0;
            while((dato = fis.read()) != -1) {
                System.out.print(dato + " ");
            }
        } catch (FileNotFoundException e) {
            System.out.println("No se ha podido encontrar el fichero.");
        } catch (IOException e) {
            System.out.println("No se ha podido leer el fichero.");
        } finally {
            try {
                if (fis != null)
                    fis.close();
            } catch (IOException e) {
                System.out.println("No se ha podido cerrar el fichero.");
            }
        }
    }
}
```

Analicemos las partes más destacadas del ejemplo anterior:

- Utilizamos la clase *FileOutputStream* para escribir datos en un fichero binario. Primero instanciamos un objeto de esta clase que prepara el fichero para escritura. Luego, utilizamos el método *write(int dato)* para escribir de byte en byte los datos en el fichero de salida. En la siguiente imagen podemos observar la descripción que nos da la documentación (*Java 17*) de este método:

  write

  ```
  public void write(int b)
            throws IOException
  ```

  Writes the specified byte to this file output stream. Implements the `write` method of `OutputStream`.

  Specified by:

  `write` in class `OutputStream`

  Parameters:

  `b` - the byte to be written.

  Throws:

  `IOException` - if an I/O error occurs.

- Utilizamos la clase *FileInputStream* para leer los datos de un fichero binario. Primero instanciamos un objeto de esta clase que prepara el fichero para lectura. Luego, utilizamos el método *read()* para leer

de byte en byte los datos del fichero; en caso de que no haya más datos a leer del fichero, el método devuelve el valor -1. En la siguiente imagen podemos observar la descripción que nos da la documentación (Java 17) de este método:

read

```
public int read()
         throws IOException
```

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Specified by:

`read` in class `InputStream`

Returns:

the next byte of data, or `-1` if the end of the file is reached.

Throws:

`IOException` - if an I/O error occurs.

- En este caso es conveniente hacer el tratamiento de las excepciones correspondientes por separado, ya que así separamos y tratamos la lógica de cada acción de forma más específica.

Recordemos que también tenemos la opción de realizar la lectura y escritura de forma más eficiente a través de un *buffer*. Para el caso de los ficheros binarios tenemos la clase *BufferedInputStream* (lectura) y *BufferedOutputStream* (escritura).

Por otra parte, también disponemos de dos clases para leer y escribir datos primitivos (int, float, etc) en ficheros binarios; en este caso, debemos conocer de antemano la estructura del fichero. Estas dos clases son:

- *DataInputstream* (para lectura)
- *DataOutputStream* (para escritura

Para el resto de los métodos de las clases mencionadas en este apartado es conveniente revisar la documentación oficial (Java 17):

- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/FileOutputStream.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/FileInputStream.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedInputStream.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedOutputStream.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/DataInputStream.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/DataOutputStream.html

CC BY NC SA

5.2. Ficheros de acceso aleatorio

Hasta ahora hemos estudiado los mecanismos que proporciona Java para leer ficheros en modo de *acceso secuencial*; esto es, como si los ficheros fueran las antiguas cintas de casete (por cierto, los casetes eran esto). La principal desventaja es que, para encontrar cualquier dato concreto, debemos recorrer el fichero desde el inicio hasta llegar al dato buscado.

Otra forma de acceder a los datos de un fichero es el modo de *acceso aleatorio*; esto es, podemos acceder a una determinada una posición del fichero para leer, escribir o modificar información en el fichero. La ventaja evidente es que no hay que recorrer las posiciones anteriores del fichero. En Java tenemos la clase *RandomAccessFile* que nos permitirá este tipo de accesos tal y como indica la documentación de Java.

Ejemplo (simplificado): Lectura/escritura de número entero con acceso aleatorio a fichero

```
// Lectura
fichero = new File("recursos\\datos_binarios.dat");
RandomAccessFile raf1 = new RandomAccessFile(fichero, "r");
raf1.seek(1);
int dato = raf1.readInt();
System.out.print(dato);

// Escritura
RandomAccessFile raf2 = new RandomAccessFile(fichero, "rw");
raf2.seek(1);
raf2.writeInt(13);
```

Lectura:

- El constructor de la clase *RandomAccessFile* necesita dos parámetros: la ruta del fichero y el modo de acceso. El modo de acceso podrá ser de lectura (*"r"*) o lectura/escritura (*"rw"*).
- Si el fichero ya existe lo abre; en caso contrario, lo crea. Por tanto, nunca sobrescribirá su contenido.
- Utilizamos el método *seek(int X)* para posicionarnos en el byte X del fichero. Hay que tener en cuenta que la indexación empieza en cero (al igual que los *arrays*).

Escritura:

- El método *readInt()* leerá un byte de tipo entero y devolverá el dato leído o -1 si es final de fichero.
- El fichero se abre en modo lectura/escritura.
- El método *writeTIPO(X)* escribe el tipo de dato primitivo (Int, Double, etc) del valor X dado.

> **Notas**
>
> - Para leer/escribir String utilizaremos los métodos *readUTF(String)* o *writeUTF(String)*.
>
> - Los anteriores ejemplos necesitan del tratamiento de las excepciones correspondientes.

CC BY NC SA

6. Persistencia de objetos en Java

La *serialización* en Java es el proceso de convertir un objeto Java en una secuencia de bytes, para que pueda ser almacenado en un archivo o transmitido a través de una red. La serialización se utiliza principalmente para la *persistencia de objetos*, es decir, para guardar el estado actual de un objeto y poder recuperarlo más tarde.

La serialización en Java se realiza utilizando la interfaz *Serializable* del paquete *java.io*. Al implementar la interfaz *Serializable*, una clase se marca como serializable, lo que significa que sus objetos pueden ser convertidos en una secuencia de bytes.

Las clases principales para g y recuperar objetos serializables son:

- ObjectOutputStream (guarda)
- ObjetInputStream (recupera)

Ambas clases heredan de las clases *OutputStream* y *InputStream*, respectivamente. A continuación, los enlaces a la documentación oficial de las clases mencionadas:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/ObjectOutputStream.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/ObjectInputStream.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/OutputStream.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/InputStream.html

Ejemplo: Designar una clase como serializable

```
import java.io.Serializable;

public class Persona implements Serializable {
}
```

Una vez hayamos marcado el objeto como serializable, Java se encargará de su serialización de forma automática.

6.1. Guardar objetos serializables (Serialización)

Supongamos que tenemos la clase Persona creada con sus atributos y métodos correspondientes y la hemos marcado como serializable. Para poder guardar los objetos que hayamos instanciado de la clase Persona, necesitamos la clase *ObjetOutputStream* y su método *writeObject(Object obj)*. No obstante, también necesitaremos de la clase *FileOutputStream* para crear y preparar el fichero binario para escritura, al igual que estudiamos en apartados anteriores.

Por otra parte, deberemos seguir tratando las excepciones que son requeridas por el uso de los métodos correspondientes.

CC BY NC SA

Ejemplo: Guardar objeto serializable

```
public class Test {
  public static void main(String[] args) {
    // Creamos un ObjectOutputStream para escribir el objeto en un archivo
    ObjectOutputStream salida = null;

    try {
      // Abrimos o creamos el fichero para escritura
      FileOutputStream fichero = new FileOutputStream("recursos\\persona.ser");

      // Instanciamos el objeto para serializar en el fichero
      salida = new ObjectOutputStream(fichero);

      Persona persona = new Persona("Anakin", 40); // nombre, edad

      // Escribimos el objeto en el archivo
      salida.writeObject(persona);
    } catch (IOException e) {
      System.out.println("No se ha podido encontrar el fichero.");
    } catch (IOException e) {
      System.out.println("No se ha podido escribir en el fichero.");
    } finally {
      // Cerramos el ObjectOutputStream
      try {
        out.close();
      } catch (IOException e) {
        System.out.println("No se ha podido cerrar el fichero.");
      }
    }
  }
}
```

Nota

El fichero donde se guardará el objeto/s creados lo podemos nombrar con cualquier extensión. Sin embargo, una de las más habituales es la extensión *.ser*

6.2. Recuperar objetos serializados (Deserialización)

En caso de querer recuperar la información de un objeto serializado necesitaremos de la clase *ObjectInputStream* y su método *readObject()*. Además, análogamente al ejemplo de la escritura, en este caso de lectura de datos necesitaremos de la clase *FileOutputStream* para crear y preparar el fichero binario para lectura.

CC BY NC SA

Ejemplo: Recuperar un objeto serializado

```
public class Test {
  public static void main(String[] args) {
    // Creamos un ObjectInputStream para leer el objeto en un archivo
    ObjectInputStream entrada = null;

    try {
      FileInputStream fichero = new FileInputStream("recursos\\persona.ser");

      // Instanciamos el objeto para lectura
      entrada = new ObjectInputStream(fichero);

      // Leemos el objeto del fichero haciendo un casting al objeto original
      Persona p = (Persona) entrada.readObject();

      System.out.println("Nombre: " + p.getNombre());
      System.out.println("Edad: " + p.getEdad());

      } catch (FileNotFoundException e) {
        System.out.println("No se ha podido encontrar el fichero.");
      } catch (ClassNotFoundException e) {
        System.out.println("No se ha podido encontrar la clase.");
      } catch (IOException e) {
        System.out.println("No se ha podido leer el objeto del fichero .");
      } finally {
        // Cerramos el ObjectOutputStream
      try {
        entrada.close();
      } catch (IOException e) {
        System.out.println("No se ha podido cerrar el fichero.");
      }
    }
  }
}
```

Salida por pantalla:

```
Nombre: Anakin
Edad: 40
```

Analicemos las partes más destacadas del ejemplo anterior:

- Leemos el objeto del fichero con el método *readObject()*.
- Debemos hacer *casting* al tipo de objeto leído; esto es, debemos saber de antemano la estructura del fichero y los objetos que hay almacenados en él.
- Debemos tratar el tipo de excepción *ClassNotFoundException* lanzada por el método *readObject()*.

BY NC SA

Es posible almacenar en el mismo fichero diferentes objetos creados. Los objetos se irán almacenando (en bytes) de forma contigua conforme se vaya añadiendo.

Veamos el siguiente ejemplo simplificado para escribir varios objetos serializados en el mismo fichero:

```
// Creamos algunos objetos para serializar
Persona p1 = new Persona("Anakin", 40);
Persona p2 = new Persona("Leia", 19);
Persona p3 = new Persona("Obijuan", 55);
// Escribimos los objetos en el archivo
salida.writeObject(p1);
salida.writeObject(p2);
salida.writeObject(p3);
```

Por otra parte, para leer un fichero que contenga varios objetos serializados sería:

```
Persona p1 = (Persona) in.readObject();
Persona p2 = (Persona) in.readObject();
Persona p3 = (Persona) in.readObject();
```

En el caso de la lectura, al llamar al método *readObject()* leerá los bytes del objeto almacenado y se situará al final de los bytes leídos. Cuando se llame otra vez a dicho método, leerá los siguientes bytes que ocupe el siguiente objeto almacenado, y así sucesivamente.

Si a priori no sabemos cuántos objetos hay almacenados, podemos utilizar un bucle para recupera los objetos serializados. El inconveniente es obtener la información de final de fichero ya que el método *readObject()* no devuelve el valor *null* en tal caso; sin embargo, sí que lanza una excepción de tipo *EOFException*. Por tanto, una posible solución sería recorrer el bucle sin parada (condición a *true*) y tratar la excepción indicada.

Ejemplo: Recuperar un número indeterminado de objetos serializado del fichero

```
try {

    // … código de preparación de fichero para deserializar

    while(true) {
        Persona p = (Persona) entrada.readObject();
        System.out.println("Nombre: " + p.getNombre());
        System.out.println("Edad: " + p.getEdad());
    }
} catch (EOFException e) {
    System.out.println("Se ha alcanzado el final del fichero.");
}
```

7. Bibliografía

Libro “Core Java Volumen I: Fundamentals”, Cary S. Horstmann

Apuntes de José Chamorro del CFGS DAW del IES Sant Vicent Ferrer (Algemesí)

https://docs.oracle.com/

BY NC SA