TEMA 9. ANÁLISIS Y ESTUDIO DE LOS FLUJOS DE ENTRADA-SALIDA. FICHEROS

Índice

Flujos de Datos (Streams). Ficheros

Hasta ahora cada vez que ejecutamos una aplicación debemos introducir los datos necesarios para su correcta ejecución ya que al finalizar la ejecución se borran de la memoria.

Cuando estamos probando una aplicación y necesitamos introducir datos muchas veces este proceso puede ser muy tedioso.

Además, en algunas ocasiones necesitamos guardar datos para una siguiente ejecución. Un ejemplo de esto son las cookies de los navegadores web.

En estos casos podemos almacenar esa información en un fichero y cargarla cuando nos haga falta.

Antes de trabajar con ficheros es muy importante saber cómo vamos a recuperar los datos (lectura) ya que el modo en que hagamos la recuperación condiciona la escritura de los mismos.

En Java un fichero es un tipo de flujo de datos (stream). Para facilitar el trabajo con flujos de datos (streams) Java dispone de la biblioteca **java.io**. Si queremos usar las clases y métodos de java.io los debemos importar escribiendo

```
import java.io.*;
```

Ficheros de Texto

Antes de poder trabajar con un fichero de texto tenemos que abrirlo del modo adecuado.

Para abrir un fichero de texto de nombre **prueba.txt** para **escribir** algo en él debemos usar un objeto **FileWriter** al que asignamos la ruta del fichero dónde queremos escribir. Además, para realizar el proceso de escritura debemos un objeto **PrintWriter** que asociamos al objeto FileWriter. Para realizar todo lo anterior escribimos

```
FileWriter fichero = null;
PrintWriter pw = null;
fichero = new FileWriter("prueba.txt");
pw = new PrintWriter(fichero);
```

Una vez abierto el fichero de texto en modo escritura podemos escribir en él usando cualquiera de los métodos de la clase PrintWriter que coinciden con los que hemos usado para escribir algo por pantalla (print, println, printf). Por ejemplo, para escribir una línea usando el método **println** (al igual que hacemos en System.out.println) escribimos

```
pw.println("Hola Mundo.");
```

Para facilitar el proceso de escritura podemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedWriter** que asociamos al objeto PrintWriter. Hay que tener en cuenta que para escribir usando el objeto BufferedWriter deberemos usar el método **write** y que este método write no escribe el caracter de fin de línea por lo que si queremos escribir el carácter de fin de línea deberemos llamar después del método write al método **newLine**. Para escribir en un fichero de texto usando un objeto BufferedWriter escribimos

```
fichero = new FileWriter("prueba.txt");
pw = new PrintWriter(fichero);
bw = new BufferedWriter(pw);

// escribo unas lineas
bw.write("Hola Mundo.");
bw.newLine();
```

Cuando no vayamos a trabajar más con el fichero de texto debemos cerrar todos los recursos que tenga asignados usando el método **close** de cada recurso. Para el ejemplo anterior escribimos

```
// cierro el fichero
bw.close();
pw.close();
fichero.close();
```

Para abrir un fichero de texto de nombre **prueba.txt** para **leer** su contenido debemos usar un objeto **FileReader** al que asignamos un objeto de tipo **File** asociado con el fichero que vamos a usar para la lectura. Además, para facilitar el proceso de lectura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedReader** y lo asociamos al objeto FileReader. Para realizar todo lo anterior escribimos

```
File archivo = null;
FileReader fr = null;
BufferedReader br = null;
archivo = new File ("prueba.txt");
fr = new FileReader (archivo);
br = new BufferedReader(fr);
```

Una vez abierto el fichero de texto en modo lectura leemos su contenido línea a línea desde el buffer usando el método **readLine** escribiendo

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

Cuando no vayamos a trabajar más con el fichero de texto debemos cerrar todos los recursos que tenga asignados usando el método **close** de cada recurso. Para el ejemplo anterior escribimos

```
br.close();
fr.close();
```

Ejercicios de Ficheros de Texto

1. Crea la clase FicheroHola que escribe el mensaje “Hola Mundo” en un fichero de texto de nombre **prueba.txt**. Abre el contenido del fichero de texto con un editor de texto. Se deben controlar las posibles excepciones.

2. Crea la clase FicheroTextoBuffer que modifica FicheroHola para que la escritura se haga usando un buffer de escritura.

3. Crea la clase FicheroTextoBufferLectura que muestra el contenido de un fichero de texto de nombre **prueba.txt** por pantalla usando un buffer de lectura.
4. Crea la clase FicheroNombres que solicita nombres que va guardando en un fichero de texto de nombre **nombres.txt** (usando **println**) hasta que se introduzca una cadena en blanco ("") que no se guarda. Después muestra el contenido del fichero por pantalla. Se deben controlar las posibles excepciones.

5. Crea la clase FicheroNombresWrite que solicita nombres que va guardando en un fichero de texto de nombre **nombres.txt** (usando **write**) hasta que se introduzca una cadena en blanco ("") que no se guarda. Después muestra el contenido del fichero por pantalla. Se deben controlar las posibles excepciones.

6. Crea la clase FicheroHolaHTML que crea un fichero de nombre **hola.html** que tiene el valor “Hola HTML” como title y dentro del body tiene un párrafo con el texto “Fichero Hola HTML generado por código”. Después abre el fichero hola.html con el navegador. Se deben controlar las posibles excepciones.

Ficheros Binarios

El modo de trabajar con ficheros binarios en Java es similar al modo de trabajar con ficheros de texto. La única diferencia está en los objetos usados a la hora de trabajar con ficheros binarios.

Por ejemplo, si queremos abrir un fichero binario de nombre **prueba.bin** para **escribir** en él debemos usar un objeto **FileOutputStream** al que pasamos como parámetro un String con la **ruta del fichero** que queremos abrir. Además, para facilitar el proceso de lectura podemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedOutputStream** y lo asociamos al objeto FileOutputStream. Para realizar todo lo anterior escribimos

```
FileOutputStream fos = new FileOutputStream("prueba.bin");
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

Una vez abierto el fichero en modo escritura escribimos en él contenido bloque a bloque usando el método **write** que recibe los **datos** a escribir, el **desplazamiento en bytes** desde la posición actual (0 para escribir a continuación), y el **número de bytes** a escribir. Para mejorar el proceso de escritura creamos un array de tipo **byte** y de nombre **datos** con el tamaño de los bytes que queramos escribir en cada escritura. Por ejemplo, si queremos escribir de 8 en 8 bytes escribimos

```
byte [] datos = new byte[8];
bos.write(datos,0,8);
```

Cuando no vayamos a trabajar más con el fichero debemos cerrar todos los recursos que tenga asignados usando el método **close** de cada recurso. Para el ejemplo anterior escribimos

```
bos.close();
fos.close();
```

Si queremos abrir un fichero binario de nombre **prueba.bin** para **leer** su contenido debemos usar un objeto **FileInputStream** al que pasamos como parámetro un String con la **ruta del fichero** que queremos abrir. Además, para facilitar el proceso de lectura debemos crear un buffer de memoria asociado al fichero para lo que creamos un objeto **BufferedInputStream** y lo asociamos al objeto FileInputStream. Para realizar todo lo anterior escribimos

```
FileInputStream fis = new FileInputStream("prueba.bin");
BufferedInputStream bis = new BufferedInputStream(fis);
```

Una vez abierto el fichero binario en modo lectura leemos su contenido usando el método **read** que devuelve el número de bytes que se han leído (devuelve **0 si no ha podido leer nada** y **-1** si ha llegado al **final del fichero**). Cuando queremos leer todo el contenido de un fichero binario podemos usar el valor devuelto por read como condición de fin ya que sabemos que **read devuelve -1 si** ha llegado al **final del fichero**
Para realizar el proceso de lectura creamos un **array** de tipo **byte** y de nombre **datos** con el tamaño de los bytes que queramos leer en cada lectura. Por ejemplo, si queremos leer de 8 en 8 bytes escribimos

```
byte [] datos = new byte[8];
bis.read(datos);
```

Cuando no vayamos a trabajar más con el fichero debemos cerrar todos los recursos que tenga asignados usando el método **close** de cada recurso. Para el ejemplo anterior escribimos

```
bis.close();
fis.close();
```

Buffer en Entrada y Salida de Datos

Si trabajamos con objetos de las clases FileInputStream, FileOuputStream, FileReader o FileWriter, las operaciones se realizan directamente en el dispositivo de almacenamiento. Esto disminuye notablemente el rendimiento del sistema ya que las operaciones sobre dispositivos físicos consumen mucho tiempo.
Para mejorar el rendimiento de las operaciones de entrada y salida de datos de usan buffers (memorias intermedias) que actúan como intermediarios entre el dispositivo físico y las aplicaciones.

Cuando una aplicación tiene que leer desde un dispositivo físico hace una petición a su buffer de lectura que se encarga de leer los datos de manera más eficiente (aprovechando los momentos ociosos del procesador). Además de almacenar los datos solicitados, el buffer almacena los datos que se encuentran a continuación por si se van a necesitar. Esto provoca que si el buffer acierta a la hora de cargar los siguientes datos no haga falta volver a cargar los datos desde el dispositivo físico lo que aumenta considerablemente el rendimiento.

Cuando una aplicación tiene que escribir en un dispositivo físico manda los datos a su buffer de escritura que se encarga de escribir los datos de manera más eficiente (aprovechando los momentos ociosos del procesador). El buffer almacena los datos de varias operaciones de escritura antes de escribir los datos lo que disminuye el número de accesos al dispositivo físico. Esto aumenta considerablemente el rendimiento.

En Java las clases BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream permiten crear un buffer intermedio a la hora de leer o escribir en un dispositivo físico.

Para mejorar el rendimiento de las operaciones de entrada y salida usaremos siempre buffers.

Ejercicios de Ficheros Binarios

7. Crea la clase FicheroCopia que copia el fichero origen de nombre **logo.png** en el fichero destino de nombre **logocopia.png**. Para ello usa un array de tipo byte con 512 posiciones. Una vez copiado, abre el fichero de imagen logocopia.png con el Explorador de Windows para comprobar que la copia se ha realizado correctamente. Se deben controlar las posibles excepciones. Para que funcione la imagen

logo.png debe de estar dentro del proyecto (en nuestro caso la ruta de nuestro proyecto es D:\PROG\eclipse\Java)

Acceso Aleatorio en Ficheros

Al **abrir un fichero** el sistema guarda la **posición actual** del mismo **en un apuntador**. Esta posición actual es de la que lee los datos o en la que escribe los datos. Cuando se leen unos datos el apuntador pasa a apuntar a la posición siguiente al último dato leído y al escribir unos datos el apuntador pasa a apuntar a la posición siguiente al último dato escrito.

Cuando trabajamos con ficheros normalmente lo hacemos de forma secuencial, primero leemos un dato, luego el siguiente, y así hasta el final del fichero.

En algunas ocasiones podemos necesitar modificar el valor de la posición para saltarnos ese acceso secuencial. A ese tipo de acceso en el que **modificamos la posición del apuntador** según nuestras necesidades se le denomina **acceso aleatorio**.

Para facilitar el acceso aleatorio en ficheros Java proporciona la clase **RandomAccessFile** que contiene los métodos **getFilePointer** que devuelve la posición actual del apuntador, **length** que devuelve el tamaño en bytes del fichero, y **seek** que recibe el valor de la posición a la que queremos mover el apuntador y lo mueve. La definición completa de la clase **RandomAccessFile** la podemos encontrar en https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html

Para crear un nuevo objeto de la clase RandomAccessFile necesitamos proporcionar dos valores o un objeto de tipo **File** o un **String** con la **ruta** del fichero, y un **String** indicando el **modo de apertura.** Los modos de apertura son:
"r". Lectura. Si el fichero no existe da un error.
"rw". Lectura y escritura. Si el fichero no existe se crea. Si el fichero existe NO se sobreescribe.

Por ejemplo, para crear un nuevo objeto de la clase RandomAccessFile que abra el fichero logo.png en modo lectura para obtener su tamaño en bytes escribimos

```
RandomAccessFile raf = new RandomAccessFile("logo.png", "r");
long fileSize = raf.length();
```

El acceso aleatorio solo modifica la posición del apuntador. El resto de operaciones siguen siendo las mismas que se realizan sobre los ficheros con acceso secuencial.

Los objetos de la clase RandomAccessFile disponen de métodos que facilitan la lectura y escritura de datos. Por ejemplo, para escribir un String en la última posición de un fichero asociado a un objeto de la clase RandomAccessFile escribimos

```
// creo un RandomAccessFile
RandomAccessFile raf = new RandomAccessFile("nombres.txt", "rw");
// obtengo fileSize
long fileSize = raf.length();
// voy a la ultima posicion
raf.seek(fileSize);
// escribo "--------" en esa posicion
String s = "--------";
raf.writeUTF(s); // también podemos usar raf.writeChars(s); la diferencia es el formato de los caracteres
// cierro raf
raf.close();
```

Ejercicios de Acceso Aleatorio

8. Crea la clase FicheroAccesoAleatorioSize que muestra la longitud en KB del fichero logo.png.

9. Crea la clase FicheroNombresAppend que abre el fichero de texto **nombres.txt** y añade el nombre “--------” al final del fichero. Después muestra el contenido del fichero por pantalla usando los métodos de lectura de la clase RandomAccessFile (readLine). Se deben controlar las posibles excepciones.

Flujos Predeterminados

Java proporciona unos flujos predeterminados para facilitar las operaciones de entrada / salida de datos.

El flujo predeterminado de salida / escritura de datos es **System.out**. La función predeterminada de este flujo es facilitar las operaciones que muestren datos por pantalla. Dentro de la clase System.out se incluyen métodos que permiten escribir datos por pantalla como por ejemplo **System.out.print()**.

El flujo predeterminado de entrada / lectura de datos es **System.in**. La función predeterminada de este flujo es facilitar las operaciones de lectura de datos por teclado. Dentro de la clase System.in se incluyen métodos que permiten leer datos por teclado como por ejemplo **System.in.read()**.

Como las operaciones de entrada de datos suelen provocar muchos problemas se ha creado otra clase Java de nombre **Scanner** que permite trabajar con System.in de manera más sencilla.

Al realizar una operación de **lectura de un dato de tipo numérico** mediante la clase Scanner se coge el valor numérico del buffer de entrada y se asigna el valor a la variable correspondiente dejando en el buffer el **carácter de fin de línea** '\n'.

Si después de leer un dato de tipo numérico se lee un dato de tipo String o char ese carácter '\n' que se ha quedado en el buffer se asigna a la variable que se quiere leer dando la sensación de que se ha saltado la lectura.

Para evitar ese problema si después de leer un dato numérico tenemos que leer un dato de tipo String o char debemos poner antes de la sentencia de lectura real de los datos una operación de lectura que elimine el carácter '\n' del buffer.

Por ejemplo, si queremos leer la variable de tipo String cadena después de leer la variable de tipo int opción debemos escribir

```
// leo opcion
System.out.print("Opción: ");
opcion = teclado.nextInt();

// limpio el buffer de entrada
teclado.nextLine();

// leo cadena
System.out.print("Cadena: ");
cadena = teclado.nextLine();
```

Ejercicios Flujos Predeterminados

10. Realiza la clase Java LeerEnteroString que lee un número entero y un String y después muestra sus valores por pantalla.

Ficheros de Datos. Concepto de Registro

En los lenguajes estructurados (por ejemplo en lenguaje C, Cobol, …) los datos que comparten características comunes se guardan en unas estructuras de datos que reciben el nombre de registros. Agrupar los datos con características comunes en registros facilita su manipulación.

En los lenguajes orientados a objetos como Java para trabajar con datos que comparten características comunes se crean clases.

Los ficheros de registros y los ficheros de objetos son dos tipos diferentes de almacenamiento de información. Nosotros nos vamos a centrar en el almacenamiento de objetos.

Operaciones con Ficheros

Las operaciones que se realizan sobre los ficheros son

- Abrir el fichero en el modo que corresponda. Se puede abrir un fichero para leer, escribir, escribir al final o una combinación de las anteriores.
- Recorrer el fichero. Se pueden leer los elementos de un fichero uno a uno hasta llegar al final del fichero. Este proceso se puede usar para buscar si un valor está en el fichero o no.
- Posicionarse en el fichero (acceso aleatorio). Podemos posicionarnos en la posición del fichero que queramos para realizar una determinada operación.

Siempre que sea posible, es conveniente minimizar el trabajo directo con ficheros. Lo ideal es cargar el contenido del fichero en memoria (por ejemplo en un ArrayList), manipularlo y, si hay que actualizarlo, grabar los datos que están en memoria en el fichero.

Ejercicios de Ficheros

11. Realiza la clase Java ArrayListCadenasMenuFicheros que modifica ArrayListCadenasMenu para que al comienzo de la ejecución del programa se cargue el contenido del fichero cadenas.txt en el ArrayList y al finalizar la ejecución, si los datos han cambiado, se graben los datos del ArrayList en el fichero cadenas.txt. Las excepciones se tienen que controlar. Si se produce algún error al manipular el fichero se mostrará un mensaje de error indicando el porqué de ese error.

Almacenamiento de Objetos en Ficheros

En los lenguajes estructurados (por ejemplo en lenguaje C) los datos que comparten características comunes se guardan en unas estructuras de datos que reciben el nombre de registros. Agrupar los datos con características comunes en registros facilita su manipulación.

En los lenguajes orientados a objetos como Java para trabajar con datos que comparten características comunes se crean clases.

Los ficheros de registros y los ficheros de objetos son dos tipos diferentes de almacenamiento de información. Nosotros nos vamos a centrar en el almacenamiento de objetos.

A la hora de almacenar los datos de un objeto en un fichero podemos almacenar uno a uno los atributos que queramos guardar en el fichero y después recuperar desde el fichero los datos de los atributos respetando ese orden o podemos **guardar el objeto completo como un conjunto de bytes** y después recuperar desde el archivo el objeto completo.

El proceso de grabar un objeto completo en un fichero como un conjunto de bytes recibe el nombre de **serialización**.

Serialización. Serializable

La **serialización** es un mecanismo que permite tratar los objetos como un conjunto de bytes independientemente del contenido que tengan. Esto simplifica mucho operaciones como grabar y recuperar los datos en ficheros o mandar los datos por la red desde un equipo y recuperarlos en otro.

Para poder utilizar la serialización de objetos en una clase, dicha clase debe implementar el interfaz **Serializable**. Por ejemplo, si queremos que la clase Racional pueda usar la serialización de objetos debemos modificar su declaración de la siguiente manera

```
public class Racional implements Serializable{…}
```

Al leer o escribir un objeto Serializable en un fichero, las clases ObjectInputStream (mediante el método **readObject**) y ObjectOutputStream (mediante el método **writeObject**) se encargan de convertir los datos de ese objeto a un array de tipo byte para realizar la lectura o la escritura del objeto Serializable como un grupo de bytes.

Por ejemplo, para escribir un objeto de la clase Racional serializada de nombre valor en el fichero **racionales.ser** debemos escribir

```
// grabo los datos en racionales.ser
FileOutputStream fos=new FileOutputStream("racionales.ser");
ObjectOutputStream oos = new ObjectOutputStream (fos);
```

```
// lo grabo
oos.writeObject(valor);

// cierro el fichero
oos.close();
fos.close();
```

Si queremos leer un objeto de la clase Racional serializada desde el fichero **racionales.ser** debemos escribir

```
FileInputStream fis=new FileInputStream("racionales.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
// convierto los bytes leídos en un objeto de la clase Racional
valor = (Racional) ois.readObject();
ois.close();
fis.close();
```

Con lo anterior es suficiente para grabar y leer objetos serializados pero si por algún motivo queremos realizar nosotros la conversión del objeto Serializable a un array de tipo byte debemos escribir

```
ByteArrayOutputStream baos= new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream (baos);
oos.writeObject(miObjetoSerializable);
oos.close();
byte[] bytes =  baos.toByteArray(); // guarda los bytes en el array de tipo byte y de nombre bytes
```

Si queremos realizar el proceso inverso, es decir, la conversión de un array de tipo byte a un objeto Serializable debemos escribir

```
ByteArrayInputStream bais= new ByteArrayInputStream(bytes);
ObjectInputStream ois = new ObjectInputStream(bais);
ClaseSerializable miObjetoSerializable = (ClaseSerializable)ois.readObject(); // convierte los bytes leídos en un objeto de la ClaseSerializable
ois.close();
```

Si la información de los objetos Serializados va a viajar a través de la red de un equipo a otro o dentro del mismo equipo paro va a ir de un proceso a otro, nos podemos encontrar con que la versión de la clase del objeto serializado sea distinta en el equipo origen y en el equipo destino. Para evitar eso, en los objetos Serializable se incluye una constante de nombre **serialVersionUID** que contiene la **versión de la clase** de tal modo que se pueda saber si la versión de la clase origen y de la clase destino son distintas. Naturalmente versiones distintas de la clase deberán tener valores distintos.

Para definir la constante serialVersionUID escribimos

```
private static final long serialVersionUID = -1664923408223878238L;
```

Algunos entornos de desarrollo, como **eclipse**, muestran un warning si una clase que implementa Serializable (o hereda de una clase que a su vez implementa Serializable) no tiene definido este campo. Además, Eclipse permite generar automáticamente ese número al definir una clase como Serializable.

Cuando leemos desde un fichero debemos de controlar si hemos llegado al **final del fichero**. Para controlar si hemos llegado al final de un objeto de tipo **FileInputStream** podemos usar el método **available**.

Por ejemplo para controlar si hemos llegado al final del fichero alcomplejos.dat debemos escribir

```
FileInputStream fis=new FileInputStream("racionales.ser");
ObjectInputStream ois = new ObjectInputStream(fis);

// mientras que no sea el final del fichero leo los datos
while(fis.available() > 0){
```

```
 valor = (Racional)ois.readObject();
 …
}
```

Ejercicios de Almacenamiento de Objetos en Ficheros

12. Modifica la clase Complejo para que permita la serialización de objetos.

13. Crea la clase Java ComplejoMainSerializable que crea un objeto de la clase Complejo y lo guarda en el fichero complejos.dat. Después lee los datos del complejo desde complejos.dat y muestra el valor del complejo leído por pantalla.

14. Crea la clase Java ComplejoMainSerializableArrayList que crea cinco objetos de la clase Complejo y los almacena en un ArrayList. Después recorre el ArrayList guardando los datos de los objetos de tipo Complejo en el fichero alcomplejos.dat. Al finalizar lee los datos desde alcomplejos.dat y los almacena en un nuevo ArrayList. Para terminar recorre el ArrayList mostrando el valor de los objetos de tipo Complejo almacenados en él.

15. Modifica la clase Racional para que permita la serialización de objetos.

16. Crea la clase Java RacionalMainSerializable que crea un objeto de la clase Racional y lo guarda en el fichero racionales.dat. Después lee los datos del racional desde racionales.dat y muestra el valor del racional leído por pantalla.

17. Crea la clase Java RacionalMainSerializableArrayList que crea cinco objetos de la clase Racional y los almacena en un ArrayList. Después recorre el ArrayList guardando los datos de los objetos de tipo Racional en el fichero racionales.dat. Al finalizar lee los datos desde racionales.dat y los almacena en un nuevo ArrayList. Para terminar recorre el ArrayList mostrando el valor de los objetos de tipo Racional almacenados en él.

18. Crea la clase Java ArrayListEnterosMenuSerializable que modifica la clase ArrayListEnterosMenu para que al inicio de la ejecución se carguen los datos desde el fichero enteros.dat y para que al final de la ejecución, si los datos han sido modificados, los grabe de nuevo en enteros.dat.

19. Crea la clase Java ArrayListCadenasMenuSerializable que modifica la clase ArrayListCadenasMenu para que al inicio de la ejecución se carguen los datos desde el fichero cadenas.dat y para que al final de la ejecución, si los datos han sido modificados, los grabe de nuevo en cadenas.dat.

20. Crea la clase Java ArrayListComplejosMenuSerializable que modifica la clase ArrayListComplejosMenu para que al inicio de la ejecución se carguen los datos desde el fichero alcomplejos.dat y para que al final de la ejecución, si los datos han sido modificados, los grabe de nuevo en alcomplejos.dat.

21. Crea la clase Java ArrayListRacionalesMenuSerializable que raliza la misma función que la clase ArrayListComplejosMenuSerializable pero con objetos de la clase Racional. Al inicio de la ejecución se cargan los datos desde el fichero racionales.dat y al final de la ejecución, si los datos han sido modificados, se graban en racionales.dat.

22. Modifica la clase Fecha para que permita la serialización de objetos.

23. Crea la clase Java FechaMainSerializable que crea un objeto de la clase Fecha y lo guarda en el fichero fechas.dat. Después lee los datos de la fecha desde fechas.dat y muestra el valor de la fecha leída por pantalla.

24. Modifica la clase Persona para que permita la serialización de objetos.

25. Crea la clase Java PersonaMainSerializable que crea un objeto de la clase Persona y lo guarda en el fichero personas.dat. Después lee los datos de la persona desde personas.dat y muestra el valor de la persona leída por pantalla.

26. Modifica la clase Alumno para que permita la serialización de objetos.

27. Crea la clase Java AlumnoMainSerializable que crea un objeto de la clase Alumno y lo guarda en el fichero alumnos.dat. Después lee los datos del alumno desde alumnos.dat y muestra el valor del alumno leído por pantalla.

28. Crea la clase Java ArrayListAlumnosMenuSerializable que realiza la misma función que la clase ArrayListComplejosMenuSerializable pero con objetos de la clase Alumno. Al inicio de la ejecución se cargan los datos desde el fichero alumnos.dat y al final de la ejecución, si los datos han sido modificados, se graban en alumnos.dat.

29. Crea la clase Java **VentanaJListRacionalesFicheros** que modifica VentanaJListRacionales para que al inicio de la aplicación cargue los datos de los objetos de la clase **Racional** que están **racionales.ser** en la lista. Al cerrar la aplicación, solo si los datos han cambiado, guardará los datos de los objetos de la clase Racional que están en la lista en el fichero racionales.ser.

30. Crea la clase Java **VentanaJListFechasFicheros** que modifica VentanaJListFechas para que al inicio de la aplicación cargue los datos de los objetos de la clase **Fecha** que están **fechas.ser** en la lista. Al cerrar la aplicación, solo si los datos han cambiado, guardará los datos de los objetos de la clase Fecha que están en la lista en el fichero fechas.ser.

31. Crea la clase Java **VentanaJListPersonasFicheros** que modifica VentanaJListPersonas para que al inicio de la aplicación cargue los datos de los objetos de la clase **Persona** que están **personas.ser** en la lista. Al cerrar la aplicación, solo si los datos han cambiado, guardará los datos de los objetos de la clase Persona que están en la lista en el fichero personas.ser.

32. Crea la clase Java **VentanaJListAlumnosFicheros** que modifica VentanaJListAlumnos para que al inicio de la aplicación cargue los datos de los objetos de la clase **Alumno** que están **alumnos.ser** en la lista. Al cerrar la aplicación, solo si los datos han cambiado, guardará los datos de los objetos de la clase Alumno que están en la lista en el fichero alumnos.ser.

Exportar los datos a XML

Exportar los datos a otro formato como puede ser XML consiste en crear un fichero de texto con la extensión XML cuyo contenido respete la estructura de un documento XML.

Por ejemplo, para crear el archivo de texto **racionales.xml** que contiene los datos de unos objetos de la clase **Racional** tenemos que escribir de forma general

```
<?xml version='1.0' encoding='UTF-8' ?>
<racionales>
…
</racionales>
```

Dentro tendremos que colocar por cada objeto de la clase Racional que queramos guardar en el documento XML las siguientes etiquetas correspondientes a las propiedades de los objetos de la clase Racional. Por ejemplo para el Racional con valores 1/2 escribimos

```
<racional>
 <numerador>1</numerador>
 <denominador>2</denominador>
</racional>
```

Para agilizar el proceso de exportar a XML podemos crear en la clase Racional un método de nombre **toXML**, parecido a toString, que genere el texto con las etiquetas XML necesarias.

Una vez generado el documento XML lo podemos usar de manera similar a como usemos cualquier otro documento XML.

Si queremos que los datos del documento XML se **validen** con un documento **XML Schema** de nombre **alumnos.xsd** tendremos que modificar la etiqueta raíz del documento, **racionales** en este ejemplo, de la siguiente manera

```
<racionales xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:noNamespaceSchemaLocation='racionales.xsd'>
```

Si queremos que los datos se transformen usando un **archivo de transformación XSLT** de nombre alumnos.xsl tendremos que añadir al inicio del documento la línea

```
<?xml-stylesheet type='text/xsl' href='alumnos.xsl' ?>
```

A partir de este momento cuando se abra el documento XML se generará la salida que hayamos indicado en alumnos.xsl, por ejemplo una tabla HTML con los datos de los objetos de la clase Racional.

Ejercicios de Exportar los datos a XML

33. Crea la clase Java **VentanaJListRacionalesFicherosXML** que modifica VentanaJListRacionalesFicheros añadiendo un nuevo botón de nombre **btnXML** con el texto "**Exportar a XML**" que al hacer clic sobre él genera un documento con formato XML de nombre **racionales.xml** que incluye la información en formato XML de todos los elementos de la lista.

34. Crea la clase Java **VentanaJListFechasFicherosXML** que modifica VentanaJListFechasFicheros añadiendo un nuevo botón de nombre btnXML con el texto "Exportar a XML" que al hacer clic sobre él genera un documento con formato XML de nombre **fechas.xml** que incluye la información en formato XML de todos los elementos de la lista.

35. Crea la clase Java **VentanaJListPersonasFicherosXML** que modifica VentanaJListPersonasFicheros añadiendo un nuevo botón de nombre btnXML con el texto "Exportar a XML" que al hacer clic sobre él genera un documento con formato XML de nombre **personas.xml** que incluye la información en formato XML de todos los elementos de la lista.

36. Crea la clase Java **VentanaJListAlumnosFicherosXML** que modifica VentanaJListAlumnosFicheros añadiendo un nuevo botón de nombre btnXML con el texto "Exportar a XML" que al hacer clic sobre él genera un documento con formato XML de nombre **alumnos.xml** que incluye la información en formato XML de todos los elementos de la lista.

Exportar los datos a HTML

Exportar los datos a HTML consiste en crear un fichero de texto con la extensión HTML cuyo contenido respete la estructura de un documento HTML.

Por ejemplo, para crear el archivo de texto tablaracionales.html que a partir los datos de unos objetos de la clase Racional genera un documento HTML que contiene una tabla con los datos de esos objetos de la clase Racional tenemos que escribir de forma general

```
<!DOCTYPE html>
<html>
<head>
 <title>Expotar datos a HTML - Txema De Miguel</title>
 <meta charset='UTF-8'>
…
</head>
<body>
…
</body>
</html>
```

Dentro del body incluiremos las etiquetas necesarias para generar una tabla. Por ejemplo, en el caso de objetos de la clase Racional escribimos

```
<table>
<tr>
 <th>Numerador</th>
 <th>Denominador</th>
```

```
</tr>
…
</table>
```

Dentro de la tabla tendremos que generar una nueva fila, tr, por cada uno de los objetos de la clase Racional que queramos escribir en el documento HTML.

Al documento HTML generado se le pueden añadir hojas de estilo externas si hace falta. Por ejemplo, para hacer que el documento generado use la hoja de estilo externa de nombre estilos.css incluimos en el head del documento HTML la línea

```
<link rel='stylesheet' href='estilos.css'>
```

También le podemos añadir funcionalidad para que use jQuery. Por ejemplo, para hacer que el documento generado ejecute el código jQuery almacenado en el documento de nombre mijquery.js incluimos en el head del documento HTML las líneas

```
<script src='https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js'></script>
<script src='mijquery.js'></script>
```

En general, podemos usar en el documento generado las mismas funcionalidades que en un documento generado de manera tradicional.

Ejercicios de Exportar los datos a HTML

37. Crea la clase Java **VentanaJListRacionalesFicherosHTML** que modifica VentanaJListRacionalesFicheros añadiendo un nuevo botón de nombre **btnHTML** con el texto "**Exportar a HTML**" que al hacer clic sobre él genera un documento con formato HTML de nombre **racionales.html** que incluye en una tabla la información de todos los elementos de la lista.

38. Crea la clase Java **VentanaJListFechasFicherosHTML** que modifica VentanaJListFechasFicheros añadiendo un nuevo botón de nombre btnHTML con el texto "Exportar a HTML" que al hacer clic sobre él genera un documento con formato HTML de nombre **fechas.html** que incluye en una tabla la información de todos los elementos de la lista.

39. Crea la clase Java **VentanaJListPersonasFicherosHTML** que modifica VentanaJListPersonasFicheros añadiendo un nuevo botón de nombre btnHTML con el texto "Exportar a HTML" que al hacer clic sobre él genera un documento con formato HTML de nombre **personas.html** que incluye en una tabla la información de todos los elementos de la lista.

40. Crea la clase Java **VentanaJListAlumnosFicherosHTML** que modifica VentanaJListAlumnosFicheros añadiendo un nuevo botón de nombre btnHTML con el texto "Exportar a HTML" que al hacer clic sobre él genera un documento con formato HTML de nombre **alumnos.html** que incluye en una tabla la información de todos los elementos de la lista.

Ejercicios de Exportar los datos a HTML con jQuery

41. Crea la clase Java **VentanaJListRacionalesjQuery** que modifica VentanaJListRacionalesFicherosHTML para que el documento con formato HTML **racionales.html** generado incluya solo un **article** que contenga la tabla con la información de todos los elementos de la lista.

42. Crea la clase Java **VentanaJListFechasjQuery** que modifica VentanaJListFechasFicherosHTML para que el documento con formato HTML **fechas.html** generado incluya solo un article que contenga la tabla con la información de todos los elementos de la lista.

43. Crea la clase Java **VentanaJListPersonasjQuery** que modifica VentanaJListPersonasFicherosHTML para que el documento con formato HTML **personas.html** generado incluya solo un article que contenga la tabla con la información de todos los elementos de la lista.

44. Crea la clase Java **VentanaJListAlumnosjQuery** que modifica VentanaJListAlumnosFicherosHTML para que el documento con formato HTML **alumnos.html** generado incluya solo un article que contenga la tabla con la información de todos los elementos de la lista.

45. Crea el documento HTML de nombre **menuajaxhtml.html**. Para controlar el formato de los elementos del menú vamos a usar la hoja de estilos **menuajaxhtml.css**. El documento menuajaxhtml.html incluye un nav con un menú con las opciones Inicio, Racionales, Fechas, Personas y Alumnos en el header y una section con id contenido. Al hacer clic sobre una de las opciones cargará en la section el contenido de su fichero asociado (inicio.html, racionales.html, fechas.html, personas.html o alumnos.html).