10.31. Se pretende mantener los datos de los clientes de un banco en un archivo de texto. De cada cliente se guardará: DNI, nombre completo, fecha de nacimiento y saldo. Implementa una aplicación que arranque mostrando en el menú:

1. Alta cliente.
2. Baja cliente.
3. Listar clientes.
4. Salir.

Implementa la clase `Cliente` con los atributos referidos. Nada más arrancar la aplicación se leerán del archivo los datos de los clientes construyendo los objetos `Cliente` de todos ellos, que se irán insertando en una tabla de clientes. Cuando se dé de alta uno nuevo, se creará el objeto correspondiente y se insertará en la tabla por orden de DNI. Para eliminar un cliente, se pedirá el DNI y se eliminará de la tabla. Al listar los clientes, se mostrará el DNI, el nombre, el saldo y la edad de todos ellos, así como el saldo máximo, el mínimo y el promedio del conjunto de los clientes. Al cerrar la aplicación, se guardarán en el archivo los datos actualizados con el mismo formato.

UNIDAD 11

Ficheros binarios

Objetivos

- Abrir y cerrar archivos binarios.
- Escribir datos primitivos y objetos en archivos binarios.
- Leer datos primitivos y objetos de archivos binarios.
- Conocer las distintas excepciones que se pueden arrojar durante la apertura y cierre de archivos binarios.
- Identificar las distintas excepciones que se pueden arrojar durante la escritura y lectura de datos en archivos binarios.
- Gestionar sistemas complejos donde se guarda y recupera información.

Contenidos

Introducción

En la unidad anterior vimos que hay dos tipos de flujos de datos en Java, los binarios (también llamados *de bytes*) y los de texto. Allí nos dedicamos a estudiar con detalle los de texto. Ahora nos ocuparemos de los flujos de datos de tipo `byte`, que nos van a permitir guardar (o transferir) y recuperar (o recibir) cualquier tipo de datos usados en un programa. No olvidemos que, para usar cualquier flujo en Java, debemos importar las clases del paquete `java.io`.

Cuando se trata de escribir (o leer) bytes en un flujo, existen dos clases básicas, `FileOutputStream` y `FileInputStream`. Pero nosotros no solemos manejar bytes individuales en nuestros programas, sino datos (eso sí, formados por bytes) más complejos, ya sean de tipos primitivos u objetos. Por eso necesitamos un intermediario capaz de convertir los datos complejos en series planas de bytes o reconstruir los datos a partir de series de bytes, en procesos de **serialización** y de **deserialización** de datos, respectivamente. Esos intermediarios son flujos llamados *envoltorio:* `ObjectOutputStream` y `ObjectInputStream`, que se crean a partir de flujos de bytes planos, como `FileOutputStream` y `FileInputStream`.

11.1. Flujos de salida binarios

Supongamos que queremos grabar en disco los enteros guardados en una tabla. Para ello empezaremos creando un flujo de salida de tipo binario, asociado al fichero donde vamos a grabarlos, que llamaremos *enteros.dat*.

```
FileOutputStream archivo = new FileOutputStream("enteros.dat");
```

El constructor puede arrojar una excepción del tipo `FileNotFoundException`, que hereda de `IOException`. La sentencia creará en el disco el archivo *enteros.dat*. Si ya existía, borrará la versión anterior y lo sustituirá por una nueva.

Como ocurría con los archivos de texto, el nombre del archivo puede incluir una ruta de acceso, con los requisitos que vimos en la Unidad 10. Una vez creado este flujo, lo «envolvemos» en un objeto de la clase `ObjectOutputStream`.

```
ObjectOutputStream out = new ObjectOutputStream(archivo);
```

El constructor de `ObjectOutputStream` puede arrojar una excepción `IOException` —excepción de entrada-salida—. Por tanto, debe ir encerrado en una estructura `try-catch`, que puede englobar también al constructor del objeto `FileOutputStream`.

La clase `ObjectOutputStream` tiene una serie de métodos que permiten la escritura de datos complejos de cualquier tipo o clase, serializándolos antes de enviarlos al flujo de salida. Para ello, las clases de estos datos deben tener implementada la interfaz `Serializable`, que no es más que una especie de sello que declara a sus objetos como susceptibles de ser serializados, es decir, convertibles en una serie plana de bytes.

Las clases implementadas por Java, como `String`, las colecciones (que veremos en la Unidad 12) y las tablas, traen implementadas la interfaz `Serializable`. Los objetos de

INFORMÁTICA Y COMUNICACIONES
11. FICHEROS BINARIOS

estas clases, así como los datos de tipo primitivo, son serializados automáticamente por Java. En cambio, las clases definidas por el usuario deben declararse como serializables en su definición, sin que esto nos obligue a implementar ningún método especial.

```
class miClase implements Serializable {
    //cuerpo de la clase
}
```

Con esto, `miClase` ya es serializable, y sus objetos susceptibles de ser enviados por un flujo binario.

`ObjectOutputStream` dispone de los siguientes métodos para la escritura de datos en un flujo de salida:

- `void writeBoolean(boolean b)`: escribe un valor `boolean` en el flujo.
- `void writeChar(int c)`: escribe el valor `char` que ocupa los dos bytes menos significativos del valor entero que se le pasa como parámetro.
- `void writeInt(int n)`: escribe un entero.
- `void writeLong(long n)`: escribe un entero largo.
- `void writeDouble(double d)`: escribe un número de tipo `double`.
- `void writeObject(Object o)`: escribe un objeto serializable.

Como ejemplo, en la Actividad resuelta 11.1, vamos a guardar en un archivo los elementos de una tabla de enteros.

Actividad resuelta 11.1

Escribir en un archivo *datos.dat* los valores de una tabla de diez enteros.

Solución

*/*Inicializamos la tabla con los enteros del 0 al 9. Luego creamos el archivo y le asociamos un flujo de salida de la clase ObjectOutputStream. A continuación, recorremos la tabla escribiendo los enteros en él*/*

```
int[] t = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
ObjectOutputStream flujoSalida = null;
try {
    flujoSalida = new ObjectOutputStream(new FileOutputStream("datos.dat"));
    for (int n : t) {
        flujoSalida.writeInt(n);
    }
} catch (IOException ex) {
    System.out.println(ex);
} finally {
    if (flujoSalida != null) {
        try {
                flujoSalida.close();
        } catch (IOException ex) {
    System.out.println(ex);
    }
}
```

En el ejemplo de la Actividad resuelta 11.1, hemos recorrido la tabla para obtener sus elementos y grabarlos por separado. Pero, en Java, una tabla es un objeto, y podríamos haberla escrito en el archivo como tal objeto, usando el método `writeObject()`. Por tanto, en este código el bucle `for` puede ser sustituido por una sentencia única:

```
flujoSalida.writeObject(t);
```

donde le hemos pasado como parámetro la referencia al objeto que queremos grabar, la tabla `t`.

En este caso grabamos la tabla como objeto, que no es lo mismo que grabar los enteros por separado. Esta distinción será importante a la hora de recuperarla.

Igualmente, para guardar una cadena de caracteres se usa el método `writeObject()`, ya que una cadena es un objeto de la clase `String`.

```
String cadena = "Sancho Panza";
flujoSalida.writeObject(cadena);
```

Actividad resuelta 11.2

Escribe como una cadena, en el fichero binario *cancionPirata.dat*, la siguiente estrofa:

```
Con diez cañones por banda,
viento en popa a toda vela,
no corta el mar, sino vuela
un velero bergantín.
```

Solución

```
/*Como no se trata de un archivo de texto, convertimos la estrofa en una cadena,
incluyendo los cambios de línea, y luego la escribimos en el flujo como un objeto
String*/
String estrofa = "Con diez cañones por banda, \n"
    + "viento en popa a toda vela, \n"
    + "no corta el mar, sino vuela \n"
    + "un velero bergantín.";
try ( ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("canconPirata.dat"))) {
    out.writeObject(estrofa);
}catch (IOException ex) {
    System.out.println(ex);
}
/*La estrofa se guardará con la codificación específica de los objetos de la clase
String y no podremos leerla directamente del archivo con un editor de texto. Si
queremos guardar texto legible con un editor de texto, deberemos usar archivos de
texto*/
```

Hasta Java 7, los flujos, tanto de texto como binarios, había que cerrarlos con el método `close()`, disponible en todas las clases de entrada y salida, incluyéndolo en un bloque `finally`. No obstante, como vimos con los archivos de texto y en la Actividad resuelta 11.2, usando una estructura `try-catch` con recursos, el cierre es automático y no tenemos que usar el método `close()`.

INFORMÁTICA Y COMUNICACIONES

Actividad resuelta 11.3

Pedir un entero `n` por consola y, a continuación, pedir `n` números de tipo `double`, que iremos insertando en una tabla. Guardar la tabla en un archivo binario.

Solución

```
try (ObjectOutputStream out = new ObjectOutputStream(
new FileOutputStream ("datos.dat"))){
    System.out.println("Número de elementos: ");
    int n = new Scanner(System.in).nextInt(); /*cantidad de valores a leer*/
    double tabla[] = new double[n]; //tabla con el tamaño adecuado
    for (int i = 0; i < tabla.length; i++) {
        System.out.print("Introduzca un número real: ");
        tabla[i] = new Scanner(System.in).useLocale(Locale.US).nextDouble ();
    }
    out.writeObject (tabla); // las tablas son objetos
} catch (IOException e) {
    System.out.println(e.getMessage ());
}
```

Actividad propuesta 11.1

Repite la Actividad resuelta 11.1 escribiendo la tabla de enteros en el archivo *datos.dat*, y no los enteros individualmente.

11.2. Flujos de entrada binarios

Para leer de fuentes de datos binarios, usaremos flujos de la clase `ObjectInputStream`, construidos a partir de un flujo de bytes planos `FileInputStream`. Por ejemplo, si leemos los datos escritos en el archivo *datos.dat* de las actividades propuesta y resuelta 11.1, crearemos un flujo de entrada asociado al archivo.

```
ObjectInputStream flujoEntrada = new ObjectInputStream(new
                                 FileInputStream("datos.dat"));
```

Esta sentencia puede producir una excepción `IOException`; por tanto, deberá ir encerrada en una estructura `try-catch`. Lo mismo ocurre a la hora de cerrarlo con el método `close()`, aunque nosotros usaremos habitualmente una estructura `try-catch` con recursos.

Los métodos de la clase `ObjectInputStream` permiten leer los mismos datos que grabamos con `ObjectOutputStream`. Por cada método de escritura de esta última hay otro de lectura de la primera. En el caso de que hayamos grabado los 10 enteros de una tabla por separado usando `writeInt()`, los podemos recuperar, también por separado, con el método `readInt()`, que puede arrojar una excepción `IOException` si hay un error de lectura o `EOFException` si se ha llegado al final del fichero.

Actividad resuelta 11.4

Leer de un archivo *datos.dat* 10 números enteros, guardándolos en una tabla de tipo `int`.

Solución

```
//usaremos una estructura try-catch con recursos
try (ObjectInputStream flujoEntrada = new ObjectInputStream (
new FileInputStream("datos.dat"))) {
     int[] t = new int[10];
     for (int i = 0; i < t.length; i++) {
          t[i] = flujoEntrada.readInt();
     }
     System.out.println(Arrays.toString(t));
} catch (IOException ex) {
     System.out.println("error lectura");
}
```

Los métodos más importantes de `ObjectInputStream` son los siguientes:

- `boolean readBoolean()`: lee un booleano del flujo de entrada.
- `char readChar()`: lee un carácter.
- `int readInt()`: lee un entero.
- `long readLong()`: lee un entero largo.
- `double readDuoble()`: lee un número real de tipo `double`.
- `Object readObject()`: lee un objeto.

Dado que las tablas son objetos, si se ha guardado la tabla `t` usando el método `writeObject()`, en vez de un bucle `for` para la lectura, usaremos una sentencia única, ya que lo que hay guardado es un objeto, no una serie de enteros.

Actividad resuelta 11.5

Leer una tabla de enteros de un archivo *datos.dat*.

Solución

```
try (ObjectInputStream flujoEntrada = new ObjectInputStream(
new FileInputStream("datos.dat"))) {
     int[] tabla = (int[]) flujoEntrada.readObject();
     System.out.println(Arrays.toString(tabla));
} catch (IOException e) {
     System.out.println("Error de entrada/salida");
} catch (ClassNotFoundException e) {
     System.out.println("El fichero no almacena un objeto tabla");
}
```

En la Actividad resuelta 11.5 el cast `(int[])` es necesario, ya que `readObject()` devuelve un objeto de la clase `Object`, que es asignado a una variable de tipo `int[]` (tabla de enteros), lo que supone una conversión de estrechamiento.

INFORMÁTICA Y COMUNICACIONES

Por otra parte, llama la atención la excepción `ClassNotFoundException`, que puede ser arrojada por el método `readObject()`. Esto se debe a que, cuando leemos un objeto de un flujo de entrada, puede ocurrir que la clase a la que pertenece no sea visible desde el lugar del código donde se invoca `readObject()`, debido a que no esté en el mismo paquete ni haya sido importada de otro.

Como ya vimos, las cadenas de texto son objetos y, si se guardaron como tales, se deben recuperar utilizando el método `readObject()`.

```
try {
       String cadena = (String) flujoEntrada.readObject();
} catch (ClassNotFoundException ex) {
       System.out.println(ex.getMessage());
}
```

Actividad resuelta 11.6

Recuperar la estrofa del archivo *cancionPirata.dat* de la Actividad resuelta 11.2 y mostrarla por consola.

Solución

```
try ( ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("cancionPirata.dat"))) {
      String cancion = (String) in.readObject();
      System.out.println(cancion);//la mostramos
} catch (IOException ex) {
      System.out.println(ex);
} catch (ClassNotFoundException ex) {
      System.out.println(ex);
}
```

A menudo desconocemos el número de datos guardados en un archivo. En este caso, para recuperarlos todos, no podemos usar un bucle `for` controlado por contador, sino que tenemos que leer hasta que se llegue al final del fichero, es decir, hasta que salte la excepción `EOFException`. Por ejemplo, si un fichero contiene una lista de enteros y no sabemos cuántos hay, para recuperarlos todos, usamos un bucle infinito del que solo nos puede sacar la excepción `EOFException` de fin de fichero.

```
try {
     while (true) {
            System.out.println(in.readInt());
     }
}catch(EOFException ex) {
     System.out.println("Fin de fichero");
}
```

Cuando se haya leído el último entero, se habrá llegado al final del fichero. Entonces se arrojará la excepción y el programa saldrá del bucle `while` y del bloque `try` para continuar en el bloque `catch`.

Actividad resuelta 11.7

Grabar en el fichero *numeros.dat* una serie de números enteros no negativos introducidos por teclado. La serie acabará cuando escribamos –1. Abrir de nuevo *numeros.dat* para lectura y leer todos los números hasta el final del fichero, mostrándolos por pantalla y copiándolos en un segundo fichero *numerosCopia.dat*.

Solución

```
try ( ObjectOutputStream salida = new ObjectOutputStream(                    new
FileOutputStream("numeros.dat"))) {
    System.out.print("Introduce entero: ");
    Scanner s = new Scanner(System.in);
    int numero = s.nextInt();
    while (numero >= 0) {
         salida.writeInt(numero);
         System.out.print("Introduce entero: ");
         s = new Scanner(System.in);
         numero = s.nextInt();
    }
} catch (IOException ex) {
    System.out.println(ex);
}
/*Abrimos flujo de entrada para leer los números grabados y de salida para
grabarlos en el archivo copia: */
try ( ObjectInputStream entrada = new ObjectInputStream(new
FileInputStream("numeros.dat"));ObjectOutputStream salida = new
ObjectOutputStream(new FileOutputStream("numerosCopia.dat"))) {
   System.out.print("[");
   while (true) {
       int numero = entrada.readInt();
       System.out.print(numero + " ");
       salida.writeInt(numero);
   }
} catch (EOFException ex) {
   System.out.println("]\nFin de fichero");
} catch (IOException ex) {
   System.out.println(ex);
}
```

11.3. Ficheros binarios y objetos complejos

Los objetos que queremos guardar en un archivo binario no siempre son tan simples como una cadena de caracteres. La mayoría pertenecen a clases con atributos, que muchas veces son también objetos. Los valores de estos atributos, en realidad, son solo referencias a los objetos propiamente dichos. Entonces se plantea la siguiente cuestión: ¿qué se guarda en el fichero, el valor del objeto referenciado o solo la referencia? Si fuera solo la referencia, no estaríamos guardando la información que nos interesa, ya que las referencias cambian en cada ejecución del programa. Si leyéramos el archivo al día siguiente en una nueva ejecución del programa, no recuperaríamos la información del objeto, sino la dirección de memoria que tenía cuando se guardó.

INFORMÁTICA Y COMUNICACIONES

Supongamos que queremos guardar una tabla de objetos de la clase `Socio`. Pasaremos al método `writeObject()` la variable `tablaSocios`, que guarda la referencia a la tabla en la memoria. Pero no olvidemos que cada componente de la tabla guarda, a su vez, la referencia a un objeto de la clase `Socio`, no el objeto propiamente dicho. ¿Qué se guarda realmente en el archivo? La respuesta es: toda la información necesaria para reconstruir la tabla cuando se vuelva a leer del archivo. Esto incluye: la propia tabla y los objetos referenciados en cada componente, con la información sobre su clase y los valores de los atributos, incluidos aquellos que referencian otros objetos, como el nombre o el `dni`. Los atributos pueden ser incluso otras tablas, como la lista de los familiares del socio, que se guardarían de la misma forma. Java rastrea todas las referencias a objetos hasta construir la estructura completa de los datos, y guarda toda la información necesaria para reconstruir de nuevo el objeto guardado, junto con todos los objetos referenciados desde él, cuando se lea más tarde con `readObject()`.

Actividad resuelta 11.8

Implementar un programa que guarde en el fichero *socios.dat* una tabla de objetos `Socio`. Después se abrirá de nuevo el fichero en modo lectura para recuperar la tabla de socios, mostrándose por pantalla.

Solución

```
/*Implementaremos una clase Socio simplificada, suficiente para ilustrar lo que
nos interesa. Para que se pueda guardar en un fichero binario, deberá implementar
la interfaz Serializable*/
class Socio implements Serializable {
        String dni;
        String nombre;
        public Socio(String dni, String nombre) {
                this.dni = dni;
                this.nombre = nombre;
        }
        @Override
        public String toString() {
                return "Socio{" + "dni=" + dni + ", nombre=" + nombre + '}';
        }
}
/*El programa principal sería: */
Socio[] tablaSocios = new Socio[4];
tablaSocios[0] = new Socio("1", "pepe");
tablaSocios[1] = new Socio("11", "ana");
tablaSocios[2] = new Socio("7", "pepa");
tablaSocios[3] = new Socio("23", "cris");
//mostramos la tabla de socios antes de guardarla:
System.out.println(Arrays.deepToString(tablaSocios));
//Creamos un flujo de salida binario y escribimos en él:
try(ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("socios.dat"))){
            salida.writeObject(tablaSocios);
}catch (IOException ex) {
            System.out.println(ex);
} //El flujo de salida se cierra automáticamente
```

```
/*Creamos un flujo de entrada y leemos de él la tabla de socios, que asignaremos
a la misma variable tablaSocios. El bloque catch recoge tanto la excepción
IOException de la apertura del flujo como ClassNotFoundException ligado al cast
(Socio[]), como ya se comentó en el parágrafo 11.2*/
try(ObjectInputStream entrada=new ObjectInputStream(
new FileInputStream("socios.dat"))){
      tablaSocios=(Socio[])entrada.readObject();
} catch (IOException | ClassNotFoundException ex) {
      System.out.println(ex);
}
//Volvemos a mostrar la tabla de socios:
System.out.println(Arrays.deepToString(tablaSocios));
```

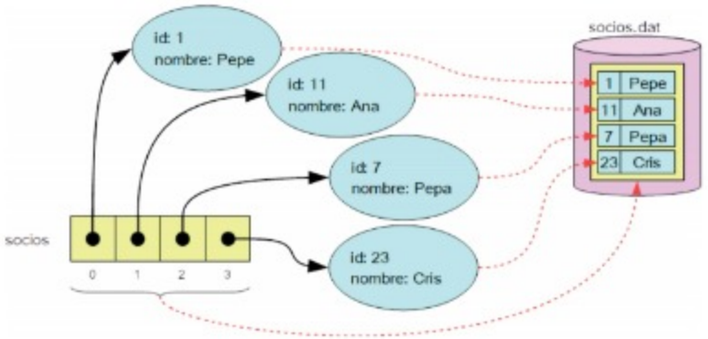


Figura 11.1. *Se guarda la tabla* `socios` *en el fichero socios.dat*

Recuerda

En una estructura `try-catch`, el bloque catch puede capturar más de un tipo de excepción. Para ello, basta escribir en el paréntesis todos los tipos separados por barras verticales, poniendo al final el nombre del parámetro que referencia la excepción, que funcionará como variable local dentro del bloque.

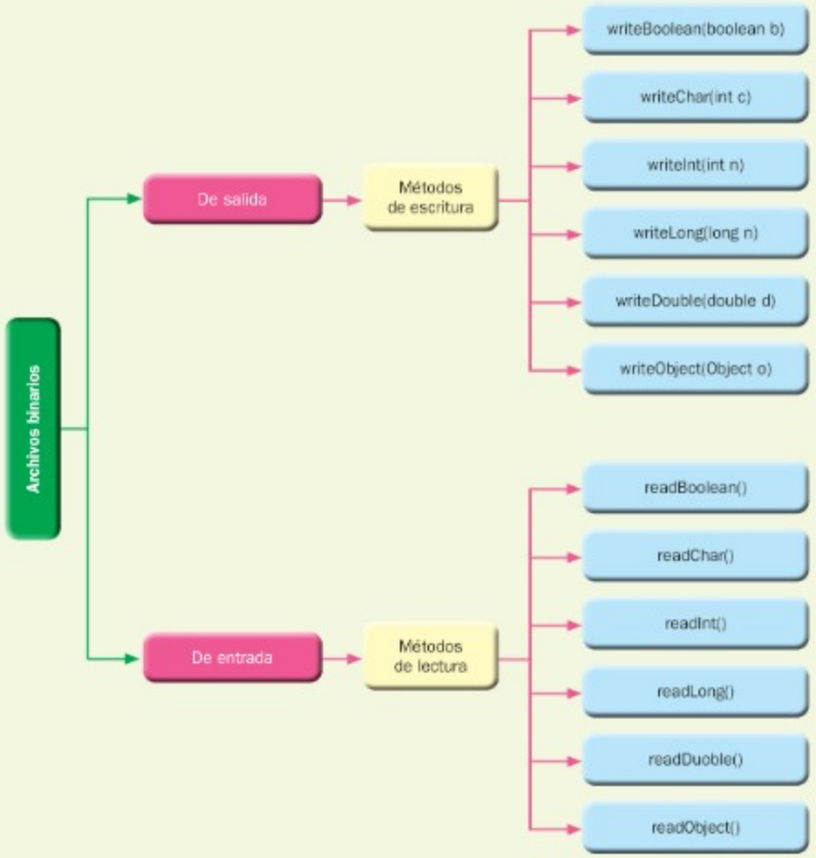
Actividad resuelta 11.9

Implementar un programa que registra la evolución temporal de la temperatura en una ciudad. La aplicación mostrará un menú que permite añadir nuevos registros de temperatura y mostrar el listado de todos los registros históricos. Cada registro constará de la temperatura en grados centígrados, introducida por teclado, y la fecha y hora, que se leerá del sistema en el momento de la creación del registro.

Solución

```
/*Definimos la clase Registro con dos atributos, la temperatura (double) y el
momento de la lectura (LocalDateTime)*/
class Registro implements Serializable {
        double temperatura;
        LocalDateTime fechaYHora;
        Registro(double temperatura) {
                this.temperatura = temperatura;
                fechaYHora = LocalDateTime.now();//lee del sistema
        }
        @Override
        public String toString() {
/*Mostramos la fecha y hora en formato local (en España, el español) corto*/
           DateTimeFormatter f = DateTimeFormatter
                .ofLocalizedDateTime(FormatStyle.SHORT)
                        .withLocale(Locale.getDefault());/*el del país local*/
           return "Registro{" + "temperatura=" + temperatura
                + ", fechaYHora=" + f.format(fechaYHora) + "}\n";
        }
}

/*En el programa principal, empezaremos creando la tabla de registros vacía: */
    Registro[] reg = new Registro[0];
/*La primera vez que se ejecute el programa, el archivo no existe y se trabaja
con la tabla vacía para insertar el primer registro. Si había registros previos,
ya existirá el fichero y se lee de él la tabla de registros, sustituyendo la
tabla vacía: */
    try ( ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("temperaturas.dat"))) {
        reg = (Registro[]) in.readObject();
    } catch (FileNotFoundException ex) {
        System.out.println("Registro vacío");
    } catch (IOException | ClassNotFoundException ex) {
        System.out.println(ex);
    }
    int opcion;
    do {
        System.out.println("1.Nuevo registro");
        System.out.println("2.Mostrar historial de registros");
        System.out.println("3.Salir");
        System.out.print("\nIntroducir opción: ");
        opcion = new Scanner(System.in).nextInt();
        switch (opcion) {
              case 1 -> {
                  System.out.print("Introducir temperatura: ");
                  double temperatura = new Scanner(System.in)
                          .useLocale(Locale.US).nextDouble();
                  Registro nuevo = new Registro(temperatura);
                  reg = Arrays.copyOf(reg, reg.length + 1);
                  reg[reg.length - 1] = nuevo;
              }
              case 2 ->
                  System.out.println(Arrays.deepToString(reg));
                }
        } while (opcion != 3);
        /*Al salir, guardamos la tabla de registros actualizada*/
    try ( ObjectOutputStream out = new ObjectOutputStream(
         new FileOutputStream("temperaturas.dat"))) {
        out.writeObject(reg);
    } catch (IOException ex) {
            System.out.println(ex);
    }
```



Actividades de comprobación

11.1. **Los ficheros binarios se diferencian de los de texto en que:**

a) Solo tienen ceros y unos.
b) Sirven tanto para escribir como para leer.
c) No sirven para guardar texto.
d) Permiten guardar todo tipo de datos, incluidos datos primitivos y objetos.

11.2. **Si queremos guardar una cadena de caracteres en un flujo binario de tipo `ObjectOutputStream`, usaremos:**

a) writeString().
b) writeChar().
c) writeObject().
d) Nada, no se puede.

11.3. **Para guardar una tabla del tipo `int[]` en un fichero binario de tipo `ObjectOutputStream`, usaremos:**

a) writeInt().
b) writeArrayInt().
c) readObject().
d) writeObject().

11.4. **Si queremos leer una tabla de `Cadenas` de caracteres del flujo binario `entrada` de tipo `ObjectInputStream`, escribiremos:**

a) String[] tabla = (String[])entrada.readObject();
b) String tabla = (String)entrada.readObject();
c) String[] tabla = entrada.readObject();
d) String[] tabla = (Object).readObject();

11.5. **Un flujo de tipo `ObjectInputStream` permite leer de:**

a) Cualquier archivo de Windows.
b) Archivos de imagen con extensión JPG.
c) Archivos creados con un flujo `ObjectOutputStream`.
d) Archivos creados con un flujo `BufferedReader`.

11.6. **Un flujo de tipo `ObjectInputStream` permite acceder a:**

a) Solo archivos del disco duro.
b) Cualquier fuente de datos primitivos u objetos de Java.
c) Únicamente a conexiones de red.
d) Solo nos permite leer de la consola.

11.7. **Si guardamos una cadena de caracteres usando un flujo `ObjectOutputStream`, podemos leerla directamente del archivo:**

a) Usando un procesador de texto.
b) Usando un editor de texto.
c) Usando una hoja de cálculo.
d) Usando un flujo `ObjectInputStream`.

11.8. **Si guardamos una serie de objetos de la clase `Cliente` con un flujo `ObjectOutputStream`, los recuperaremos:**
- a) En el mismo orden en que se guardaron.
- b) En orden inverso.
- c) En un orden aleatorio.
- d) Nunca se pueden recuperar.

11.9. **Los flujos binarios se cierran:**
- a) Con el método `close()`.
- b) Apagando el ordenador.
- c) Abortando el programa.
- d) Con el método `cerrar()`.

11.10. **Hay que cerrar los flujos binarios:**
- a) Siempre.
- b) Una vez al día.
- c) Solo si no se han abierto con una estructura `try-catch` con recursos.
- d) Nunca.

Actividades de aplicación

11.11. Pide un valor `double` por consola y guárdalo en un archivo binario.

11.12. Abre el fichero de la Actividad de aplicación 11.11, lee el valor `double` contenido en él y muéstralo por pantalla.

11.13. Escribe un programa que lea de un fichero binario una tabla de números `double` y después muestre el contenido de la tabla por consola.

11.14. Introduce por teclado una frase y guárdala en un archivo binario. A continuación, recupérala y muéstrala por pantalla.

11.15. Implementa un programa que lea números enteros desde el fichero *numeros.dat* y los vaya guardando en los ficheros *pares.dat* e *impares.dat*, según su paridad.

11.16. Implementa una aplicación que gestione una lista de nombres ordenada por orden alfabético. Al arrancar se leerá de un fichero los nombres insertados anteriormente y se pedirán nombres nuevos hasta que se introduzca la cadena «fin». Cada nombre que se introduzca deberá añadirse a los que ya había, de forma que la lista permanezca ordenada. Al terminar, se guardará en el fichero la lista actualizada.

11.17. Escribe un texto, línea a línea, de forma que, cada vez que se pulse Intro, se guarde la línea en un archivo binario. El proceso se termina cuando introduzcamos una línea vacía. Después el programa lee el texto completo del archivo y lo muestra por pantalla.

11.18. Un libro de firmas es útil para recoger los nombres de todas las personas que han pasado por un determinado lugar. Crea una aplicación que permita mostrar el libro de firmas o insertar un nuevo nombre (comprobando que no se encuentre repetido) usando el fichero binario *firmas.dat*.

11.19. Por motivos puramente estadísticos se desea llevar constancia del número de llamadas recibidas cada día en una oficina. Para ello, al terminar cada jornada laboral se guarda dicho número al final de un archivo binario. Implementa una aplicación con un menú, que nos permita añadir el número correspondiente cada día y ver la lista completa en cualquier momento.

11.20. Implementa una aplicación que permita guardar y recuperar los datos de los clientes de una empresa. Para ello, define la clase `Cliente`, que tendrá los atributos: `id` (identificador de cliente), `nombre` y `telefono`. Los objetos `Cliente` se insertarán en una tabla.

Para realizar las distintas operaciones, la aplicación tendrá el siguiente menú:
1. Añadir nuevo cliente.
2. Modificar datos.
3. Dar de baja cliente.
4. Listar los clientes.

La información se guardará en un fichero binario, que se cargará en la memoria al iniciar la aplicación y se grabará en disco, actualizada, al terminar.

11.21. Repite la Actividad de aplicación 11.20, pero insertando los objetos `Cliente` en un objeto `Lista` para `Object`, como el definido en la Actividad resuelta 9.11.

11.22. Implementa una aplicación que gestione los empleados de un banco. Para ello se definirá la clase `Empleado` con los atributos `dni`, `nombre` y `sueldo`. Los empleados se guardarán en un objeto de la clase `Lista` para objetos de la clase `Object`. La aplicación cargará en la memoria, al arrancar, la lista de empleados desde el archivo binario *empleados.dat* y mostrará un menú con las siguientes opciones: 1. Alta empleado; 2. Baja empleado; 3. Mostrar datos empleado; 4. Listar empleados, y 5. Salir. Al pulsar 5, se grabará en el disco la lista actualizada y terminará el programa.

11.23. Implementa el método, `Integer[] fusionar(String ficherol, String fichero2)`, al que se le pasan los nombres de dos ficheros binarios que contienen dos listas ordenadas de objetos `Integer`, y devuelve una tabla ordenada con todos los elementos de los dos ficheros fusionados.

11.24. Implementa el método, `void fusionar(String ficheroBase, String ficheroNuevo)`, que añade a `ficheroBase`, los elementos de `ficheroNuevo`, ambos ordenados. Al final, `ficheroBase` contiene la lista ordenada de todos los elementos de ambos ficheros.

11.25. En una tabla de cadenas se guardan los nombres de 4 ficheros binarios. En cada uno de ellos se guarda una tabla de números enteros ordenados en sentido creciente. Implementa una aplicación donde se introduce por teclado un número entero. El programa debe determinar si ese número se halla en alguno de los 4 ficheros y, en caso afirmativo, en cuál de ellos y en qué lugar de la tabla correspondiente.

Actividades de ampliación

11.26. Se quiere mantener un registro de las temperaturas máxima y mínima diaria en una estación meteorológica. Define la clase `Registro` con los atributos `tempMax`, `tempMin` y `fecha`, cuyos valores se introducen por teclado. Los dos primeros como valores `double` y el tercero como cadena con el formato `dd/mm/aaaa`. Implementa un programa que muestre por pantalla un menú con las opciones: 1. Nuevo registro; 2. Mostrar historial; 3. Mostrar estadísticas, y 4. Salir. La opción 2 mostrará en cuatro columnas las fechas, los valores máximo y mínimo diario y la variación (la diferencia entre el máximo y el mínimo) diaria. La opción 3 mostrará los valores medios, máximos y mínimos de las temperaturas máximas, de las mínimas y de las variaciones diarias.

Todos los registros se insertarán en una tabla, que se guardará en un archivo binario de forma que, al arrancar la aplicación, se leerá del archivo y al salir de ella (opción 4) se volverá a guardar actualizada.

11.27. Implementa la clase `Deportista` para gestionar la sección de deportes de un club social. Los atributos serán el DNI, el nombre, la fecha de nacimiento y el deporte que practica (enumerado), que deberá ser uno de los que ofrece el club: natación, remo, vela y waterpolo. Escribe una aplicación que gestione los datos de los deportistas, utilizando una tabla cuya longitud deberá ajustarse con las altas y bajas, y un menú que incluya las opciones: 1. Alta; 2. Baja; 3. Modificación de datos (todos los atributos salvo el DNI, que es inalterable); 4. Listar por orden alfabético de nombres; 5. Listar por orden de edad, y 6. Salir. Los datos se guardarán en un archivo binario, de donde se leerán al arrancar la aplicación y volverán a grabarse actualizados al salir.

11.28. Implementa la clase `Socio` para gestionar un club. Sus atributos serán: el número de socio, que se adjudicará consecutivamente según el orden de alta en el club, el nombre, la fecha de nacimiento, la fecha de alta, el teléfono y la dirección de correo electrónico. Escribe un programa que gestione las altas, las bajas y las modificaciones de los datos (salvo el número de socio, que es inalterable una vez asignado). Entre las funcionalidades de la aplicación deberán incluirse un listado por orden alfabético de nombres y otro por antigüedad en el club. Toda esta información se mantendrá en un archivo binario.

11.29. Desarrolla la Actividad 11.28 para añadir a la ficha de cada socio una lista de familiares a su cargo. Para ello, define la clase `Familiar` con los atributos: `dni`, `nombre` y `fecha de nacimiento`. Además, añade la opción de listar los datos de un socio incluyendo la lista de sus familiares ordenada por edad.

11.30. Con la clase `Jornada` de la Actividad de ampliación 9.28, implementa una aplicación que gestione una lista con las jornadas de los trabajadores, controlando las entradas y salidas (lo que comúnmente se llama «fichar»). El programa pedirá el DNI del usuario. A continuación, presentará un menú: 1. Entrada, y 2. Salida. Al elegir la opción se leerá la fecha y hora, que se asignará al atributo correspondiente. Con esta información se creará un registro de la jornada. La aplicación terminará cuando se introduzca como `dni` un número clave que solo conoce un directivo responsable. Los registros se insertarán, según el orden natural descrito en la Actividad de ampliación 9.28, en una tabla redimensionable, que se grabará en disco al finalizar la aplicación y se volverá a cargar al arrancar al día siguiente.

© Ediciones Paraninfo

11.31. Se quieren mantener los datos de los clientes de un banco en un archivo binario. De cada cliente se guardará: DNI, nombre completo, fecha de nacimiento y saldo. Implementa una aplicación que arranque mostrando el menú:

1. Alta cliente.
2. Baja cliente.
3. Listar clientes.
4. Salir.

Implementa la clase `Cliente` con los atributos referidos. Los objetos `Cliente` irán insertados en un objeto `Lista` de tipo `Object`. Nada más arrancar la aplicación se leerá del archivo la lista de clientes. Cuando se dé de alta uno nuevo, se creará el objeto correspondiente y se insertará en la lista por orden de DNI. Para eliminar un cliente, se pedirá el DNI y se eliminará de la lista. Al listar los clientes, se mostrará el DNI, el nombre, el saldo y la edad de todos ellos, así como el saldo máximo, el mínimo y el saldo promedio del conjunto de los clientes. Al cerrar la aplicación, se guardará en el archivo la lista actualizada.

11.32. Con la clase `Llamada` de la Actividad de ampliación 9.31, crea un registro de las llamadas efectuadas en una centralita, que se guardarán en el archivo binario *centralita.dat*. Al arrancar la aplicación se leerán los datos del archivo y a continuación se mostrará el menú:

1. Nuevo registro de llamada; 2. Listar las llamadas de un número de teléfono; 3. Listar todas las llamadas, y 4. Salir.

En el apartado 1, las fechas y horas se introducirán como cadenas con el formato «dd/MM/yyyy HH:mm». En los apartados 2 y 3 se mostrará el número de teléfono del titular, el del interlocutor, la fecha y hora de inicio (con el formato aludido antes) y la duración en minutos de cada llamada. Los registros se insertarán en una tabla por su orden natural. Al terminar la aplicación se guardará la tabla actualizada en el mismo archivo.