PROGRAMACIÓN UD-11

FICHEROS BINARIOS

ÍNDICE

[INTRODUCCIÓN 3](#_Toc130881679)

[FLUJOS DE SALIDA BINARIOS 3](#_Toc130881680)

[FLUJOS DE ENTRADA BINARIOS 5](#_Toc130881681)

[FICHEROS BINARIOS Y OBJETOS COMPLEJOS 6](#_Toc130881682)

[ACCESO ALEATORIO A UN FICHERO 8](#_Toc130881683)

# INTRODUCCIÓN

En este tema trataremos los ficheros binarios (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.

Tenemos que recordar 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. El problema es que 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.

Estos intermediarios son flujos llamados envoltorio: ObjectOutputStream y ObjectInputStream, que se crean a partir de flujos de bytes planos, como FileOutputStream y FileInputStream.

# FLUJOS DE SALIDA BINARIOS

Imaginaros 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 lanzar 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.

Al igual que ocurría con los archivos de texto, el nombre del archivo puede incluir una ruta de acceso. 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 objecto FileOutputStream.

La clase ObjectOutputStream tiene una serie de métodos que permiten la escritura de datos complejos 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 siguiente unidad) y las tablas, traen implementadas la interfaz Serializable. Los objectos de 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Con esto, miClase ya es serializable, y sus objetos son 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 pe 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.

En la actividad anterior, teníamos dos opciones: una de ellas, recorrer la tabla para obtener sus elementos y grabarlos por separado, y la segunda opción, sabiendo que, en Java, una tabla es un objeto, podríamos haberla escrito en el archivo como tal objeto, usando el método writeObject( ). Por lo que, en este último caso, el bucle for de código puede ser sustituido por una única sentencia:



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.

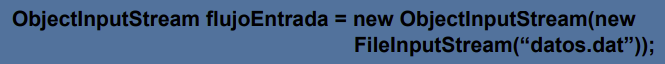
Un letrero de color blanco

Descripción generada automáticamente con confianza media

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, usando una estructura try-catch con recursos, el cierre es automático y no tenemos que usar el método close( ).

# 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 la actividad resuelta 11.1, creamos un flujo de entrada asociado al archivo.



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 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 grabados los 10 enteros de una tabla por separado usando writeInt( ), los podemos recuperar 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.

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 readDouble( ): lee un número real de tipo double.
* Object readObject( ): lee un objecto.

Dado que las tablas son objectos, 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 objecto, no una serie de enteros.

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

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 deberán recuperar utilizando el método readObject( ).

Texto, Aplicación

Descripción generada automáticamente

A menudo desconocemos el número de datos guardados en un archivo. En este caso, para recuperarlos todos, no podemos usar el bucle for controlado por un 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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.

# 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 fuese 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.

Supongamos que queremos guardar una tabla de objectos 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( ).

Diagrama

Descripción generada automáticamente

En una estructura try-catch, el bloque catch puede capturar más de un tipo de excepción. Para ello, basta con 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.

# ACCESO ALEATORIO A UN FICHERO

En Java, existe la posibilidad de poder acceder de forma aleatoria a un fichero, es decir, indicar la posición o punto concreto en el archivo a partir del cual se empezará a leer o escribir. Es importante indicar que la información almacenada en este tipo de archivos de acceso aleatorio se guarda en forma de bytes. Este tipo de ficheros no son stream, y se numeran por un índice que empieza por cero.

La clase de Java que nos va a permitir realizar este tipo de acceso se denomina RandomAccessFile.

Para el uso de este tipo de ficheros, es importante conocer cuántos bytes ocupa cada tipo de variable, ya que esto nos va a permitir colocarnos en el punto que queramos del fichero. Como recordatorio, veamos la siguiente tabla:

Tabla

Descripción generada automáticamente

Hay que tener en cuenta que los datos tipo String (tipo texto), en java son considerados como objetos. Por este motivo, un dato tipo String, se considera un array de caracteres tipo char. Esto quiere decir que la información de tipo String, ocupará dos bytes por cada carácter tipo char. A esta información, se le deben sumar los bytes ocupados por los espacios en blanco y los saltos de línea.

La clase RandomAccessFile permite utilizar un fichero de acceso aleatorio en el que el programador define el formato de los registros.

RandomAccessFile objFile = new RandomAccessFile( ruta, modo );

Los constructores de la clase son:

* RandomAccessFile(String path, String modo);
* RandomAccessFile(File objetoFile, String modo);

El argumento modo indica el modo de acceso en el que se abre el fichero.

Los valores permitidos para este parámetro son:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

La clase RandomAccessFile implementa los interfaces DataInput y DataOutput. Para abrir un archivo en modo lectura haríamos:

RandomAccessFile in = new RandomAccessFile(“input.dat”, “r”);

Para abrirlo en modo lectura y escritura:

RandomAccessFile inOut = new RandomAccessFile(“input.dat”, “rw”);

Esta clase permite leer y escribir sobre el fichero, no se necesitan dos clases diferentes.

Hay que especificar el modo de acceso al construir un objeto de esta clase: sólo lectura o lectura/escritura.

Dispone de métodos específicos de desplazamiento como seek y skipBytes para poder moverse de un registro a otro del fichero, o posicionarse directamente en una posición concreta del fichero.

La clase RandomAccessFile también dispone de métodos como:

* long getFilePointer( ); Devuelve la posición actual del puntero del fichero. Indica la posición (en bytes) donde se va a leer o escribir.
* long length(); Devuelve la longitud del fichero en bytes.
* void seek(long pos); Coloca el puntero del fichero en una posición pos determinada. La posición se da como un desplazamiento en bytes desde el comienzo del fichero. La posición 0 indica el principio del fichero. La posición length( ) indica el final del fichero.

Además, dispone de métodos de lectura/escritura:

* public int read( ) Devuelve el byte leído en la posición marcada por el puntero. Devuelve -1 si alcanza el final del fichero. Se debe utilizar este método para leer los caracteres de un fichero de texto.
* public final String readLine( ) Devuelve la cadena de caracteres que se lee, desde la posición marcada por el puntero, hasta el siguiente salto de línea que se encuentre.
* public xxx readXxx() Hay un método read para cada tipo de dato básico: readChar, readInt, readDouble, readBoolean, etc.
* public void write(int b) Escribe en el fichero el byte indicado por parámetro. Se debe utilizar este método para escribir caracteres en un fichero de texto.
* public final void writeBytes(String s) Escribe en el fichero la cadena de caracteres indicada por parámetro.
* public final void writeXxx(argumento) También existe un método write para cada tipo de dato básico: writeChar, writeInt, writeDouble, writeBoolean, etc.

# CLASE FILE

La clase File nos va a permitir desplazarnos por los directorios, mostrar su contenido, borrar ficheros, renombrar ficheros… Tiene los siguientes métodos:

* exists: Indica si existe o no el fichero.
* isDirectory: Indica si el objeto File es un directorio.
* isFile: Indica si el objeto File es un fichero.
* isHidden: Indica si el objeto File esta oculto.
* getAbsolutePath: Devuelve una cadena con la ruta absoluta del fichero o directorio.
* canRead: Indica si se puede leer.
* canWrite: Indica si se puede escribir.
* canExecute: Indica si se puede ejecutar.
* setReadable(true|false): Convierte el fichero en leíble o no.
* setWritable(true|false): Convierte el fichero a escribible o no.
* setExecutable(true|false): Convierte el fichero en ejecutable o no.
* getName: Devuelve una cadena con el nombre del fichero o directorio
* getParent: Devuelve una cadena con el directorio padre.
* length: Indica el tamaño del fichero apuntado en bytes.
* listFiles: Devuelve un array de File con los directorios hijos. Solo funciona con directorios.
* list: Devuelve un array de String con los directorios hijos. Solo funciona con directorios.
* mkdir: Permite crear el directorio en la ruta indicada. Solo se creara si no existe.
* mkdirs: Permite crear el directorio en la ruta indicada, también crea los directorios intermedios. Solo se creara si no existe.
* createNewFile: Permite crear el fichero en la ruta indicada. Solo se creara si no existe. Debemos controlar la excepcion con IOException.
* Delete: Elimina el fichero apuntado