PRÁCTICA 02 2DAM 21/10/2023

INTRODUCCIÓN A LOS FLUJOS DE ENTRADA Y SALIDA

Manuel Martín Santamaría

Índice

Indicaciones de entrega

Una vez realizada la tarea elaborarás un único documento de texto donde figuren las respuestas correspondientes. El envío se realizará a través de la plataforma de la forma establecida para ello, y el archivo se nombrará siguiendo las siguientes pautas:

apellido1_apellido2_nombre_SIGxx_Tarea

Asegúrate que el nombre no contenga la letra ñ, tildes ni caracteres especiales extraños. Así por ejemplo la alumna Begoña Sánchez Mañas para la primera unidad del MP de DAW, debería nombrar esta tarea como…

sanchez_manas_begona_PSP_Tarea01

Actividades

[Actividad 1]

¿Qué son los streams y para qué sirven?

Son clases que envuelven una fuente (o destino) de datos, implementando cierta funcionalidad, esta nos permite operar con dicha fuente de una forma fácil y rápida, dándonos control sobre el flujo de datos desde o hacia la fuente.

[Actividad 2]

¿De qué tipo pueden ser?

Según el sentido de los datos:

- De entrada (InputStreams): Envuelven una fuente de datos. Permiten la lectura de datos desde la fuente hacia nuestro programa.

- De salida (OutputStreams): Envuelven el destino de los datos. Permiten la escritura de datos desde nuestro programa al destino.

[Actividad 3]

¿Qué viaja a través de ellos?

Bytes o caracteres dependiendo del tipo del stream en cuestión. Pueden transferir información desde o hacia ficheros, strings, la red…
Los bytes pueden representar cualquier tipo de información, desde texto hasta imágenes, audio o cualquier otro dato binario.

[Actividad 4]

¿Qué excepción lanzan?

Excepciones relacionadas con la entrada y/o salida de datos, todas ellas heredan de IOException. Esta se puede dar por ejemplo si durante la ejecución del programa no se puede acceder a un fichero en la creación del stream, si hay un error de lectura / escritura…

[Actividad 5]

¿Qué se realiza para no sobreescribir un fichero a la hora de añadir texto? Es decir, ¿qué hay que hacer para añadir texto al final de un fichero?

Para evitar sobreescribirlo, se puede instanciar el flujo de salida hacia el fichero en modo adición (si este lo permite), mediante una variable booleana en su constructor.
Al hacer esto, cuando escribamos, se añadirá al final del fichero.

Si el flujo de salida no tiene ese parámetro en el constructor, como el Writer o BufferedWriter, se hace uso de los métodos que permiten esta característica, como los métodos append().

[Actividad 6]

¿Qué es un FileOutputStream?

Es una clase que crea un flujo de salida conectado a un archivo . Los bytes se escriben directamente en el archivo, sobrescribiendo su contenido existente si ya existe o creándolo si el archivo no está presente a no ser de que haya sido iniciado en modo append mediante su constructor. Este no ofrece ninguna funcionalidad para escribir datos en formato legible, escribe bytes puros.

[Actividad 7]

¿Qué hay que hacer si se quiere enviar cualquier otro tipo de dato que no sea byte por un OutputStream?

Para enviar cadenas de texto a través de un OutputStream, se puede utilizar un objeto de tipo PrintWriter o un OutputStreamWriter.

En el caso de caracteres, se puede usar un OutputStreamWriter para convertirlos a bytes antes de enviarlos.

Para enviar enteros u otros tipos de datos primitivos, antes hay que convertirlos a su representación en bytes.

Para poder enviar objetos han de ser serializables. Para ello, la clase del objeto debe implementar la interfaz Serializable. Esto permite que el objeto sea convertido en una secuencia de bytes, transmisibles a través del OutputStream.

[Actividad 8]

¿En qué se diferencia un PrintStream de un DataOutputStream?

Se diferencian en cómo formatean y almacenan los datos.
PrintStream se utiliza para la salida de texto legible, mientras que DataOutputStream se utiliza para la escritura de datos binarios primitivos.

[Actividad 9]

¿Sería correcto hacer lo siguiente?

```
Public void escribe(OutputStream out){
    try{
        DataOutputStream output =new DataOutputStream(out);
        output.writeChars(“nombre”);
        out.writeInt(19);
        output.close();
    }
        catch(IOException ex){
        ex.PrintExtra();
    }
}
```

OutputStream no tiene el método “writeInt()”, en todo caso sería “output.writeInt(19)“.

Hay un error en el manejo de excepciones, en lugar de "ex.PrintExtra();", imagino que a lo que te refieres es al método "ex.printStackTrace();" (para imprimir información sobre la excepción).

El close cierra el flujo de datos hacia la fuente, así que sería más eficiente cerrarlo desde otro sitio al terminar de operar con el fichero. O si sólo se va a trabajar con él en este método, cerrarlo en el finally, con su respectiva estructura try catch para diferenciar si el error se ha producido en la escritura o en el cierre del flujo..

[Actividad 10]

¿Qué se hace para no leer datos vacíos y crear ficheros corruptos?

- Comprobación del tamaño del fichero, si su tamaño es igual a cero, el fichero está vacío.
- Validación de los datos antes de procesarlos.
- Uso de excepciones para capturar y gestionar posibles errores durante la lectura de ficheros.
- La correcta elección y gestión del flujo hacia el fichero.
- Asegurarse de cerrar el flujo al fichero al terminar de operar con él.
- Implementar mecanismos de control de concurrencia (si se va a realizar un acceso simultáneo al archivo por múltiples hilos).

[Actividad 11]

¿Qué hay que hacer si se desea realizar de forma seguida dos InputStreams?

Cuando se desea realizar dos "InputStreams" de forma seguida, es importante recordar que estos flujos son secuenciales y no pueden compartir el mismo origen de datos. Puede ser necesario cerrar el primer "InputStream" antes de abrir el segundo. Antes de cerrar cada uno, puede ser conveniente vaciar el buffer si estos emplean un buffer para la lectura, porque aunque el recolector de basura de java se encarga de eliminar los recursos que no son necesarios de memoria, si no se vacía de manera adecuada el buffer, podría quedar almacenado en él durante un tiempo indefinido información sensible o provocar una corrupción del fichero.

Si los dos "InputStreams" apuntan a un origen de datos diferente, es conveniente manejarlos en estructuras try catch diferentes, para aislar mejor el origen del problema si se lanza una excepción.

[Actividad 12]

¿En qué se diferencian los Writer y los Readers de los Input y Output Streams? Cuando hay que leer/escribir texto ¿qué es más recomendable usar?

Los Input y Output Streams son más generales, se utilizan para la transferencia de bytes. Son eficientes en la lectura y escritura de datos binarios, pero cuando se trata de texto, trabajar con ellos es un poco tedioso, ya que hay que realizar conversiones manuales de bytes a caracteres y viceversa, lo que se traduce a un código más largo y propenso a errores.

Los Writers y Readers están diseñados para trabajar con datos de texto. Están optimizados para trabajar con texto y simplifican la codificación y la decodificación de caracteres, lo que es esencial cuando se trabaja con diferentes juegos de caracteres, como UTF-8, UTF-16, etc…

[Actividad 13]

¿Qué sucede al leer y al escribir con una codificación?

Internamente, la máquina virtual Java trabaja con datos en formato Unicode. No obstante, el formato de los datos transferidos a la JVM o desde ella corresponde con la propiedad file.encoding.

Los datos que entran en la JVM se convierten de file.encoding a Unicode, y los datos que salen de la JVM se convierten de Unicode a file.encoding.

Para leer o escribir datos en un programa Java utilizando una codificación de caracteres diferente a la establecida en la propiedad file.encoding, se pueden emplear las clases Java de E/S java.io.InputStreamReader, java.io.FileReader, java.io.OutputStreamReader y java.io.FileWriter.

Estas clases permiten especificar un valor de file.encoding que tiene prioridad sobre la propiedad file.encoding predeterminada por la JVM.

[Actividad 14]

¿Qué hay que hacer para leer/escribir bien un fichero con una codificación distinta a la de por defecto?

En java Existen varios tipos de flujos que admiten como parámetro del constructor la codificación con la cual han de interpretar los bytes que atraviesan el mismo, como por ejemplo FileInputStream , FileOutputStream, InputStreamReader, OutputStreamWriter..

[Actividad 15]

¿Qué es la serialización?

Es un proceso que permite convertir un objeto en una secuencia de bytes, para poder almacenarlos o transmitirlos a otro programa o a una base de datos y que permite la posterior recuperación del objeto (deserialización).

En Java se realiza implementando en la clase a serializar la interfaz Serializable, lo que permite que sus objetos sean convertidos en una secuencia de bytes utilizando el mecanismo de ObjectOutputStream. Así mismo, permite que sean deserializados mediante el mecanismo ObjectInputStream.

[Actividad 16]

¿Qué dos Stream se usan para la serialización?

Como mencioné en el ejercicio anterior, en java es posible serializar objetos a través de un ObjectOutputStream y deserializarlos mediante un ObjectInputStream.

[Actividad 17]

¿Qué tiene que cumplir una clase para ser serializable?

Se debe implementar en la clase a serializar la interfaz Serializable, si esta clase contiene objetos o colecciones de objetos en composición, estos también deben de implementar esta interfaz, o bien estos, junto con cualquier atributo que no deseamos que se incluyan en el proceso de serialización, deben presentar el modificador “transient” en su declaración.

Si cumple con los requisitos anteriormente mencionados, se puede hacer uso de ObjectOutputStream para serializar instancias de objetos de la clase en cuestión.

[Actividad 18]

¿Qué se necesita para deserializar?

Debe cumplir los requisitos mencionados en el ejercicio anterior, pero en adición, la clase mediante la cual se serializaron ha de estar bien definida en nuestro programa y que la clase no han de haber sido modificados desde su serialización, entonces podremos emplear un ObjectInputStream para deserializar las instancias de los objetos serializados.

[Actividad 19]

¿Qué sucede con los objetos serializados si su clase cambia?

Si se modifica la clase, dependiendo del cambio, podría afectar a la deserialización de los objetos, si se añaden atributos, a no ser que añadamos a su declaración el modificador “transient”, lanzará el error ‘InvalidClassException’, si eliminamos uno de los atributos, si este no era originalmente “transient”, lanzará el mismo error. Si por otro lado modificamos los métodos, no afectará a la deserialización, pero los objetos serán manejados conforme a los métodos actuales, no conforme a los cuales poseían guando fueron serializados.

Para asegurarnos de que los objetos se serializan y deserializan de forma adecuada, podemos adoptar diferentes estrategias o enfoques, tales como implementar en nuestra clase el atributo “‘serialVersionUID” para controlar la compatibilidad de las versiones de los objetos o implementar los métodos “readObject ” y “writeObject” de forma personalizada sobreescribiendo los predeterminados de las clases ObjectOutputStream y ObjectInputStream respectivamente para que se adapten a los requisitos de nuestro programa.

[Actividad 20]

¿Qué sucede si se mantiene el número de versión de serialización y se cambia los atributos de la clase?

Los objetos se podrán deserializar, pero sin embargo, si hemos añadido atributos que no constaban en la clase en el momento en el cual fueron serializados, si estos son declaraciones para objetos (incluidos Strings) se iniciarán en null en la deserialización y si son declaraciones de enteros, booleanoss… se iniciarán en 0 ó false respectivamente…
Si por otro lado modificamos los métodos, tampoco afectará a la deserialización, pero los objetos serán manejados conforme a los métodos actuales, no conforme a los cuales poseían guando fueron serializados.