UNIVERSIDAD CATOLICA DE COLOMBIA
FACULTAD DE INGENIERIA DE SISTEMAS

CURSO: JAVA BASICO
PROFESOR: EMERSON CASTAÑEDA SANABRIA

TEMA: Entrada y Salida

OBJETIVOS:

- Aprender a utilizar los flujos para realizar operaciones complejas de Entrada y Salida.
- Estudiar la jerarquía de clases del paquete java.io.
- Aprender a realizar lectura y escritura de datos y objetos en archivos, como primera técnica de almacenamiento para aplicaciones que manejen información persistente.

CONTENIDO:

1. Flujos
2. Clases de java para Entrada y Salida
3. Lectura y Escritura de Archivos
4. Lectura y Escritura de Objetos en Archivos

DESARROLLO:

1. Flujos

La entrada y salida en java se basa en el uso de flujos. Los flujos son secuencias de bytes que viajan desde un origen a un destino a través de una vía de comunicación. Cuando un programa escribe en un flujo es el origen de este. Cuando lee desde un flujo, es el destino de éste. La vía de comunicación depende del tipo de E/S que se realice y puede consistir en trasferencias de memoria a memoria, sistemas de archivos, redes y otras formas de E/S.

Los flujos no representan una cuestión compleja. Su potencia reside en la capacidad de abstraer los detalles de la vía de comunicación desde las operaciones de entrada y salida. Esto permite llevar a cabo procesos de entrada y salida mediante un conjunto habitual de métodos. Estos métodos pueden modificarse y ampliarse para que proporcionen una mayor capacidad de E/S.

2. Clases de java para Entrada y Salida

Java define dos clases principales de flujos: InputStream y OutputStream. A partir de estos flujos pueden establecerse subclases a fin de que proporcionen diversas funciones de E/S.

A continuación se muestra la distribución de parte de la jerarquía de clases del paquete java.io:

```
class java.io.InputStream
      class java.io.ByteArrayInputStream
      class java.io.FileInputStream
      class java.io.FilterInputStream
            class java.io.BufferedInputStream
            class java.io.DataInputStream (implements java.io.DataInput)
            class java.io.LineNumberInputStream
            class java.io.PushbackInputStream
      class java.io.ObjectInputStream (implements java.io.ObjectInput,
      java.io.ObjectStreamConstants)
      class java.io.PipedInputStream
      class java.io.SequenceInputStream
      class java.io.StringBufferInputStream
class java.io.OutputStream
      class java.io.ByteArrayOutputStream
```

```
class java.io.FileOutputStream
class java.io.FilterOutputStream
      class java.io.BufferedOutputStream
      class java.io.DataOutputStream (implements java.io.DataOutput)
      class java.io.PrintStream
class java.io.ObjectOutputStream (implements java.io.ObjectOutput,
java.io.ObjectStreamConstants)
class java.io.PipedOutputStream
```

Las clases InputStream y OutputStream poseen subclases complementarias. Por ejemplo ambas tienen subclases para realizar procesos de entrada y salida por medio de Buffers de memoria, archivos y canales. Las subclases de InputStream se encargan de la entrada y las subclases de OutputStream de la salida.

La clase InputStream

La clase InputStream es una clase abstracta que constituye los cimientos da la jerarquía de la clase Input de java. Como tal, proporciona métodos que son heredados pos todas las clases InputStream.

- El metodo read()

El metodo read() es el mas importante de la jerarquía de clases InputStream. Lee un byte de datos desde un flujo de entrada y se bloquea si no hay datos disponibles. Cuando se bloquea un método, hace que el hilo en el que se esta ejecutando espere hasta que estén disponibles los datos. Esto no se supone un problema en programas divididos en múltiples hilos. El método read() adopta diversas formas sobrecargadas. Pueden leer un único byte o un arreglo de bytes en función de la forma que haya adoptado. Devuelve el número de bytes leídos o -1 si se encuentra un final de archivo sin haber leído ningún dato de éste.

- El método available()

El método available() devuelve el numero de bytes que pueden leerse sin ocasionar un bloqueo. Se utiliza para examinar el flujo de entrada y averigua la cantidad de datos disponibles. Sin embargo, en función del tipo de flujo de entrada quizás no de el resultado esperado o no sea util, razon por la cual no es buena idea fiarse ciegamente de este método para realizar procesos de entrada.

- El método close()

El método close() cierra un flujo de entrada y libera los recursos asociados a éste. En todos los casos es buena costumbre el cerrar un flujo para garantizar la finalización correcta de los procesos.

- El método skip()

El método skip() omite un numero especifico de bytes de entrada. Como parámetro utiliza un valor long.

La clase OutputStream

La clase OutputStream es una clase abstracta que constituye la base de la jerarquía de flujo de salida. Proporciona un conjunto de métodos que son análogos a los métodos de InputStream.

- El método write()

El método write() permite escribir bytes en el flujo de salida. Proporciona tres formas sobrecargadas para escribir un único byte, un arreglo de bytes o un fragmento de u arreglo. El método write(), al igual que el método read(), puede bloquearse en el momento en que intenta escribir en un flujo. El bloqueo pone en modo de espera al hilo que esta ejecutando el método write() hasta que no termina la operación de grabación.

El método flush()

El método flush() hace que todo dato almacenado en buffer se escriba inmediatamente en el flujo de salida. Algunas clases de OutputStream soportan la función de almacenamiento provisional y anulan este método a fin de vaciar sus buffers y escribir todos los datos almacenados provisionalmente en el flujo de salida.

El medodo close()

Por lo general es mas importante cerrar los flujos de entrada que los de salida, de modo que cualquier dato que se escribe en el flujo se almacena antes de designarlo y perderlo. El método close() de OutputStream se utiliza del mismo modo que InputStream.

3. Lectura y Escritura de Archivos

Java permite la escritura y lectura de archivos mediante flujos con las clases File, FileDescriptor, FileInputStream y FileOutputStream. Además, soporta la E/S de acceso aleatorio o directo mediante las clases File, FileDescriptor y RandomAccessFile.

La clase File

La clase fie se utiliza para acceder a objetos de archivo y de directorio. Emplea los convenios de nomenclatura del sistema operativo local. La clase File agrupa estas convenciones mediante constantes de clase.

File incorpora constructores para crear archivos o directorios, estos constructores admiten rutas de acceso y nombre de archivo y directorio tanto absolutos como relativos.

La clase file proporciona un gran numero de métodos de acceso que pueden utilizarse para crear, borra y renombrar archivos, además de proporcionar acceso a la ruta y el nombre del archivo y determinar si un objeto File es un archivo o un directorio. Asimismo estos métodos compruebas los permisos de acceso para lectura y escritura.

La clase FileDescriptor

La clase FileDescriptor permite acceder a los descriptores de un archivo mantenidos por los sistemas operativos para acceder a archivos o directorios. Esta clase no permite conocer la información específica que mantiene el sistema operativo. Proporciona un único método, valid(), que se emplea para determinar si un objeto de descriptor de archivo es valido en su momento.

La clase FileInputStream

La clase FileInputStream permite leer los datos de un archivo en forma de flujo. Los objetos de la clase FileInputStream se crean utilizando como argumento una cadena de nombre de archivo o un objeto File o FileDescriptor. FileInputStream anula los métodos de la clase InputStream y proporciona dos métodos nuevos, finalize() y getFD(). El método finalize() se emplea para cerrar un flujo cuando es procesado de retorno de recursos de java. El método getFD() se utiliza para obtener acceso al FileDescriptor que esta asociado a un flujo de entrada.

La clase FileOutputStream

La clase FileOutputStream permite escribir una salida en un flujo de archivo. Los objetos de la clase FileOutputStream se crean del mismo modo que los de la clase FileInputStream. Anula los métodos de la clase OutputStream y soporta los métodos finalize() y getFD() al igual que FileInputStream.

La clase RandomAccessFile

La clase RandomAccessFile ofrese la posibilidad de realizar operaciones de E/S directamente en posiciones especificas de un Archivo. El nombre “random access” se basa en el hecho de que es posible leer y escribir datos en ubicaciones aleatorias de un archivo en lugar de un flujo continuo de información. El método de

acceso aleatorio lo soporte el método seek(), que permite asignar una ubicación arbitraria al puntero correspondiente a la posición actual del archivo. RandomAccessFile implementa las interfaces DataInput y DataOutput. Esto ofrece la posibilidad de realizar operaciones de entrada y salida utilizando todo los objetos y tipo des datos primitivos.

La clase RandomAccessFile también soporta la posibilidad de establecer permisos de básicos de lectura y escritura de archivos, mediante los cuales autoriza acceso de solo lectura o solo escritura. Para ellos se transfiere un argumento r o rw de modo al constructor de RandomAccessFile, con el que se indica el tipo de acceso al archivo.

RandomAccessFile introduce varios métodos nuevos, además de los que hereda de Object e implementa de DataInput y DataOutput. Estos métodos incluyen seek(), getFielPointer() y length(). Los anteriores métodos asignan al puntero al archivo una ubicación específica, devuelven la posición actual del puntero y la longitud del archivo en bytes respectivamente.

4. Lectura y Escritura de Objetos en archivos

Con el fin de aprovechar las características de un lenguaje orientado a objetos, en todo su esplendor, evitando así hacer híbridos en el almacenamiento de los objetos persistentes (combinar objetos con relacional), es decir, en lugar de almacenar registros equivalentes a los datos encapsulados en un objeto y luego recuperarlos para crear una instancia del mismo. Existe la posibilidad de almacenar los objetos tal cual como se encuentren en determinado momento y posteriormente hacer la recuperación de los mismos sin hacer esfuerzos adicionales en conversiones intermedias para llevarlos a una representación relacional, es de esta manera como: guardamos objetos y recuperamos objetos.

Las clases que intervienen para conseguir este objetivo, son las siguientes: FileOutputStream, ObjectOutputStream para la escritura de objetos y FileInputStream, ObjectInputStream para la recuperacion de objetos previamente almacenados en un archivo. Otro elemento que hace parte del procedimiento es la Interface Serializable del paquete java.io. esta posibilita a las clases que la implementan, su representación como cadenas de bytes para asi ser almacenados y recuperados como información binaria.

A continuación se presenta un ejemplo sencillo de cómo es posible llevar a cabo estas operaciones:

Como primera medida es necesario que las clases de los objetos a almacenar implementen la interface Serializable, a continuación un ejemplo con una clase llamada ObjGuardable.

```
class ObjGuardable implements Serializable{
	private int i;
	private float f;
	private boolean b;
	private String str1;
	public static String str2;

	ObjGuardable(){
		i=10;f=5.5f;b=true;str1="abc";
		str1=str2;

	}

	ObjGuardable(int p1,float p2,boolean p3, String p4){
		i=p1;f=p2;b=p3;str1=p4;
		str2=str1;
	}

	public String toString(){
		String aux= "i="+i+" f=" +f+" b="+b+" str1="+str1+" str2="+str2;
		return aux;
	}
}
```

Como se puede apreciar la clase ObjGuardable implementa la interface Serializable , lo que permite que sus instancias se puedan representar como cadena de bytes. De tal manera ya se cuenta con objetos con capacidades para ser representados como una cadenas de bytes y así posteriormente ser almacenados y recuperados en un archivo de datos.

A continuación la clase LeeEscObjetos, una clase que utiliza intancias de la clase ObjGuardable junto con las clases mencionadas, para llevar a cabo el almacenamiento y la recuperación de objetos en un archivo.

```
import java.io.*;

public class LeeEscObjetos
{
	public static void main(String arg[]){

		ObjGuardable og1= new ObjGuardable();
		ObjGuardable og2= new ObjGuardable(21,23.3f,true,"cadena");

		try{
			FileOutputStream ostream = new FileOutputStream("t.tmp");
			ObjectOutputStream oos = new ObjectOutputStream(ostream);

			oos.writeObject(og1);
			oos.writeObject(og2);
			oos.flush();
			ostream.close();

			FileInputStream istream = new FileInputStream("t.tmp");
			ObjectInputStream ois = new ObjectInputStream(istream);

			ObjGuardable ogr1 = (ObjGuardable)ois.readObject();
			ObjGuardable ogr2 = (ObjGuardable)ois.readObject();

			istream.close();

			System.out.println("1er objeto almacenado: "+og1);
			System.out.println("2do objeto almacenado: "+og2);
			System.out.println("1er objeto recuperado: "+ogr1);
			System.out.println("2do objeto recuperado: "+ogr2);
		}
		catch(Exception e){
			e.printStackTrace();
		}

	}
}
```

Básicamente la clase LeeEscObjetos crea dos instancias de la clase ObjGuardable, los objetos og1 y og2. Luego procede a la creación de un flujo de salida al archivo t.tmp que se le asocia a un objeto de la clase ObjecOutputStream, objeto mediante el cual se realiza la escritura de las instancias de la clase ObjGuardable en el archivo.

El segundo paso consiste el la creación de un flujo de entrada desde el archivo t.tmp, asociado a una instancias de la clase ObjectInputStream, a través de la cual se recuperan instancias de la clase ObjGuardable previamente almacenadas en dos nuevas instancias de la misma clase, llamadas ogr1 y ogr2. Para terminar se comprueba la información encapsulada en cada uno de los objetos, imprimiendo la información en la consola.