Centro Asociado Palma de Mallorca

Introducción Práctica de Programación Java

Antonio Rivero Cuesta

Sesión

XI

Entrada y Salida

El paquete **`java.io`** contiene casi todas las clases que podamos necesitar para la entrada y salida de datos.

Vamos a utilizar los **`streams`**, ***flujos***, para poder leer y escribir datos en los archivos.

Un **`stream`** puede ser definido como una secuencia de datos.

Un **`stream`** trata la comunicación de información entre el programa y el exterior.

La clase **`InputStream,`** flujo de bytes de entrada, se utiliza para leer datos de una fuente.

El **`OutputStream`** se utiliza para escribir datos en un destino.

Los flujos se pueden utilizar solos o combinados.

Para trabajar con los flujos de datos tanto de entrada como de salida hacemos lo siguiente:

Entrada de Datos

Se crea un flujo de datos de entrada.

Se leen datos con los métodos apropiados.

Se cierra el flujo de datos.

Salida de Datos

Se crea un flujo de datos de salida.

Se escriben datos utilizando los métodos apropiados del objeto flujo.

Se cierra el flujo de datos.

Flujo

Es un objeto que se utiliza para realizar una entrada o salida de datos.

Representa un canal de información del que se puede leer o escribir datos de forma secuencial.

Existen dos tipos de flujos en Java:

- Los que utilizan bytes.
- Los que utilizan caracteres.

Flujos de Entrada	Flujos con bytes	Flujos con caracteres
	InputStream ByteArrayInputStream FileInputStream FilterInputStream BufferedInputStream DataInputStream LineNumberInputStream PushBackInputStream ObjectInputStream PipedInputStream SequenceInputStream StringBufferInputStream	Reader BufferedReader LineNumberReader CharArrayReader FilterReader PushBackReader InputStreamReader FileReader PipedReader StringReader

Flujos de Salida	Flujos con bytes	Flujos con caracteres
	OutputStream ByteArrayOutputStream FileOutputStream FilterOutputStream BufferedOutputStream DataOutputStream PrintStream ObjectOutputStream PipedOutputStream	Writer BufferedWriter CharArrayWriter FilterWriter OutputStreamWriter FileWriter PipedWriter PrintWriter StringWriter

Java proporciona un flujo para la entrada predeterminada llamado **`System.in`** que el sistema abre al empezar la ejecución del programa.

Este flujo lee, por defecto, del teclado.

Así mismo se dispone del flujo **`System.out`** para la salida predeterminada.

Este flujo escribe, por defecto, en la pantalla, en la consola de Java.

Ambos flujos predeterminados son flujos de bytes.

Scanner

Una de las utilidades de la clase Scanner es la obtención de datos tecleados.

La clase Scanner se encuentra en el paquete **`java.util`**.

```
import java.util.Scanner;
```

Tenemos que crear un objeto de la clase Scanner asociado al dispositivo de entrada.

Si el dispositivo de entrada es el teclado escribiremos:

```
Scanner sc = new Scanner(System.in);
```

Se ha creado el objeto **sc** asociado al teclado representado por **System.in**

Una vez hecho esto podemos leer datos por teclado.

Para leer podemos usar el método

`nextXXX()`

donde XXX indica en tipo.

Por ejemplo nextInt() para leer un entero, nextDouble() para leer un double, etc.

Métodos
`nextByte()`
`nextDouble()`
`nextFloat()`
`nextInt()`
`nextLong()`
`nextShort()`
`next()`
`nextLine()`

Ejemplos

```
byte b = teclado.nextByte();
double d = teclado.nextDouble();
float f = teclado.nextFloat();
int i = teclado.nextInt();
long l = teclado.nextLong();
short s = teclado.nextShort();
String p = teclado.next();
String o = teclado.nextLine();
```

File

La clase **`File`** no sirve para leer ni para escribir en un archivo.

Permite entre otras operaciones:

- Obtener el tamaño del archivo.
- Obtener el nombre completo, incluida la ruta.
- Cambiar el nombre.
- Eliminar el nombre.
- Saber si es un directorio o un archivo.
- Si es un directorio, obtener la lista de los archivos y directorios que contiene.
- Crear un directorio.

getPath()

Devuelve la ruta con la que se creó el objeto **`File`**. Puede ser relativa o no.

getAbsolutePath()

Devuelve la ruta absoluta asociada al objeto **`File`**.

getCanonicalPath()

Devuelve la ruta única absoluta asociada al objeto **`File`**.

Puede haber varias rutas absolutas asociadas a un **`File`** pero solo una única ruta canónica.

Lanza una excepción del tipo **`IOException`**.

Filtro

Sirven para que el método **`list`** devuelva solo aquellos archivos o carpetas que:

- Cumplan una determinada condición.
- Tengan una extensión determinada.
- Contengan en su nombre una cadena determinada.
- Empiecen por...

Un filtro es un objeto de una clase que implementa el **`interface FilenameFilter`**.

`FilenameFilter` tiene un solo método llamado **`accept`** que devuelve un valor de tipo **`boolean`**:

```
public interface FilenNameFilter{
    boolean accept (File ruta,String nombre);
}
```

El método recibe el directorio donde se encuentra el archivo (objeto **`File`**) y el nombre del archivo (**`String`**).

Este método lo utiliza el método **`list`** de **`File`** para decidir si un archivo o directorio determinado se incluye o no en el **`array`** que devuelve.

Si **`accept`** devuelve true se incluye y si devuelve false no se incluye.

El método **`list`** llama de forma automática al método **`accept`** para cada uno de los archivos o directorios.

Ejemplo de creación y uso de un filtro:

Vamos a crear un filtro para obtener todos los archivos que tiene una extensión determinada.

Como es un objeto de una clase que implementa el interface **`FileNameFilter`**, tenemos que crear esta clase.

La clase se llamará **`Filtro`** y debe implementar el método **`accept`** de **`FilenameFilter`**.

En este caso como queremos saber si un archivo tiene una determinada extensión el método **`accept`** lo podemos escribir utilizamos el método **`endsWith`** de **`String`**.

Ejemplo:

- Filtro

FileReader y FileWriter

Permiten leer y escribir en un fichero.

Podemos abrir un fichero de texto para leer usando la clase **`FileReader`**.

Esta clase tiene métodos que nos permiten leer caracteres.

`FileReader` no contiene métodos que nos permitan leer líneas completas, pero sí **`BufferedReader.`**

Podemos construir un **BufferedReader** a partir del **FileReader** de la siguiente forma:

```
File archive = new File("C:\\archivo.txt");
FileReader fr = new FileReader (archivo);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

Como opción para leer un fichero de texto línea por línea, podría usarse la clase **Scanner.**

Lo primero que debemos hacer es importar estas clases y las que controlan las excepciones.

Después debemos crear un objeto de alguna de estas clases, que deben estar dentro de un **`try-catch`.**

Es importante controlar las excepciones.

Cuando creamos un objeto, abrimos un **`stream`** entre nuestro programa y el exterior.

Cuando debemos de usarlo debemos cerrar el **`stream`** con el método **`close()`.**

Escribimos la ruta del fichero.

Si usamos **`FileWriter`** y se escribe una ruta de fichero que no existe lo crea.

Para **`FileReader`** si que debe existir el fichero, sino lanzara una excepción.

Usamos doble barra \\ porque es un carácter de escape para poner \.

Para Escribir

Usaremos el método **`write`** de **`FileWriter`**.

Este método puede usar como parámetro:

- Un **`String`** con lo que queremos escribir.
- Un número que se corresponderá un carácter de la tabla ASCII.

Para Leer

Usaremos el método **`read`** de **`FileReader.`**

Este método no tiene parámetros pero devuelve un número que si le hacemos un casting a **`char`** este será legible por nosotros.

Esto lo podemos mostrar por pantalla o incluso pasarlo a otro fichero, crear otro objeto.

Cuando se termina el fichero, el método **`read`** devuelve **−1,** indicando que no hay más caracteres**.**

BufferedReader

Las clases **`BufferedReader`** y **`BufferedWriter`** las podemos encontrar en **`java.io`**.

Tienen la misma función que **`FileReader`** y **`FileWriter`**.

Leer y escribir en ficheros.

Pero optimizan estas funciones.

Si sólo usamos:

- FileInputStream
- FileOuputStream
- FileReader
- FileWriter

Cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro.

Si escribimos o leemos pocos caracteres cada vez el proceso se hace costoso y lento con muchos accesos a disco duro.

- BufferedReader
- BufferedInputStream
- BufferedWriter
- BufferedOutputStream

Añaden un buffer intermedio.

Cuando leamos o escribamos, esta clase controlará los accesos a disco.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido.

La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

Únicamente posee el método **`readLine()`** para leer la entrada y este siempre devuelve **`String.`**

Para obtener un número debemos leer primero como **`String`** y luego enviarle dicho **`String`** al método **`parseInt()`** de la clase **`Integer`**.

Se crean así:

```
FileWriter fw = new FileWriter("D:\\fichero1.txt");
FileReader fr = new FileReader("D:\\fichero1.txt");

BufferedReader br = new BufferedReader(fr);
BufferedWriter bw = new BufferedWriter(fw);
```

Otra forma de crearlos:

```
BufferedReader br = new
BufferedReader(new
FileReader("D:\\fichero1.txt"));

BufferedWriter bw = new
BufferedWriter(new
FileWriter("D:\\fichero1.txt"));
```

La ventaja de los buffered es que **`BufferedReader`** nos permite leer una línea completa, en lugar de carácter a carácter como hacia **`FileReader`**, cuando el fichero termina, devuelve **`null`**, no un -1 como en **`FileReader`**.

Con BufferedWriter también podemos añadir una línea, como si pulsáramos un Enter.