GESTIÓN DE FICHEROS

Módulo: Acceso a datos

Ciclo: Desarrollo de Aplicaciones Multiplataforma

Planificación

- Del 23 de septiembre al 3 de octubre → Realización de ejercicios resueltos
- Del 7 al 15 de octubre → Realización del proyecto
- Día 20 de octubre → Entrega del proyecto UD1
- Día 17 de octubre (jueves) → Examen UD 1

Sep.	9	10	11	12	13	14	15
	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
	30	1	2	3	4	5	6
Oct.	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
	28	29	30	31	1	2	3

Introducción

- Los ficheros permiten hacer **persistente** la información en el sistema.
- Para la gestión de ficheros usaremos el lenguaje **Java**.
- Veremos cómo crear, actualizar y procesar ficheros con Java.
- Las operaciones de E/S en Java las proporcionan dos librerías: **java.io** y **java.nio**
- **Definición de operación de E/S**? ¿?

Introducción

- Los ficheros permiten hacer **persistente** la información en el sistema.
- Para la gestión de ficheros usaremos el lenguaje **Java**.
- Veremos cómo crear, actualizar y procesar ficheros con Java.
- Las operaciones de E/S en Java las proporcionan dos librerías: **java.io** y **java.nio**
- **Definición de operación de E/S**? Aquellas operaciones que constituyen un flujo de información entre el programa y el exterior.

Librería java.io

- Contiene las clases necesarias para gestionar las operaciones de entrada y salida con Java.
- Estas clases de E/S las podemos agrupar fundamentalmente en:
  - Clases para **leer entradas desde** un flujo de datos.
  - Clases para **escribir entradas a un** flujo de datos.
  - Clases para **operar con ficheros en el sistema** de ficheros local.
  - Clases para **gestionar la serialización** de objetos.

Clase File

- La clase File permite representar de forma abstracta ficheros y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows…
- Las instancias de la clase File representan nombres de archivos, no los archivos en sí mismos.

Clase File

- Para representar un fichero o directorio, tenemos que crear un objeto *File*. En el constructor debemos indicar la ruta en la que se encuentra el fichero o directorio:

```
File fichero = new File("/home/Escritorio/fichero.txt");
```

Clase File

- Una vez que ya tenemos creado el objeto, podemos utilizar los diferentes métodos de la clase. Algunos de los más importantes son:
- **createNewFile():** crea un nuevo fichero. Devuelve *true* si se ha podido crear.
- **delete():** elimina un fichero o directorio. Si es un directorio, debe estar vacío.
- **mkdir():** crea un directorio. Devuelve *true* si se ha podido crear.
- **exists():** comprueba si un fichero o directorio existe.
- **renameTo()**: permite renombrar el archivo

Clase File

- **getName():** devuelve el nombre del fichero o directorio.
- **length():** devuelve el tamaño en bytes del fichero.
- Lista el contenido del directorio.
  - **list()**: devuelve un vector de String con los nombre de los archivos
  - **listFiles()**: devuelve un vector de objetos de tipo File.
  - **listRoots**(): devuelve los nombre de archivos de la raíz.
- Puedes consultar todos sus métodos en la documentación: Enlace

Interfaz FilenameFilter

- La interfaz FilenameFilter se puede usar para crear filtros que establezcan criterios de filtrado sobre una serie de ficheros.
- Para usar un filtro, debemos crear una clase que implemente la interfaz (*implements FilenameFilter*). En esta clase debemos implementar el siguiente método:

```
public boolean accept(File dir, String name)
```

- El parámetro *dir* es el directorio en el que se encuentra el fichero. El parámetro *name* es el nombre del fichero.

Interfaz FilenameFilter

- El método ***accept*** recibe el nombre de un fichero y devuelve ***true*** si cumple un criterio determinado. En caso contrario devolverá *false*.

- Por ejemplo, si queremos filtrar sólo aquellos ficheros con extensión *.txt*:

```
public boolean accept(File dir, String name) {
        return name.endsWith(".txt");
}
```

- Devuelve *true* solo para aquellos ficheros de texto que se encuentren en el directorio *dir*.

Interfaz FilenameFilter

- Para poder filtrar los ficheros de un directorio, podemos usar el método *list()*, al que le debemos pasar un objeto de la clase que implementa a la interfaz *FilenameFilter*:

```
File directorio = new File("/home/Escritorio");
String[] ficherosDeTexto = directorio.list(new FiltrarExtension(".txt"));
```

- La variable *ficherosDeTexto* es un array que almacena los nombres de todos los ficheros del directorio que tienen extensión *.txt*.
- Puedes consultar la documentación de la interfaz: https://docs.oracle.com/javase/7/docs/api/java/io/FilenameFilter.html

File.separator

- Linux y windows utilizan diferentes caracteres para separar las rutas de los archivos:
  - Linux: /
  - Windows: \\
- Si se quisiese construir una ruta que funcionase en cualquier sistema, en lugar de utilizar los caracteres indicados anteriormente se utilizaría:

File.separator

Ej.: new File (rutaRelativa + File.separator + “nombrefichero.txt”)

Flujos o *streams*

- En Java, las operaciones de entrada/salida (E/S) se hacen a través de un flujo o *stream*.
- Un **flujo** es una abstracción de todo aquello que produce o consume información.

  Es decir, es una abstracción que nos permite comunicar información entre una fuente y un destino (en disco duro, memoria, red…).
- Cualquier programa que necesite obtener o enviar información, necesita abrir un *stream*.

Flujos o *streams*

- En Java hay dos tipos de flujos:
  - **Byte streams** (8 bits):
    - proporciona lo necesario para la gestión de entradas y salidas de bytes.
    - Uso está orientado a la lectura y escritura de datos binarios.
    - Utiliza las clases abstractas **InputStream** y **OutputStream**, donde destacan los métodos read() y write().
  - **Character streams** (16 bits):
    - Permiten manejar flujos de caracteres Unicode.
    - Utilizan con las clases abstractas **Reader** y **Writer**, destacan los métodos read() y write().

Byte streams

- Estos flujos nos permiten leer y escribir bytes en un fichero.
- Existen varias clases que heredan de InputStream y OutputStream, pero las que nos interesan son **FileInputStream** y **FileOutputStream**.
- Estas clases permiten trabajar con **archivos binarios**.
- FileInputStream: https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html
- FileOutputStream: https://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html

Métodos de InputStream

- **int read()**:
  - Lee el siguiente byte y lo devuelve como entero
  - Devuelve: -1 (Si llega al final)
- **int read(byte[] buffer):**
  - Lee byte a byte y lo almacena en el buffer.
  - No se puede garantizar el mínimo número de bytes leídos
  - El máximo de bytes será el tamaño del buffer.
  - Devuelve: número de bytes leídos.
- **int read(byte[] buffer, int offset, int len):** similar al método anterior. Intenta leer *len* bytes y los almacena en buffer a partir de la posición *offset*

Métodos de InputStream

- **int available():**
  - Devuelve el número de bytes disponibles para leer en el flujo de entrada.
  - Método informativo ⇒ no garantiza que en el momento de la lectura las condiciones no hayan cambiado.
- **long skip (long bytesToSkip):**
  - Salta y descarta tantos bytes como indique el parámetro *bytesToSkip*.
  - **No hay garantía** de que se salten exactamente *bytesToSkip*.
  - Devuelve: número real de bytes saltados.
  - Si *bytesToSkip* <= *0*, no se producirá ningún salto.
- **void close():** cierra el flujo

Métodos de OutputStream

- **void write(int byte)**: Escribe el byte en el fichero
- **void write(byte[] data):** Escribe todos los bytes del vector *data* en el fichero.
- **void write(byte[] data, int offset, int len):** escribe *len* bytes del vector *data* a partir de la posición *offset*
- **void flush()**: vacía la secuencia de salida forzando la escritura de los bytes del buffer.
- **void close()**: cierra el flujo.

Decoradores

- InputStream y OutputStream están muy bien pero un poco cortas de funcionalidades
- Dos opciones:
  - Incorporar nuevas utilidades a las clases anteriores <= Opción “más fácil”
  - Decoradores <= Opción que eligió la gente de Java
- **Decorador**: Consiste en envolver dos o más objetos (uno dentro de otro) para **aportar** cierta **funcionalidad extra** diferente a la original.
- Ventaja: crear diferentes instancia en función de lo que se necesite.

Decoradores

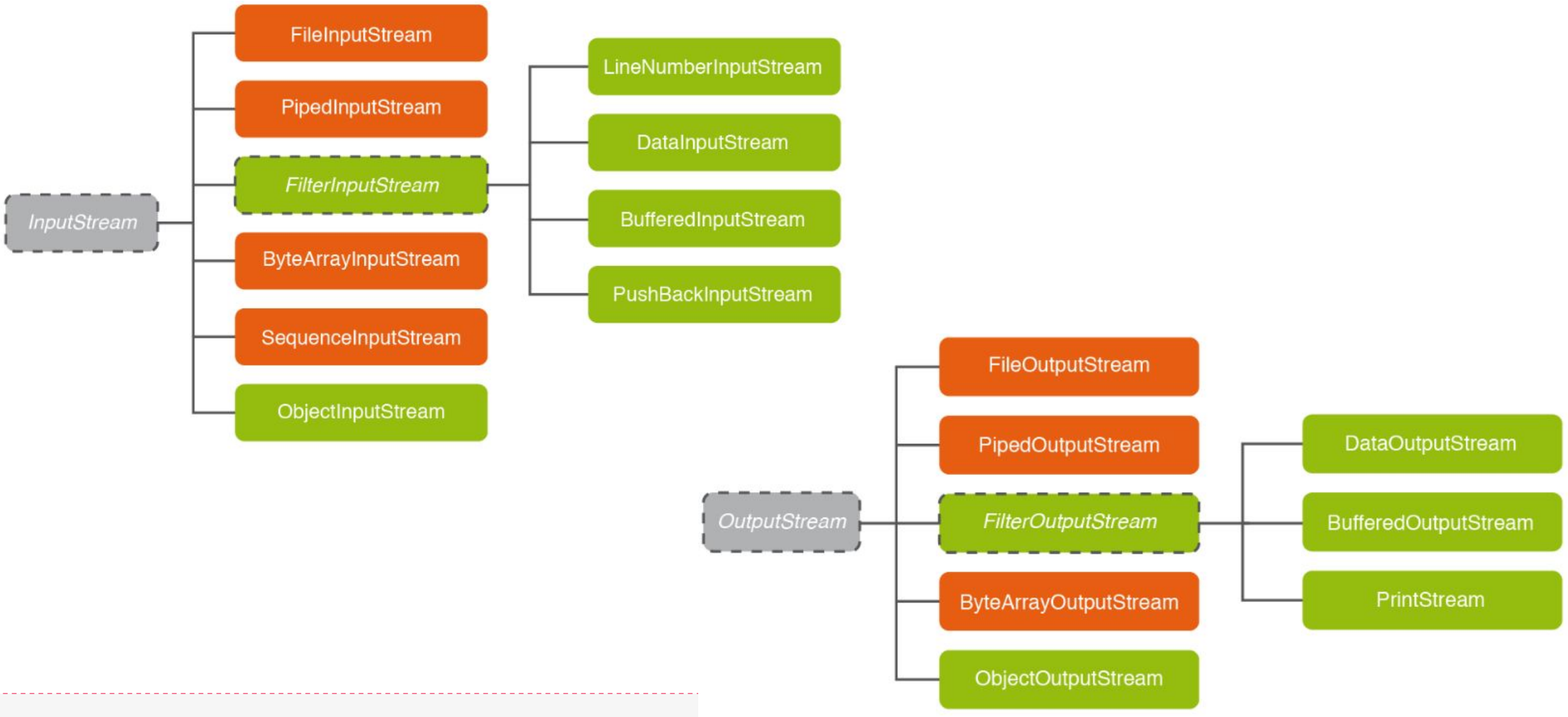
- Ejemplo:
  - Quiero escribir en un fichero ⇒ FileOutpuStream
  - Tamén querría tener un buffer para agilizar la transferencia ⇒ BufferedOutputStream
  - Ya que estamos, quiero que convierta datos básicos en bytes ⇒ DataOutputStream
- Solución:

```
FileOutputStream fundido =  new  FileOutputStream(new File(fichero)) ;
BufferedOutputStream bos =  new  BufferedOutputStream(fundado);
DataOutputStream streamFinal =  new  DataOutputStream(bos);
```

o bien:

```
DataOutputStream streamFinal =  new  DataOutputStream(
    new BufferedOutputStream(new FileOutputStream (new File(fichero)))
);
```

Decoradores



Objetos Serializable

- Si queremos guardar un objeto y sus atributos en un fichero, podemos utilizar la interfaz *Serializable*.
- Esta interfaz no tiene métodos ni atributos. Simplemente sirve para indicar que un objeto es “serializable”.
- Usaremos las clases **ObjectInputStream** y **ObjectOutputStream**.
- Atributo *SerialversionUID*:
  - permite diferenciar versiones de objetos. Permite serializar en una versión y deserializar en otra.
  - tiene que ser privado.

Objetos Serializable: Leyendo objetos del pasados

- Ejemplo de una agenda:
  - Ejecuto el programa, guardo un objeto Contacto (ObjectOutputStream) y fin de la ejecucción.
  - Vuelvo a realizar la misma operación pero con otro objeto Contacto.
  - Leo el contenido (ObjectInputStream) para pintarlo ⇒ ¿ERROR?
- ¿Cuál es el problema? Existencia de dos cabeceras.
  - *Dibuxiño*

Character streams

- Estos flujos nos permiten leer y escribir en un fichero, utilizando caracteres Unicode.
- Las clases **FileReader** y **FileWriter** nos permiten leer y escribir desde **ficheros de texto**.
- FileReader: https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html
- FileWriter: https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html

Character streams

- Dentro de las clases utilizadas para trabajar con flujos de caracteres tenemos:
  - FileInputStream, FileOutputStream
  - FileReader, FileWriter
  - BufferedReader, BufferedWriter
  - BufferedInputStream, BufferedOutputStream
  - InputStreamReader, OutputStreamReader ⇒ Adaptadores ⇒flujo orientado a bytes transformado en flujo orientado a caracteres

Formas de acceso a un fichero

- Hemos visto cómo trabajar con:
  - dos tipos de ficheros:
    - de texto
    - binarios
  - dos tipos de acceso a los ficheros:
    - secuencial
    - aleatorio

Acceso secuencial

- ¿?

Acceso secuencial

- Para acceder a una posición tenemos que recorrer todos los elementos anteriores.
- Se utilizan las clases Reader y Writer para el acceso a flujos de caracteres y texto general.
- Subclases: **FileReader** (lectura), **FileWriter** (escritura)
- Posibilidad de usar buffers:
  - BufferedWriter sobre un FileWriter
  - BufferedReader sobre un FileReader

Acceso aleatorio

- ¿?

Acceso aleatorio

- Para acceder a una posición no es necesario recorrer los elementos anteriores.
- Se utilizan las clase RandomAccessFile:

RandomAccessFile(File file, String mode)

RnadomAccessFile(String file, String mode)

- Donde, *mode* puede ser ‘r’ (solo lectura) o ‘rw’ (lectura y escritura)
- Operaciones de búsqueda:
  - Seek(long position)
  - skipBytes(int desplazamiento)

Seek y skipBytes mismo propósito pero skip más rápido y eficiente