PERSISTENCIA y SERIALIZACION

Fuentes:
1. Transparencias Darwin Durant. Persistencia de archivos
2. Lenguaje Java Avanzado

Introducción:

- Todas las aplicaciones realizadas hasta ahora obtenían los datos necesarios para su ejecución de la entrada estándar y visualizaban los resultados en la salida estándar.
- De esta forma los datos estarán disponibles mientras dure la ejecución de la aplicación. Si finalizamos la ejecución de la aplicación y la ejecutamos de nuevo, tendremos que volver a ingresar de nuevo todos los datos.
- La solución para hacer que los datos **persistan** de una ejecución a otra es almacenarlos en un archivo en el disco. Entonces cada vez que se ejecute la aplicación podrá leer el archivo almacenado en disco y recuperar los datos que requiera, los mismos que serán cargarlos en la memoria y podrán ser manipularlos.

Datos de una aplicación

- Como Java es un lenguaje orientado a objetos, la mayoría de aplicaciones tienen un **conjunto de clases de datos** (modelo de datos).En la memoria residen las instancias de estas clases de datos y los componentes de vista y controlador (interfaz de usuario).
- Para almacenar los datos de forma permanente una aplicación debe tomar el modelo de datos de la memoria y escribirlo en disco.
- Los datos almacenados en un **archivo o fichero (FILE)** son de manera permanente de modo que pueden ser manipulados en cualquier momento.

Índice

- Introducción a los flujos de E/S
- Entrada y salida estándar
- Acceso a ficheros
- Acceso a recursos
- Acceso a URLs
- Codificación de datos
- Serialización de objetos

Flujos de E/S

- Las aplicaciones muchas veces necesitan enviar datos a un determinado destino o leerlos de una determinada fuente
  - Ficheros en disco, red, memoria, otras aplicaciones, etc
  - Esto es lo que se conoce como E/S
- Esta E/S en Java se hace mediante flujos (*streams*)
  - Los datos se envían en serie a través del flujo
  - Se puede trabajar de la misma forma con todos los flujos, independientemente de su fuente o destino

Tipos de flujos según el tipo de datos

- Según el tipo de datos que transportan, distinguimos
  - Flujos de bytes (con sufijos `InputStream` y `OutputStream`)
  - Flujos de caracteres (con sufijos `Reader` y `Writer`)

- Superclases

	Entrada	Salida
Bytes	`InputStream`	`OutputStream`
Caracteres	`Reader`	`Writer`

Tipos de flujos según su propósito

- Distinguimos:
  - Canales de datos

    Simplemente llevan datos de una fuente a un destino

    Ficheros: `FileInputStream, FileReader, FileOutputStream, FileWriter`

    Memoria: `ByteArrayInputStream, CharArrayReader, ...`

    Tuberías: `PipedInputStream, PipedReader, PipedWriter, ...`

  - Flujos de procesamiento

    Realizan algún procesamiento con los datos

    Impresión: `PrintWriter, PrintStream`

    Conversores de datos: `DataOutputStream, DataInputStream`

    Bufferes: `BufferedReader, BufferedInputStream, ...`

Acceso a los flujos

- Todos los flujos tienen una serie de métodos básicos

Flujos	Métodos
`InputStream, Reader`	`read, reset, close`
`OutputStream, Writer`	`write, flush, close`

- Los flujos de procesamiento
  - Se construyen a partir de flujos canales de datos
  - Los extienden proporcionando métodos de más alto nivel, p.ej:

Flujos	Métodos
`BufferedReader`	`readLine`
`DataOutputStream`	`writeInt, writeUTF,...`
`PrintStream, PrintWriter`	`print, println`

Objetos de la E/S estándar

- En Java también podemos acceder a la entrada, salida y salida de error estándar
- Accedemos a esta E/S mediante flujos
- Estos flujos se encuentran como propiedades estáticas de la clase `System`

	Tipo de flujo	Propiedad
Entrada	`InputStream`	`System.in`
Salida	`PrintStream`	`System.out`
Salida de error	`PrintStream`	`System.err`

Salida estándar

- La salida estándar se ofrece como flujo de procesamiento `PrintStream`
  - Con un `OutputStream` a bajo nivel sería demasiado incómoda la escritura
- Este flujo ofrece los métodos `print` y `println` que permiten imprimir cualquier tipo de datos básico
  - En la salida estándar

```
System.out.println("Hola mundo");
```

  - En la salida de error

```
System.err.println("Error");
```

Flujos de ficheros

- Canales de datos para acceder a ficheros

	Entrada	Salida
Caracteres	`FileReader`	`FileWriter`
Binarios	`FileInputStream`	`FileOutputStream`

- Se puede acceder a bajo nivel directamente de la misma forma que para cualquier flujo

- Podemos construir sobre ellos flujos de procesamiento para facilitar el acceso de estos flujos

Lectura y escritura de ficheros

```
public void copia_fichero() {
     int c;
try {
  FileReader in = new FileReader("fuente.txt");
      FileWriter out = new FileWriter("destino.txt");
       while( (c = in.read()) != -1)
  {
    out.write(c);
  }
  in.close();
  out.close();
    } catch(FileNotFoundException e1)
      {  System.err.println("Error: No se encuentra el
      fichero");
  } catch(IOException e2) {
    System.err.println("Error leyendo/escribiendo fichero");
  }
}
```

Uso de flujos de procesamiento

```
public void escribe_fichero() {
      FileWriter out = null;
     PrintWriter p_out = null;
     try {
    out = new FileWriter("result.txt");
         p_out = new PrintWriter(out);
    p_out.println("Este texto será escrito
         en el fichero");
} catch(IOException e) {
    System.err.println("Error al escribir en
         el fichero");
} finally {
    p_out.close();
}
}
```

Sistema de ficheros

- La clase `File` contiene utilidades para trabajar con el sistema de ficheros
  - Constantes para indicar los separadores de directorios (‘/’ ó ‘\’) Hace las aplicaciones independientes de la plataforma
  - Crear, borrar o renombrar ficheros y directorios
  - Listar los ficheros de un directorio
  - Comprobar y establecer los permisos sobre ficheros
  - Obtener la ruta de un fichero
  - Obtener datos sobre ficheros (tamaño, fecha, etc)
  - Etc...

Acceso a recursos

- Los recursos incluidos en un JAR no se encuentran directamente en el sistema de ficheros
  - No podremos utilizar los objetos anteriores para acceder a ellos
- Accedemos a un recurso en el JAR con

```
getClass().getResourceAsStream("/datos.txt");
```

- Anteponiendo '/' se busca de forma relativa al raíz del JAR
- Si no, buscará de forma relativa al directorio correspondiente al paquete de la clase actual

URLs

- URL = *Uniform Resource Locator*
  - Cadena para localizar los recursos en Internet

- Se compone de

```
protocolo://servidor[:puerto]/recurso
```

  - P.ej. `http://www.ua.es/es/index.html`

    Se conecta al servidor `www.ua.es`

    A través del puerto por defecto (puerto `80`)

    Utilizando protocolo `HTTP` para comunicarse Solicita el recurso `/es/index.html`

URLs en Java

- Se encapsulan en la clase `URL`

```
URL url = new URL("http://www.ua.es/es/index.html");
```

- Es obligatorio especificar el protocolo
  - P.ej. `www.ua.es` es una URL mal formada
- Si la URL está mal formada se producirá una excepción `MalformedURLException`

```
try {
    URL url = new URL("http://www.ua.es/es/index.html");
} catch(MalformedURLException e)
    {  System.err.println("Error: URL mal
    construida");
}
```

Lectura del contenido

- Podemos leer el contenido de la URL abriendo un flujo de entrada con

```
InputStream in = url.openStream();
```

- Leeremos de este flujo de la misma forma que con cualquier otro flujo
  - Con los métodos a bajo nivel (byte a byte)
  - O utilizando un flujo de procesamiento
- P.ej, si la URL corresponde a un documento HTML obtendremos el código fuente de este documento

Codificación

- Podemos codificar de forma sencilla los datos para enviarlos a través de un flujo de bytes (en serie)
- Utilizaremos un flujo `DataOutputStream`

```
String nombre = "Jose";
 int edad = 25;

ByteArrayOutputStream
baos = new
ByteArrayOutputStream()
;

DataOutputStream dos = new DataOutputStream(baos);
 dos.writeUTF(nombre);
dos.writeInt(edad);

dos.close();
baos.close();
```

Descodificación

- Para descodificar estos datos del flujo realizaremos el proceso inverso
- Utilizamos un flujo `DataInputStream`

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
 DataInputStream dis = new DataInputStream(bais);

String nombre = dis.readUTF();
 int edad = dis.readInt();

dis.close();
bais.close();
```

Entrada/Salida de objetos

- Si queremos enviar un objeto a través de un flujo deberemos convertirlo a una secuencia de bytes
- Esto es lo que se conoce como *serialización*
- Java serializa automáticamente los objetos
  - Obtiene una codificación del objeto en forma de array de bytes
  - En este array se almacenarán los valores actuales de todos los campos del objeto serializado

Objetos serializables

- Para que un objeto sea serializable debe cumplir:
  - Implementar la interfaz `Serializable`

```
public MiClase implements Serializable {
    ...
}
```

Esta interfaz no obliga a definir ningún método, sólo marca el objeto como serializable

  - Todos los campos deben ser

Datos elementales u
Objetos
serializables

Flujos de objetos

- Para enviar o recibir objetos tendremos los flujos de procesamiento

  `ObjectInputStream`

  `ObjectOutputStream`

- Estos flujos proporcionan respectivamente los métodos

  `readObject`

  `writeObject`

- Con los que escribir o leer objetos del flujo
  - Utilizan la serialización de

¿Preguntas...?