Programación Orientada a Objetos

Tema 7-2:
Librería I/O

Contenidos

Tema 7-2: Librería I/O

1. LIBRERÍA I/O (ENTRADA/SALIDA)
2. FICHEROS, LA CLASE **`FILE`**
3. FICHEROS DE TEXTO
4. SERIALIZACIÓN DE OBJETOS

LIBRERÍA I/O

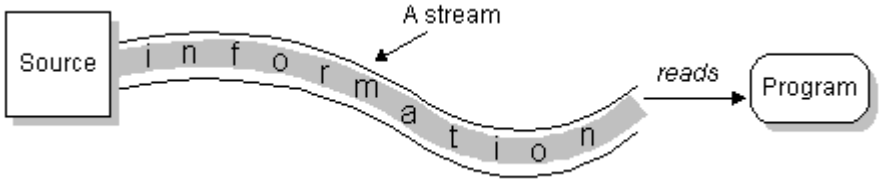
Normalmente las aplicaciones necesitan **leer o escribir información desde o hacia una fuente externa de datos**.

- La información puede estar en cualquier parte, en un fichero, en disco, en algún lugar de la red, en memoria o en otro programa.
- También puede ser de cualquier tipo: objetos, caracteres, imágenes o sonidos.
- La comunicación entre el origen de cierta información y el destino se realiza mediante un '**stream**' (flujo) de información.

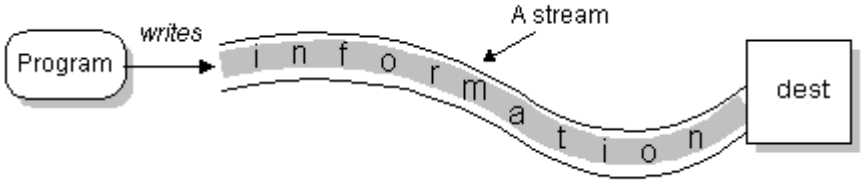
Un '`stream`' es un objeto que hace de intermediario entre el programa y el origen o destino de la información.

LIBRERÍA I/O

Para traer la información, un programa abre un 'stream' sobre una fuente de información (un fichero, memoria, un socket) y lee la información, de esta forma:



De igual forma, un programa puede enviar información a un destino externo abriendo un 'stream' sobre un destino y escribiendo la información en este, de esta forma:



Los algoritmos para leer y escribir:

ABRIR Un stream
MIENTRAS haya información
  LEER / ESCRIBIR información
CERRAR el stream

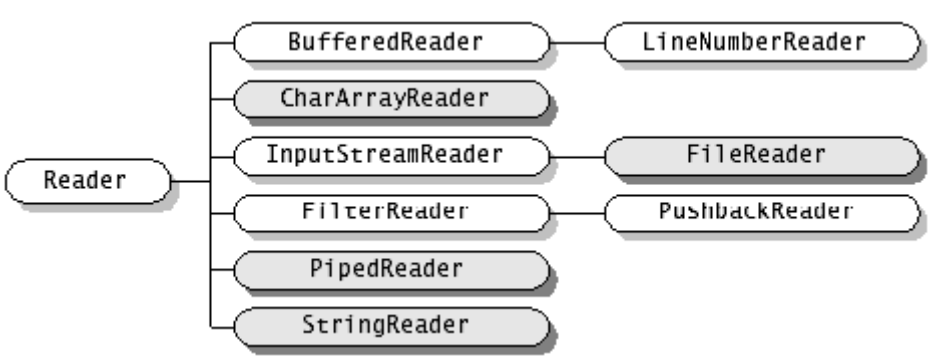
El paquete *java.io* contiene una colección de clases **stream** que soportan estos algoritmos para leer y escribir.

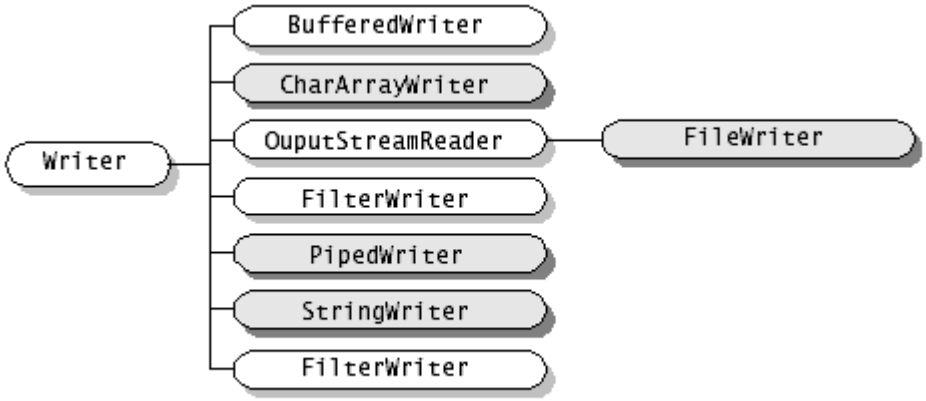
Estas clases están divididas en dos árboles basándose en los tipos de datos (caracteres o bytes) sobre los que opera.

Los Streams de Caracteres

`Reader` y **`Writer`** son las superclases abstractas para 'streams' de caracteres en **`java.io`**.

- **`Reader`** proporciona el API y una implementación para `readers` ('streams' que leen caracteres de 16-bits)
- **`Writer`** proporciona el API y una implementación para `writers` ('streams' que escriben caracteres de 16-bits).



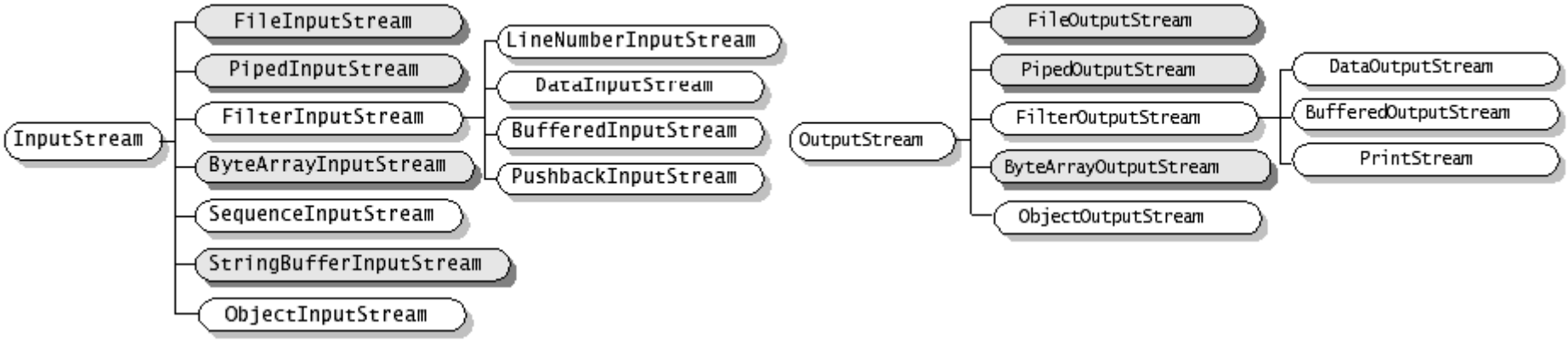


LIBRERÍA I/O

Los Streams de Bytes

Los programas deberían usar los 'streams' de bytes, descendientes de **`InputStream`** y **`OutputStream`**, para leer y escribir bytes de 8-bits.

Estos 'streams' se usan normalmente para leer y escribir datos binarios, tales como imágenes y sonidos.



La clase `File`

Nos permite recuperar información acerca de un archivo o directorio.

Los objetos de la clase **`File`** no abren archivos ni proporcionan herramientas para procesar archivos. Se utilizan en combinación con objetos de otras clases de **`java.io`** para especificar los archivos o directorios que se van a manipular.

```
import java.io.File;
public class PruebaFile {
    //muestra información acerca de un fichero y un directorio
    public static void main(String[] args) {
        File fichero = new File("ejemplo.txt");
        if (fichero.exists() && fichero.isFile()) {
            System.out.println("\n- Información del fichero:");
            System.out.println("El fichero tiene el nombre: " + fichero.getName());
            System.out.println("El fichero tiene el path: " + fichero.getAbsolutePath());
            System.out.println("Longitud del fichero: " + fichero.length());
        }
        File directorio = new File("C:\\Program Files\\Java");
        if (directorio.exists() && directorio.isDirectory()) {
            String listado[] = directorio.list();
            System.out.println("\n- Listado del directorio:");
            for (int i = 0; i < listado.length; i++) { System.out.println(listado[i] + "\n");
           }
        }
   }
}
```

Lectura de un fichero de texto

Si necesitamos leer la información almacenada en un fichero de texto que contiene caracteres especiales tales como acentos y eñes debemos combinar las clases **`FileInputStream`**, **`InputStreamReader`** y **`BufferedReader`**.

- Mediante la clase **`FileInputStream`** indicaremos el fichero a leer (es un 'stream' de bytes).

- La clase **`InputStreamReader`** se encarga de **leer bytes y convertirlos a carácter** según unas reglas de conversión definidas para cambiar entre 16-bit Unicode y otras representaciones específicas de la plataforma.

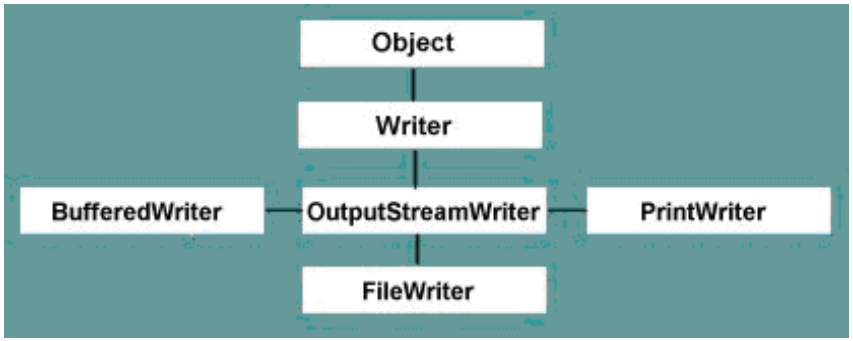
- Mediante la clase **`BufferedReader`** leeremos el texto desde el `InputStreamReader` almacenando los caracteres leídos en un buffer de almacenamiento temporal.
  - Esta clase tiene el método **`readLine()`** para leer una línea de texto a la vez.

Lectura de un fichero de texto

```
import java.io.*;
public class LeeFicheroEspecial {
    public static void main(String[] args) {
        String cad;
        try {
            FileInputStream fis = new FileInputStream("ejemplo.txt");
            InputStreamReader isr = new InputStreamReader(fis, "ISO-8859-1");
            BufferedReader br = new BufferedReader(isr);
            while ((cad = br.readLine()) != null) {
                System.out.println(cad);
            }
            //Cerramos el stream
            br.close();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}
```

Escritura de un fichero de texto

- Para crear un 'stream' de salida, tenemos la clase **`Writer`** y sus descendientes.



- La clase **`PrintWriter`** proporciona métodos que facilitan la escritura de valores de tipo primitivo y objetos en un 'stream' de caracteres.
- Los métodos principales que proporciona:
  - El método **`println`** añade una nueva línea después de escribir su parámetro.
  - La clase **`PrintWriter`** se basa en un objeto **`BufferedWriter`** para el almacenamiento temporal de los caracteres y su posterior escritura en el fichero.

Escritura de un fichero de texto

```
import java.io.*;
public class EscribeFichero {
   public static void main(String[] args)  {
        String cad1 = "Esto es una cadena.";
        String cad2 = "Esto es otra cadena.";
        try {
            PrintWriter salida
                = new PrintWriter(new BufferedWriter(new FileWriter("salida.txt")));
            salida.println(cad1);
            salida.println(cad2);

            //Cerramos el stream
            salida.close();
        } catch (IOException ioe) {
            System.out.println("Error IO: "+ioe.toString());
        }
   }
}
```

SERIALIZACIÓN DE OBJETOS

Cuando ejecutamos una aplicación OO lo normal es crear **múltiples instancias de las clases** que tengamos definidas en el sistema. Cuando cerramos esta aplicación todos los objetos que tengamos en memoria se pierden.

Para solucionar este problema los lenguajes de POO nos proporcionan unos mecanismos especiales para poder guardar y recuperar el estado de un objeto y de esa manera poder utilizarlo como si no lo hubiéramos eliminado de la memoria. Este tipo de mecanismos se conoce como **persistencia de los objetos**.

En **Java** hay que implementar una interfaz y utilizar dos clases

- `Interfaz` **`Serializable`** (interfaz vacía, no hay que implementar ningún método)
- Streams: **`ObjectOutputStream`** y **`ObjectInputStream`**.

Por ejemplo: `class` *Clase* `implements` `Serializable`, a partir de esta declaración los objetos que se basen en esta clase pueden ser persistentes.

SERIALIZACIÓN DE OBJETOS

Las clases `ObjectOutputStream` y `ObjectInputStream` permiten leer y escribir objetos, es decir, escribir y leer los bytes que representan al objeto. El proceso de transformación de un objeto en un 'stream' de bytes se denomina **serialización**.

- Los objetos creados por `ObjectOutputStream` y `ObjectInputStream` serán almacenados en ficheros, utilizan los 'streams' de bytes `FileOutputStream` y/o `FileInputStream`, ficheros de acceso secuencial.

  Para serializar objetos necesitamos

  1. Crear un objeto `FileOutputStream`, que nos permita escribir bytes en un fichero
     ```
     FileOutputStream fos = new FileOutputStream("fichero.dat");
     ```
  2. Crear un objeto `ObjectOutputStream`, al que le pasamos el objeto
     ```
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     ```
  3. Almacenar objetos mediante `writeObject()`
     ```
     oos.writeObject(objeto);
     ```
  4. Cuando terminemos, debemos cerrar el fichero escribiendo
     ```
     fos.close();
     ```

SERIALIZACIÓN DE OBJETOS

Para recuperar los objetos serializados necesitamos

1. Crear un objeto `FileInputStream`, nos permita leer bytes de un fichero

```
FileInputStream fis = new FileInputStream("fichero.dat");
```

2. Crear un objeto `ObjectInputStream`, al que le pasamos el objeto anterior

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

3. Leer objetos mediante `readObject()`

```
(ClaseDestino) ois.readObject();
```

Necesitamos realizar una conversión a la "ClaseDestino" debido a que Java solo guarda `Objects` *en el fichero.*

4. Cuando terminemos, debemos cerrar el fichero

```
fis.close();
```

Nota :

- Los atributos **`static`** no se serializan de forma automática.
- Los atributos que pongan **`transient`** no se serializan

SERIALIZACIÓN DE OBJETOS

Ejemplo de serialización de objetos de tipo persona 1:

```
public class Persona implements Serializable { … } // declaración de la clase persona
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1”); … ;
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4”);
//Serialización de las personas
FileOutputStream fosPer = new FileOutputStream("copiasegPer.dat");
ObjectOutputStream oosPer = new ObjectOutputStream(fosPer);
oosPer.writeObject(obj1); … ; oosPer.writeObject(obj4);
//Lectura de los objetos de tipo persona
FileInputStream fisPer  = new FileInputStream("copiasegPer.dat");
ObjectInputStream oisPer = new ObjectInputStream(fisPer);
try {
        while (true) {
              Persona per = (Persona) oisPer.readObject();
              System.out.println (per.toString());
        }
} catch (EOFException e) {
        System.out.println ("Lectura de los objetos de tipo Persona finalizada");
}
fisPer.close();
```

SERIALIZACIÓN DE OBJETOS

Ejemplo de serialización de objetos de tipo persona 2:

```
public class Persona implements Serializable { … } // declaración de la clase persona
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1”); …
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4”);
//Introducimos los objetos en una tabla hash
HashMap<String, Persona> personas = new HashMap<>();
personas.put(obj1.getDni(), obj1); … ; personas.put(obj4.getDni(), obj4);
//Serialización de la tabla hash personas
FileOutputStream fosPer = new FileOutputStream("copiasegPer.dat");
ObjectOutputStream oosPer = new ObjectOutputStream(fosPer);
oosPer.writeObject(personas);
fosPer.close();
//Lectura de los objetos de tipo persona a través de la tabla hash personas
FileInputStream fisPer = new FileInputStream("copiasegPer.dat");
ObjectInputStream oisPer = new ObjectInputStream(fisPer);
try {
     while (true) {
        personas = (HashMap) oisPer.readObject();
        System.out.println (personas.toString());
     }
} catch (EOFException e) {
       System.out.println ("Lectura de los objetos de tipo Persona finalizada");
}
fisPer.close();
```

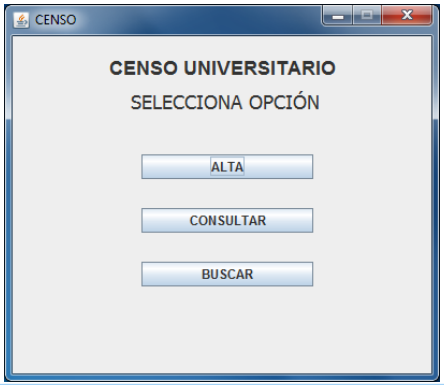
EJEMPLO USO DE COLECCIONES Y PERSISTENCIA

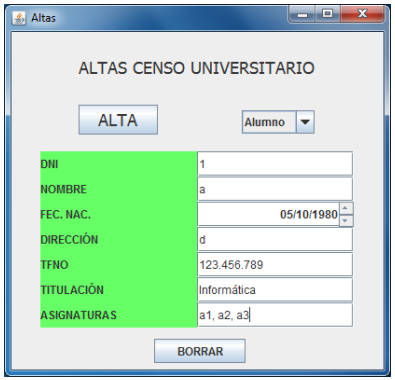
PROGRAMAS TEORIA

En `\Programas\Tema7\CensoUni` encontrareis códigos del ejemplo.

- El ejemplo utiliza un **`ArrayList`** para gestionar objetos de tipo Persona en un censo universitario con profesores y alumnos.

- También se utiliza la persistencia para almacenar los datos cuando la aplicación se cierra y la generación de ficheros de tipo texto.





Programación Orientada a Objetos

Tema 7-2:

Librería I/O