Programación Orientada a Objetos

Tema 7:
Persistencia

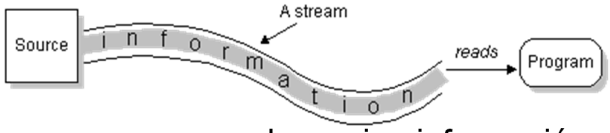
Contenidos

- Tema 7: Persistencia

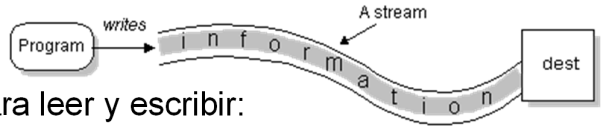
1. LIBRERÍA I/O
2. FICHEROS
3. FICHEROS DE ACCESO DIRECTO
4. FICHEROS DE TEXTO
5. SERIALIZACIÓN DE OBJETOS
6. EJEMPLO USO DE COLECCIONES Y PERSISTENCIA

- 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.

- 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:

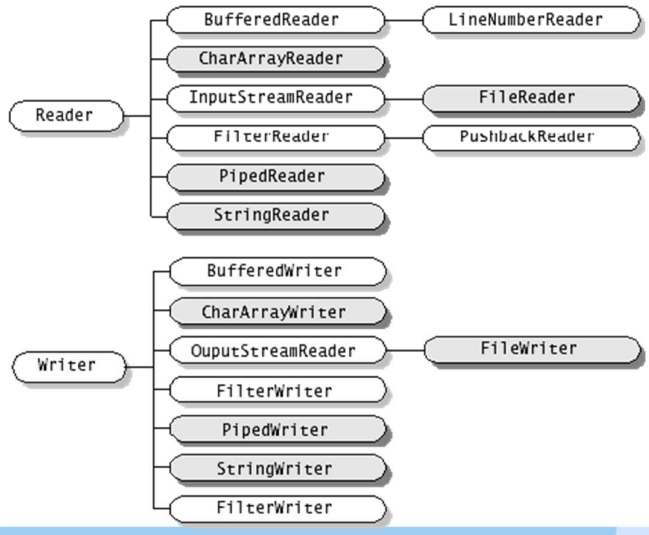


- Los algoritmos para leer y escribir:

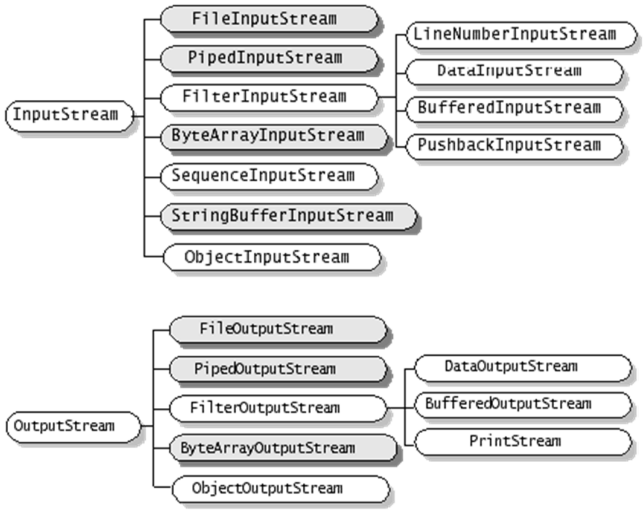
abrir un stream
mientras haya información
leer o escribir información
cerrar el stream

- 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.

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



- **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 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.
  - Ejemplo:

```
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");
         }
      }
   }
}
```


FICHEROS DE ACCESO ALEATORIO

- **La clase RandomAccessFile:**
  - Nos permiten tener un acceso instantáneo a la información.
  - Hay que determinar el tamaño de cada registro.
  - Métodos importantes:
    - seek(): Nos permite desplazarnos a un nuevo registro del archivo.
    - getFilePointer(): Permite averiguar en qué lugar del archivo nos encontramos.
    - length(): Determina el tamaño máximo del archivo.
    - Constructores: El primer parámetro indica el nombre del fichero y el segundo el modo: r (lectura); rw (lectura/escritura).
    - mark(): Marcar una posición.
    - reset(): Reposicionar a la marca.

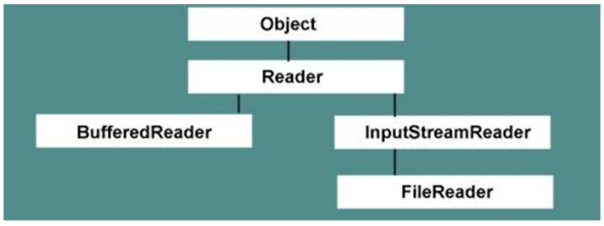
FICHEROS DE ACCESO ALEATORIO

- **Ejemplo RandomAccessFile:**

```
import java.io.IOException; import java.io.RandomAccessFile;
public class PruebaRandomAccessFile {
   public static void main(String[] args) {
      try {
         RandomAccessFile raf = new RandomAccessFile("ficheroAleatorio.dat", "rw");
         for (int i = 0; i < 7; i++) {
            raf.writeDouble(i * 1.5);
         }
         raf.close();
         presenta();
         raf = new RandomAccessFile("ficheroAleatorio.dat", "rw");
         raf.seek(5 * 8); //nos situamos en la posición 5.
         raf.writeDouble(3.14);
         raf.close();
         presenta();
      } catch (IOException ioe) {
         System.out.println("Error de IO: " + ioe.getMessage());
      } catch (Exception e) {
         System.out.println("Error: " + e.getMessage());
      }
   }
   private static void presenta() throws IOException {
      RandomAccessFile raf = new RandomAccessFile("ficheroAleatorio.dat", "r");
      System.out.println("Lectura del raf:");
      for (int i = 0; i < raf.length()/8; i++) {
         System.out.println("Valor: " + raf.readDouble());
      }
      raf.close();
   }
```


FICHEROS DE TEXTO

- **Lectura de un fichero de texto 1:**
  - Si necesitamos leer la información almacenada en un fichero de texto haremos uso de los streams de caracteres siguientes:



  - **Reader** es una clase base abstracta **para leer streams de caracteres**.
  - 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.
  - **FileReader** es una clase para **leer ficheros de caracteres**. Los constructores de esta clase asumen que la codificación de caracteres es la de por defecto y el tamaño del buffer de bytes es el apropiado.
  - La clase **BufferedReader** lee texto desde un stream de entrada de caracteres, almacenando los caracteres leídos. Proporciona un buffer de almacenamiento temporal. Esta clase tiene el método *readLine()* para leer una línea de texto a la vez. Sin embargo, BufferedReader no puede usarse por sí mismo, debemos envolver un Reader dentro de un BufferedReader.

FICHEROS DE TEXTO

- **Lectura de un fichero de texto 1:**

```
import java.io.*;
public class LeeFichero {
    public static void main(String[] args) {
            String cad;
            try {
                    BufferedReader br =
                    new BufferedReader(new FileReader("ejemplo.txt"));
                    while ((cad = br.readLine()) != null) {
                            System.out.println(cad);
                    }
                    //Cerramos el stream
                    br.close();
            }
            catch (IOException ioe) {
                    System.out.println(ioe);
            }
    }
}
```


FICHEROS DE TEXTO

- **Lectura de un fichero de texto 2:**
  - 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).
  - Mediante la clase **InputStreamReader** traduciremos cada byte leído del **FileInputStream** al carácter correspondiente siguiendo un tipo determinado de codificación.
  - Por último, mediante la clase **BufferedReader** leeremos el texto desde el **InputStreamReader** almacenando los caracteres leídos.

FICHEROS DE TEXTO

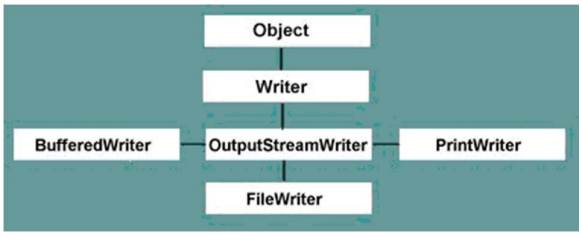
- **Lectura de un fichero de texto 2:**

```
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);
      }
   }
}
```


FICHEROS DE TEXTO

- **Escritura de un fichero de texto 1:**
  - Para crear un stream de salida, tenemos la clase **Writer** y sus descendientes. Podemos realizarlo de dos formas distintas, bien usando un objeto de tipo **BufferedWriter** o bien uno de tipo **PrintWriter**.



  - La primera forma de escribir un fichero de texto es utilizar la clase **BufferedWriter** que proporciona un buffer de almacenamiento temporal. Los datos se escriben en el buffer, y cuando éste se llena, los contenidos se escriben a un fichero, lo que reduce el número de veces que se tiene que acceder al disco o a la red.
  - Un objeto BufferedWriter envuelve un objeto **FileWriter**, entonces se llama al método *write* para escribir los datos en el fichero especificado. Es importante utilizar los métodos flush y close para vaciar el buffer y cerrar el fichero para liberar recursos.

- **Escritura de un fichero de texto 1:**

```
import java.io.*;
public class EscribeFichero1 {
	public static void main(String[] args)  {
		String cad1 = "Esto es una cadena.";
		String cad2 = "Esto es otra cadena.";
		try {
			BufferedWriter bw =  new BufferedWriter(new FileWriter("salida.txt"));
			bw.write(cad1);
			bw.newLine();
			bw.write(cad2);
			bw.flush();

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


- **Escritura de un fichero de texto 2:**

  - La otra forma de escribir ficheros de texto consiste en utilizar la clase **PrintWriter** ya que proporciona métodos que facilitan la escritura de valores de tipo primitivo y objetos en un stream de caracteres.

  - Los métodos principales que proporciona son ***print*** y ***println,*** más convenientes de utilizar que los métodos *write*. 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.

FICHEROS DE TEXTO

- **Escritura de un fichero de texto 2:**

```
import java.io.*;
public class EscribeFichero2 {
      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

- ObjectOutputStream y ObjectInputStream permiten leer y escribir grafos de 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 ObjectOutputStream y ObjectInputStream deben ser almacenados en ficheros, para hacerlo utilizaremos los streams de bytes FileOutputStream y FileInputStream, **ficheros de acceso secuencial**.
- Para serializar objetos necesitamos:
  - Un objeto FileOutputStream que nos permita escribir bytes en un fichero como por ejemplo:
    FileOutputStream fos = new FileOutputStream("fichero.dat");
  - Un objeto ObjectOutputStream al que le pasamos el objeto anterior de la siguiente forma:
    ObjectOutputStream oos = new ObjectOutputStream(fos);
  - Almacenar objetos mediante writeObject() como sigue:
    oos.writeObject(objeto);
  - Cuando terminemos, debemos cerrar el fichero escribiendo:
    fos.close();

Universidad de Alcalá
Departamento Ciencias de la Computación

SERIALIZACIÓN DE OBJETOS

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

- Para recuperar los objetos serializados necesitamos:
  - Un objeto FileInputStream que nos permita leer bytes de un fichero, como por ejemplo:
    FileInputStream fis = new FileInputStream("fichero.dat");
  - Un objeto ObjectInputStream al que le pasamos el objeto anterior de la siguiente forma:
    ObjectInputStream ois = new ObjectInputStream(fis);
  - Leer objetos mediante readObject() como sigue:
    (ClaseDestino) *ois.readObject();*
    Necesitamos realizar una conversión a la "ClaseDestino" debido a que Java solo guarda Objects en el fichero.
  - Cuando terminemos, debemos cerrar el fichero escribiendo:
    fis.close();

SERIALIZACIÓN DE OBJETOS

- Ejemplo de serialización de objetos de tipo persona 1:

```
public class Persona implements Serializable { … }
class Fecha implements Serializable { … }
/*************************************************************************************************/
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1"); …
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4");
/*************************************************************************************************/
//Serialización de las personas 1
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 { … }
class Fecha implements Serializable { … }
/*************************************************************************************************/
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<String, Persona>();
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

- El siguiente 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.