PROMETEO

PROGRAMACIÓN

Manejo de archivos y flujos de datos en Java

Lectura y escritura de datos

Hasta ahora, hemos trabajado con programas que **leen y escriben información por consola**.

Hemos utilizado la clase **Scanner** para leer datos desde la **entrada estándar**, es decir, información introducida por el usuario a través del teclado.

Esto nos ha permitido:

- Leer números, textos y otros tipos de datos.
- Interactuar con el usuario durante la ejecución del programa.
- Probar fácilmente nuestros programas en consola.

Para mostrar información, hemos usado:

- System.out.print()
- System.out.println()
- System.out.printf()

Estos métodos muestran resultados por pantalla, pero **la información se pierde al cerrar el programa**.

Lectura y escritura de datos

A partir de ahora, además de trabajar con la consola, **leeremos y escribiremos información usando archivos**.

📂 Lectura desde archivos

Esto nos permitirá:

- Trabajar con datos persistentes.
- Leer grandes cantidades de información automáticamente.
- Simular situaciones reales como ficheros de datos, configuraciones o registros.

📁 Escritura en archivos

Además de escribir por pantalla, **podremos guardar información en archivos** para:

- Almacenar datos de forma permanente.
- Generar informes y listados.
- Guardar resultados para utilizarlos más adelante.

Fundamentos de Lectura y Escritura

El manejo de archivos y flujos de datos es una parte esencial de muchas aplicaciones en Java. Permite almacenar, leer, procesar y escribir información en diversos formatos. Java proporciona una rica biblioteca de clases en el paquete java.io para manejar operaciones de entrada y salida.

- **Entrada**:
    a. Implica leer datos de diversas fuentes, como teclado, archivos o redes.
    b. Se realiza utilizando **InputStreams**, que permiten obtener datos para que el programa los procese.

- **Salida**:
    a. Implica escribir datos en un destino, como la consola, archivos o redes.
    b. Se realiza utilizando **OutputStreams,** que permiten mostrar o guardar los resultados del programa.

¿Qué son los flujos (Streams)?

Los **flujos (streams)** son abstracciones que se utilizan para realizar operaciones de entrada y salida en Java.

Proporcionan una manera:

- Uniforme
- Eficiente
- Independiente del origen o destino de los datos

Gracias a los flujos, Java puede trabajar de la misma forma con:

- Archivos
- Redes
- Memoria

Tipos de flujos

En Java existen distintos tipos de flujos, diseñados para adaptarse a diferentes necesidades.

Los flujos se clasifican principalmente:

- Según el tipo de datos que manejan
- Según la forma en la que se procesan esos datos

Esta clasificación permite elegir el flujo adecuado para cada situación.

Tipos de flujos

Flujos de bytes y flujos de caracteres

- **Flujos de Bytes**
  - Trabajan con datos binarios (bytes).
  - Se utilizan para archivos no textuales.
  - Ejemplos: imágenes, audio, vídeo.

- **Flujos de Caracteres**
  - Trabajan con texto.
  - Manejan caracteres en lugar de bytes.
  - Son ideales para leer y escribir archivos de texto.

Clases relativas a flujos

Java ofrece numerosas clases para trabajar con flujos de datos.

Estas clases permiten:

- Leer información
- Escribir información
- Adaptar el flujo a diferentes formatos y necesidades

Cada clase está especializada en un tipo concreto de operación, lo que facilita el trabajo con archivos y datos.

Clases relativas a flujos

Ejemplos de clases de flujos de bytes

- **InputStream**

  Clase base para leer datos en forma de bytes.
- **FileInputStream**

  Permite leer datos directamente desde un archivo.
- **OutputStream**

  Clase base para escribir datos en forma de bytes.
- **FileOutputStream**

  Permite escribir datos directamente en un archivo.

Estas clases se usan, por ejemplo, para trabajar con:

- Imágenes - Archivos binarios - Audio o vídeo

Clases relativas a flujos

Ejemplos de clases de flujos de caracteres

- **Reader**

  Clase base para leer caracteres.
- **FileReader**

  Permite leer texto desde un archivo.
- **Writer**

  Clase base para escribir caracteres.
- **FileWriter**

  Permite escribir texto en un archivo.

Estas clases son ideales para trabajar con **archivos de texto**.

Jerarquía de clases

Las clases de flujos en Java están organizadas en una **jerarquía**, basada en la herencia.

Esto significa que:

- Existen clases generales (padres)
- Existen clases más específicas (hijas)
- Las clases hijas heredan comportamiento de las clases base

Jerarquía de clases

Ejemplo de jerarquía (flujos de bytes)

```
InputStream
   └── FileInputStream
```

- **InputStream** define cómo se leen bytes.
- **FileInputStream** implementa la lectura desde archivos.

Ejemplo de jerarquía (flujos de caracteres)

```
Reader
   └── FileReader
```

- **Reader** define la lectura de caracteres.
- **FileReader** permite leer texto desde archivos.

Operaciones comunes

Cuando se trabaja con flujos en Java, normalmente se siguen estos pasos:

1. Abrir el flujo
2. Leer o escribir datos
3. Cerrar el flujo
4. Gestionar posibles errores

Cerrar los flujos correctamente es fundamental para evitar problemas en el programa.

Lectura de datos

Usando `FileInputStream`

- Lee un archivo byte por byte, ideal para datos binarios.

```java
import java.io.FileInputStream;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("imagen.jpg")) {
            int byteLeido;
            while ((byteLeido = fis.read()) != -1) {
                System.out.println(byteLeido); // Procesa cada byte
            }
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

Lectura de datos

Usando `FileReader`

- Maneja datos de texto, procesando caracteres en lugar de bytes.

```java
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("archivo.txt")) {
            int caracter;
            while ((caracter = fr.read()) != -1) {
                System.out.print((char) caracter);
            }
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

Lectura de datos

Usando `BufferedReader`

- `BufferedReader` permite leer archivos línea por línea, lo que lo hace eficiente para procesar texto.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("archivo.txt"))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

Escritura de datos

Usando FileOutputStream

- Escribe datos byte por byte.

```java
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("salida.dat")) {
            String texto = "Ejemplo de escritura en bytes";
            fos.write(texto.getBytes());
        } catch (IOException e) {
            System.out.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```

Escritura de datos

Usando FileWriter

```java
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {

        try (FileWriter fw = new FileWriter("archivo.txt")) {
            String texto = "Hola desde FileWriter";

            fw.write(texto);

        } catch (IOException e) {
            System.out.println("Error al escribir el archivo: " + e.getMessage());
        }

    }
}
```

Escritura de datos

Usando BufferedWriter

- BufferedWriter permite escribir grandes cantidades de texto de manera eficiente.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("archivo.txt"))) {
            bw.write("Primera línea");
            bw.newLine();
            bw.write("Segunda línea");
        } catch (IOException e) {
            System.out.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```


PROMETEO