Java.IO

INTRODUCCIÓN

Las aplicaciones Java pueden guardar datos de forma persistente utilizando archivos o sistemas más avanzados como bases de datos. El API **java.io** permite interactuar con archivos y flujos para leer y escribir datos, garantizando que la información se preserve entre ejecuciones del programa.

LA CLASE FILE

La clase `File` es una de las más utilizadas (antigua) del paquete `Java.io` se utiliza para **leer información** sobre **archivos** y **directorios** existentes, listar el contenido de un directorio o crear/eliminar archivos y directorios.

Una **instancia de una clase** `File` representa la **ruta a un archivo o directorio específico en el sistema de archivos**, pero **no contiene los datos del archivo o directorio** (el archivo podría no existir).

1. Creación de un Objeto de tipo File

```
File javaFile = new File("/home/otto/apuntes/javaio.txt");
```

Constructores de la clase `File`

```
public File(String pathname) //Crea archivo a través de una ruta
public File(File parent, String child) //Crea archivo a través de unaruta principa
public File(String parent, String child)//Crea un archivo a través de una ruta y
public File(URI uri) //Crea un archivo a tavés de una URI
```

2. El Objeto File

La clase `File` en Java representa una **ruta** a un archivo, pero no está conectada a un archivo real a menos que se realicen operaciones sobre él. Permite comprobar si un archivo existe, leer propiedades del archivo, modificar su nombre o ubicación, y eliminarlo. Al operar con archivos, la JVM y el sistema operativo el sistema de archivos realizan las acciones basadas en los métodos de la clase `File`. Si intentas operar en un archivo que no existe o no tienes acceso, algunos métodos lanzarán excepciones, mientras que otros devolverán `false` si la operación no puede realizarse.

3. Métodos Clase File

`boolean delete()`	Borra el archivo o directorio y devuelve `true` sólo si la operación se completó con éxito. Si esta instancia es un directorio, **el directorio debe estar vacío para poder eliminarse** .
`boolean exists()`	Devuelve `true` si un el archivo sobre el que se aplica el método existe
`String getAbsolutePath()`	Obtiene el nombre absoluto del archivo o directorio en el sistema de archivos
`String getName()`	Obtiene el nombre del archivo o directorio
`String getParent()`	Obtiene el directorio principal en el que se encuentra la ruta (null si no hai ninguno)
`boolean isDirectory()`	Comprueba si el archivo en el que se aplica el método es un directorio
`boolean isFile()`	Comprueba si el archivo en el que se aplica el método es un archivo
`long lastModified()`	devuelve el tiempo trascurrido (milisegundos) desde 1/1/1970 hasta la fecha de ultima modificación
`long length()`	Obtiene el número de bytes del archivo
`File[] listFiles()`	Devuelve una lista de los archivos contenidos dentro de un directorio
`boolean mkdir()`	Crea un directorio en la ruta especificada
`boolean mkdirs()`	Crea un directorio en la ruta especificada, incluyendo cualquier directorio anterior inexistente
`boolean renameTo(File dest)`	Cambia el nombre del archivo o directorio especificado (devuelve true si se ha completado con éxito)

LA CLASE RANDOMACCESSFILE

La clase `RandomAccessFile` en Java permite el **acceso no secuencial (aleatorio)** a archivos, lo que facilita la lectura y escritura en cualquier parte del archivo.

1. Características principales:

- **Modos de apertura**:
  - `"r"`: Solo **lectura**.
  - `"rw"`: **Lectura** y **escritura**.
  - `"rwd"`: **Lectura** y **escritura**, **sincronizado**.
- **Puntero de archivo**:
  Esta clase emplea la notación de
  puntero a archivo para especificar la posición actual en el archivo.
  - Inicialmente apunta al principio del archivo (posición 0).
  - La posición se modifica con cada operación de lectura o escritura.

2. Métodos para gestionar la posición del puntero:

- `int skipBytes(int n)`: Mueve el puntero hacia adelante “n” bytes.
- `void seek(long pos)`: Sitúa el puntero en la posición especificada.
- `long getFilePointer()`: Devuelve la posición actual del puntero.

Excepciones:

- Lanza `EOFException` si se alcanza el final del archivo antes de leer el número deseado de bytes.
- Lanza `IOException` si hay un error diferente, como un flujo cerrado.

3. Lectura con RandomAccessFile

📌 *El método `read()` de la clase `RandomAccessFile` posee dos constructores, uno para leer byte a byte y otro para leer un array de bytes*

- **Lectura de un byte desde `read()`**

  La **lectura** un byte desde un `RandomAccessFile` **se realiza usando su método `read()`**:

```
RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamundo.
int miByte = file.read();
```

- **Lectura de un array de bytes: `read(byte[])`**

  También es posible leer un array de bytes con un `RandomAccessFile`:

```
RandomAccessFile randomAccessFile = new RandomAccessFile("progran

byte[] dest     = new byte[1024]; // Array de bytes donde se almacenarán
int    offset   = 0;
int    length   = 1024;
int    bytesLeidos = randomAccessFile.read(dest, offset, length);
```

4. Escritura con RandomAccessFile

📌 *El método `write()` de la clase `RandomAccessFile` posee dos constructores, uno para leer byte a byte y otro para leer un array de bytes*

- **Escritura de un byte con `write`**

  El método `write()` de `RandomAccessFile` toma un entero como parámetro. El byte se escribirá en la posición actual del puntero del archivo en el `RandomAccessFile`:(Si hubiese algún byte anteriormente en esa posición, será sobrescrito)

```
RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamur
```

```
file.write(67); // Código ASCII para 'C'
```

- **Escritura de un array de bytes con** `write`

```
RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamur

byte[] bytes = "Hello World".getBytes("UTF-8");
file.write(bytes); //Escribe todos los bytes
file.write(bytes, 2, 5); //escribe los bytes de las posiciones determinadas
```

Al igual que en el método `read()` , en el método `write` el puntero avanza automáticamente una posición despues de ser llamado.

5. Cierre

Después de usar la clase `RandomAccessFile` se debe utilizar el método `close()` para cerrar la instancia

```
RandomAccessFile file = new RandomAccessFile("c:\\programas\\holamun
do.kt", "rw");
file.close();
```

U**tilizar un bloque** `finally` **garantiza que ambos flujos se cierren incluso si se produce un error**

En el caso de haber utilizado la sentencia *try-with-resources* no será necesario aplicar este método

Ejemplo de uso RandomAccessFile

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

public class RegistroEstudiantes {

    public static void main(String[] args) throws IOException { // En realidad es

        try (RandomAccessFile file = new RandomAccessFile("E:\\programas\\es
```

```
        Scanner scanner = new Scanner(System.in);

        System.out.println("Introduce el número de estudiantes: ");
        int numEstudiantes = scanner.nextInt();
        file.writeInt(numEstudiantes);

        for (int i = 0; i < numEstudiantes; i++) {
            System.out.println("Introduce el nombre del estudiante " + (i + 1) + "
            String nombre = scanner.next();
            file.writeUTF(nombre);
        }

        System.out.println("Introduce el número del estudiante a leer: ");
        int numEstudiante = scanner.nextInt();

        file.seek(0);
        int numEstudiantesGuardados = file.readInt();

        if (numEstudiante > numEstudiantesGuardados) {
            System.out.println("No hay tantos estudiantes guardados.");
        } else {
            file.seek(4); // Saltamos el número de estudiantes
            for (int i = 0; i < numEstudiante - 1; i++) {
                file.readUTF();
            }
            System.out.println("El estudiante " + numEstudiante + " es: " + file.re
        }
    }
  }
}
```

FLUJOS DE ENTRADA/SALIDA

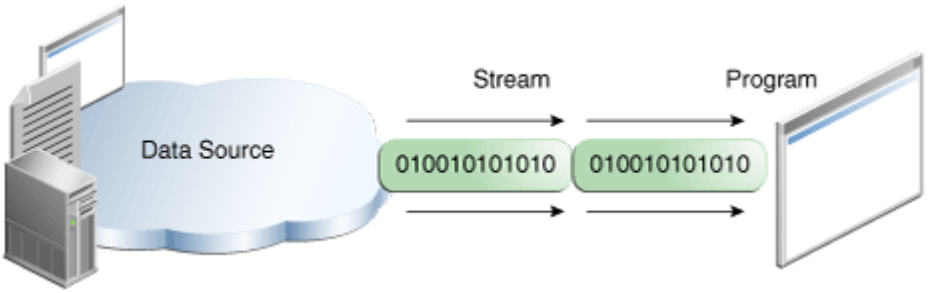
> 💡 Los flujos de E/S en Java permiten la entrada y salida de datos entre un programa y diversas fuentes o destinos, como archivos, dispositivos y otros programas.

Las clases que nos permiten **crear**, **acceder** y **manipular** flujos pertenecen a la API `Java.IO` . Los flujos son una **secuencia de datos que son leídos por el programa en bloques**, dependiendo del método con el que estemos leyendo el flujo, estos bloques se dividirán de una manera u otra en forma de bytes, objetos, caracteres...

1. Tipos de Flujos según Entrada y Salida

- **Flujos de entrada**

Representan una **fuente de entrada** de datos al programa y pueden provenir de diversas fuentes y en distintos tipos de datos

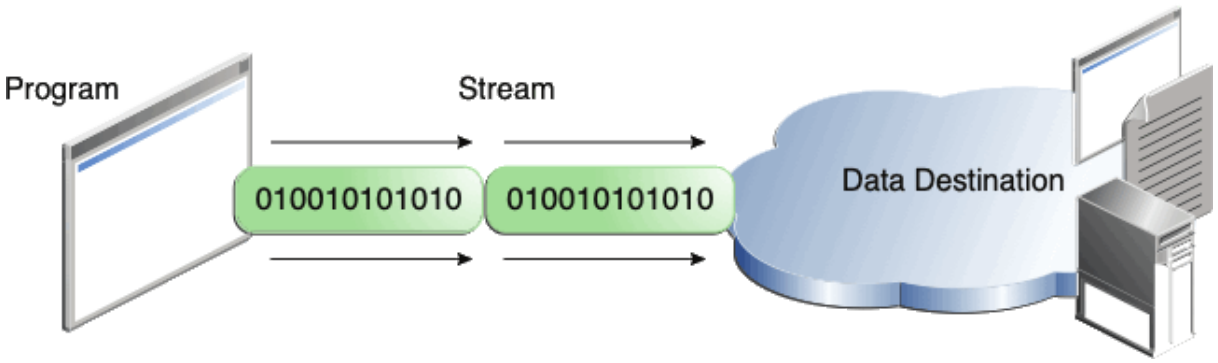


▼ Tipos de datos:

- Flujos de Bytes: `InputStream`
- Flujos de Caracteres: `Reader`
- Flujos de objetos: `ObjectOutputStream`
- Flujos de Arrays: `ByteArrayIntputStream`
- Flujos de datos primitivos: `DataInputStream`
- Dispositivos: `System.in`

- **Flujos de salida**

Representan una fuente de los datos del programa a algún destino interno o externo del ordenador



▼ Tipos de datos

- Flujos de Bytes: `OutputStream`
- Flujos de Caracteres: `Writer`
- Flujos de objetos: `ObjectInputStream`
- Flujos de Arrays: `ByteArrayOutputStream`
- Flujos de datos primitivos: `DataOutputStream`
- Dispositivos: `System.out`

2. Flujos de Bytes vs Flujos de Caracteres

⚠️ La mayoría de las clases de **flujos de entrada tienen una clase de flujo de salida correspondiente, y viceversa.**

En la API de `java.IO` se definen dos tipos de flujos para la lectura y escritura de flujos

Flujos de bytes:

- **Los flujos de bytes leen/escriben datos binarios (0 y 1)** y tienen nombres de clase que **terminan**
- Leen en **bloques de bytes y no pueden manejar caracteres Unicode.en** `InputStream` **o** `OutputStream`.
- **Todas** las clases **descienden (heredan) de** `InputStream` **y** `OutputStream`.
- Hay muchas clases de flujos de bytes, como: `FileInputStream` **y** `FileOutputStream`. Todos los restantes flujos funcionan del mismo modo sólo difieren en la

forma de construirlos.

Flujos de caracteres:

- Los flujos de caracteres **leen/escriben datos de texto** y tienen nombres de clase que **terminan en** `Reader` **o** `Writer`.
- Automáticamente, **transforma caracteres Unicode (formato de Java) al conjunto de caracteres local**.
- Todas las clases **descienden de** `Reader` **y** `Writer`.
- Hay muchas clases de flujos de carácter, como : `FileReader` (usa internamente `FileInputStream`), `FileWriter` (usa internamente `FileOutpuStream`). Todos los restantes flujos funcionan de igual modo, sólo difieren en la forma de construirlos.

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

public class CopiaArchivos {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("otto.txt");
            out = new FileOutputStream("nohaycole.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally { // Hay que cerrar el flujo en cualquier condición.
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
```

```
        }
    }
}
```

FLUJOS DE BYTE

> 💡 **Los flujos de bytes leen/escriben datos binarios (0 y 1)** y tienen nombres de clase que terminan en `InputStream` o `OutputStream` (**Clases que heredan** o de las que descienden todas sus variables que veremos a continuación).

Los Flujos de bytes **leen la información en bloques de 8 bits** (1 byte) y **no pueden manejar caracteres Unicode**

▼ 1. IntputStream

- * `ByteArrayInputStream`: Contiene un búfer interno que permite leer los bytes directamente desde la memoria.
- * `ObjectInputStream`: Lee **objetos Java serializado**s desde un **flujo de entrada.**
- * `FileInputStream`: Crea un flujo de entrada sobre un **archivo** en el sistema de archivos.
- `AudioInputStream`: Permite la lectura de **datos de audio** desde un archivo o cualquier otro recurso que contenga sonido.
- `FilterInputStream`: Proporciona clases decoradoras que modifican la funcionalidad básica de un flujo de entrada.
    - * `BufferedInputStream`: Almacena los **bytes** leídos en un búfer interno para mejorar la eficiencia.
    - * `DataInputStream`: Permite leer **datos primitivos** de Java desde un flujo de entrada.
    - `PushbackInputStream`: *Permite "devolver" bytes al flujo de entrada*, para que puedan ser leídos de nuevo.

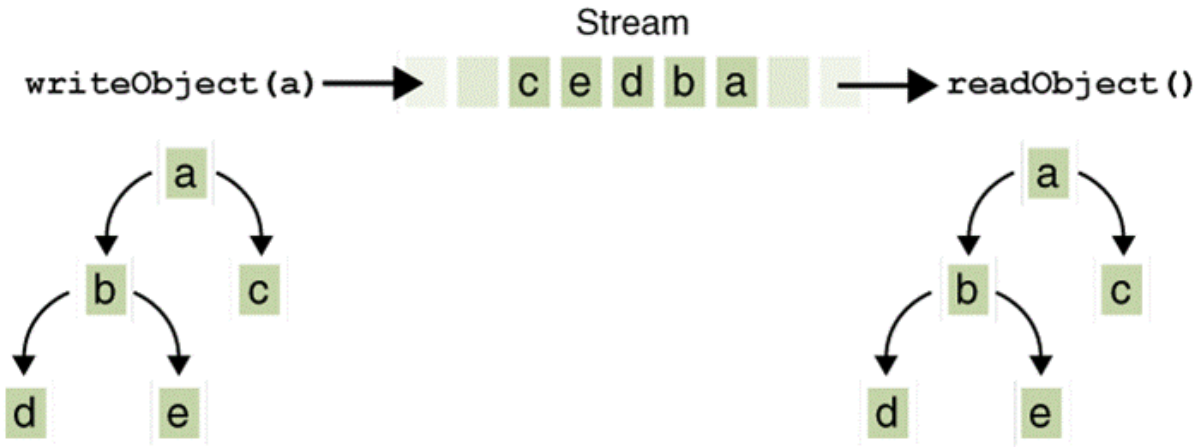
- `PipedInputStream`: Implementa un flujo de entrada que puede conectarse a un `PipedOutputStream`, permitiendo la comunicación entre hilos.
- `SequenceInputStream`: **Concatena múltiples flujos de entrada** para que puedan ser leídos secuencialmente como si fueran un solo flujo.

▼ 2.OutputStream

- *`ByteArrayOutputStream`: Implementa un flujo de salida que escribe datos en un array de bytes, expandiéndose automáticamente. Los datos pueden recuperarse con `toByteArray()` y `toString()`.
- *`FileOutputStream`: Crea un **flujo de salida sobre un archivo**
- *`ObjectOutputStream`**: escribe objetos Java serializados en un flujo de salida.**
- `PipedOutputStream` : Implementa un **flujo de salida conectado a un** `PipedInputStream`**,** permitiendo la comunicación entre diferentes hilos de ejecución.
- `FilterOutputStream`:
  - *`BufferedOutputStream` : Mejora la eficiencia al **almacenar temporalmente los datos en un búfer** antes de escribirlos en el flujo de salida subyacente.
  - *`PrintStream`: proporciona métodos para **imprimir representaciones de datos primitivos y objetos** en un flujo de salida. un ejemplo de uso es `System.out`.
  - `DataOutputStream` : Escribe **datos primitivos** de Java en un flujo de salida.
  - `ZipOutputStream`: Facilita la escritura de datos en **formato comprimido en archivos ZIP.**

3. ObjectInputStream y ObjectOutputStream

`ObjectInputStream`: lee objetos Java serializados del flujo de entrada y los deserializa. `ObjectOutputStream`: escribe objetos Java serializados en un flujo de salida.



Para emplear las clases `ObjectInputStream`, `ObjectOutStream` los objetos a leer (escribir **deben implantar la interface: Serializable** (dicha interface no tiene métodos para implantar)

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

public class SerializacionEjemplo {
    public static void main(String[] args) {
        // Archivo en el que se guardará el objeto serializado
        String nombreArchivo = "persona.ser";

        // Crear un objeto Persona
        Persona persona1 = new Persona("Juan", 25);

        // Serialización: escribir el objeto en un archivo
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputS
             // Escribiendo el objeto en el archivo
            oos.writeObject(persona1);
            System.out.println("Objeto serializado: " + persona1);
          } catch (IOException e) {
            System.err.println("Error durante la serialización: " + e.getMessage());
        }
```

```
        // Deserialización: leer el objeto desde el archivo
        try (FileInputStream fis = new FileInputStream(nombreArchivo);
             ObjectInputStream ois = new ObjectInputStream(fis)) {

            // Leyendo el objeto desde el archivo
            Persona personaRecuperada = (Persona) ois.readObject();
            System.out.println("Objeto deserializado: " + personaRecuperada);

        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Error durante la deserialización: " + e.getMessage(
        }
    }
}
```

4. Lectura desde URL

📖 Para leer desde una URL, se puede emplear la clase `URL` y `openStream()`:

```
import java.io.*;

public class LeerURL {
    public static void main(String[] args) throws Exception {
        URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/d
        URL url = uri.toURL();

        try (InputStream is = url.openStream();
             InputStreamReader isr = new InputStreamReader(is); // es un puente d
             int c;
             while ((c = isr.read()) != -1) {
                 System.out.print((char) c);
             }
        }
```

```
    }
}
```

URLConnection

El método `openConnection()` de `URL` devuelve un objeto de tipo `URLConnection`:

```
URI uri = new URI("https://manuais.pages.iessanclemente.net/plantillas/dan
URL url = uri.toURL();
URLConnection urlConnection = url.openConnection();
urlConnection.getInputStream();
```

HttpURLConnection

Permite añadir elementos específicos de HTTP, como el tamaño del contenido, o el tipo de archivo:

```
URL url = new URI("https://manuais.pages.iessanclemente.net/plantillas/dam
HttpURLConnection httpConnection = (HttpURLConnection) url.openConnec
httpConnection.getInputStream();
httpConnection.setRequestMethod("HEAD");
long tamanho = httpConnection.getContentLengthLong();
```

FLUJOS DE CARACTERES

📖 Los flujos de caracteres **leen y escriben datos de texto transformando automáticamente caracteres Unicode (formato de java) al conjunto de caracteres local**. Todas las variables de estos flujos heredan de las clases abstractas `Reader` y `Writer`

La principal característica que define a este tipo de flujos es que el programador se puede desentender de la traducción entre los tipos de caracteres que manejen las dos partes del programa

```
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;

public class CopiarCaracteres {
   public static void main(String[] args) throws IOException {

      FileReader inputStream = null;
      FileWriter outputStream = null;

      try {
         inputStream = new FileReader("otto.txt");
         outputStream = new FileWriter("nohaycole.txt");

         int c;
         while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
         }
      } finally {
         if (inputStream != null) {
            inputStream.close();
         }
         if (outputStream != null) {
            outputStream.close();
         }
      }
   }
}
```

Las clases `Filereader` y `FileWriter` acceden a la información en bloques de 16 bits (caracter)a diferencia de `FileInputStream` y `FileOutputStream` que lo hacen en bloques de 8 bits (entero)

> ❗ Los **flujos de caracteres** suelen “envolver” a los **flujos de bytes** para manejar la lectura y escritura de texto. Los flujos de caracteres (como `FileReader` y `FileWriter`) usan flujos de bytes (`FileInputStream` y `FileOutputStream`) para la E/S física, traduciendo entre bytes y caracteres automáticamente.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.IOException;

public class CopiarCaracteresConFlujosBytes {
public static void main(String[] args) {
try (InputStreamReader inputStreamReader = new InputStreamReader(new Fil
        OutputStreamWriter outputStreamWriter = new OutputStreamWriter(ne
      int c;
      // Leer carácter por carácter usando InputStreamReader y escribirlo usar
      while ((c = inputStreamReader.read()) != -1) {
         outputStreamWriter.write(c);
      }

      System.out.println("Copia completada con éxito.");
   } catch (IOException e) {
      e.printStackTrace();
   }
   }
}
```

FLIJOS DE ENTRADA/SALIDA CON BUFFER

Los flujos de la API `Java.IO` se pueden dividir en dos grupos. de bajo nivel(sin buffer), que son con los que hemos trabajado hasta ahora. estos se conectan directamente con la fuente de datos y procesa sus datos o recursos en bruto, es decir, sin filtrar. Esto hace que la actividad pueda ser ineficiente

1. Flujos de alto nivel

```
try (var br = new BufferedReader(new FileReader("noHayCole.txt"))) {
   System.out.println(br.readLine());
```

```
}
```

En este ejemplo, `FileReader` actúa como el flujo de bajo nivel para la lectura, mientras que `BufferedReader` es el flujo de alto nivel que utiliza un `FileReader` como entrada. Muchas operaciones del flujo de alto nivel, como `read()` o `close()`, se delegan al flujo de bajo nivel subyacente. Sin embargo, otras operaciones modifican o **añaden nueva funcionalidad** a los métodos del flujo de bajo nivel.

> 💡 En este caso el uso de Bufferes agrega funcionalidades come el método `.readLine()`, asi como mejoras en el rendimiento de acceso a datos

▼ **Clases de Flujos incompatibles entre si:**

```
new BufferedInputStream(new FileReader("z.txt")); // NO COMPILA por mezclar clases de Reader con clases de InputStream

new BufferedWriter(new FileOutputStream("z.txt")); // NO COMPILA po r mezclar clases de Writer con clases de OutputStream

new ObjectInputStream(new FileOutputStream("z.txt")); // NO COMPIL A por mezclar clases de InputStream con clases de OutputStream

new BufferedInputStream(new InputStream()); // NO COMPILA porque I nputStream es una clase abstracta
```

FLUJOS RESUMEN GENERAL

> 💡 Estas tablas son de suma importancia ya que encapsulan los diferentes tipos de Flujos tanto de entrada como salida que existen en java

Clase	Contenido	Descripción
InputStream	byte	Clase abstracta para todas los flujos de entrada de bytes
OutputStream	byte	Clase abstracta para todas los flujos de salida de bytes
Reader	caracter	Clase abstracta para todas los flujos de entrada de caracteres
Writer	caracter	Clase abstracta para todas los flujos de salida de caracteres

Tabla clases abstractas de flujos

Clase	Bajo/Alto Nivel	Descripción
FileInputStream	Bajo	Lee datos de archivos como bytes
FileOutputStream	Bajo	Escribe datos de archivos como bytes
FileReader	Bajo	Lee datos de archivos como caracteres
FileWriter	Bajo	Escribe datos de archivos como caracteres
BufferedInputStream	Alto	Lee datos de bytes de un flujo de entrada existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento
BufferedOutputStream	Alto	Escribe datos de bytes en un flujo de salida existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento
BufferedReader	Alto	Lee datos de caracteres de un objeto Reader existente de manera bufferizada, lo que mejora la eficiencia y el rendimiento
BufferedWriter	Alto	Escribe datos de caracteres en un objeto Writer existente de manera bufferizada, lo que mejora la eficiencia y elrendimiento
ObjectInputStream	Alto	Deserializa tipos de datos primitivos de Java y gráficos de objetos de Java a partir de un flujo de entrada existente
ObjectOutputStream	Alto	Serializa tipos de datos primitivos de Java y gráficos de objetos de Java en

Clase	Bajo/Alto Nivel	Descripción
		un flujo de salida existente
PrintStream	Alto	**Escribe representaciones formateadas de objetos Java en un flujo binario**
PrintWriter	Alto	**Escribe** representaciones **formateadas de objetos Java en un flujo de caracteres**

Tabla clases de flujos