

# **UNIDAD 5: LECTURA Y ESCRITURA DE INFORMACIÓN**

**Módulo Profesional: Programación**

## ÍNDICE

RESUMEN INTRODUCTORIO.....	3
INTRODUCCIÓN.....	3
CASO INTRODUCTORIO .....	3
1. INTRODUCCIÓN .....	4
2. CONCEPTO DE FLUJOS DE E/S.....	4
2.1 Flujos de bytes (byte streams) .....	5
2.2 Flujos de caracteres .....	7
2.3 Flujos de líneas.....	8
3. ENTRADA SALIDA DESDE LA LÍNEA DE COMANDOS .....	10
4. FLUJOS DE DATOS.....	11
5. FLUJOS DE OBJETOS.....	14
6. OBJETOS DE TIPO FILE .....	16
7. ARCHIVOS DE ACCESO ALEATORIO.....	18
RESUMEN FINAL .....	23

## RESUMEN INTRODUCTORIO

En esta unidad se explicarán las operaciones de E/S que se realizan normalmente en un programa centrándose en un lenguaje de programación orientado a objetos que será Java. Veremos el concepto de flujo de entrada y de salida, los tipos de flujos según se clasifiquen. Posteriormente veremos cómo se realiza la entrada y la salida desde o hacia la línea de comandos. Nos centraremos finalmente en los flujos de datos y en los flujos de objetos, viendo también el concepto de serialización de objetos. Finalmente, hablaremos acerca del objeto File de Java, para terminar con los archivos de acceso aleatorio. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

## INTRODUCCIÓN

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde el teclado, o bien enviar información a la pantalla. La comunicación entre el origen de cierta información y el destino se realiza mediante un flujo (stream) de información. Este intercambio de información puede ser entre la aplicación y el exterior o viceversa, entre archivos y la aplicación o viceversa, etc.

## CASO INTRODUCTORIO

Una vez que has realizado en tu empresa el desarrollo de la última aplicación, te encuentras con el problema de que tienes que enviar datos a la aplicación a través de la línea de comandos y almacenar cierta información en archivos de acceso aleatorio. Para esta tarea es de vital importancia que conozcas cómo se trabaja con los flujos en el lenguaje de programación Java.

Al finalizar la unidad el alumnado:

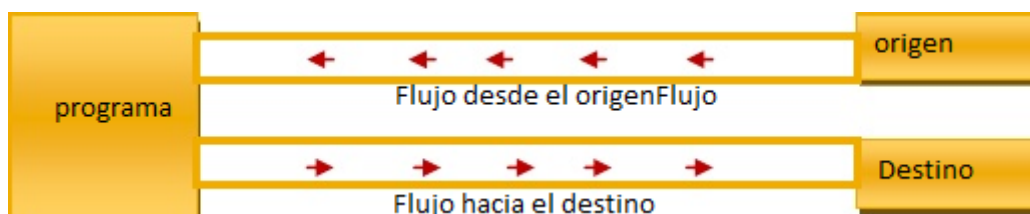
- Será capaz de trabajar con datos provenientes de distintas entradas: teclado, archivos, etc.
- Conocerá las clases que implementan estos flujos de entrada y salida.
- Trabajará con ficheros de acceso aleatorio.

## 1. INTRODUCCIÓN

Un **flujo** es un objeto que hace de intermediario entre el programa, y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el flujo sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que hace que la programación sea más fácil.

Los algoritmos para leer y escribir datos son siempre, más o menos, los mismos:

Leer	Escribir
Abrir un flujo desde un origen. Mientras haya información leer información. Cerrar el flujo.	Abrir un flujo hacia un destino. Mientras haya información escribir información. Cerrar el flujo.



## 2. CONCEPTO DE FLUJOS DE E/S

Ahora sabremos que significan los siguientes conceptos de flujos:

- Un flujo de **entrada/salida** (I/O stream, Input/Output stream) representa una fuente desde la cual se reciben datos o un destino hacia el cual se envían datos.
- Un flujo de datos puede **provenir o dirigirse** hacia archivos en disco, dispositivos de comunicaciones, otros programas o arrays en memoria.
- Los datos pueden ser **bytes, tipos primitivos, caracteres** propios de un idioma local, u **objetos**.



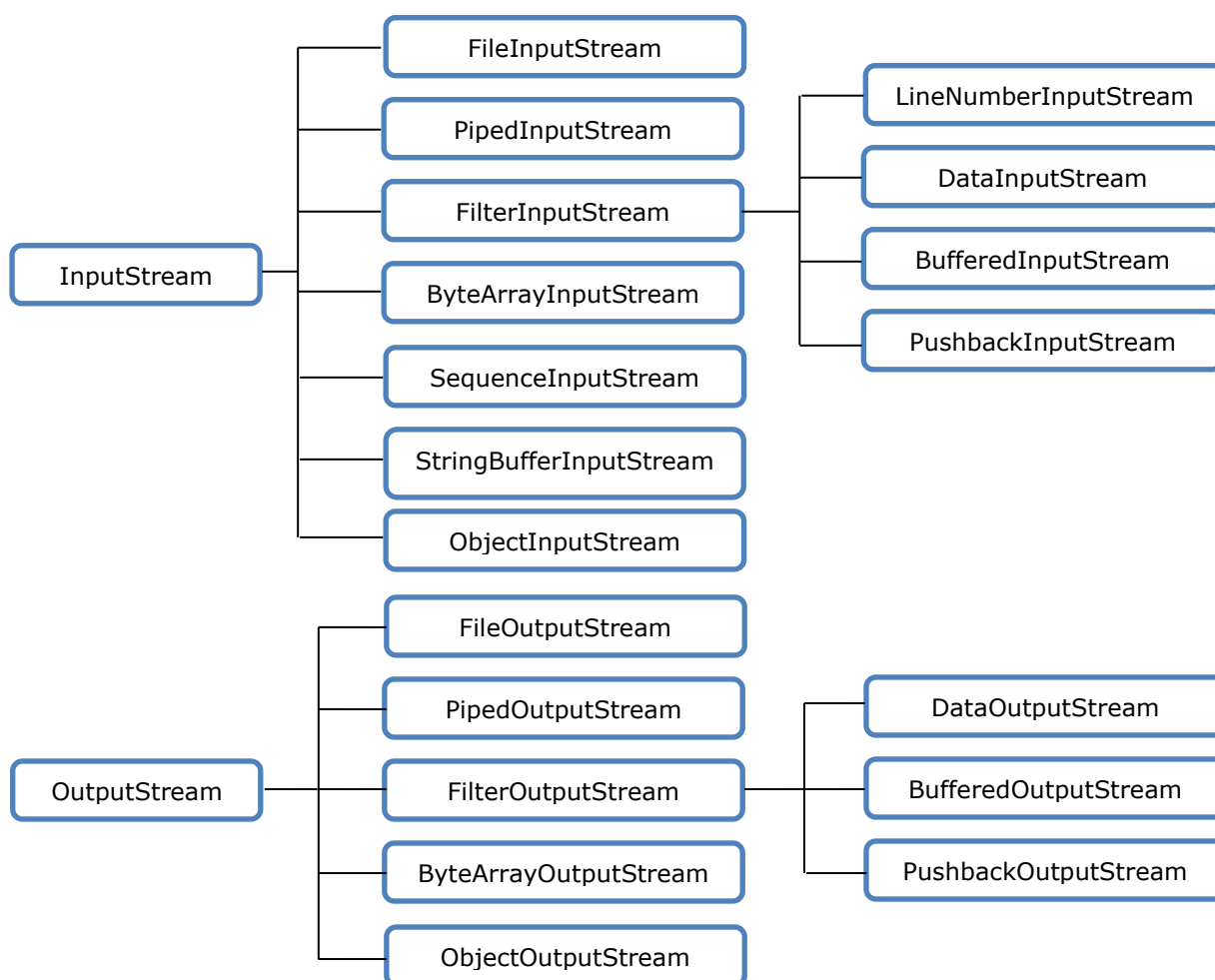
### ARTÍCULO DE INTERÉS

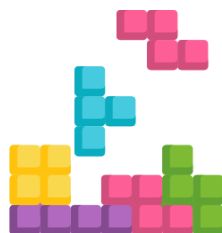
Para ampliar información sobre los flujos y algunas clases utilizadas se recomienda visitar:

- [Flujos de datos](#)

## 2.1 Flujos de bytes (byte streams)

Los flujos de bytes realizan operaciones de entrada y salida basándose en **bytes** de 8 bits. Todas las clases de flujos de bytes descienden (heredan) de las clases `InputStream` y `OutputStream`. Las clases `FileInputStream` y `FileOutputStream` manipulan flujos de bytes provenientes o dirigidos hacia archivos en disco.





### EJEMPLO PRÁCTICO

El siguiente ejemplo copia su propio texto fuente desde Archivo.txt hacia CopiaBytes.txt

```
package flujosbytes;

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

public class CopiaBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try { //Se crean los flujos de entrada y salida
            in = new FileInputStream("Archivo.txt");
            out = new FileOutputStream("CopiaBytes.txt");
            int c; //Cada byte se guarda en una variable de tipo int
            while ((c=in.read())!=-1) {
                out.write(c);
            }
        } finally {
            if (in!=null)
                in.close();
            if (out!=null)
                out.close();
        }
    }
}
```

El método read() devuelve un valor entero, lo cual permite indicar con el valor -1 el final del flujo. El tipo primitivo int puede almacenar un byte. Mantener flujos abiertos implica un gasto de recursos; deben cerrarse estos flujos para evitar malgastar recursos. El programa anterior cierra los flujos en el bloque finally. En este bloque se verifica que los flujos fueron efectivamente creados (sus referencias no son null) y luego se cierran.



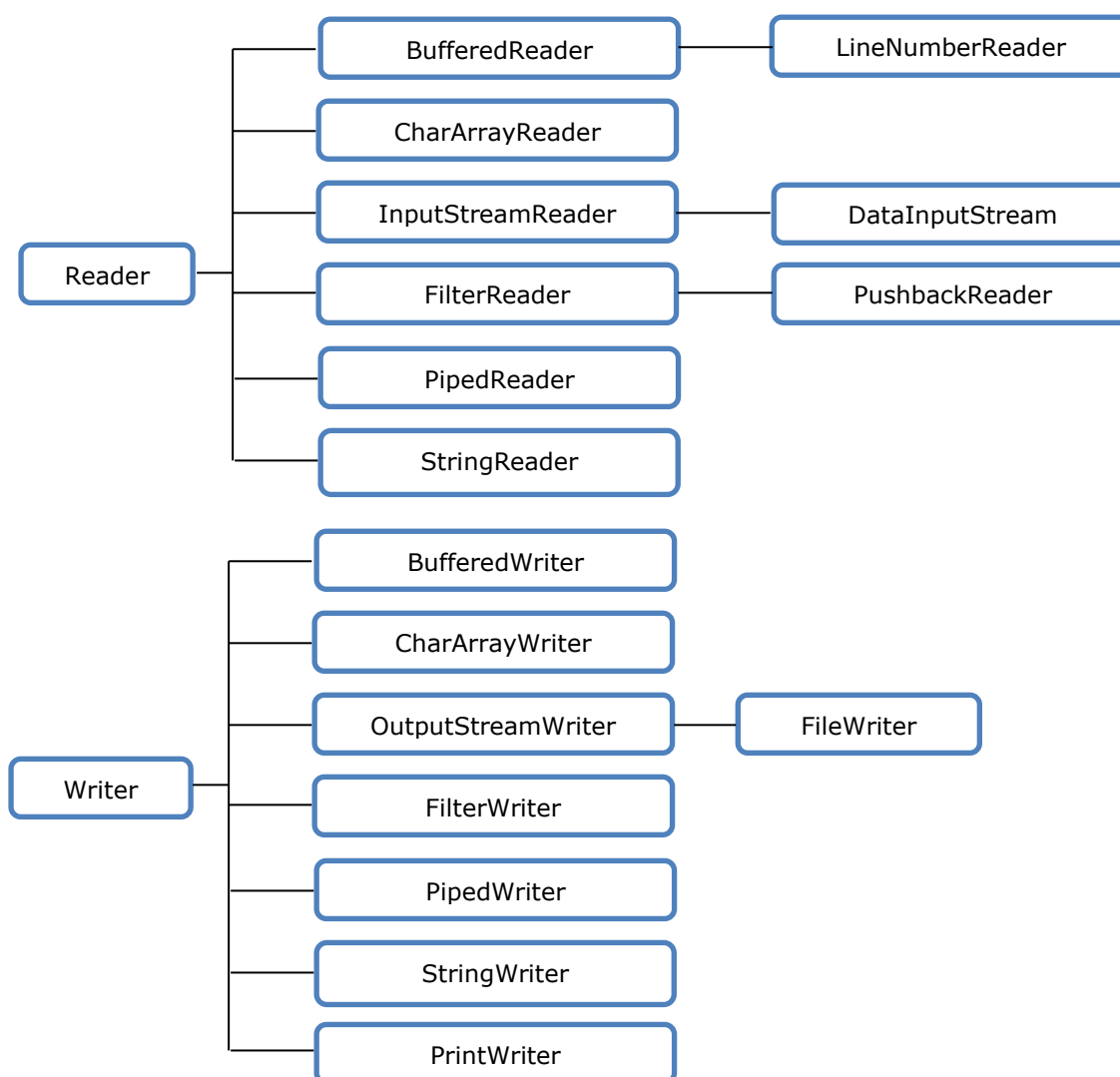
### ENLACE DE INTERÉS

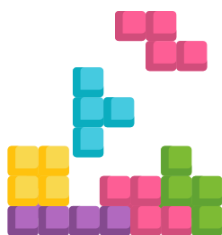
Si quiere saber más sobre la clase InputStream visite el enlace a continuación:

- [Java:InputStream](#)

## 2.2 Flujos de caracteres

El uso de flujos de bytes sólo es apto para las operaciones más elementales de entrada salida; es preciso usar los flujos más adecuados según los tipos de datos a manejar. En el ejemplo anterior, como se sabe que es un archivo con caracteres, lo mejor es usar los flujos de caracteres definidos en las clases `FileReader` y `FileWriter`. Estas clases heredan de `Reader` y `Writer`, y están destinadas a la lectura y escritura de caracteres en archivos.





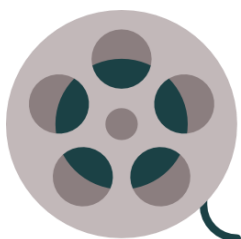
### EJEMPLO PRÁCTICO

Ahora veremos un ejemplo:

```
package flujosbytes;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopiaCaracteres {
    public static void main(String[] args) throws IOException {
        FileReader in = null;
        FileWriter out = null;
        try { //Se crean los flujos de entrada y salida
            in = new FileReader("Archivo.txt");
            out = new FileWriter("CopiaCaracteres.txt");
            int c; //Cada byte se guarda en una variable de tipo int
            while ((c=in.read())!=-1) {
                out.write(c);
            }
        } finally {
            if (in!=null)
                in.close();
            if (out!=null)
                out.close();
        }
    }
}
```



### VIDEO DE INTERÉS

En este caso se recomienda un **canal completo** en el que se muestran pequeños vídeos sobre la programación en Java. En concreto tiene dos videos muy interesantes sobre Entrada y Salida en Java.

- [Curso de Java desde 0](#)

## 2.3 Flujos de líneas

Para la lectura y escritura por líneas se emplean las clases `BufferedReader` y `PrintWriter`.





### ENLACE DE INTERÉS

En el siguiente enlace puedes analizar un ejemplo completo utilizando la clase `BufferedReader`.

- [Java – Clase `java.io.BufferedReader`](#)



### EJEMPLO PRÁCTICO

Ahora veremos un ejemplo:

```
package flujosbytes;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Copialineas {
    public static void main(String[] args) throws IOException {
        BufferedReader in = null;
        PrintWriter out = null;
        try { //Se crean los flujos de entrada y salida
            in = new BufferedReader(new FileReader("Archivo.txt"));
            out = new PrintWriter(new FileWriter("CopiaLineas.txt"));
            String linea; //Cada byte se guarda en una variable de tipo String
            while ((linea=in.readLine())!=null) {
                out.println(linea);
            }
        } finally {
            if (in!=null)
                in.close();
            if (out!=null)
                out.close();
        }
    }
}
```

El ejemplo anterior usa un flujo de entrada con "**buffer**". Un **buffer** es un área de memoria utilizada como almacenamiento intermedio para mejorar la eficiencia de las operaciones de entrada salida: escribir o leer de memoria es mucho más rápido que escribir o leer de dispositivos periféricos.

Cuando se usan buffers sólo se lee o escribe en el dispositivo final cuando el buffer está **lleno**, reduciendo la cantidad de operaciones de lectura y escritura sobre los dispositivos lentos (más lentos que la memoria). Las clases disponibles para entrada salida con buffer son `BufferedInputStream` y `BufferedOutputStream` para flujos de bytes, `BufferedReader` y `BufferedWriter` para flujos de caracteres. `Reader` y `Writer` son las clases bases de la jerarquía para los flujos de caracteres. Para leer o escribir datos binarios tales como imágenes o sonidos.

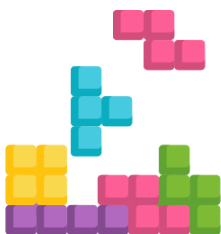
### 3. ENTRADA SALIDA DESDE LA LÍNEA DE COMANDOS

En Java existen varios flujos para la interacción con el usuario en la línea de comandos. Estos flujos se denominan flujos estándar (standard streams), y son comunes en varios sistemas operativos. Por defecto estos flujos leen del teclado y escriben en pantalla. Estos flujos pueden redirigirse hacia archivos u otros programas. En Java hay tres flujos estándar:

- La entrada estándar (**Standard Input**), accesible a través del objeto `System.in`;
- La salida estándar (**Standard Output**), accesible a través del objeto `System.out`;
- La salida de mensajes de error, accesible a través del objeto `System.err`;

Estos objetos se definen automáticamente y no requieren ser abiertos. La entrada estándar está asignada al teclado. La salida estándar está asignada a la pantalla. Para usar la entrada estándar como flujo de caracteres se "envuelve" el objeto `System.in` en un objeto `InputStreamReader`.

```
InputStreamReader cin = new InputStreamReader(System.in);
```



#### EJEMPLO PRÁCTICO

El siguiente ejemplo solicita al usuario ingresar una línea de caracteres, finalizando con la tecla Enter, y la muestra después en pantalla.

```

package flujosbytes;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EntradaEstandar {

    public static void main(String[] args) throws IOException {
        BufferedReader stdin = null;
        stdin = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Introduzca caracteres y pulse Return: ");
        String linea = stdin.readLine();
        System.out.println("Ha escrito: "+linea);
        stdin.close();
    }
}

```



### ARTÍCULO DE INTERÉS

Una alternativa a los flujos estándar es Console. En el siguiente enlace podrás aprender más sobre esta clase:

- [Uso de java.io.Console](#)

## 4. FLUJOS DE DATOS

Los flujos de datos soportan operaciones de entrada/salida de datos de tipo primitivo (boolean, char, byte, short, int, long, float, y double) así como cadenas de caracteres (String).



### EJEMPLO PRÁCTICO

El siguiente ejemplo escribe en un archivo una serie de datos correspondientes a una factura de venta, los vuelve a leer y los muestra en pantalla (Parte 1 de 2 – Escritura de datos en el archivo Factura.txt).

```
package flujosbytes;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FlujoDeDatos {

    static final String archDatos = "Factura.txt";
    static final double[] precios = {18.00, 160.00, 25.00, 14.00, 2.50};
    static final int[] cants = {4,2,1,4,50};
    static final String[] items = {"Marcador Azul", "Papel A4 500 hojas", "Borrador", "DVD", "Sobres A4"};
    static void main(String[] args) throws IOException {
        DataOutputStream out = null;
        try {
            out = new DataOutputStream (new BufferedOutputStream(new FileOutputStream(archDatos)));
            for (int i = 0; i < precios.length; i++) {
                out.writeDouble(precios[i]);
                out.writeInt(cants[i]);
                out.writeUTF(items[i]);
            }
        } finally {
            out.close();
        }
    }
}
```



## EJEMPLO PRÁCTICO

El siguiente ejemplo escribe en un archivo una serie de datos correspondientes a una factura de venta, los vuelve a leer y los muestra en pantalla (Parte 2 de 2 – Lectura de datos del archivo Factura.txt).

```
DataInputStream in = null;
double total = 0.0;
try {
    in = new DataInputStream (new BufferedInputStream(new FileInputStream(archDatos)));
    double precio;
    int cant;
    String item;
    try {
        while(true){
            precio = in.readDouble();
            cant = in.readInt();
            item = in.readUTF();
            System.out.format(" %4d %25s a %6.2f€ c/u %8.2f€\n", cant, item, precio, cant * precio);
            total += cant*precio;
        }
    } catch (EOFException e) {
        System.out.format("\t\t\t\t\t TOTAL %8.2f€\n", total);
    }
} finally {
    in.close();
}
}
```

Este programa produce la siguiente salida por la consola:

4	Marcador Azul a	18,00€ c/u	72,00€
2	Papel A4 500 hojas a	160,00€ c/u	320,00€
1	Borrador a	25,00€ c/u	25,00€
4	DVD a	14,00€ c/u	56,00€
50	Sobres A4 a	2,50€ c/u	125,00€
	TOTAL		598,00€

Después de importar las clases, el programa define variables estáticas para el nombre de archivo (archDatos) y arrays para los componentes de cada línea de factura (precios, cants, items). El flujo de salida sólo puede ser creado como envoltorio de un objeto flujo de bytes existente, por lo que se crea uno con `new BufferedOutputStream(...)`, que a su vez requiere un objeto existente de flujo de salida hacia archivo, que se crea con `new FileOutputStream(archDatos)`, todo en la sentencia:

```
out=new DataOutputStream (new BufferedOutputStream(new  
FileOutputStream(archDatos)));
```

Los elementos de cada array se escriben usando métodos propios de sus tipos de datos (`writeDouble()`, `writeInt()`, `writeUTF()`) para los caracteres en el tipo `String`.

La lectura de los datos requiere un flujo de entrada, que se construye también como envoltorio de un objeto flujo de bytes existente, en la sentencia

```
in = new DataInputStream(new BufferedInputStream(new  
FileInputStream(archDatos)));
```

La lectura de los datos se realiza también con métodos propios de sus tipos de datos: `readDouble()`, `readInt()`, `readUTF()`.

El método `format()` disponible en el objeto `System.out`, que es de tipo `PrintStream`, permite dar formato a la línea de salida. El fin de archivo se detecta a través de la captura de la excepción `EOFException`.

Para valores monetarios existe un tipo especial, `java.math.BigDecimal`. No se ha usado en este ejemplo por ser objetos y no tipos primitivos; los objetos no pueden tratarse como flujos de datos, deben tratarse como flujos de objetos.

## 5. FLUJOS DE OBJETOS

Los flujos de objetos permiten realizar operaciones de entrada salida de objetos. Muchas de las clases estándar soportan serialización de sus objetos, implementando la interfaz `Serializable`. La serialización de objetos permite guardar el objeto en un archivo escribiendo sus datos en un flujo de bytes. Es posible luego leer desde el archivo el flujo de bytes y reconstruir el objeto original. Las clases de flujos de objetos son `ObjectInputStream` y `ObjectOutputStream`. Estas clases implementan las interfaces `ObjectInput` y `ObjectOutput`, subinterfaces de `DataInput` y `DataOutput`.

En consecuencia, todos los métodos de entrada/salida que estaban disponibles para flujos de datos primitivos estarán implementados también para flujos de objetos.

El siguiente programa implementa la misma aplicación pero usando objetos `BigDecimal` para los precios, y un objeto `Calendar` para la fecha. Si el método `readObject()` no devuelve el tipo correcto, el casting puede lanzar la excepción `ClassNotFoundException`, lo cual es notificado en el método `main()` mediante la cláusula `throws`.



### EJEMPLO PRÁCTICO

Flujo de objetos (1 de 2):

```
package flujosbytes;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.math.BigDecimal;
import java.util.Calendar;

public class FlujoDeObjetos {

    static final String archDatos = "Factura.txt";
    static final BigDecimal[] precios = {new BigDecimal(18.00), new BigDecimal(160.00), new BigDecimal(25.00),
        new BigDecimal(14.00), new BigDecimal(2.50)};
    static final int[] cants = {4,2,1,4,50};
    static final String[] items = {"Marcador Azul", "Papel A4 500 hojas", "Borrador", "DVD", "Sobres A4"};

    public static void main(String[] args) throws ClassNotFoundException, IOException {
        ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(archDatos)));
            out.writeObject(Calendar.getInstance());
            for (int i = 0; i < precios.length; i++) {
                out.writeObject(precios[i]);
                out.writeInt(cants[i]);
                out.writeUTF(items[i]);
            }
        } finally {
            out.close();
        }
    }
}
```



## EJEMPLO PRÁCTICO

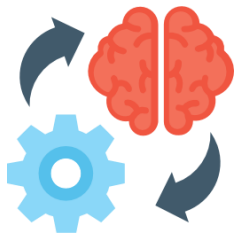
Flujo de objetos (2 de 2):

```
ObjectInputStream in = null;
try {
    in = new ObjectInputStream (new BufferedInputStream(new FileInputStream(archDatos)));
    Calendar fecha = null;
    BigDecimal precio;
    int cant;
    String item;
    BigDecimal total = new BigDecimal("0");
    fecha = (Calendar) in.readObject();
    System.out.format("El %tA, %<tB %<te, %<tY, se compró:%n", fecha);
    try {
        while(true){
            precio = (BigDecimal) in.readObject();
            cant = in.readInt();
            item = in.readUTF();
            System.out.format(" %4d %25s a %6.2f€ c/u %8.2f€%n", cant, item, precio,
                precio.multiply(new BigDecimal(cant)));
            total = total.add(precio.multiply(new BigDecimal(cant)));
        }
    } catch (EOFException e) {
        System.out.format("\t\t\t\t\t TOTAL %8.2f€%n", total);
    }
    finally {
        in.close();
    }
}
}
```

La fecha será la fecha del día en que se ejecute el programa:

```
El jueves, agosto 9, 2018, se compró:
  4      Marcador Azul a 18,00€ c/u    72,00€
  2      Papel A4 500 hojas a 160,00€ c/u 320,00€
  1      Borrador a 25,00€ c/u    25,00€
  4      DVD a 14,00€ c/u    56,00€
 50      Sobres A4 a 2,50€ c/u    125,00€
                        TOTAL    598,00€
```

Existen operaciones de entrada salida sobre archivos que no pueden tratarse como flujos de datos. La clase File permite examinar y manipular archivos y directorios, de forma independiente de la plataforma (MS Windows, Linux, MacOS). Los archivos se pueden acceder también en forma no secuencial o aleatoria (random access files); existen clases específicas para acceder a los archivos sin necesidad de recorrerlos ordenadamente.

**RECUERDA**

Todos los métodos de entrada/salida que estaban disponibles para flujos de datos primitivos estarán implementados también para flujos de objetos.

## 6. OBJETOS DE TIPO FILE

Las instancias de la clase **File** representan nombres de archivo, no a los archivos en sí. El archivo correspondiente a un nombre puede no existir.

Un objeto de clase **File** permite examinar el nombre del archivo, descomponerlo en su rama de directorios, o crear el archivo si no existe pasando el objeto de tipo **File** a un constructor adecuado como **FileWriter(File f)**. Que recibe como parámetro un objeto **File**.

Para archivos existentes, a través del objeto **File** un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Estas operaciones pueden hacerse independientemente de la plataforma sobre la que esté corriendo el programa.

Si el objeto **File** se refiere a un archivo existente un programa puede usar este objeto para realizar una serie de operaciones sobre el archivo:

- **delete()** borra el archivo inmediatamente;
- **deleteOnExit()** lo borra cuando finaliza la ejecución de la máquina virtual Java.
- **setLastModified()** permite fijar la fecha y hora de modificación del archivo:

```
new File("factura.txt").setLastModified(new Date().getTime());
```

- **renameTo()** permite renombrar el archivo.
- **mkdir()** crea un directorio, **makedirs()** también, pero crea los directorios superiores si no existen.
- **list()** y **listFiles()** listan el contenido de un directorio. **list()** devuelve un array de **String** con los nombres de los archivos, **listFiles()** devuelve un array de objetos **File**.



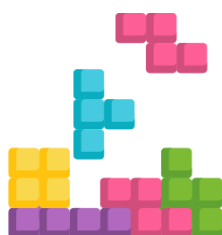
- **createTempFile()** crea un nuevo archivo con un nombre único y devuelve un objeto `File` que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándose de tener un nombre de archivo no repetido.
- **listRoots()** devuelve una lista de nombres de archivo correspondientes a la raíz de los sistemas de archivos. En Microsoft Windows serán de formato `a:\` y `c:\`, en UNIX, MacOS y Linux será el directorio raíz único `/`.



### ARTÍCULO DE INTERÉS

Se recomienda realizar la siguiente lectura para ampliar y clarificar conceptos sobre la clase `File` disponible en Java:

- [La clase File](#)



### EJEMPLO PRÁCTICO

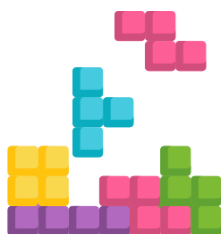
El siguiente ejemplo lista los archivos en el directorio raíz `c:\`

```
package flujosbytes;

import java.io.File;

public class Dir {

    public static void main(String[] args) {
        System.out.println("Archivos en el directorio actual: ");
        File ficheros = new File("c:\\");
        String[] archivos = ficheros.list();
        for (int i = 0; i < archivos.length; i++) {
            System.out.println(archivos[i]);
        }
    }
}
```



### EJEMPLO PRÁCTICO

El resultado de ejecutar el programa anterior podría ser, por ejemplo:

```
Archivos en el directorio actual:
$Recycle.Bin
$SysReset
Archivos de programa
Documents and Settings
hiberfil.sys
Intel
pagefile.sys
PerfLogs
Program Files
Program Files (x86)
ProgramData
Recovery
swapfile.sys
swsetup
System Volume Information
System.sav
Users
Windows
```

## 7. ARCHIVOS DE ACCESO ALEATORIO

Un **archivo de acceso aleatorio** permite leer o escribir datos en forma no secuencial. El contenido de un archivo suele consistir en un conjunto de partes o registros, generalmente de distinto tamaño.

La **búsqueda de información en el archivo** equivale a ubicar un determinado registro. En el **acceso secuencial** es preciso leer el archivo pasando por todos sus registros hasta llegar al registro que se desea ubicar. En promedio debe leerse la mitad del archivo en cada búsqueda. Si el tamaño de los registros es conocido puede crearse un **índice** con punteros hacia cada registro. La búsqueda de un registro comienza entonces por ubicar ese registro en el índice, obtener un puntero hacia el lugar del archivo donde se encuentra el contenido de ese registro, y desplazarse hacia esta posición directamente. El acceso aleatorio descrito es **mucho más eficiente** que el acceso secuencial.

La clase `java.io.RandomAccessFile` implementa las interfaces `DataInput` y `DataOutput`, lo cual permite leer y escribir en el archivo. Para usar `RandomAccessFile` se debe indicar un **nombre de archivo** para abrir o crear si no existe.

Se debe indicar también si se abrirá para **lectura** o también para **escritura** (para poder escribir es necesario también poder leer).

La siguiente sentencia abre un archivo de nombre archiuno.txt para lectura, y la siguiente abre el archivo archidos.txt para lectura y escritura:

```
//Archivo de solo lectura

RandomAccessFile f1 = new RandomAccessFile("archiuno.txt","r");

//Archivo de lectura/escritura

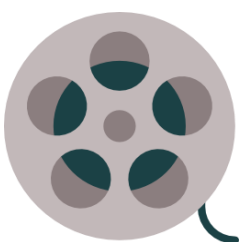
RandomAccessFile f2 = new RandomAccessFile("archidos.txt","rw");
```

Una vez abierto el archivo pueden usarse los métodos **read()** o **write()** definidos en las interfaces **DataInput** y **DataOutput** para realizar operaciones de entrada/salida sobre los archivos.

La clase **RandomAccessFile** maneja un puntero al archivo (file pointer). Este puntero indica la **posición actual** en el archivo. Cuando el archivo se crea, el puntero al archivo se coloca en el 0, apuntando al principio del archivo. Las sucesivas llamadas a los métodos **read()** y **write()** ajustan el puntero según la cantidad de bytes leídos o escritos.

Además de los métodos de entrada/salida que ajustan el puntero automáticamente, la clase **RandomAccessFile** tiene métodos específicos para manipular el puntero al archivo:

- **int skipBytes(int):** mueve el puntero hacia adelante la cantidad especificada de bytes.
- **void seek(long):** ubica el puntero justo antes del byte especificado en el entero long.
- **long getFilePointer():** devuelve la posición actual del puntero, el número de byte indicado por el entero long devuelto.



#### VIDEO DE INTERÉS

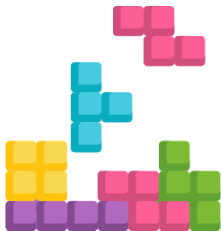
En el siguiente vídeo se muestra como trabajar con flujos de datos que son entrada y salida de información desde consola y en modo gráfico.

- [Entrada y Salida de datos en java](#)

El siguiente ejemplo muestra el uso de un archivo de acceso aleatorio y el valor de los punteros. Crea una tabla de raíces cuadradas de los números del 0 al 9 expresada como decimales doble precisión tipo `double`, de tamaño 8 bytes.

Realizar las siguientes tareas:

- Calcular los cuadrados, guardarlos en un archivo de acceso aleatorio y cerrar el archivo.
- Abrir el archivo recién creado, desplazar el puntero 40 bytes (5 `double` de 8 bytes cada uno),
- Leer el registro ubicado a partir del byte 40 (raíz cuadrada del número 5: 2,23...),
- Verificar el avance del puntero a 48 (avanzó un `double` en la lectura), cambiar su valor por el número arbitrario 333.0003 y cerrar el archivo.
- Abrir nuevamente el archivo, ahora en sólo lectura, y mostrar punteros y valores.
- Intentar escribir en el archivo de sólo lectura, lanzando y capturando la excepción.



### EJEMPLO PRÁCTICO

Parte 1:

```
package aleatorio;

import java.io.IOException;
import java.io.RandomAccessFile;

public class Aleatorio {

    public static void main(String[] args) {
        RandomAccessFile rf = null;
        try {
            rf = new RandomAccessFile("dobles.dat", "rw");
            for (int i = 0; i < 10; i++) {
                rf.writeDouble(Math.sqrt(i));
            }
            rf.close();
        } catch (IOException e) {
            System.out.println("Error de E/S parte 1:\n"+e.getMessage());
        }
        try {
            rf = new RandomAccessFile("dobles.dat", "rw");
            rf.seek(5*8); //Avanza el puntero 5 registros * 8 bytes
            System.out.println("\nPuntero antes de lectura: "+rf.getFilePointer());
            System.out.println(" Valor:"+rf.readDouble());
            System.out.println("\nPuntero después de lectura: "+rf.getFilePointer());
            rf.seek(rf.getFilePointer()-8); //Restaura el puntero a registro 6
            System.out.println("\nPuntero restaurado: "+rf.getFilePointer()+"\n");
            rf.writeDouble(333.0003); //Cambia el registro 6 y avanza el puntero
            rf.close();
        } catch (IOException e) {
            System.out.println("Error de E/S parte 2:\n"+e.getMessage());
        }
    }
}
```



## EJEMPLO PRÁCTICO

Parte 2:

```
try {
    rf = new RandomAccessFile("dobles.dat", "r");
    for (int i = 0; i < 10; i++) {
        System.out.print("\nPuntero en: "+rf.getFilePointer()+" ");
        System.out.println("valor: "+rf.readDouble());
    }
    rf.writeDouble(1.111); //Intento de escribir en archivo de solo lectura
    rf.close();
} catch (IOException e) {
    System.out.println("Error de E/S parte 3:\n"+e.getMessage());
}
}
```

Resultado de la ejecución del programa:

```
Puntero antes de lectura: 40
Valor:2.23606797749979

Puntero después de lectura: 48

Puntero restaurado: 40

Puntero en: 0 valor: 0.0
Puntero en: 8 valor: 1.0
Puntero en: 16 valor: 1.4142135623730951
Puntero en: 24 valor: 1.7320508075688772
Puntero en: 32 valor: 2.0
Puntero en: 40 valor: 333.0003
Puntero en: 48 valor: 2.449489742783178
Puntero en: 56 valor: 2.6457513110645907
Puntero en: 64 valor: 2.8284271247461903

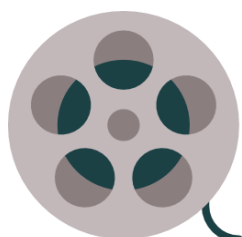
Puntero en: 72 valor: 3.0
Error de E/S parte 3:
Acceso denegado
```



### ENLACE DE INTERÉS

Se recomienda la visión de los siguientes enlaces que complementan el temario desarrollado en Java:

- [Entrada y Salida con Java](#)
- [Ejercicios propuestos y resueltos de ficheros en Java](#)



### VIDEO DE INTERÉS

A continuación, encontrará un video con ejemplos y ejercicios resueltos de ficheros en Java

- [Java - Ficheros](#)

## RESUMEN FINAL

Cualquier programa que se desarrolle en Java y que tenga la necesidad de recibir o enviar datos lo hará a través de lo que se ha definido como un flujo (*stream*). La vinculación de un stream a con un dispositivo físico concreto la va a realizar Java. Por lo tanto, las clases y los métodos que utilicemos van a ser las mismas sin tener en cuenta el dispositivo con el cual vamos a interactuar. Java se va a encargar de realizar esa tarea y será el que se comunique con el teclado, el monitor o cualquier otro dispositivo.

En esta unidad se ha comenzado analizando el concepto de flujo de entrada/salida. Posteriormente se han clasificado los flujos dependiendo de si están orientados a bytes, a caracteres o a líneas. Se ha visto cómo se realiza la entrada y la salida desde la línea de comandos. Más adelante se ha centrado la atención en los flujos de datos y los flujos de datos, analizando el concepto de serialización. Después se ha conocido qué es el objeto File para finalizar con los archivos de acceso aleatorio.