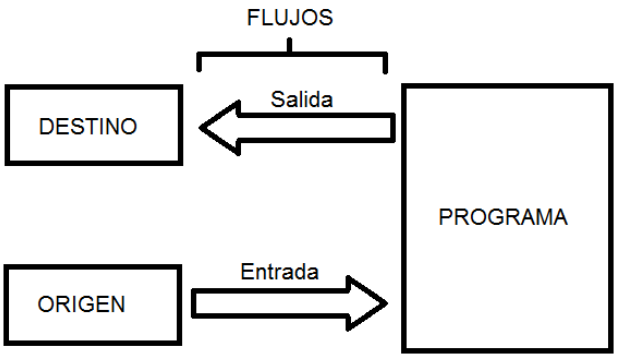
FLUJOS DE ENTRADA Y SALIDA

Los programas nos entregan información producto de las funciones que se le han definido, esta debe provenir de alguna fuente de datos (discos, CD-RW, memoria) y seguramente necesitaremos enviar esta información o datos hacia otro lugar para mostrarla o almacenarla. Estos son los flujos (en inglés stream) de información.



"Un flujo es una ruta seguida por los datos de un programa. Un flujo de entrada envía datos desde una fuente a un programa, y un flujo de salida envía datos desde un programa hacia un destino".

En JAVA, un flujo es un objeto que sirve de intermediario entre el origen y el destino de los datos. Esto tiene la ventaja que el programa leerá y escribirá en el flujo de información sin importar el origen o el destino (la pantalla, un archivo, la memoria, Internet, etc.). Además, tampoco va a tener relevancia el tipo de dato que se encuentra en este objeto. Por otro lado, esto significa un nuevo nivel de abstracción pues al programa ya no le importa saber nada acerca del dispositivo del cual vienen o al cual van los datos.

Así, para leer información el programa tiene que abrir un flujo (objeto), de la misma manera que tiene que hacerlo para escribirla o enviarla. Para ello JAVA contiene una serie de clases que son parte del paquete java.io. Un programa que use flujos de entrada/salida (E/S) deberá importar el paquete: `import java.io.*.`

Existen dos tipos de flujos: los de Entrada, que sirven para leer datos, y los de Salida, que se usan para guardar datos. En ambos casos, los flujos pueden ser flujos de bytes o flujos de caracteres.

Los Flujos de Bytes, se utilizan para manejar bytes, enteros u otros tipos simples en el flujo, con valores que van desde 0 a 255.

Los Flujos de Caracteres, manejan archivos de texto u otras fuentes de texto. Cualquier clase de datos que comprenda texto debería utilizar este tipo de flujos.

1. Flujos de Entrada

La clase **InputStream** es la que se encarga de establecer el flujo de bytes de entrada de información. Es una clase abstracta que es superclase de todas las subclases que representan este flujo. En este sentido, el método más importante es *read(),* que son varios métodos sobrecargados que leen bytes ya sea individualmente o en conjunto.

Por otro lado, tenemos la superclase **Reader**, que al igual que la superclase InputStream lee desde el origen, pero sus subclases leen caracteres, es decir, *char* en vez de *byte*.

2. Flujos de Salida

La clase **OutputStream** es la superclase de todas las clases que representan un flujo que se encarga de escribir *bytes* un destino. Su método más importante es *write()*

También, existen otra superclase que en vez de escribir o enviar bytes escribe caracteres: **Writer**.

3. Flujo E/S Estándar

El paquete `java.lang` proporciona, por intermedio de la clase **System** tres flujos que son abiertos una vez que el programa se carga en memoria para el uso de la salida estándar, normalmente el monitor:

System.in, que es una subclase de la clase **InputStream** y que hace referencia a la entrada estándar del sistema, normalmente el teclado. Se usa para leer datos introducidos por el usuario.

System.out, que es subclase de **PrintStream** que a su vez es subclase de **OutputStream** y que hace referencia a la salida estándar. Se utiliza para mostrar datos al usuario.

System.err, que es subclase de **PrintStream** que a su vez es subclase de **OutputStream** y que hace referencia a la salida estándar. Se utiliza para mostrar mensajes de error al usuario.

4. Flujo E/S desde y hacia Archivos

Una gran parte de la vida de una aplicación en memoria, se dedicará a trabajar con archivos o archivos, para intercambiar datos entre distintos dispositivos de almacenamiento, que se utilizan refiriéndolos con una ruta de directorio y un nombre.

Las operaciones básicas que se realizan con un archivo son: abrir el archivo, leer datos o escribirlos y cerrar el archivo.

a. Flujos de Bytes.

Esta es la manera básica de leer o escribir sobre un archivo. Para leer bytes se debe abrir un flujo de entrada hacia el origen, y para escribir sobre él se abre un flujo de salida hacia el destino.

Para crear el flujo de entrada del archivo se usa un objeto de la clase **FileInputStream**, a cuyo constructor se le pasa como argumento una cadena que contiene la ruta del archivo y su nombre, con su extensión, o sólo el nombre del archivo si se encuentra en el directorio actual de trabajo. Por ejemplo:

```
FileInputStream archivo = new FileInputStream("Ruta\Nombre.ext");
```

Una vez creado el flujo, se podrán leer bytes desde el flujo usando el método *read(),* el cual devuelve un valor entero que contiene la representación del próximo byte en el flujo y devuelve -1 cuando llega al final del flujo.

```
/* Lee los bytes desde un archivo */
import java.io.*;
public class EjemFile {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

     String archivo = "";
     int enteroByte = 0;

      System.out.print("Ingrese el nombre del archivo: ");
      archivo = s.nextLine();

     try{
            FileInputStream f = new FileInputStream(archivo);

            enteroByte = f.read();

            while (enteroByte != -1){
                System.out.print((char)enteroByte);
                enteroByte = f.read();
            }

            f.close();
     }
     catch (IOException e){
            System.out.println(e);
     }
    }
}
```

Este programa lee bytes desde un archivo existente. La ruta y/o nombre del archivo se ingresa por teclado. Se debe considerar la forma como cada sistema operativo maneja el acceso a sus directorios; así si se trabaja con sistemas UNIX o Linux se usa la barra inclinada normal (/) para separar los directorios, en cambio, es sistemas DOS y Windows se usa la barra invertida (\), pero como representa la secuencia de escape se debe usar así: '\\'. En este programa, se crea el flujo, se lee el archivo byte a byte y se imprimen hasta el fin del archivo con el que se está trabajando.

Sin embargo, en este caso, estamos leyendo el archivo byte a byte. Sería una mejor solución, traer a memoria una mayor cantidad de información para trabajar con ella en vez de ir leyendo cada vez desde el archivo, mejorando así la velocidad de ejecución de la aplicación. Esto se hace a través de un buffer, "...un lugar donde se pueden guardar datos antes de ser utilizados por un programa...".

En el caso del flujo de entrada, el buffer se llena con datos que no se han utilizado aún. Y cuando el programa los requiera los encontrará en el buffer antes de ir a buscarlo a la fuente origen (archivo físico). La clase con la que se logra esto es: **BufferedInputStream**, que crea un flujo de entrada almacenado en un buffer reservado para un objeto **InputStream** que se le pasa como argumento y que, en este caso particular, será un objeto **FileInputStream**. Para leer datos desde la entrada del buffer se usa el método *read(),* sin argumentos, el cual devuelve un entero de de valor 0 a 255 que representa el byte leído. Si llega al final del flujo y no quedan datos devuelve un -1. Para ejemplificar, tomaremos el mismo ejemplo anterior y le agregaremos una línea y modificaremos otra:

```
...
try{
       FileInputStream f = new FileInputStream(archivo);
       BufferedInputStream buffer = new BufferedInputStream(f);

           enteroByte = buffer.read();
...
```

En el caso de los flujos de salida, este se crea con la clase **FileOutputStream**, a la cual se le pasa como argumento la ruta y el nombre del archivo. Si el archivo ya existe, se borrará su contenido. En caso de no existir se creará, pero si existe, y se quiere agregar su contenido al final del archivo existe el constructor *FileOutputStream (String rutaArchivo, [true/false]),* que permite esta operación cuando se pasa *true* como el argumento booleano.

Es el método *write(Entero)* el que permite escribir valores enteros y bytes en el flujo. En este caso, es necesario cerrar expresamente el flujo hacia el archivo mediante el uso del método *close().*

Como ejemplo, el siguiente programa toma la entrada por teclado y la envía a un archivo de texto (datos.txt). Si el archivo no existe, lo creará, y si ya existe lo sobreescribirá. La entrada es por un flujo estándar la cual termina al presionar la tecla “Enter” (10):

```
import java.io.*;

public class EjemFileOut {

   public static void main(String[] args) {
      int letra;

      System.out.print("Ingrese un texto: ");

      try{
         FileOutputStream f = new FileOutputStream("datos.txt");

         do{
            letra = System.in.read();
            f.write(letra);
         }while(letra != 10);

         f.close();
      }
      catch (IOException e){
         System.out.println(e);
      }
   }
}
```

Para hacer lo anterior, pero sin perder el contenido original del archivo si este ya existe, si no que agregar la final los bytes que se escriben en el flujo, se escribe el programa exactamente igual, pero se utiliza el constructor que permite agregar los enteros a partir del final del archivo, para lo cual se reemplaza la siguiente línea:

```
FileOutputStream f = new FileOutputStream("datos.txt", true);
```

También se puede establecer un buffer para los flujos de salida a través de la clase **BufferOutputStream**, donde se pasa como argumento el objeto del flujo de salida establecido para el archivo a escribir. Se usa el método *write(entero)*, para escribir bytes en el archivo, que representen valores enteros de 0 a 255. Los datos se escriben en el flujo y sólo se escriben en el archivo cuando el buffer se llena o cuando se llama expresamente al método *flush()*.

```
import java.io.*;

public class EjemFileOut {
   public static void main(String[] args) {
      int letra;

      System.out.print("Ingrese un texto: ");

      try{
         FileOutputStream f = new FileOutputStream("datos.txt");
         BufferOutputStream buffer = new BufferOutputStream(f);

         do{
            letra = System.in.read();
            buffer.write(letra);
         }while(letra != 10);

         Buffer.flush();  // Vacia el buffer al archivo
         f.close();
      }
      catch (IOException e){
         System.out.println(e);
      }
   }
}
```

b. Flujos de Caracteres

Estos flujos se usan para trabajar con cualquier texto que represente un conjunto de caracteres UNICODE, que incluye los caracteres ASCII. Pueden ser archivos de texto, HTML, archivos fuentes, etc.

El ***flujo de entrada*** para un archivo de texto, se establece a través de la clase **FileReader**. Esta clase es subclase de **InputStreamReader**, la cual lee un flujo de bytes y los convierte a enteros "que representan" valores de los caracteres UNICODE. El método a utilizar es *read()*, el cual devuelve enteros, que es necesario convertir a *char* para visualizar los caracteres y devuelve -1 cuando se dejan de leer caracteres. Por ejemplo :

```
import java.io.*;
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner s = new Scanner(System.in);

    String archivo = "";
    int enteroByte = 0;

    System.out.print("Ingrese el nombre del archivo: ");
    archivo = s.nextLine();

    try{
      FileReader f = new FileReader(archivo);
      enteroByte = f.read();

      while (enteroByte != -1){
        System.out.print((char)enteroByte);
        enteroByte = f.read();
      }

      f.close();
    }
    catch (IOException e){
      System.out.println(e);
    }
  }
}
```

Al igual que en los casos anteriores, se puede usar un buffer para trabajar con los datos del flujo de entrada en memoria y no directamente con el archivo. Para ello se utiliza la clase **BufferadReader**, en donde se pasa como argumento un objeto *Reader* que sirve para comunicarse con el archivo, en este caso, **FileReader**. También tiene un método *read()* otro método interesante es que se usa para leer una línea entera del buffer: *readLine(),* que

devuelve un objeto *String* con la línea y un valor *null* para la cadena cuando ya no quedan más líneas. Por ejemplo, el mismo programa anterior pero con modificaciones:

```
try{
      FileReader f = new FileReader(archivo);
      BufferedReader buffer = new BufferedReader(f);
      String linea;

      while (linea != null){
        linea = buffer.readLine();
        if (linea != null) System.out.print(linea + "\n");
      }
      f.close();
```

El ***flujo se salida*** para escribir caracteres sobre archivos de texto, usa la clase **FileWrite**. Con uno de sus constructores, *FileWriter(String ruta_archivo),* se reemplaza el contenido del archivo original y lo reemplaza por el nuevo, si este existía; o en caso contrario; lo crea.

Ejemplo :

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

  public static void main(String[] args) {
    int letra;
    System.out.print("Ingrese un texto: ");

    try{
      FileWriter f = new FileWriter("texto.txt");

      do{
        letra = System.in.read();
        f.write(letra);
      }while(letra != 10);

      f.close();
    }
    catch (IOException e){
      System.out.println(e);
    }
  }
}
```

Por otro lado, si no queremos perder el contenido de nuestro archivo tenemos el constructor *FileWriter(String ruta/archivo, boolean [true/false])*, que escribe desde el final de un archivo existente cuando se pasa como parámetro el valor boleano "true". Para esto se remplaza la siguiente línea:

```
FileWriter f = new FileWriter("texto.txt", true);
```

Finalmente, se puede utilizar un buffer para escribir nuestros caracteres en memoria y después mandarlos al archivo. Para establecer el buffer, se usa el constructor *BufferedWriter(Writer)*, donde "*Writer*" es el objeto que se crea para establecer el flujo. También tiene el método *write()*, y el método *newLine()*, que manda el o los caracteres de "nueva línea" según la plataforma: en DOS retorno de carro más salto de línea , y en UNIX y sistemas similares salto de línea. Utilizar este método es mejor pues el programa se hace realmente multiplataforma. Se usa de la misma manera que lo hemos visto en los ejemplos anteriores. Sin embargo, para poder utilizar el buffer se requiere también el método *flush(),* para vaciar el contenido del buffer en el archivo.

```
try{
     FileWriter f = new FileWriter("texto.txt");
     BufferedWriter buffer = new BufferedWriter(f);

     do{
        letra = System.in.read();
        buffer.write(letra);
     }while(letra != 10);
```

c. Flujos de Datos

Los bytes y caracteres son datos también, pero por un archivo de datos entenderemos una colección de información que se almacena en algún soporte magnético. Estos pueden ser tipos primitivos u objetos. En el último caso, se habla de un conjunto de registros, que contienen los mismos campos, que pueden ser datos primitivos o un tipo definido por el usuario, es decir otro objeto referenciado por el objeto que se almacena. En la POO, se habla de objetos en vez de "registros" y de "atributos" en vez de campos.

Flujos de datos de tipos primitivos.

Muchas veces trabajar con bytes y caracteres puede llegar a ser muy restrictivo. JAVA permite manejar flujos con los tipos primitivos, de modo que el manejo de ellos sea transparente para la aplicación.

Un flujo de entrada de datos de tipos primitivos, se hace con la clase **DataInputStream**, a la cual se le pasa como argumento un objeto de flujo existente.

El flujo de salida, por su lado, es controlado por la clase **DataOutputStream**, que también recibe como argumento un objeto de flujo, esta vez de salida.

Los métodos de lectura y escritura para flujos de datos de tipos primitivos: boolean, byte, short, int, long, float y double son:

Entrada	Salida	Significado
readBoolean()	writeBoolean(boolean)	Lee-escribe tipo boolean
readByte()	writeByte(int)	Lee tipo Byte - escribe tipo Byte un valor entero
	writeBytes(String)	Escribe una cadena como secuencia de bytes
readChar()	writeChar(int)	Lee tipo Char - Escribe como Char un valor entero
readChars(String)		Lee una cadena como una secuencia de caracteres
readShort()	writeShort(int)	Lee tipo Short - Escribe como Short un valor entero
readInt()	writeInt(Int)	Lee - escribe tipo entero
readLong()	writeLong(Long)	Lee -escribe tipo long
readFloat()	writeFloat(Float)	Lee - escribe tipo float
readDouble()	writeDouble(Double)	Lee - escribe tipo double
readUTF(DataInput)	writeUTF(String)	Lee formato UTF-8 desde objeto DataInput - Escribe una cadena en formato UTF-8

Cada método devuelve un valor de retorno del mismo tipo sobre el que trabaja.

En el caso de *readUTF()* y *writeUTF()*, los datos obtenidos en el origen o escritos en el destino, son convertidos a un formato portable entre distintos lenguajes y plataformas. En este caso, se usa el estándar de caracteres UTF-8, que es un derivado de UNICODE.

Para trabajar con los flujos de datos primitivos, se debe crear un flujo asociado al origen o destino de los datos, después se asocia un objeto **DataInputStream** o **DataOutputStream** a este flujo y se lee o escribe según sea el caso. Por ejemplo estos 2 casos:

```
String cadena;
FileOutputStream f = new FileOutputStream("archivo.dat");
DataOutputStream d = new DataOutputStream(f);
d.writeUTF(cadena);

long dato;
FileInputStream f = new FileInputStream("enterosLong.dat");
DataInputStream d = new DataInputStream(f);
d.readLong(dato);
```

La extensión no es relevante, pero suele usarse ".dat", para indicar que se trata de un archivo de datos binarios. A partir de ahora, utilizaremos la clase File, de cual hablaremos a continuación, con el objeto de identificar los archivos sobre los que se trabajaran. Primero vamos a crear un archivo con datos de distinto tipo y después lo leeremos.

Como ejemplo crearemos un programa, que ingrese registros en un archivo de datos y otro que los lea y los muestre en pantalla.

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

    public static void main(String[] args) throws IOException {
        FileOutputStream f = new FileOutputStream("datos.dat",true);
        DataOutputStream d = new DataOutputStream(f);

        String cadena = "Esto esta que arde";
        int entero = 22;

        d.writeUTF(cadena);
        d.writeInt(entero);

        f.close();
    }
}
```

Este programa permite ingresar el archivo de datos en el que vamos a guardar la información. Crea el objeto **File** que nos permitirá determinar si el archivo de datos existe o no. Si no existe se crea. Si existe vamos a escribir a partir del final del mismo. Se crea el *stream* de salida, y un objeto **DataOutputStream** que nos va permitir escribir los datos en el archivo.

Para leer los datos guardados tenemos el siguiente programa :

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

  public static void main(String[] args) throws IOException {
    FileInputStream f = new FileInputStream("datos.dat");
    DataInputStream d = new DataInputStream(f);

    String cadena;
    int entero;

    try{
      do{
        cadena = d.readUTF();
        entero = d.readInt();
        System.out.println("Cadena: "+cadena+" entero:"+entero);
      }while(true);
    }
    catch(EOFException e){
      System.out.print("Fin del archivo");
    }
    f.close();
  }
}
```

En la definición del método *main*, se usa la palabra reservada *throws* para señalar o indicar que el método en cuestión puede lanzar excepciones, en este caso, de la clase **IOException**. Esto se hace para evitar tener que declarar, en cada caso que ésta se produzca, un try y un catch, que se producirían a nivel de la lectura de los datos contenidos en el archivo. En estas situaciones, la excepción no es atrapada en el mismo método, no hay un catch. Esto es simplemente un ejemplo para mostrar cómo evitar que se lancen excepciones. Por otro lado, en los métodos que eventualmente llamen a este tipo de métodos están obligados a atrapar la excepción.

Al igual que en el programa anterior, se declaran los objetos que hacen la lectura. Se crea un objeto File con el objeto de verificar si el archivo existe y controlar la marca fin de archivo (EOF). Si existe, se crean e inicializan los objetos del *stream* de entrada. Luego se leen los datos del archivo y se ponen en pantalla. Finalmente, se cierra el flujo de datos.

Indudablemente, si queremos almacenar una colección de datos relacionados entre sí, una mejor y más lógica solución es el uso de objetos en vez del uso de bytes para ir guardando y leyendo datos representados por tipos primitivos.

Flujo de datos con objetos

Esto se conoce como "*Seriación de Objetos*". La idea es almacenar objetos con datos en un archivo para hacerlos "persistentes", esto es que los datos no se pierdan cuando se deje de usar al aplicación en memoria principal. Ello porque la Memoria de Acceso Aleatorio (RAM) o Principal, es volátil y los datos desaparecen una vez que ha terminado la aplicación que los controla. Se denomina "*deseriación*" a la operación de recuperar los datos desde el archivo para usarlos en memoria.

Las clases que permiten estas operaciones son **ObjectOutputStream**, que permite escribir en el flujo de salida, y **ObjectInputStream**, que permite reconstruir los objetos desde el flujo de entrada. Ambas forman parte del paquete java.io. Actúan sobre un flujo ya definido por otro objeto. Por ejemplo :

```
//actúa sobre el archivo
FileOutputStream f = new FileOutputStream("datos.dat");

//actúa sobre los objetos
ObjectOutputStream obj = new ObjectOutputStream(f);
```

La seriación es posible, sólo cuando se implementa en la clase la interfaz Serializable. Que tiene por objeto identificar los objetos serializables y no contiene ningún método :

```
import java.io*;
public class MiClase implements Serializable {
```

Cuando un objeto tiene atributos que referencian a otros objetos, estos últimos deben escribirse también en el flujo. De esto se encarga el método *writeObject()*, que recorre las referencias recursivamente, escribiendo todos ellos. Por otro lado, el método *readObject(),* se encargará de recorrer recursivamente las referencias para recuperar los datos de los objetos referenciados.

Como ejemplo, se crea una nueva clase denominada "Articulo", con sus respectivos atributos y métodos:

```
import java.io.Serializable;
public class Articulo implements Serializable {
  private int cod;
  private String nombre;
  private int precio;
  private boolean estado;

  public Articulo(){}

  public Articulo (int cod,String nombre,int precio,boolean estado){
    this.cod = cod;
    this.nombre = nombre;
    this.precio = precio;
    this.estado = estado;
  }

  public int getCod() { return cod; }
  public boolean isEstado() { return estado; }
  public String getNombre() { return nombre; }
  public int getPrecio() { return precio; }
}
```

Se importa el paquete java.io para el uso de la interface "Serializable". Se definen dos constructores: uno vacío por defecto y otro que recibe como argumentos los atributos de la clase.

El siguiente programa, tiene por objetivo crear el archivo que va a seriar los objetos con aquellos datos que ingresemos por teclado (objeto *art*). Si se abre el archivo de datos con un editor de textos se verá que tiene un encabezado señalando, entre otras cosas, la clase de objetos que almacena. Es por ello, que la seriación se debe hacer en una sola sesión. Para agregar objetos al archivo, o modificarlos o borrarlos, se usa la clase **RandomAccessFile** del paquete java.io, que permite acceder a cualquier posición (registro) dentro del archivo. Luego en el mismo programa se realiza la lectura de los datos desde el archivo a través del objeto *art2:*

```
import java.io.*;
import java.util.Scanner;

public class Main {

  public static void main(String[] args) throws IOException {
    FileOutputStream f = new FileOutputStream("datos.dat");
    ObjectOutputStream ob = new ObjectOutputStream(f);
    Scanner ing = new Scanner(System.in);

    /*
     * Escritura del objeto hacia un archivo
     */
    // Ingreso de datos
    String cod, nom, pre, est;

    System.out.print("Ingrese codigo: ");
    cod = ing.nextLine();

    System.out.print("Ingrese nombre: ");
    nom = ing.nextLine();

    System.out.print("Ingrese precio: ");
    precio = ing.nextLine();

    Articulo art = new Articulo(Integer.parseInt(cod),
                                nom,
                                Integer.parseInt(precio),
                                false);
    ob.writeObject(art);

    f.close();

    /*
     * Lectura del objeto desde un archivo
     */
    Articulo art2 = new Articulo(); // Leer

    FileInputStream f2 = new FileInputStream("datos.dat");
    ObjectInputStream ob2 = new ObjectInputStream(f2);

    try{
      art2 = (Articulo)ob2.readObject();
    }
    catch(ClassNotFoundException e){
      System.err.println("Clase no encontrada");
    }

    System.out.println("El codigo es: "+art2.getCod());
    System.out.println("El nombre es: "+art2.getNombre());
    System.out.println("El precio es: "+art2.getPrecio());
```

```
    if(art2.isEstado())
      System.out.println("Articulo Activo");
    else
      System.out.println("Articulo Desactivado");

    f2.close();
  }
}
```