JAVA: SECUENCIA DE APRENDIZAJE nº 15 (Ficheros)

RESULTADOS Y COMPETENCIAS PERSEGUIDOS

En consonancia a los resultados de aprendizaje y competencias que el alumno debe completar cursando este módulo, la secuencia de aprendizaje contribuye a conseguir los siguientes:

R5) Realiza operaciones de entrada y salida de información, utilizando procedimientos específicos del lenguaje y librerías de clases. (Criterios a, c, d y e)

C2) El trabajo regular y la constancia.
C3) El aprendizaje con los demás.
C4) El aprendizaje por sí mismo.
C5) La autonomía responsable.
C7) El dominio de las herramientas de desarrollo de aplicaciones.

En esta secuencia de aprendizaje, vamos a adentrarnos en el territorio de la persistencia, porque el de la entrada/salida de datos tuvo un recorrido inicial cuando aprendimos a pedir por teclado datos y los mostramos por pantalla. Y por conocimientos anteriores teníamos claro que llega un momento en que la información que necesita/genera un programa en Java (lo mismo que uno en C) está en/requiere almacenamiento permanente. De bueno tiene que mucho de lo aprendido en lenguaje C nos vale, pero hay que hacer unas pocas matizaciones para entenderlas del todo ahora en Java.

Flujos de comunicación

Dado que en todo proceso de comunicación entre un usuario y un programa se realiza mediante flujos de información, el desarrollo de programas en Java no se iba a quedar al margen. Desde pedir un dato por teclado hasta mostrarlo por pantalla pasando por almacenarlo en un disco duro, el abanico de posibilidades que se abre ante el desarrollador no es moco de pavo.

Para ello, muchos lenguajes entre ellos Java, han diseñado mecanismos que permiten ese trasiego de los datos de un lado para otro. El principal se denomina **flujo** (***Stream***, en inglés) y no deja ser una secuencia ordenada de datos que se transmite desde una fuente hasta una destino, pudiendo ser estos orígenes y destinos desde ficheros y cadenas hasta un dispositivo, bien sea teclado, pantalla, altavoz o red.

Para gestionar adecuadamente un fichero desde un programa, es importante tener en cuenta el tipo de contenido que almacenará. Como ya se dijo al principio, Java ha diseñado flujos de caracteres y flujos de bytes para leer/grabar datos de/en los ficheros:

Dirección de flujo de datos	Acceso de los datos	Tipo de información
Entrada *Salida* *Entrada/Salida*	*Secuencial* *Directo*	*De caracteres (texto)* *De bytes (binarios)*

Por último es bueno saber cómo se utilizan estos flujos. Si dejamos de fijarnos un momento en el tipo de dato que manejan y los miramos en abstracto, observan el patrón genérico siguiente:

1) Creación del flujo a una fuente de datos
2) Lectura (flujo en una dirección) o escritura (en la otra) de los datos disponibles
3) Cierre del flujo

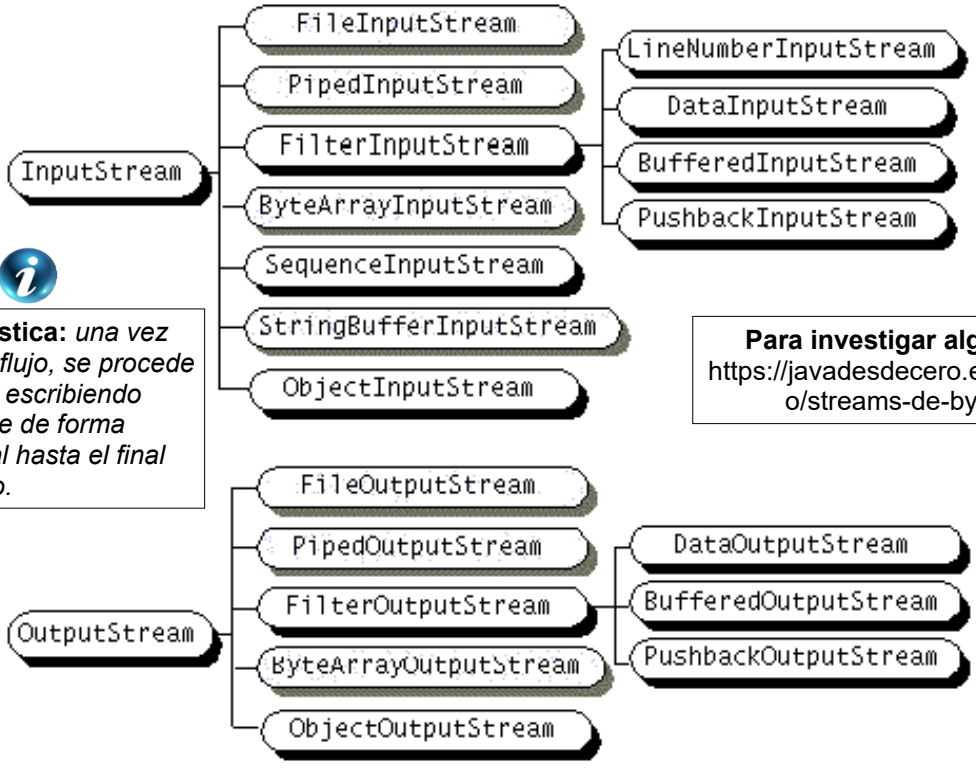
¿Te acuerdas del *stdin* en C? Sí, ese parásito del fflush(). Pues *stdin* en Java tiene su homólogo, aunque tiene unos primos para jugar cuando se aburre. Son los denominados flujos predeterminados y son:

System.in: entrada estándar (por defecto asignada al teclado, mirar clase Scanner)
System.out: salida estándar (por defecto asignada a pantalla, ¿te suena con println()?)
System.err: error estándar (por defecto asignado a pantalla, por donde se muestran los errores)

Deducimos entonces que las secuencias de comportamiento de los flujos son prácticamente idénticas, se vincule al dispositivo físico real al que se vincule. De ahí que las mismas clases y métodos de E/S que Java aplica un tipo de dispositivo como una consola, puede aplicarlo igualmente a un fichero de disco duro para el movimiento de los datos.

Clases que manejan flujos

Pero Java ya se sabe..., con eso de buscar el equilibrio en todo momento entre la reusabilidad de las clases y la precisión en la funcionalidad, ha construido una jerarquía de clases dentro del paquete **java.io** partiendo de las más abstractas para ir perfilándolas según las necesidades del desarrollador.



Característica: *una vez abierto el flujo, se procede leyendo o escribiendo byte a byte de forma secuencial hasta el final del fichero.*

Para investigar algo más:
https://javadesdecero.es/avanzado/streams-de-bytes/.

La primera remesa viene de la mano de las clases abstractas **InputStream** y **OutputStream** que están trufadas de métodos para manejar ***flujos de bytes*** (*recuerda, bytes son 8 bits*) y que las siguientes clases derivadas terminan de concretar:

Métodos de InputStream	int read()	Lee el siguiente byte desde el InputStream. Si lee -1 indica que no hay más bytes
	int available()	Devuelve el nº de bytes que se pueden leer del InputStream
	void close()	Cierra el InputStream
Métodos de OutputStream	void write(int b)	Escribe un byte en el OutputStream
	void write(byte[] a)	Escribe todos los bytes del array a en el OutputStream
	void flush()	Vacía el flujo de salida actual del OutputStream
	void close()	Cierra el OutputStream y escribe los contenidos que haya en ese momento

Este vídeo hasta el minuto 5:55 te cuenta algo como lo que viste en C pero adaptado a Java: **https://www.youtube.com/watch?v=etQN4EfYN7k**

A partir de ahí, y para saber elegir de entre las clases que pueden interesar en cada circunstancia, se muestran a continuación las funcionalidades de las utilizadas habitualmente:

1.) FileOutputStream: permite escribir bytes en un fichero binario
2.) BufferedOutputStream: acopla un buffer interno a un OutputStream para escribir bytes mejorando rendimiento
3.) DataOutputStream: define métodos como writeBoolean, writeByte, writeInt, etc..., que permiten escribir datos de tipo primitivo desde un OutputStream
4.) ByteArrayOutputStream: posibilita utilizar un array de bytes como un OutputStream utilizando un buffer interno que mejora el rendimiento
5.) FileInputStream: permite leer bytes de un fichero binario
6.) BufferedInputStream: acopla un buffer interno a un InputStream para leer datos mejorando el rendimiento
7.) DataInputStream: define métodos como readBoolean, readByte, readInt, etc..., que permiten leer datos de tipo primitivo desde un InputStream
8.) ByteArrayInputStream: posibilita utilizar un array de bytes como un InputStream utilizando un buffer interno que mejora el rendimiento

Ejemplos de manejo de ficheros binarios

Programa que lee nºs enteros por teclado y los escribe en el fichero **datos.dat**. La lectura de datos acaba cuando se introduce un "-1".

```
import java.io.DataOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Scanner;
public class EscrituraFicheroBinario {
   public static void main(String[] args) {
      Scanner pt = new Scanner(System.in);
      FileOutputStream fos = null;
         System.out.print("Introduce nº entero. (-1 = FIN");
         num = pt.nextInt();
      }
   } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe.getMessage());
   } catch (IOException ioe) {
      System.out.println(ioe.getMessage());
   } finally {
      try {
```

```
        DataOutputStream salida = null;
        int num;
        try {
            //Crea el fichero si no existe y lo abre para escritura
            fos = new FileOutputStream("src/datos.dat");
            salida = new DataOutputStream(fos);
            //Lee de teclado
            System.out.print("Introduce nº entero. (-1 = FIN");
            num = pt.nextInt();
            while (num != -1) {
                //Graba en el fichero el dato
                salida.writeInt(num);
                if (fos != null) {
                    fos.close();
                }
                if (salida != null) {
                    salida.close();
                }
            } catch (IOException ioe) {
                System.out.println(ioe.getMessage());
            }
        }
    }
}
```

Programa que lee nºs enteros por teclado y los escribe en el fichero **datos.dat** ya existente al final de éste. La lectura de datos acaba cuando se introduce un "-1".

```
import java.io.DataOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Scanner;
public class EscrituraFicheroBinario {
    public static void main(String[] args) {
        Scanner pt = new Scanner(System.in);
        FileOutputStream fos = null;
        DataOutputStream salida = null;
        int num;
        try {
            //Abre el fichero existente para escribir por el final
            fos = new FileOutputStream("src/datos.dat", true);
            salida = new DataOutputStream(fos);
            //Lee de teclado
            System.out.print("Introduce nº entero. (-1 = FIN");
            num = pt.nextInt();
            while (num != -1) {
                //Graba en el fichero el dato tecleado
                salida.writeInt(num);
                System.out.print("Introduce nº entero. (-1 = FIN");
                num = pt.nextInt();
            }
        } catch (FileNotFoundException fnfe) {
            System.out.println(fnfe.getMessage());
        } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        } finally {
            try {
                if (fos != null) {
                    fos.close();
                }
                if (salida != null) {
                    salida.close();
                }
            } catch (IOException ioe) {
                System.out.println(ioe.getMessage());
            }
        }
    }
}
```

*Utilizando un segundo constructor de **FileOutputStream()**, que tiene la posibilidad de pasarle un **true** como parámetro **append**, se consigue añadir datos al fichero.*

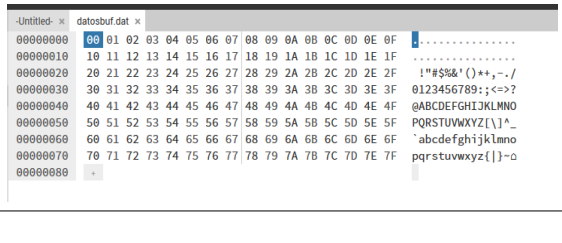
Programa que lee el contenido del fichero creado en el código anterior. Utilizaremos un bucle infinito para leer los datos. Cuando se llega al final del fichero se lanza la EOFException que se utiliza para salir del bucle while.

```
import java.io.DataInputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class LecturaFicheroBinario {
    public static void main(String[] args) {
        } catch (FileNotFoundException fnfe) {
            System.out.println(fnfe.getMessage());
        } catch (EOFException eofe) {
            System.out.println("Fin de fichero");
        } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        } finally {
            try {
```

```
        FileInputStream fis = null;
        DataInputStream entrada = null;
        int num;
        try {
          fis = new FileInputStream("src/datos.dat");
            entrada = new DataInputStream(fis);
            while (true) {
               //Lectura de un número entero del fichero
               num = entrada.readInt();
               //Se muestra lo leído en pantalla
               System.out.println(num);
            }
            if (fis != null) {
               fis.close();
            }
            if (entrada != null) {
               entrada.close();
            }
         } catch (IOException ioe) {
            System.out.println(e.getMessage());
         }
      }
   }
}
```

Programa que coge los 128 enteros (del 0 a 127) generados por un bucle for y los escribe en el fichero binario **datosbuf.dat**. En este caso se emplea el flujo BufferedOutputStream.

```
import java.io.FileOutputStream;
import java.io.BufferedOutputStream;
public class EscrituraFicheroBinario {
   public static void main(String[] args) throws java.io.IOException {
      FileOutputStream fos = new FileOutputStream("src/datosbuf.dat");
      BufferedOutputStream salida = new BufferedOutputStream(fos);
      for (int i = 0; i < 128; i++) {
         salida.write(i);
      }
      salida.flush();
      fos.close();
   }
}
```



Como sabes ya, el contenido de un fichero binario no es visible mediante un bloc de notas o similar. Pero no todo está perdido; puedes visualizarlo utilizando un editor hexadecimal como el que puedes encontrar en [https://hexed.it/].

Programa que lee los nºs enteros almacenados antes en el fichero binario **datosbuf.dat**, va pasando cada lectura "bufferizada" a una tabla de 50 bytes y muestra por pantalla tanto el número de bytes leídos cada vez como el contenido llegado a esas posiciones de la tabla

```
import java.io.FileInputStream;
import java.io.BufferedInputStream;
public class LecturaFicheroBinario {
   public static void main(String[] args) throws java.io.IOException {
      byte bufferEntrada[] = new byte[50];
      int numBytesLeidos;
      FileInputStream fis = new FileInputStream("src/datosbuf.dat");
      BufferedInputStream entrada = new BufferedInputStream(fis);
      while (true) {
         numBytesLeidos = entrada.read(bufferEntrada);
         System.out.println("Leidos " + numBytesLeidos + " bytes");
         if (numBytesLeidos <= 0) break;
         for (int i = 0; i < 50; i++) {
            System.out.println("En "+i+" hay " + bufferEntrada[i]);
            bufferEntrada[i] = 0;
         }
```

```
    }
    fis.close();
  }
}
```

Si quieres aprender algo más con este código puedes visitar una versión mejorada del mismo en el aula virtual: **FileTestIn.java**

Hasta ahora se han vistos códigos que realizan lecturas y escrituras secuenciales sobre ficheros binarios. Pero en este tipo de ficheros binarios también es posible implementar lecturas y escrituras accediendo directamente a éstos. Java ha dispuesto la clase **RandomAccessFile** para realizar accesos a un fichero de forma directa (también conocida como aleatoria). Dicha clase expone dos constructores:

RandomAccessFile(String rutafich, String modo);	**RandomAccessFile(File objetoFile, String modo);**
*Parámetro **modo** indica el modo de acceso al abrirlo* [**"r"** sólo lectura] o [**"rw"** lectura y escritura]	
Ejemplo: **RandomAccessFile** fichero = new **RandomAccessFile(**"src/alumnos.dat", **"r");**	
Ejemplo: File fich = new File ("src/alumnos.dat"); **RandomAccessFile** fichero = new **RandomAccessFile(**fich, **"rw");**	

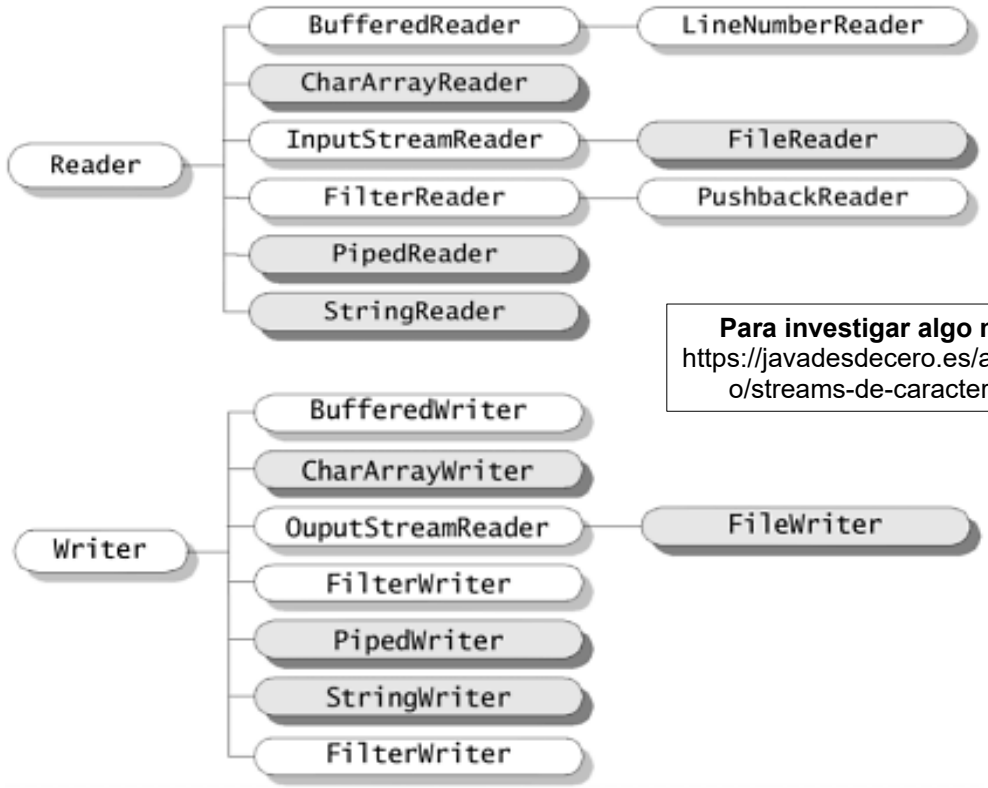
A continuación, un ejemplo de cómo abrir un fichero binario existente y añadirle datos al final del mismo:

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

public class LecturaDirectaBinario {
   static Scanner sc = new Scanner(System.in);
   static RandomAccessFile fichero = null;

   public static void main(String[] args) {
      int num;
      try {
         //Abre el fichero en modo lectura-escritura
         fichero = new RandomAccessFile("src/datosbuf.dat", "rw");
         //Muestra el contenido del fichero antes de añadir
         mostrarFichero();
         System.out.print("Teclea un número entero que se añadirá al final del fichero: ");
         num = sc.nextInt();
         //Sitúa el puntero al final del fichero
         fichero.seek(fichero.length());
         //Se graba el dato en el fichero
         fichero.writeInt(num);
         //Muestra el contenido del fichero después de añadir
         mostrarFichero();

        } catch (FileNotFoundException fnfe) {  //Por si no encuentra el fichero
         System.out.println(fnfe.getMessage());
        } catch (IOException ioe) {   //Por si alguien intenta escribir en un fichero de sólo lectura
         System.out.println(ioe.getMessage());
      } finally {
         try {
            if (fichero != null) {
               fichero.close();
```

```
            }
        } catch (IOException eio) {
            System.out.println(eio.getMessage());
        }
    }
  }

  public static void mostrarFichero() {
    int num;
    try {
      //Sitúa el puntero al principio del fichero
      fichero.seek(0);
      while (true) {
        //Lectura de un entero del fichero
        num = fichero.readInt();
        System.out.println(num);
      }
     } catch (EOFException eofe) {  //Por si llega al final del fichero
      System.out.println("Fin de fichero");
    } catch (IOException ioe) {
      System.out.println(ioe.getMessage());
    }
  }
}
```

Estos vídeos complementan cómo Java trata los ficheros que manejan datos a nivel de bytes: **https://www.youtube.com/watch?v=38YBRnJtQEw** **https://www.youtube.com/watch?v=v6ctWhhTFrk**

La segunda remesa viene de la mano de las clases abstractas **Reader** y **Writer** que están trufadas de métodos para manejar ***flujos de caracteres*** (*recuerda, char son 16 bits*) y que las siguientes clases derivadas terminan de concretar:



Para investigar algo más: https://javadesdecero.es/avanzado/streams-de-caracteres/.

Métodos de Reader	int read()	Lee el siguiente byte del flujo y lo almacena en formato entero. Si lee -1 indica que no hay más caracteres.
	int read(byte[]buffer)	Lee bytes del flujo de entrada y los almacena en el buffer. Lee hasta llenar el buffer.
	int read(byte[]buffer, intposInicio, intdespl)	Lee bytes del flujo de entrada y los almacena en el buffer. La lectura la almacena en el array pero a partir de la posición indicada, el número máximo de bytes leídos es el tercer parámetro.
	void mark(int bytes)	Marca la posición actual en el flujo de entrada. Cuando se lea el número de bytes indicado, la marca se elimina.
	boolean markSupported()	Devuelve true si en el flujo de entrada es posible marcar mediante el método mark.
	boolean ready()	Devuelve verdadero si el flujo de entrada está listo.
	void reset()	Coloca el puntero de lectura en la posición marcada con mark.
	long skip()	Se salta de la lectura el número de bytes indicados.
	void close()	Cierra el flujo de entrada. Cualquier acceso posterior generaría una IOException.
Métodos de Writer	void write(int byte)	Escribe un byte en el flujo de salida.
	void write(byte[] a)	Escribe todos los bytes del array de bytes en el flujo de salida.
	void write(byte[] buffer, int posInicial, int numBytes)	Escribe el array de bytes en el flujo de salida, pero empezando por la posición inicial y sólo la cantidad indicada por numBytes.
	void write(String texto)	Escribe los caracteres en el String en el flujo de salida.
	void write(Stringbuffer, int posInicial, int numBytes)	Escribe el String en el flujo de salida, pero empezando por la posición inicial y sólo la cantidad indicada por numBytes.
	void flush()	Vacía el buffer del flujo de salida.
	void close()	Cierra el flujo de salida. Cualquier acceso posterior generaría una IOException.

Característica: *una vez abierto el flujo, se procede leyendo o escribiendo caracter a caracter de forma secuencial hasta el final del fichero.*

Ficheros de texto

Dado que ahora se va a presentar el manejo básico de un **fichero de texto**, lo lógico es utilizar **flujos de caracteres**. Por tanto, tiraremos de las clases **FileReader**, **FileWriter**, **PrintWriter, BufferedReader**, etc..., que heredan todas de las abstractas **Reader** y **Writer**. Y, ojo, que a esta fiesta es fácil que se apunte la clase **Scanner** que por dentro está compuesta de algunas de las mencionadas. Por último, es bueno reseñar que cuando el contenido de un fichero es textual, el único acceso posible es el secuencial, lo que quiere decir es que para llegar a un caracter concreto del fichero hay que pasar por narices por los caracteres anteriores.

A continuación, se presenta una serie de códigos que permiten "jugar" con ellos con el objetivo de entrenar la habilidad del manejo de ficheros en diferentes circunstancias:

Programa Java que lee texto por teclado y lo escribe en un fichero de texto llamado **mitexto.txt**. Se trata de leer una línea de texto por teclado y escribirla en el fichero, pudiéndose repetir hasta que se introduce por teclado la cadena FIN que indica el final de lectura (FIN no va al fichero). **Si el fichero tuviese información previa, la machaca.**

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class EscrituraFicheroTexto {
   public static void main(String[] args) {
      //Se crea el flujo de entrada para recibir del teclado
      Scanner pt = new Scanner(System.in);
      //Se crea el flujo de salida para grabar en un fichero
      PrintWriter salida = null;
      try {
         //Se crea el fichero y salida es su referencia
         salida = new PrintWriter("src/mitexto.txt");
         String cadena;
           System.out.println("Introduce un texto, pulsa Enter
al acabar e introduce entonces la cadena FIN:");
         //Se introduce por teclado una cadena de texto
(pulsando Enter al final)
         cadena = pt.nextLine();
         //Comprueba si la cadena introducida contiene texto
normal o sólo la cadena FIN
         while (!cadena.equalsIgnoreCase("FIN")) {
            //Se graba la cadena en el fichero
            salida.println(cadena);
            //Se introduce por teclado una nueva cadena de texto
en nueva línea (pulsando también Enter al final)
            cadena = pt.nextLine();
         }
         //Momento barrendero LIPASAM
         salida.flush();
      } catch (FileNotFoundException fnfe) {
         //Si no encuentra referencia del fichero que avise
         System.out.println(fnfe.getMessage());
      } finally {
         //Pase lo que pase, me cierras el flujo (conexión) con el
fichero liberando recursos
         salida.close();
      }
   }
}
```

Nota: *aunque la clase **Scanner** también puede utilizarse para la lectura de ficheros en Java, en realidad, aquí la clase **Scanner** se utiliza como hasta ahora; para pedir datos por teclado.*

Programa Java que lee texto por teclado y lo escribe en un fichero de texto llamado **mitexto.txt**. Se trata de leer una línea de texto por teclado y escribirla en el fichero, pudiéndose repetir hasta que se introduce por teclado la cadena FIN que indica el final de lectura (FIN no va al fichero). **Ahora se respetará el contenido anterior del fichero añadiéndose debajo.**

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class EscrituraFicheroTexto {
   public static void main(String[] args) {
      Scanner pt = new Scanner(System.in);
      FileWriter fw = null;
      PrintWriter salida = null;
      String cadena;
      try {
         fw = new FileWriter("src/mitexto.txt", true);
         salida = new PrintWriter(fw);
           System.out.println("Introduce un texto, pulsa Enter
al acabar e introduce entonces la cadena FIN:");
         cadena = pt.nextLine();
         while (!cadena.equalsIgnoreCase("FIN")) {
            salida.println(cadena);
            cadena = pt.nextLine();
         }
      } catch (IOException ioe) {
         System.out.println(ioe.getMessage());
      }
      salida.close();
   }
}
```

Programa Java que lee de un fichero de texto llamado **mitexto.txt**. Se trata de leer una línea del fichero y mostrarla por pantalla, hasta que se llegue al final del fichero (no hay más líneas) y readLine() devuelve null.

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

public class LeeFicheroTexto {
   public static void main(String[] args) {
      FileReader fr = null;
      try {
         fr = new FileReader("src/mitexto.txt");
         BufferedReader entrada = new BufferedReader(fr);
         //Se lee la primera línea del fichero
         String cadena = entrada.readLine();
         //Mientras no se llegue al final del fichero
         while (cadena != null) {
            //Se muestra por la pantalla
            System.out.println(cadena);
            //Se leen las siguientes líneas que haya en el fichero
            cadena = entrada.readLine();
         }
      } catch (FileNotFoundException fnfe) {
         System.out.println(fnfe.getMessage());
      } catch (IOException ioe) {
         System.out.println(ioe.getMessage());
      } finally {
         try {
            if (fr != null) {
               fr.close();
            }
         } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
         }
      }
   }
}
```

Programa Java que lee de un fichero de texto llamado **mitexto.txt**. Se trata de leer una línea del fichero y mostrarla por pantalla, pero ahora **utilizando la clase Scanner para coger datos del fichero**.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class LeeFicheroTexto {
   public static void main(String[] args) {
      File f = new File("src/mitexto.txt");
      String cadena;
      Scanner entrada = null;
      try {
         //se crea un Scanner asociado al fichero
         entrada = new Scanner(f);
         //mientras no se alcance el final del fichero
         while (entrada.hasNext())  {
            //se lee una línea del fichero
            cadena = entrada.nextLine();
            //se muestra por pantalla
            System.out.println(cadena);
         }
      } catch (FileNotFoundException fnfe) {
         System.out.println(fnfe.getMessage());
      } finally{
         entrada.close();
      }
   }
}
```

A continuación un ejemplo de cómo abrir un fichero de texto existente y modificarle información al mismo:

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

public class LecturaDirectaTexto {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      RandomAccessFile fichero = null;
      String palabra, cadena;
      StringBuilder auxBuilder;
```

```
        long pos = 0;
        int indice;
        try {
           //Abre el fichero para lectura-escritura
           fichero = new RandomAccessFile("src/texto.txt", "rw");
           //Se pide la palabra a buscar
           System.out.print("Teclea palabra: ");
           palabra = sc.nextLine();
           //Lectura del fichero línea a línea
           cadena = fichero.readLine();
           while(cadena!=null) {
              //Busca la palabra en la línea leída
              indice = cadena.indexOf(palabra);
              //mientras la línea leída contenga esa palabra (controla si está repetida)
              while(indice!=-1){
                 //Pasa la línea leída a un StringBuilder para poder modificar contenido
                 auxBuilder = new StringBuilder(cadena);
                 auxBuilder.replace(indice, indice+palabra.length(), palabra.toUpperCase());
                 cadena = auxBuilder.toString();
                 //Posiciona el puntero al principio de la línea actual (según valor de pos) y la sobrescribe
                 fichero.seek(pos);
                 fichero.writeBytes(cadena);
                 //Comprueba si hay repitición de la misma palabra en la propia línea
                 indice = cadena.indexOf(palabra);
              }
              //Almacena posición de la línea actual que va a leer
              pos = fichero.getFilePointer();
              cadena = fichero.readLine();
           }
        }
        catch (FileNotFoundException fnfe) {
           System.out.println(fnfe.getMessage());
        }
        catch (IOException ioe) {
           System.out.println(ioe.getMessage());
        }finally {
           try {
              if (fichero != null) {
                 fichero.close();
              }
           }
           catch (IOException eio) {
              System.out.println(eio.getMessage());
           }
        }
     }
  }
```

Este vídeo a partir del minuto 6:00 te cuenta algo sobre el manejo de ficheros de texto en Java: **https://www.youtube.com/watch?v=etQN4EfYN7k**

¿Por qué hay flujos que se apellidan **Buffered**? Pues imaginemos un supermercado sin carritos. Cada dos o tres artículos que cogiera con las manos, tendría que llevarlos a la caja y así dando multitud de viajes de ida y vuelta. Pues hay que pensar en un buffer como en un carrito de la compra. En ellos se dejan temporalmente los datos hasta que lo llenamos haciendo menos viajes cuando usas el carrito.

Cualquiera de estos flujos, **FileInputStream**, **FileOuputStream**, **FileReader** o **FileWriter** que se usen a pelo en un código Java, cada vez que realizan una operación de lectura/escritura,

trabajarán directamente contra el disco duro. De forma que nos podemos encontrar leyendo/escribiendo sólo unos pocos caracteres cada vez, haciendo de este proceso un recorrido penitencial, con muchos accesos al disco duro.

Para aliviar esta circunstancia los ingenieros de Sun Microsystems implementaron un mecanismo, ya conocido en el mundillo informático, consistente en añadir un **buffer intermedio** de forma que mientras se escriben datos (caracteres o bytes), éstos se van guardando en dicho buffer hasta que tenga datos suficientes como para hacer la escritura eficiente. Pero al igual que si lo que nos viene de golpe es una lectura brutal de datos; el buffer aguanta la avalancha y los acaba entregando en cómodos plazos. Es una de las maneras de conseguir también leer/escribir línea por línea de/en un fichero como puedes imaginar.

Como siempre, Java presenta sus mecanismos en forma de clase y las que se emplean para estos menesteres son: **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream**.

Con esto se consiguen accesos a disco más eficientes, de ahí que el programa se ejecute más rápido. La diferencia se hace más ostensible cuanto mayor tamaño presenta el fichero del que se quiere leer o escribir.

Este vídeo complementa cómo Java trata los buffers para el manejo de ficheros:
https://www.youtube.com/watch?v=YCCE4sbmWrw

Operaciones con ficheros

De todos es sabido que los ficheros son susceptibles de ser manipulados a nivel de sistema. Aunque hay que distinguir cómo Java, según sea la versión 5, 6 o 7, trata este tipo de situaciones de maenjo de ficheros. Y alguien se estará preguntando, ¿por qué no ceñirnos únicamente a lo más nuevo? Porque como Java fue un lenguaje que vino hace años para quedarse, y efectivamente se ha quedado, un desarrollador puede encontrarse con que tenga que enfrentarse con códigos hechos en versiones como la 5, la 6, la 7 o la 8. Y es bueno saber de dónde venimos.

Por eso, pero sin cebarnos con todas las versiones posibles porque sino escribiríamos una biblia, se procede a presentar cómo Java lidia con operaciones como crear, borrar, renombrar, mover y copiar ficheros, así como crear o borrar directorios de dos maneras (cuando sea relevante). Dado que las de versiones más antiguas son directamente asumidas por las versiones más recientes, no hay problemas de compatibilidad. Otra historia sería, pero no es nuestro caso, que quisiéramos con compiladores y *runtimes* antiguos querer ejecutar códigos hechos con las últimas versiones de Java.

Ahí van estos ejemplos para poder entrenarnos un poco:

Borrar un fichero: en el ejemplo siguiente se aplican las dos maneras de borrar un fichero desde un código Java: la primera lo borra con efecto inmediato (**delete()**), mientras que el segundo deja a la JVM agendarlo para cuando pueda (**deleteOnExit()**).

```java
import java.io.File;
public class BorraFich {
   public static void main(String[] args) {
      File miFich = new File("src/cualquier.txt");
       //Una forma de comprobar que el fichero existe
      System.out.println("miFich.exists()->"+miFich.exists());
      boolean borrado = miFich.delete();
      if (borrado) {
         System.out.println("Eliminado: "+miFich.getName());
      }
      else {
         System.out.println("Problema al borrar el fichero.");
      }
   }
}
```

```java
import java.io.File;
public class BorraFich {
   public static void main(String[] args) {
      File miFich = new File("src/cualquier.txt");
       //Mostramos la ruta del fichero a borrar
      System.out.println("Ruta -> "+miFich.getPath());
       //Ordenamos el borrado a la JVM
      miFich.deleteOnExit();
      if (miFich.exists()) {
         System.out.println("El fichero aún existe");
      }
      else {
         System.out.println("Mira que pronto lo borró");
      }
   }
}
```

Crear/Borrar un directorio: en el ejemplo siguiente se muestra cómo borrar un directorio desde un código Java. A diferencia del anterior se ha hecho un dos en uno (prescindiendo de la variable borrado), creando/borrando y comprobando si todo ha ido bien.

```java
import java.io.File;
public class CreaDir {
   public static void main(String[] args) {
      File miDir = new File("src/creadodir");
      if (miDir.mkdir()) {
         System.out.println("Creado: "+miDir.getName());
      }
      else {
         System.out.println("Problema al crear directorio.");
      }
   }
}
```

```java
import java.io.File;
public class BorraDir {
   public static void main(String[] args) {
      File miDir = new File("src/creadodir");
      if (miDir.delete()) {
         System.out.println("Eliminado: "+miDir.getName());
      }
      else {
         System.out.println("Problema al borrar directorio.");
      }
   }
}
```

Renombrar/Mover un fichero: en el ejemplo básico siguiente tiramos de un método llamado ***renameTo()*** de la ya conocida clase **java.io.File**, que es como tradicionalmente se ha hecho en Java y sigue funcionando.

```java
import java.io.File;
public class RenombrarFich {
   public static void main(String[] args) throws java.io.IOException {
      File miFichAnt = new File("src/datos.txt");
      File miFichDesp = new File("src/datos.csv");
       //Mostramos la ruta del fichero a renombrar
      System.out.println("Ruta -> "+miFichAnt.getPath());
      boolean conforme = miFichAnt.renameTo(miFichDesp);
      if (!conforme) {
         System.out.println("Problema al cambiar el nombre del fichero");
       }
   }
}
```

El código es igual sólo que si a:

```java
File miFichDesp = new File("datos.csv");
```

se le modifica la ruta donde se moverá el fichero, va y lo mueve:

```java
File miFichDesp = new File("src/datos.txt");
```

Pero atentos, porque Java también permite mover ficheros de una forma alternativa. Para ello se implementó la clase **java.nio.file.Files**. Con esto se buscaba que Java pudiera trabajar a más bajo nivel y de forma más eficiente. Sí, sí; java.**nio** (new io) está dotada de una serie **de** métodos estáticos para las operaciones de manejo de ficheros, entre los cuales está el método **move()**, que mueve un fichero de un directorio a otro. En este ejemplo, es digno de recalcar que el método **move()** recibe un argumento **CopyOptions**, con el que podemos especificar que sobrescribe el fichero de destino si ya existía. Aunque, hay que destacar que con la versión 7 de Java se implementó años después un nuevo **nio** (conocido oficiosamente como NIO2) que introdujo la utilización de la clase **Path**.

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
public class MoverFich {
   public static void main(String[] args) throws java.io.IOException {
      Path miFichAnt = FileSystems.getDefault().getPath("src/carpeta1/ejemplo1.txt");
      Path miFichDesp  = FileSystems.getDefault().getPath("src/carpeta2/ejemplo1.txt");
      try {
         Files.move(miFichAnt, miFichDesp, StandardCopyOption.REPLACE_EXISTING);
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Copiar un fichero: en el ejemplo siguiente vamos a utilizar las clases tradionales de manejo de ficheros que se incluyen en java.io como hasta ahora. Se propone en este caso un cambio estético, método llamado desde ***main()*** y con los ficheros

```
import java.io.*;
public class CopiarFich {
   public void copiarFichero(String fichOrig, String fichDest) {
      System.out.println("===============");
      System.out.println("Desde: " + fichOrig);
      System.out.println("Hacia: " + fichDest);
      System.out.println("===============");
      try {
         File fichEnt = new File(fichOrig);
         File fichSal = new File(fichDest);
         FileInputStream entrada = new FileInputStream(fichEnt);
         FileOutputStream salida = new FileOutputStream(fichSal);
         int c;
         while ((c = entrada.read()) != -1)
            salida.write(c);
         entrada.close();
         salida.close();
      } catch(IOException ioe) {
            System.err.println("¡Hubo un error de E/S!");
      }
   }
   public static void main(String[] args) {
         if(args.length == 2)
```

```
            new CopiarFich.copiarFichero("src/"+args[0], "src/"+args[1]);
         else
            System.out.println("Error: Debe ingresar dos parámetros");
   }
}
```

Pero también debemos saber que se puede implementar la versión de copiar ficheros con el paquete **java.nio**, que pasea el método ***getChannel()*** perteneciente la clase **FileChannel**.

```
import java.io.*;
import java.nio.channels.FileChannel;
public class CopiarFich {
   public static void main(String[] args) throws java.io.IOException {
      File miFichOrig = new File ("src/datos.txt");
      File miFichDest = new File ("src/pruebas/datos.txt");
      if(!destino.exists()) {
         destino.createNewFile();
      }
      FileChannel entrada = null; FileChannel salida = null;
      try {
         entrada = new FileInputStream(miFichOrig).getChannel();
         salida = new FileOutputStream(miFichDest).getChannel();
         salida.transferFrom(entrada,0, entrada.size());
      }
      finally {
         if(entrada != null) {
            entrada.close();
         }
         if(salida != null) {
            salida.close();
         }
      }
   }
}
```

En este ejemplo, es conveniente resaltar que el método **copy()** recibe un argumento **CopyOptions**, con el que podemos provocar que sobrescriba el fichero de destino si ya existe.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class CopiarFich {
   public void copiarArchivo(String origenArchivo, String destinoArchivo) {
      try {
         Path origenPath = Paths.get(origenArchivo);
         Path destinoPath = Paths.get(destinoArchivo);
         Files.copy(origenPath, destinoPath, StandardCopyOption.REPLACE_EXISTING);
         } catch (FileNotFoundException ex) {
            System.err.println("¡No se encontró el fichero!");
         } catch (IOException ex) {
            System.err.println("¡Hubo un error de E/S!");
         }
   }
```

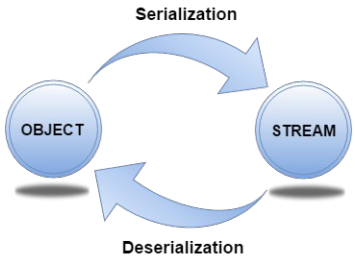
```
    public static void main(String[] args) {
        if(args.length == 2)
            new CopiarFich.copiarArchivo("src/"+args[0], "src/"+args[1]);
        else
            System.out.println("Error: Debe ingresar dos parámetros");
    }
}
```

Por último, es bueno conocer que la clase **File** tiene más métodos, sobre todo cuando pueden aportar información interesante acerca de los ficheros:

```
import java.io.File;
public class InfoFich {
    public static void main(String[] args) {
        File miFich = new File("src/datos.txt");
        if (miFich.exists()) {
            System.out.println("Nombre del Fichero: " + miFich.getName());
            System.out.println("Ruta Absoluta: " + miFich.getAbsolutePath());
            System.out.println("Habilitado para escritura: " + miFich.canWrite());
            System.out.println("Habilitado para lectura: " + miFich.canRead());
            System.out.println("El tamaño del fichero en bytes " + miFich.length());
        } else {
            System.out.println("El fichero no existe.");
        }
    }
}
```

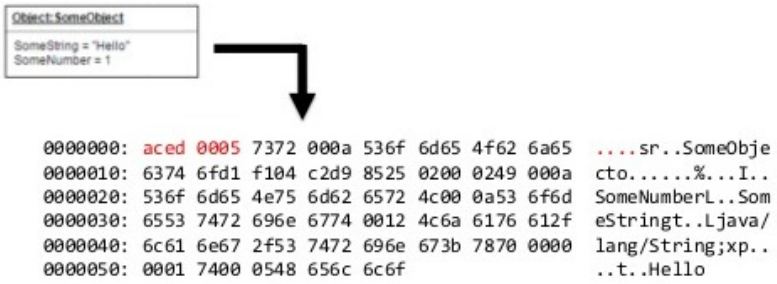
Serialización

La **serialización** es el proceso por el cual conseguimos transformar un objeto en una secuencia (serie) de bytes de forma que pueden ser posteriormente leídos para reconstruir el objeto original. El objeto serializado pueda guardarse en un fichero o puede enviarse por red para reconstruirlo en otro lugar. Por si vale el ejemplo, viene a ser lo que se hace con el Nescafé: se elabora café, se provoca una evaporación del agua hasta que sólo queda el polvillo, se guarda en un bote de cristal y cuando se quiere tomar café no hay que echarle agua de nuevo y listo.



Eso sí, no todos los objetos son susceptibles de ser serializados. Un objeto es serializable si su clase implementa la interfaz vacía **Serializable** (sin ningún método) que se encuentra en el paquete **java.io**. Para que lo entiendas fácil, la interfaz **Serializable** simplemente es la marca que deja Java

para indicar aquellas clases cuyos objetos se podrán serializar. Todos los tipos primitivos Java son serializables, así como las tablas (arrays) y los String. Si un objeto que pretende ser serializable contiene atributos que son referencias a otros objetos, éstos a su vez deben ser serializables.



Las clases **ObjectOutputStream** y **ObjectInputStream** se encargan de realizar los procesos en Java de serialización (pasar el café a polvo) y deserialización (agua al polvo reconstituir el café) respectivamente. Son las encargadas de escribir o leer el objeto de un archivo. Como ya vimos en uno de los cuadros presentados al principio de la secuencia, heredan de **InputStream** y **OutputStream**, de hecho son prácticamente iguales a **DataInput** y **OutputStream** con la diferencia de incorporar los métodos **readObject()** y **writeObject()** que son los que permiten grabar directamente objetos. Sirvan estos dos fragmentos de código para nuestra primera bajada al terreno de lo práctico que nos permitan pasar de las palabras a los hechos:

Transforma objeto en bytes y lo guarda en fichero	Transforma bytes en fichero a objeto

FileOutputStream fos = new FileOutputStream("src/fich.dat");
//ObjectOutputStream es usada por Java para serializar objetos
ObjectOutputStream oos = new ObjectOutputStream(fos);
ClaseSerializable o1 = new ClaseSerializable();
ClaseSerializable o2 = new ClaseSerializable();
// Escribe el objeto en el fichero
oos.writeObject(o1);
oos.writeObject(o2);
	FileInputStream fis = new FileInputStream("src/fich.dat");
	//ObjectInputStream es la usada para deserializar objetos
	ObjectInputStream ois = new ObjectInputStream(fis);
	ClaseSerializable o1 = new ClaseSerializable();
	ClaseSerializable o2 = new ClaseSerializable();
	// Lee el objeto del fichero
	o1 = (ClaseSerializable)ois.readObject();
	o2 = (ClaseSerializable)ois.readObject();

Si lo que queremos ya es lanzarnos a hacer una prueba de serialización y deserialización, ahí van estos dos códigos simples pero ilustrativos. Este primero se encarga de serializar. Observa la extensión del fichero **(.ser)** en el que se almacenará el objeto serializado. Es la convención que se usa en Java aunque no es obligatorio pero sí recomendable.

```
public class Alumno implements java.io.Serializable {
   public String nombre;
   public String direc;
   public transient int nss;
   public int id;

   public void comprobandoDatos() {
      System.out.println("Nombre: " + nombre + "Dirección " + direc);
   }
}
```

Ojo con **transient**

Este vídeo complementa lo explicado hasta ahora sobre serialización:
https://www.youtube.com/watch?v=POj5owpInuY

```
import java.io.*;
public class DemoSerializar {
   public static void main(String[] args) {
      // Crea el objeto que va a ser serializado
      Alumno alum = new Alumno();
      alum.nombre = "Pepe Da Rosa";
      alum.direc = "c/ Humor, 12, Sevilla";
      alum.nss = 11122333;
      alum.id = 101;
      try {
         // Preparando el fichero que recibirá los datos del objeto a serializar
         FileOutputStream fichSalida = new FileOutputStream("src/alumno.ser");
         // Creación del flujo para la serialización
         ObjectOutputStream salida = new ObjectOutputStream(fichSalida);
         // Momento de serializar
         salida.writeObject(alum);
         salida.close();      fichSalida.close();
         System.out.printf("Los datos serializados guardados en src/alumno.ser");
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }
}
```

Y ahora toca el turno de reconstituir el objeto a su estado natural (deserializarlo) y para ello hay que leer del fichero donde se guardó la información del mismo:

```
import java.io.*;
public class DemoDeserializar {
   public static void main(String [] args) {
      //Preparando el objeto que recibirá la información
      Alumno alum = null;

      try {
         // Preparando el fichero que almacena los datos serializados
         FileInputStream fichEnt = new FileInputStream("src/alumno.ser");
         // Creación del flujo para la deserialización
         ObjectInputStream entrada = new ObjectInputStream(fichEnt);
         //Casting imprescindible porque readObject() devuelve obviamente Object
         alum = (Alumno) entrada.readObject();
         entrada.close();      fichEnt.close();

      } catch (IOException ioe) {
         ioe.printStackTrace();    return;

      } catch (ClassNotFoundException cnfe) {
         System.out.println("Clase Alumno no encontrada");
         cnfe.printStackTrace();  return;
      }

      System.out.println("Alumno deserializado...");
      System.out.println("Nombre: " + alum.nombre);
      System.out.println("Address: " + alum.direc);
      System.out.println("NSS: " + alum.nss);
      System.out.println("Expediente: " + alum.id);
   }
}
```

Al ver el resultado en pantalla acuérdate de **transient** para explicar lo sucedido y arreglarlo.

Investiga un poco

Investiga los métodos que contienen las clases **File** y **Files** con objeto de confeccionar una comparativa en forma de tabla que permita conocer qué métodos hacen operaciones similares en una y otra clase.

Follow The Code

Indica qué salida sería esperable por pantalla si se ejecuta este código Java y de paso averigua la utilidad del método ***available()***.

```
import java.io.*;
public class Demo1Ftc {
   public static void main(String [] args) {
      try {
         byte tabInt [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("src/test.txt");
         for(int i = 0 ; i < tabInt.length ; i++) {
            os.write( tabInt[i] );
         }
         os.close();
         InputStream is = new FileInputStream("src/test.txt");
         int tam = is.available();
         for(int i = 0 ; i < tam ; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException ioe) {
         System.out.print("Excepción capturada");
      }
   }
}
```

Follow The Code

Indica qué salida sería esperable por pantalla si se ejecuta este código Java y de paso averigua la utilidad del método ***list()***.

```
import java.io.File;
public class Demo2Ftc {
   public static void main(String [] args) {
      File file = null;
      String[] paths;
      try {
         file = new File("/tmp");
         paths = file.list();
         for(String path:paths) {
            System.out.println(path);
         }
      } catch (Exception e) {    e.printStackTrace();      }
   }
}
```

Cuando llegue el momento de probarlo, elige la ruta del directorio que quieras. **tmp** es por poner algo.

Entrenamiento 1

Prueba los códigos expuestos de ejemplo desde el principio de esta secuencia.

Entrenamiento 2

Te puede servir para entrenar las habilidades básicas de operación con ficheros bajo el manto de la clase File el siguiente vídeo: **https://www.youtube.com/watch?v=TBzGXYqFq3w**

Entrenamiento 3

Te puede servir para entrenar las habilidades básicas de operación con ficheros bajo el manto de la clase File el siguiente vídeo: **https://www.youtube.com/watch?v=vTLho2lhSQg**

Ejercicio 1

Implementa un programa en Java que rellene un fichero de nombre **numeros.dat** con una cantidad aleatoria de números enteros también generados de manera aleatoria.

Ejercicio 2

Implementa un programa en Java que permita la modificación de un número entero almacenado en el fichero **numeros.dat** utilizando el acceso aleatorio. Es por ello que primero se debe solicitar al usuario la posición que ocupa dicho entero (comprendida entre 1 y el número de enteros que contiene el fichero) que pretendemos modificar, de forma que una vez localizado se muestre en pantalla su valor. Hecho eso, se pedirá entonces al usuario el valor de sustitución que se escribirá en la posición indicada del fichero.

Y te preguntarás, ¿cómo sé yo el número de enteros que contiene el fichero? Pues hombre, con un poco de raciocinio, ya conocemos de otras sesiones métodos que calculan el tamaño de alguna estructura. Además ya sabes cuanto ocupa un entero, así que no te resultará muy difícil dar con ello.

Ejercicio 3

Implementa un programa en Java que muestre al usuario un menú mediante el cual permita por una parte el volcado de una tabla con los 100 primeros números naturales a un fichero de texto y por otro mostrar por pantalla el contenido del fichero de texto creado. La tercera opción que es salir del programa se da por sobreeentendida.

Ejercicio 4

Implementa un programa en Java que lee un fichero de texto, cuenta el número de veces que aparecen cada una de las vocales del alfabeto y muestra en pantalla tanto el texto como el número de vocales de cada tipo.

Ejercicio 5

Implementa un programa en Java que acceda a un directorio que contenga ficheros y subdirectorios de forma que muestre por pantalla un listado de dichos ficheros y subdirectorios.

Ejercicio 6

Échale un vistazo al siguiente código e implementa el código de serialización que habría que ejecutar previamente para que este funcione correctamente.

```
import java.io.*;
public class EjemploLeerSerial {
   public static void main(String[] args) {
      try {
         FileInputStream fos=new FileInputStream("src/fich.dat");
         ObjectInputStream os = new ObjectInputStream(fos);
         Coche c;
         boolean finalArchivo = false;
         while (!finalArchivo) {
            c = (Coche) readObject();
            System.out.println(c);
         }
      }
      catch(EOFException eofe) {
         System.out.println("Se alcanzó el final");
      }
      catch(ClassNotFoundException cnfe) {
         System.out.println("Error el tipo de objeto no es compatible");
      }
      catch(FileNotFoundException fnfe) {
         System.out.println("No se encontró el archivo");
      }
      catch(IOException ioe) {
         System.out.println("Error "+ioe.getMessage());
         ioe.printStackTrace();
      }
   }
}
```