Los archivos de texto se buscan o se crean en la carpeta de inicio del proyecto

FICHEROS DE TEXTO Y FICHEROS BINARIOS

Son conceptos que debes de tener claro. Vuelve a leer el boletín 1 si es necesario.

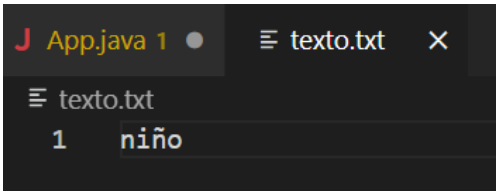
LEER UN FICHERO DE TEXTO

Hay muchas formas posibles de leer de un fichero de texto, algunas fundamentales:

- Podemos leer de carácter en carácter con read() de Reader.
- Podemos leer de n en n caracteres read(char[] b). Donde n es el tamaño de un array de caracteres.
- Podemos leer líneas con readLine() de BufferedReader.
- Podemos obtener con lines() un Stream con la información del fichero y procesarlo al estilo programación funcional

LEER UN FICHERO DE TEXTO DE CARÁCTER EN CARÁCTER

Veamos un ejemplo, para empezar creamos con ide u otro editor un un fichero texto.txt asegurándonos que se graba con utf-8. Ojo en windows que es donde a veces se utilizan codificaciones no Unicode.




```
import java.io.FileReader;
import java.io.IOException;
class App {
    public static void main(String[] args) throws IOException {
        FileReader in = new FileReader("mitexto.txt");
        int unByte;
        while ((unByte = in.read()) != -1) {
            System.out.print((char) unByte);
        }
    }
}
```

Recuerda la discusión de flujo de bytes vs flujo de caracteres del boletín 1. El método read() de FileReader trabaja duro para ensamblar bytes e interpretarlos para obtener un carácter escrito en utf-8 (FileReader por defecto usa utf-8).

Los famosos caracteres ASCII(los primeros 128 valores) están incluidos en todos los códigos con el mismo valor, pero para caracteres no ASCII cada código tiene sus propios valores. Así por ejemplo, las mayúsculas y minúsculas sin acentuar son caracteres ASCII pero las letras acentuadas no. En el ejemplo anterior podríamos tener problemas con la letra *o* acentuada si la codificación del fichero no coincide con la codificación que usa el método read().

FileReader utiliza por defecto UTF-8 podemos demostrarlo con

```
import java.io.FileReader;
import java.io.IOException;
class App {
    public static void main(String[] args) throws IOException {
        FileReader in = new FileReader("texto.txt");
        System.out.println(in.getEncoding());
    }
}
```

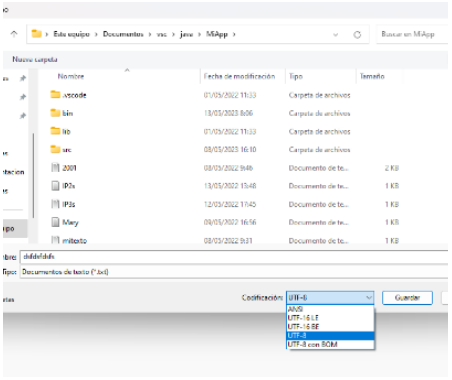
Si el fichero estuviera en otra codificación para que nuestro flujo trabaje con el código del fichero, lo indicamos a través de la clase puente InputStreamReader. Recuerda que InputStreamReader es una clase que transforma un flujo de bytes en flujo de caracteres. En el constructor de InputStreamReader podemos indicar con qué codificación queremos trabajar en el flujo de caracteres. Para trabajar sobre un objeto InputStreamReader no puedo usar FileReader(por diseño de JDK) y en el ejemplo se usa BuferedReader que si "traga" un InputStreamReader

```
import java.io.BufferedReader;
import java.io.FileInputStream;

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

class App {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("texto.txt");
        InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
        BufferedReader br = new BufferedReader(isr);
        int c;
        while ((c = br.read()) != -1) {
            System.out.println((char) c);
        }
    }
}
```

Si el fichero usa UTF-8 le OK los caracteres de niño. Entonces si leo el fichero con la codificación windows me leerá mal el fichero

```
InputStreamReader isr = new InputStreamReader(fis,"iso-8859-1");
```

Puedes experimentar en windows usando el bloc de notas y grabando el fichero con distintas codificaciones para concluir que tiene que haber correspondencia entre código en el que esta grabado el fichero y el código que usamos para leer.

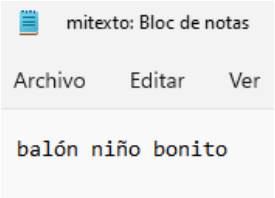


ACCEDER A DISCO CON BUFFER

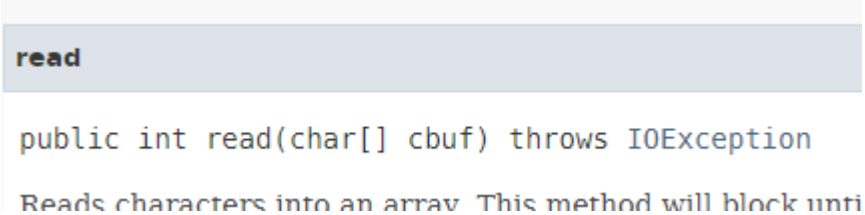
LEER DE N EN N CARACTERES USANDO UN ARRAY DE CHAR

El array se utilizará como un buffer que evita tener que acceder al disco para leer tan sólo un carácter.

El término buffer en programación se refiere a una zona de memoria intermedia. Por ejemplo, el buffer de teclado será una zona de memoria donde se va almacenando todo lo que teclea el usuario hasta que pulsa ENTER. Para acceder a disco también interesa usar buffer en este caso para evitar leer sólo un carácter ya que sería un esquema de acceso a disco muy ineficiente. Lo que normalmente se hace es leer muchos bytes juntos y almacenarlos en buffer, luego el programa irá cogiendo la información de dicho buffe. Los buffer de disco los gestionan automáticamente las clases de acceso a disco, no obstante hay una suerte de método de buffer manual que consiste en utilizar un array de char para el acceso.



Ahora el read() utiliza un array de char.



En el ejemplo, como el array tiene un tamaño de 4 caracteres, cada read() devuelve 4 caracteres para llenar el array/buffer, salvo la última lectura que según el texto puede devolver menos caracteres

```
import java.io.FileReader;
public class App {
    public static void main(String[] args) throws Exception {
        char[] b = new char[4];
        int leidos;
        try (FileReader f = new FileReader("mitexto.txt")) {
            while ((leidos = f.read(b)) != -1) {
              // System.out.println(new String(b));
              System.out.println(new String(b,0,leidos));
            }

        }
    }
}
```

CLASES CON BUFFER

Como indicamos, hay clases que trabajan internamente con buffers., muchas de ellas su nombre comienzan por "Buffer". Las clases que usan buffer son más eficientes. Clases que usan Buffer para leer son por ejemplo *BufferedInputStream* y *BufferedReader*. El buffer está en memoria RAM(como nuestro array del ejemplo anterior). Si no utilizamos buffer cada read() implica acceder a disco. Esto es ineficiente. Si utilizamos Buffer para leer la estrategia es que cada vez que se accede a disco se lee un bloque de bytes y luego cada read() o readLine() que invoquemos coge

los bytes de ese buffer, no directamente de disco. El programador no tiene que hacer nada para trabajar con buffer, salvo indicar el tamaño del buffer o sin indicar tamaño alguno se trabaja con el el tamaño por defecto. Es responsabilidad del programador acordarse de cerrar apropiadamente el flujo para vaciar el buffer antes de cerrarlo.

LEER DE LÍNEA EN LÍNEA

Ya que un fichero de texto lo vemos normalmente organizado en líneas, al trabajar con ficheros de texto, una forma atractiva y conveniente de leer un fichero de texto es de línea en línea. Esto es posible con el método readLine() de la clase BufferedReader ¡ya lo conocemos!.

Para crear un objeto BufferedReader que lea de disco necesitamos un objeto Reader, por ejemplo un FileReader. No es necesario ahora un objeto "puente" InputStreamReader que antes usábamos para transformar el flujo de bytes del teclado a flujo de caracteres.

Constructors

Constructor and Description
`BufferedReader(Reader in)` Creates a buffering character-input stream that uses a default-sized input buffer.
`BufferedReader(Reader in, int sz)` Creates a buffering character-input stream that uses an input buffer of the specified size.

Creamos con bloc de notas un fichero de texto 2001.txt con codificación UTF-8 que contenga el siguiente texto de *2001:una odisea en el espacio* (la famosa película).

Dave Bowman: Hola, HAL. ¿Me estás leyendo, HAL?
HAL: … Afirmativo, Dave. Te leo.
Dave Bowman: Abre las compuertas de la capsula, HAL.
HAL: Lo siento, Dave. Temo que no puedo hacer eso.
Dave Bowman: ¿Cuál es el problema?
HAL: Creo que sabes al igual que yo cuál es el problema.
Dave Bowman: ¿De qué estás hablando, HAL?
HAL: Esta misión también es importante para mí como para permitir que la pongas en peligro.
Dave Bowman: No sé de qué estás hablando, HAL.
HAL: Sé que tú y Frank estaban planeando desconectarme, y temo que eso es algo que no puedo permitir que suceda.
Dave Bowman: … ¿De dónde diablos sacaste esa idea, HAL?
HAL: Dave, si bien tomaron minuciosas precauciones en la capsula para impedir que les oyera, pude ver el movimiento de sus labios.
Dave Bowman: …Está bien, HAL... Entraré por la escotilla de emergencia.
HAL: Sin tu casco espacial, Dave, encontrarás eso bastante difícil.
Dave Bowman: HAL, no quiero discutir más contigo. Abre las puertas.
HAL: Dave, esta conversación ya no tiene ningún sentido... Adiós.

Podemos leerlo de la siguiente forma

```
import java.io.*;
class App{
  public static void main(String [] arg) throws IOException {
    FileReader fr = null;
    BufferedReader br = null;
    fr = new FileReader ("2001.txt");
    br = new BufferedReader(fr);
    String linea;
    while((linea=br.readLine())!=null)
        System.out.println(linea);
    fr.close();
    br.close();
  }
}
```

Estás 4 líneas

```
    FileReader fr = null;
```

```
BufferedReader br = null;
fr = new FileReader ("2001.txt");
br = new BufferedReader(fr);
```

se suelen escribir de forma compacta

```
BufferedReader br = new BufferedReader(new FileReader ("fich.txt"));
```

Y también podemos prescindir de la instrucción
fr.close();
Ya que como el flujo BufferedReader "envuelve" al flujo FileReader, al cerrar el primero también se cierra el segundo, por tanto, sólo es necesario un close(). Aunque esto tiene matices, y es posible que interese hacer el close() por separado.

Si no necesitamos un tratamiento especial en cierres , lo más sencillo es usar try con declaración de recursos para que se cierren los flujos automáticamente.

```
import java.io.*;
class App{
  public static void main(String [] arg) {
    try (BufferedReader br = new BufferedReader(new FileReader ("2001.txt"));) {
      String linea;
      while((linea=br.readLine())!=null)
        System.out.println(linea);
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Recuerda que puede interesarnos declarar los flujos por separado y el *try with resources* lo permite.

```
import java.io.*;
class App{
  public static void main(String [] arg) {

    try(FileReader fr = new FileReader("2001.txt");
        BufferedReader br= new BufferedReader(fr)) {
      String linea;
      while((linea=br.readLine())!=null)
        System.out.println(linea);
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

PROGRAMACIÓN FUNCIONAL: EL MÉTODO lines()

Un flujo de E/S es una fuente de datos típica para los Streams de programación funcional. Por ejemplo, el método lines() nos devuelve un Stream<String>, de forma que el código anterior lo pudimos haber escrito.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.stream.Stream;

class App{
  public static void main(String [] arg) {
```

```
    try(FileReader fr = new FileReader("2001.txt");
        BufferedReader br= new BufferedReader(fr)) {
        Stream<String> stream= br.lines();
        stream.forEach( s-> System.out.println(s));
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Otro ejemplo en el que se constata el buen maridaje entre flujos de E/S y programación funcional "De las líneas que le corresponden a HAL, dime cuantos caracteres tiene la línea más larga "

```
import java.io.*;
class App {
        public static void main(String[] arg) throws FileNotFoundException, IOException {
        try (BufferedReader br = new BufferedReader(new FileReader("2001.txt"))){
        int resultado= br.
                lines().
             filter(s->s.startsWith("HAL"))
                .mapToInt(s->s.length())
                .max()
                .getAsInt();
        System.out.println("la línea de mayor longitud(de las de HAL) tiene "+ resultado + " caracteres");
        }
        }
}
```

- El método lines() nos devuelve un Stream de líneas de texto en forma de String.
- Cada linea se selecciona si comienza por HAL. La función filter devuelve otro Stream de líneas seleccionadas
- con mapToInt convertimos el stream de String en un IntStream de números enteros, donde cada número entero indica el número de caracteres de la línea original (tal y como escribimos la expresión lambda)
- A un IntStream le podemos aplicar la función max(). Si consultamos el max() del API. Observa que devuelve un OptionalInt

OBJ

- Un objeto OptionalInt se pasa a int con getAsInt()

ESCRIBIR UN FICHERO DE TEXTO

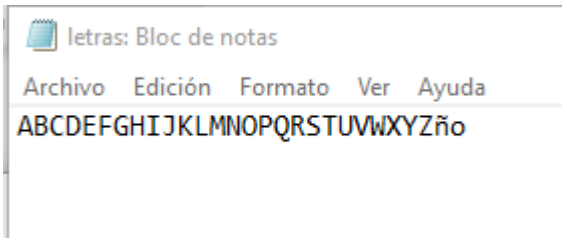
Como siempre...hay muchas posibilidades, veremos algunos ejemplos.

Ejemplo escribiendo de carácter en carácter

```
import java.io.*;
public class App{
   public static void main(String[] args){
```

```
        try(FileWriter fw = new FileWriter("letras.txt");) {
            for(char c='A'; c<='Z'; c++)
                fw.write(c);
            fw.write('ñ');fw.write('o');
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Podemos visualizar el contenido con el bloc y observamos que se imprimió bien la ñ ya que Filewriter escribe en utf-8 y bloc de notas sabe leer en utf-8

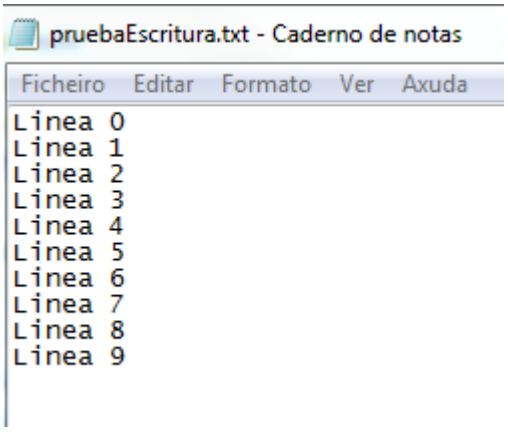


Para obtener un fichero como el anterior pudimos haber utilizado un flujo de bytes los caracteres A-Z es ok ya que su código utf-8 ascii cabe en un byte pero con la ñ falla.

```
import java.io.*;
public class App{
    public static void main(String[] args){
        try(FileOutputStream fos = new FileOutputStream("letras.txt");) {
            for(char c='A'; c<='Z'; c++)
                fos.write(c);
            fos.write('ñ');fos.write('o');
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Ejemplo escribiendo líneas

El fichero de texto que nos va a crear el programa tendrá el siguiente aspecto



Para hacer esto, podríamos utilizar el método write() de FileWriter. Fíjate que con read() de FileReader no podía leer strings(y usaba BufferedReader) pero con write() de FileWriter, si puedo escribir strings. El salto de línea tengo que añadirlo manualmente. Observa también que para simplificar y ahorrar el catch uso throws IOException en el main, no es malo esto en un ejemplo para ganar concisión. El cierre de flujos "se lo encargamos a " try-with-resources.

```
import java.io.*;
public class App{
    public static void main(String[] args) throws IOException{
        try(FileWriter fichero = new FileWriter("pruebaEscritura.txt");){
            String s1="Línea ";
            for (int i = 0; i < 10; i++){
                String s2=s1+i + "\r\n";
                fichero.write(s2);
            }
        }
    }
}
```

Observa que con teclado el salto de línea se representaba simplemente con \n pero aquí(en ficheros de texto) "puede" ser necesario también el retorno de carro \r. Conseguir una nueva línea puede tener sus matices según codificación y sistema operativo. No nos interesa entrar en detalles, probamos y nos quedamos con lo que funciona. Lo que siempre nos funciona es utilizar como separador el String que devuelve

```
System.lineSeparator()
```

Escritura de líneas utilizando buffer

En muchos ejemplos de la web, por razones de eficiencia ya explicadas, observamos que se utiliza la escritura con buffer. Tanto FileWriter como BufferedWriter usan Buffer para escribir, lo que ocurre es que BufferedWriter para escritura intensiva en disco está más optimizada al respecto.

Aprovechamos el siguiente ejemplo para analizar el trabajo con Buffer con conceptos también aplicables a la lectura con buffer. Cuando se trabaja con Buffer, sólo se graba en el disco una vez que se llena el buffer o se invoca a flush(). Si consultas en el api observas que el close() una de las cosas que hace es invocar al método flush(). El método flush() se encarga de actualizar el disco con el contenido del buffer.

Ejecuta el siguiente ejemplo ¡no tiene *fichero.close()* ni usa try-with-resources!

```
import java.io.*;
```

```
public class App {
    public static void main(String[] args) throws IOException {
        FileWriter fichero = new FileWriter("pruebaEscritura.txt");
        for (int i = 0; i < 10; i++) {
            fichero.write("linea" + i + "\r\n");
        }
     }
}
```

Prueba el ejemplo anterior para i<10, para i<100 y para i<1000. En los dos primeros casos si comprobamos el fichero, observamos que está vacío ya que no se llegó nunca a llenar el buffer ni se invocó a flush() o a close()
Si añadimos un close() vemos que ya funciona para i<10

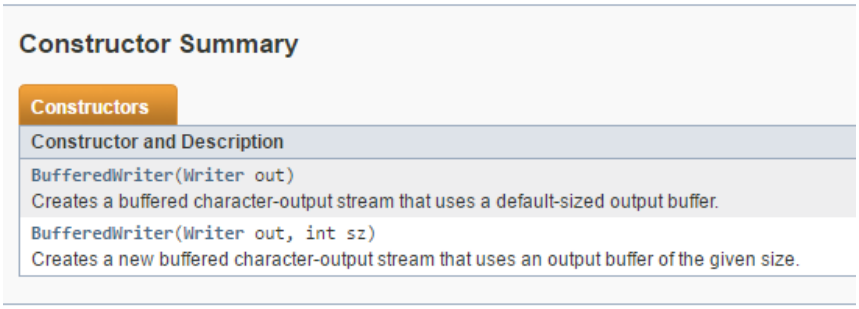
```
import java.io.*;
public class App {
    public static void main(String[] args) throws IOException {
        FileWriter fichero = new FileWriter("pruebaEscritura.txt");
        for (int i = 0; i < 10; i++) {
            fichero.write("linea" + i + "\r\n");
        }
        fichero.close();
     }
}
```

Como curiosidad, observa que en el siguiente código que tras cada write() hace un flush() , es decir, actualiza el contenido del buffer en el disco y por tanto, su efecto es equivalente a no usar buffer accediendo a disco para cada write

```
import java.io.*;

public class App{
    public static void main(String[] args) throws IOException{
        FileWriter fw = new FileWriter("letras.txt");

        for(int i=0; i<10;i++){
            fw.write("linea "+ i+ "\r\n");
            fw.flush();
        }
    }
}
```

BufferedWriter.

Ahora utilizamos la clase BufferedWriter. Para escrituras intensivas tiene un rendimiento superior, entre otras cosas, porque el tamaño del buffer es configurable. Observa la segunda versión de constructor.



Constructor Summary

Constructors

Constructor and Description
`BufferedWriter(Writer out)` Creates a buffered character-output stream that uses a default-sized output buffer.
`BufferedWriter(Writer out, int sz)` Creates a new buffered character-output stream that uses an output buffer of the given size.

Además BufferedWriter funciona con cualquier writer no necesariamente tiene que usar ficheros de disco como en estas pruebas, podría por ejemplo usarla para trabajar con sockets de red. En cambio FileWriter sólo trabaja con flujos de disco

```
import java.io.*;

public class App{
   public static void main(String[] args) throws IOException{
      FileWriter fw = new FileWriter("escrituraConBuffer.txt");
      BufferedWriter bfw =new BufferedWriter(fw);
      String s1="Línea";
      for (int i = 0; i < 10; i++)
         bfw.write(s1+i+"\r\n");
      bfw.close();
      fw.close();
   }
}
```

¿Qué ocurriría si no hubiera cerrado el flujo con Buffer, bfw.close()?

Pruébalo comentando bfw.close()

Aunque cierre el flujo FileWriter, el resultado puede ser inesperado. Como los datos están en el buffer, es posible que parte (o todos) de los datos aún no se hubieran guardado en el disco y si edito el fichero con el bloc de notas tras ejecutar el programa, observo que puede "faltar" información.

¿Qué ocurriría si no hubiera cerrado el flujo FileWriter, fw.close()?

Si cierro el flujo bfw se cierra automáticamente el flujo fichero. Sólo sería interesante cerrar específicamente el flujo fichero fw si quiero controlar qué pasa si se cierra bFichero y ocurre un error que dejaría sin cerrar el flujo fichero.

¿Qué ocurriría si hubiera cerrado los flujos al revés)?

```
fw.close();
bfw.close();
```

Ocurriría una IOException ya que en el momento de hacer bfw.close() el flujo BufferedWriter intenta utilizar un flujo FileWriter que no existe(ya cerrado).

Conclusión: cierra todos los flujos. Si unos flujos envuelven a otros, ciérralos en el orden contrario al que se abren, es decir, primero los más exteriores. Recuerda que salvo para aplicaciones que requieran ultra robustez, es suficiente cerrar el flujo más exterior.

Escribir con la clase PrintWriter

Y todavía hay más posibilidades para escribir ficheros de texto, como último ejemplo, se utiliza la clase PrintWriter. Esto nos permite trabajar con los métodos print y println igual que con System.out, esto es muy cómodo y se ve a menudo.

```
import java.io.*;

public class App{
    public static void main(String[] args) throws IOException{
        PrintWriter pw = new PrintWriter(new FileWriter("pruebaPrintWriter.txt"));
        for (int i = 0; i < 10; i++)
            pw.println("Linea " + i);
        pw.close();
    }
}
```

Una conclusión:

Además analizar cuestiones concretas de ficheros de texto también observamos que la E/S en java está diseñada con un gran número de clases y que unas usan a otras en base a múltiples combinaciones. Esto es un lío para aprender a programar pero es ventajoso en la programación de alto nivel.

Ejercicio U9_B3_E1:
Escribe en App, código de forma que args[0] sea un fichero xml que puede contener con \n y \t y crea un fichero de salida xml con el nombre indicado en args[1] sin \n ni \t

Intenta resolver el problema a la manera tradicional y también con estilo programación funcional

Puedes hacer un xml en un ficheor de texto por ejemplo con

```
<agenda>
  <persona>
    <nombre>Elías</nombre>
    <apellido>Lozano</apellido>
    <edad>7</edad>
  </persona>
  <persona>
    <nombre>Román</nombre>
    <apellido>López</apellido>
    <edad>6</edad>
  </persona>
  <persona>
    <nombre>Telma</nombre>
```

```
    <apellido>Lozano López</apellido>
    <edad>3</edad>
  </persona>
</agenda>
```