Control de excepciones. Lectura y escritura de información.

1. CONTROL DE EXCEPCIONES.

Supongamos que estamos ejecutando el código de un programa. Este se va ejecutando hasta que llega a una instrucción que JAVA no puede realizar porque hay un error en la misma. El programa se detiene y lanza una excepción (throws). El programa deberá manejar esta excepción para no acabar de manera brusca. Estas excepciones pueden producirse por muchas razones: división de 0, lectura de archivo incorrecta, cuando se recibe un formato incorrecto, etc.

En JAVA existen muchos tipos de excepciones. Cuando se produce una excepción se crea un objeto de una determinada clase, que mantendrá la información sobre el error producido y nos proporcionará los métodos necesarios para obtener dicha información. Estas clases heredan de la clase ***Throwable***, por tanto se mantiene una jerarquía en las excepciones, ver figura 1.

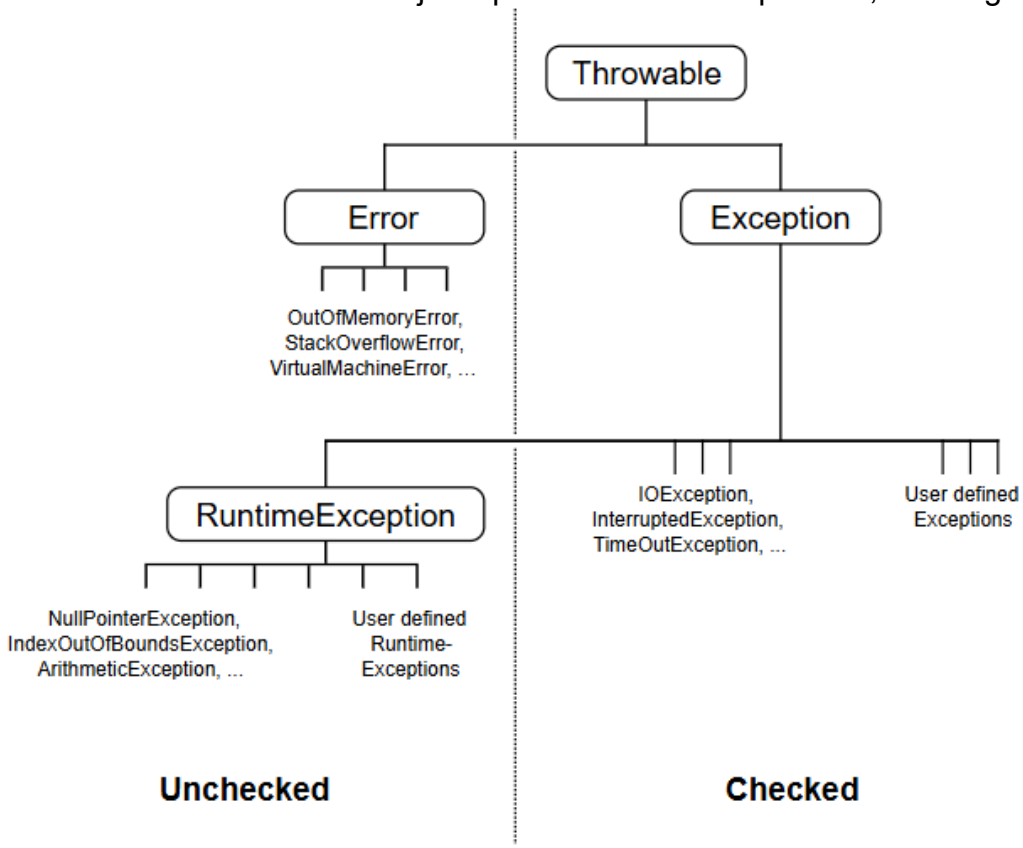


Figura 1.

Viendo la figura 1 debemos diferenciar entre un error y una excepción:

- **Error**: son problemas que el programador no puede solucionar como falta de memoria del sistema, error del sistema… En este caso el programa se detiene porque, como hemos dicho anteriormente, el programador no puede resolver esto.
- **Excepción:** se produce en las situaciones en las que la aplicación es capaz de resolver el problema de manera transparente al usuario lanzando advertencias. Existen dos tipos de excepciones:
    - **Checked:** se consideran excepciones recuperables, es decir, situaciones que un programa debería manejar adecuadamente en tiempo de ejecución y, por tanto, son aquellas que el compilador obliga a manejar explícitamente, ya sea mediante un try-catch o declarando que el método las lanza con throws (se explica más adelante).
    - **Unchecked:** Son subclases de RuntimeException y generalmente indican errores de programación que deberían corregirse en lugar de manejarse en tiempo de ejecución, por lo que no es obligado su manejo aunque puede hacerse.

Para manejar las excepciones lanzadas tenemos dos opciones. Capturarlas y decir al programa lo que debe hacer en el caso de que se produzcan. Para ello usaremos la estructura **try - catch - finally.** O informar de que el método lanza, throws, esta excepción que deberá ser manejada en donde se llama al método.

Veamos a continuación el uso del try-catch:

```
//Bloque1
try {
// Bloque2. Instrucciones que pueden provocar excepción.
} catch (TypeException ex1) {
// Bloque3. Instrucciones cuando se produce una excepción
} catch (TypeException ex2) {
// Instrucciones cuando se produce una excepción
} finally {
//Bloque4. Instrucciones que se ejecutan, tanto si hay como
si no hay excepciones
}
//Bloque5
```

Donde:

- **Try**: Aquí van las instrucciones propias del programa. Las que lanzarán la excepción.
- **Catch**: Aquí van las instrucciones que hará el programa cuando se produzca una excepción del tipo descrito en "*TypeException".* El código deberá tener tantas partes catch como excepciones queramos tratar. Si no se produce la excepción este código no se ejecuta.
- **Finally**: Son trozos de código que se ejecutarán haya o no excepción.

Los bloques se ejecutarían de la siguiente manera:

- Sin error: 1 -> 2 -> 4 -> 5
- Excepción de tipo descrito: 1 -> 2* -> 3 -> 4 -> 5
- Excepción de otro tipo: 1 -> 2* -> 4

Por tanto, el bloque finally se ejecutará después de try si no hay excepciones, después de catch si se capturó la excepción, o justo después de la excepción si está no se "captura". Como se puede apreciar el bloque 5 no se ejecuta si se produce una excepción y está no se trata. Veamos todo esto con un ejemplo:

```
public class DivisionCero {
    public static void main(String[] args) {
        final int numerador=10;
        final int denominador=0;
        float division= 0;

        System.out.println ("VOY A DIVIDIR");


        try{
         division = numerador/denominador;
         System.out.println ("DESPUÉS DE LA DIVISIÓN");
        }
        catch(ArithmeticException ex){
          division =1;
          System.out.println ("SE PRODUCE EXCEPCIÓN");
        }
        finally{
          System.out.println ("división = " + division );
        }


        System.out.println ("TERMINO EL PROGRAMA");
    }}
```

Este programa generará la siguiente salida:

VOY A DIVIDIR

SE PRODUCE EXCEPCIÓN

división = 1.0

TERMINO EL PROGRAMA

Si cambiamos el tipo de excepción para que se produzca una excepción de otro tipo:

```
public class DivisionCero {
    public static void main(String[] args) {
        final int numerador=10;
        final int denominador=0;
        float division= 0;

        System.out.println ("VOY A DIVIDIR");


        try{
         division = numerador/denominador;
         System.out.println ("DESPUÉS DE LA DIVISIÓN");
        }
        catch(NullPointerException ex){
          division =1;
          System.out.println ("SE PRODUCE EXCEPCIÓN");
        }
        finally{
          System.out.println ("división = " + division );
        }


        System.out.println ("TERMINO EL PROGRAMA");
    }
}
```

Al ejecutar este programa dará de salida:

VOY A DIVIDIR

división = 0.0

Exception in thread "main" java.lang.ArithmeticException: / by zero

at
divisioncero.DivisionCero.main(DivisionCero.java:13)

C:\Users\usuario\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1

Como se aprecia el bloque que va dentro de finally se ha ejecutado, lo que está fuera no.

1.1 CREACIÓN DE EXCEPCIONES PROPIAS.

En Java tenemos muchísimos tipos de excepciones pero puede ocurrir que, como programadores, queramos definir nuestras propias excepciones asociadas a nuestro programa. Por ejemplo, imaginemos que en nuestro programa estamos metiendo números y le pedimos al usuario que estos estén en un rango dado o no nos valdrán..

Como hemos visto las excepciones en Java no son más que clases que derivan de la clase Throwable. Así pues Java nos permitirá crear nuestras propias excepciones que no serán más que clases que heredan de la Exception (vinos que esta clase definía las clases que el programador podía corregir) o una subclase de esta.

```
public class Excepcionnombre extends Exception {
    public Excepcionnombre (String msg) {
        super(msg);
    }
```

Si analizamos esto vemos que se le pasa de argumento un String y que llamamos al constructor de la supercase donde inicializamos el mensaje de error de la excepción y la clase base se encarga de configurar el mensaje personalizado, de acuerdo con el msg.

Una vez que tenemos definido nuestra clase error… ¿cómo le decimos al sistema cuando se produce el error? Para ello es necesario “lanzar” la excepción con la palabra reservada **throw**.

```
throw new Excepcionnombre (mensaje);
```

Con este código podemos lanzar, no solo las excepciones creadas por nosotros, sino cualquier otra.

Además al definir un método que puede lanzar una excepción lo marcaremos con la palabra clave **throws** más el tipo de excepción que lanza. Se puede definir más de un tipo de excepción lanzada.

Recordemos cómo definimos el main cuando lanza una excepción de entrada/salida:

```
public static void main (String [] args) throws
IOException{
….
}
```

Consideremos el ejemplo anterior que el número debe estar en un rango (en el ejemplo entre -5 y 100 .

```
static void rango(int num, int den)throws
ExcepcionIntervalo{
        if((num>100)||(num<-5)){
            throw new ExcepcionIntervalo("Números fuera del
intervalo");
        }
```

NOTA: Son excepciones Unchecked todas aquellas excepciones que heredan de la clase RuntimeException, estás excepciones no necesitan ser declaradas en el método.

Para tratar esta excepción lanzada recurriremos a los bloques vistos anteriormente try y catch.

Veamos un ejemplo:

```
public class Matematicas {
    public double dividir(double a, double b) throws
Exception {
        if (b == 0) {
            throw new Exception("El argumento b no puede
ser 0");
        }
        return a / b;
  }
}
```

El main que va a tratar la excepción lanzada por esta clase será:

```
public class main {
	public static void main(String[] args) {
		Matematicas matematicas=new Matematicas();
		try {
			double c=matematicas.dividir(-1.6, 0);
		} catch (Exception ex) {
			//Tratar la excepción
		}
	}
}
```

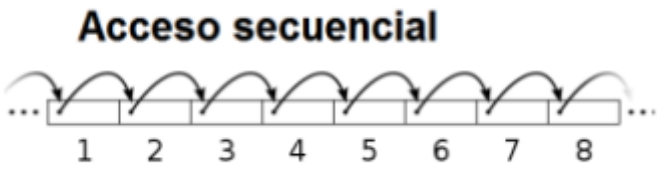
2. LECTURA Y ESCRITURA DE INFORMACIÓN EN FICHEROS.

Antes de explicar cómo se realiza la lectura/escritura de los ficheros veamos algunos conceptos que tenemos que tener claros al hacer este tipo de operaciones.

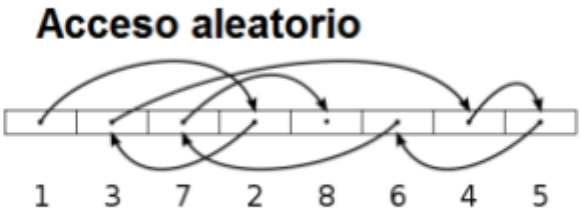
2.1 CLASIFICACIÓN DE LOS FICHEROS SEGÚN SU ORGANIZACIÓN.

Según se organicen la información en los ficheros podemos encontrar ficheros:

- **Secuenciales**: recibe este nombre porque los datos se almacenan uno detrás de otro, y el acceso a los mismos será de forma secuencial, es decir, tengo que recorrer el archivo desde el principio hasta llegar al dato que quiero recuperar. Para saber cuando hemos llegado al final del archivo existe la marca **EOF.** Los lenguajes de programación emplean el carácter EOF en los procesos de lectura de datos. Van recorriendo el archivo de forma secuencial, línea a línea, dentro de un bucle hasta leer la marca EOF y de la misma manera al escribir un dato nuevo esta escritura se hará al final del archivo y se añadirá automáticamente la marca EOF. Este tipo de ficheros solo permiten un tipo de operación a la vez o lectura o escritura y además, al ser el tipo de operación diferente, tendremos que cerrar el fichero antes de realizar la otra operación.



- **Aleatorios:** Los archivos de acceso aleatorio nos permiten ir directamente a recuperar el registro deseado, sin necesidad de leer antes todos los anteriores, para ello es necesario conocer exactamente en qué posición, dentro del archivo, se encuentra el registro que buscábamos. Esto se consigue gracias a que la longitud de registro estaba estrictamente definida. Es decir, todos los registros tenían la misma longitud, ocupaban el mismo espacio en memoria, y se emplea el número de registro, para posicionarnos en el registro deseado. Además es interesante saber que estos ficheros permiten las operaciones de lectura y escritura al mismo tiempo, no hace falta cerrar el archivo.

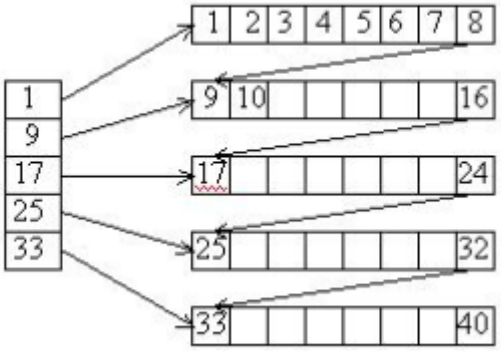


	Campo 1																												
byte ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
valor	F	r	u	t	a	s		G	u	t	i	e	r	r	e	z														

	Campo 2																												
byte ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
vañpr	A	n	t	o	n	i	o		G	u	t	i	e	r	r	e	z													

	Campo 3								
byte ->	1	2	3	4	5	6	7	8	9
valor	6	0	7	4	5	4	5	4	5

- **Secuenciales indexados:** Los registros se organizan en una secuencia basada en un campo clave, este proporciona una capacidad de búsqueda para llegar rápidamente al registro deseado. Para ello se recorren los campos claves de manera secuencial hasta llegar al valor deseado y así obtener la dirección del registro.



Una vez dicho esto aclarar que para grandes cantidades de información los datos se almacenan en base de datos y, por esto y por simplificar los procesos, los archivos que suelen usar los programas son secuenciales.

2.2 LA CLASE FILE.

La clase **File** en java representa un fichero como directorios.

```
import java.io.File;
```

…

```
File f = new File("un_path/un_fichero.txt");
```

Esta clase nos permite hacer las siguientes operaciones:

- Crear un fichero.
- Comprobar si existe un determinado fichero
- Comprobar si un item es un archivo o un directorio.
- Comprobar los permisos de un fichero.
- Borrar un fichero.

- Obtener la ruta completa de un fichero.
- Obtener el nombre de un fichero.
- Obtener la ruta del directorio que contiene un fichero. Es decir, obtener su ruta padre.
- Obtener el tamaño de un fichero.
- Comprobar si un fichero o directorio está oculto.
- Listar todos los archivos y directorios de un determinado directorio.
- Crear un directorio.
- Cambiar los permisos de un fichero.

Para ello podremos, entre otros, usar los siguientes métodos:

- **Constructores**:
  - public File(String nombreFichero|path);
  - public File(String path, String nombreFichero|path);
  - public File(File path, String nombreFichero|path);
- **canRead()/canWrite()/canExecute()**: devuelven un booleano (true/false) en el caso de que el archivo se pueda leer, escribir o ejecutar respectivamente.
- **createNewFile():** Crea el fichero asociado al objeto File. Devuelve true si se ha podido crear. Para poder crearlo el fichero no debe existir. Lanza una excepción del tipo IOException.
- **delete()**: Elimina el fichero o directorio. Si es un directorio debe estar vacío. Devuelve true si se ha podido eliminar.
- **exists()**: Devuelve true si el fichero o directorio existe.
- **getName()**: Devuelve el nombre del fichero o directorio (String).
- **getAbsolutePath()**: Devuelve la ruta absoluta asociada al objeto File (String).
- **getPath():** Devuelve la ruta con la que se creó el objeto File. Puede ser relativa o no (String).
- **isFile(), isDirectory()**: Devuelve true si es un fichero o un directorio válido respectivamente.
- **length():** Devuelve el tamaño en bytes del fichero (long). Devuelve 0 si no existe. Devuelve un valor indeterminado si es un directorio.
- **list():** Devuelve un array de String con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File. Si no es un directorio devuelve null. Si el directorio está vacío devuelve un array vacío.
- **listFiles():** Devuelve un array de File con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto File. Si no es un directorio devuelve null. Si el directorio está vacío devuelve un array vacío.
- **mkdir():** Crea el directorio. Devuelve true si se ha podido crear.

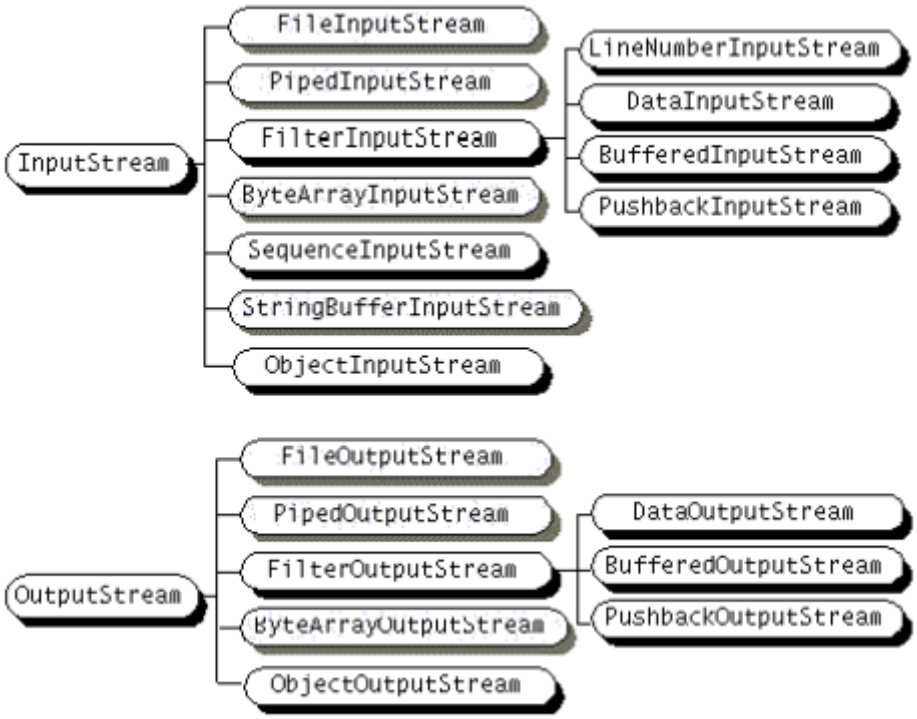
2.3 FLUJOS DE ENTRADA/SALIDA EN JAVA.

Hasta ahora se han hecho operaciones siguiendo los flujos de datos estándar (teclado, pantalla), para ello se han usado campos estáticos de la clase java.lang.System (System.in implementa la entrada estándar; System.out implementa la salida estándar; System.err implementa la salida de error)

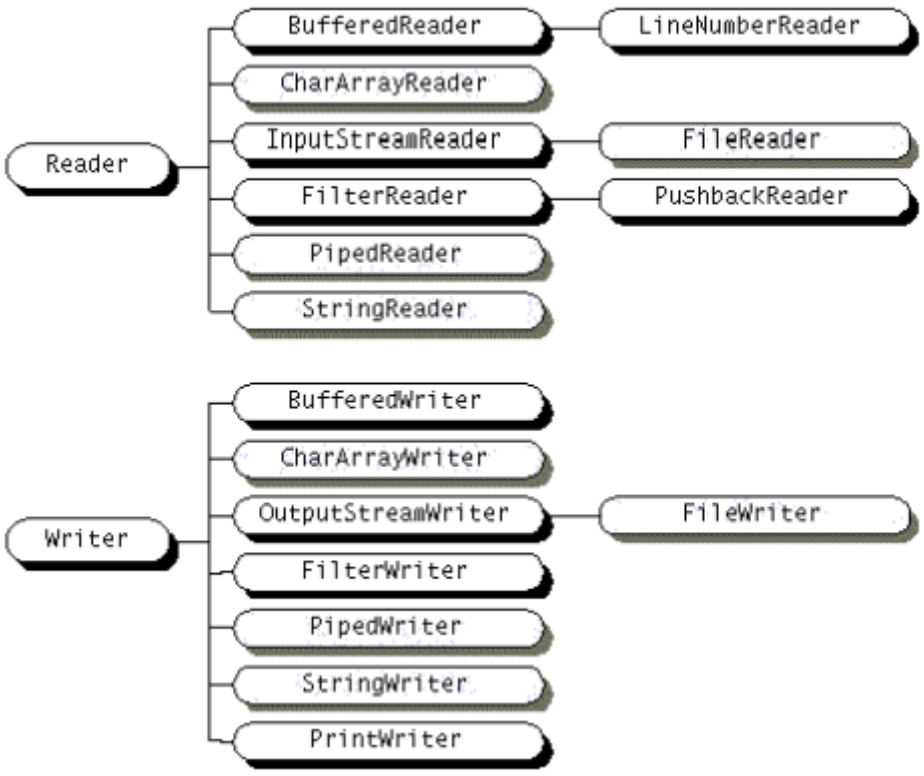
Cuando se trata de archivos podemos hablar de flujo de bytes o flujo de carácteres. El primero lo usaremos cuando mandemos información entre equipos y el segundo cuando la información necesite ser leída/interpretada por personas.

En java tenemos clases ya definidas que nos permiten tratar estos flujos de información:

- Para byte:



- Para carácteres:



Todas estas clases se encuentran en el paquete **java.io**, por lo que tendremos que importarlo antes.

Vamos a centrarnos en los flujos de caracteres. Recordemos que cuando abrimos un fichero para lectura o escritura solo vamos a poder hacer una de las dos operaciones. No es necesario decir cual porque ya se usará una clase u otra dependiendo de la operación que vayamos a realizar.

2.4 CLASE FILEREADER.

Para la lectura de archivos vamos a usar la clase InputStreamReader, es un puente que transforma los bytes a caracteres, y su subclase FileReader. Esta última nos servirá para decir dónde se encuentra el archivo a leer.

Para crear el objeto lector tendremos que crearlo primero usando:

```
FileReader nombre = new FileReader(String name);
FileReader nombre = new FileReader(File file);
```

Veamos los métodos que tiene la clase FileReader:

- método read:
  - **int read()** => lee un solo carácter del lector. Devuelve un entero, -1 si es el final del archivo.
  - int read(char() array) => lee los caracteres del lector y los almacena en la matriz especificada.
  - int read(char() array, int start, int length) => lee el número de caracteres igual a la longitud del lector y los almacena en la matriz especificada comenzando desde el inicio de la posición.

- **void close()** => cierra el flujo lector.

Estos métodos lanzan la excepción IOException por lo que tendremos que capturarla o relanzarla.

Veamos un ejemplo con esto:

```
class lee_fichero{
    //método de la clase lee_fichero que leerá el fichero

    //public lee_fichero(){
    //}


    public void lee() throws IOException {
    int c;
    char letra;
      try {
            //lanza excepcion FileNotFoundException
            //este flujo de datos hay que cerrarlo
            FileReader entrada = new FileReader
("C:/Users/usuario/Desktop/hola.txt");
```

```
            //usamos el bucle dowhile para que me vaya
leyendo caracter a caracter
            do {
                //lanza excepcion IOException ex
                //devuelve codigo entero
                c=entrada.read();
                //System.out.print(c);

                //convertimos ese entero a un caracter
                letra = (char)c;
                System.out.print(letra);
            }while (c!=-1);

             System.out.println();
             //cerramos el flujo de entrada para no consumir
recursos
             entrada.close();
        }
        catch (IOException ex) {
         System.out.println ("archivo no encontrado");
        }

    }
}
```

2.5 CLASE FILEWRITER.

Para la lectura de archivos vamos a usar la clase OutputStreamWriter, es un puente que transforma los caracteres a bytes, y su subclase FileWriter. Esta última nos servirá para decir dónde se encuentra el archivo a escribir.

Para usar el objeto tendremos que instanciarlo primero. Podemos usar dos constructores, este que sobreescribe el fichero:

```
FileWriter nombre = new FileWriter(String name);
FileWriter nombre = new FileWriter(File file);
```

O el siguiente que escribe a continuación:

```
FileWriter nombre = new FileWriter(String name, true);
FileWriter nombre = new FileWriter(File file, true);
```

Veamos los métodos que tiene la clase FileWriter:

- método write:

- **void write()** => escribe un solo carácter.
- **void write (String str)** => escribe una cadena.
- **void write (String str, int inicio, int len)** => escribe una parte de una cadena. Aquí *inicio* es el desplazamiento desde el que empezar a escribir caracteres y len es el número de caracteres a escribir.

- **void flush()** => limpia el flujo
- **void close()** => vacía y cierra el flujo lector.

Estos métodos lanzan la excepción IOException por lo que tendremos que capturarla o relanzarla.

Veamos un ejemplo con esto:

```
class escribe_fichero{

    public void escribe() {

        String frase = "voy a guardar esto";

        try{
          //imprimir sin sobreescribir en un fichero
existente
          FileWriter salida = new FileWriter
     ("C:/Users/usuario/Desktop/hola2.txt", true);


            //imprime la cadena
            salida.write(frase);
            salida.close();
        }
        catch(IOException ex){
            System.out.println ("problema al escribir");
        }
    }
}
```

2.6 CLASE BUFFEREDREADER Y BUFFEREDWRITER.

Las clases FileReader y FileWriter hacen la lectura/escritura directamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro. Para optimizar este proceso se usan las clases **BufferedReader** y **BufferedWriter** que añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.

- Si vamos escribiendo, se guardarán los datos hasta que tenga bastantes datos como para hacer la escritura eficiente.
- Si queremos leer, la clase leerá muchos datos de golpe, aunque sólo nos dé los que hayamos pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez.

Esta forma de trabajar hace los accesos a disco más eficientes y el programa correrá más rápido. La diferencia se notará más cuanto mayor sea el fichero que queremos leer o escribir.

Otra de las ventajas que presenta BufferedReader es que nos permite leer líneas del fichero y no carácter a carácter. Veamos cómo usarlo. Lo primero a tener en cuenta es que para tener un objeto BufferedReader debemos partir de un FileReader, de esta manera para definir los objetos tendríamos:

```
//crea un objeto BufferedReader con tamaño de buffer por defecto
BufferedReader in = new BufferedReader(new FileReader(Path));

//crea un objeto BufferedReader con tamaño de buffer definido
BufferedReader in = new BufferedReader(new FileReader(Path), tamaño);
```

La apertura del fichero y su posterior lectura pueden lanzar excepciones que debemos capturar. Por ello, la apertura del fichero y la lectura debe meterse en un bloque *try-catch*.
Además, el buffer hay que cerrarlo cuando terminemos con él, tanto si todo ha ido bien como si ha habido algún error en la lectura después de haberlo abierto. Por ello, se suele poner al *try-catch* un bloque *finally* y dentro de él, el *close()* del fichero.

La clase BufferedReader presenta, entre otros, los siguientes métodos:

- **read:**
  - **String readLine()** => Lee una línea del texto.
  - **void read()** => Lee un solo carácter, este será un entero, hay que hacer un casting.

- **boolean ready()** => devuelve true si aún hay caracteres en el flujo para leer y en caso contrario false (cuando llega al final del fichero).
- **void mark (int limitetamaño)** => establece una marca en la posición del apuntador del flujo. El parámetro int le dice el número máximo de caracteres que desea poder retroceder. Si lee demasiados datos más allá de la posición marcada, entonces la marca puede "invalidarse", y llamar a reset() fallará con una excepción.
- **void reset()** => regresará el apuntador a la última marca que haya sido hecha.

Ejemplo de lectura usando BufferedReader:

```
public static void lecturaLinea() throws IOException{

     BufferedReader br = new BufferedReader(new
     FileReader("C:/Users/usuario/Desktop/hola.txt"));
     String linea;

     //ready() retornará false cuando se llege al EOF
while (br.ready()) {
//lee línea del fichero y la imprime por pantalla
linea = br.readLine();
System.out.println(linea);
}
}
```

Ejemplo de lectura usando el método mark y reset de BufferedReader:

```
public static void lectura() throws IOException {
  try{
     BufferedReader br = new BufferedReader(new FileReader
     ("C:/Users/usuario/Desktop/hola.txt"));
     String linea;



     // Leer y mostrar las primeras dos líneas del archivo
     String linea= br.readLine();
     System.out.println(linea);
     linea= br.readLine();
     System.out.println(linea);
```

```
     //  Establecer  una  marca  en  la  posición
actual del flujo de entrada de caracteres, línea 2
            reader.mark(100);
     //Leer dos líneas más.
     linea = br.readLine();
     System.out.println(linea);
     linea = br.readLine();
     System.out.println(linea ne);


//regresamos el apuntador del flujo al inicio que es donde
hicimos la marca
br.reset();


     //volvemos a leer una línea
linea = br.readLine();
System.out.println(linea);
}
 catch (IOexception ex){
  System.Out.println("Error de lectura del fichero");
  }
 finally{
   br.close();
}
```

Este programa leerá las líneas de la siguiente manera: 1-2-3-4-3.

Para crear un objeto BufferedWriter:

```
BufferedWriter bw=new BufferedWriter(new FileWriter(Path));
```

Hay que tener en cuenta que el fichero se crea pero si ya existe su contenido se pierde. Si necesitamos abrirlo sin perder su contenido:

```
BufferedWriter bw=new BufferedWriter(new FileWriter(Path,
boolean append));
```

Si el parámetro **append** es **true** significa que los datos se van a añadir a los existentes. Si es **false** los datos existentes se pierden.

La clase BufferedWriter presenta, entre otros, los siguientes métodos:

- **write:**
  - **void write (int a)** => escribe un solo carácter especificado por int a.
  - **void write (String str, int pos, int length)** => escribe una parte de la string desde la posición pos hasta la longitud del número de caracteres.
- **void newLine()** => rompe / separa la línea.
- **void flush()** => vacía el búfer de escritura.
- **void close()** => vacía el búfer de escritura y luego lo cierra.

Veamos un ejemplo de esto:

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args) {
      try {
         FileWriter fw = new FileWriter("output.txt");
         BufferedWriter bw= new BufferedWriter(fw);
         String message = "Hola carabola";


         // Escribe en el buffer
         bw.write(message);
        }


     catch (IOException e) {
       System.out.println("Error al escribir en el
     archivo");
        }


     finally {
            try {
               //Evito un NullPointerException en caso de
               que el objeto bufferedWriter no se haya
               inicializado correctamente.
                if (bw != null) {
                    bw.close();
                }
            } catch (IOException e) {
              System.out.println("Error al cerrar el
archivo");
            }
        }
    }
}
```

Aunque la clase FileWriter proporciona el método write() para escribir cadenas de caracteres se suele utilizar esta clase junto con la clase **PrintWriter** para facilitar la escritura. La clase **PrintWriter** permite escribir caracteres en el fichero de la misma forma que en la pantalla.
Para ello habrá que crear el objeto PrintWriter a partir de un FileWriter:
Ejemplo:

```
try{
        FileWriter fichero = new
FileWriter("c:/prueba.txt");
        PrintWriter  pw = new PrintWriter(fichero);

//guardo en el fichero la línea que es del 1 al 10
     for (int i = 1; i < 11; i++)
      pw.println("Linea " + i);

}
catch (Exception e) {
            System.out.println ("se produce excepción");
 }
 finally {
           try {
           // aseguramos que se cierra el fichero.
           if (null != fichero)
              fichero.close();
           }
     catch (Exception e2) {
               System.out.println ("se produce excepción");
           }
}
```