

# Lectura y Escritura de Ficheros en Java

---

Veamos cómo leer y escribir ficheros de texto y binarios en java. También como acceder a dichos ficheros a través de los **Buffer** de java y alguna cosilla más.

Cualquier duda sobre el tema o de java en general, suelo atender en el [foro de java](http://foro.chuidiang.org) (<http://foro.chuidiang.org>).

## Lectura de un fichero de texto en java

---

Hay varias formas de leer un fichero de texto en Java. Veamos varias de ellas

### Uso de FileReader con BufferedReader

En la forma tradicional, se hace con la clase *FileReader* (<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/FileReader.html>). Esta clase tiene métodos que nos permiten leer caracteres. Sin embargo, suele ser habitual querer las líneas completas, bien porque nos interesa la línea completa, bien para poder analizarla luego y extraer campos de ella. *FileReader* no contiene métodos que nos permitan leer líneas completas, pero sí *BufferedReader*. Afortunadamente, podemos construir un *BufferedReader* a partir del *FileReader* de la siguiente forma:

```
File archivo = new File ("C:/archivo.txt");
FileReader fr = new FileReader (archivo);
BufferedReader br = new BufferedReader(fr);
...
String linea = br.readLine();
```

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 fichero 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 usar *try-with-resources*. En esta estructura, detrás de *try*, entre paréntesis, se abre el fichero y se almacena en una variable. Si la variable implementa la interface *Closeable*, se cerrará automáticamente al terminar el bloque *try*, independientemente de que salte o no una excepción.

```
try (BufferedReader br = new BufferedReader(...)) {
    ...
} catch (Exception e) {
    ...
}
// Aquí BufferedReader se habrá cerrado automáticamente, sin hacerlo explícitamente.
```

El siguiente es un código completo con todo lo mencionado.

```
import java.io.*;

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

        try (FileReader fr = new FileReader("C:/archivo.txt")) {
            BufferedReader br = new BufferedReader(fr);
            // Lectura del fichero
            String linea;
            while((linea=br.readLine())!=null)
                System.out.println(linea);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

## Recuerdos para toda la vida

Explore templos antiguos y sumérjase en la impresion  
belleza.

Qatar Airways

Re

Como opción para leer un fichero de texto línea por línea, podría usarse la clase *Scanner* en vez de el *FileReader* y el *BufferedReader*. Ver el ejemplo del [Ejemplo de lectura de un fichero con Scanner](#)

## Uso de Files y Path

El uso conjunto de la clase *Files* y *Paths* nos permite leer ficheros de texto de otras formas. La clase *Paths* nos devuelve una instancia *Path* que representa el fichero

```
Path file = Paths.get("c:/archivo.txt");
```

a partir de aquí, *Files* nos ofrece un par de formas de leer el fichero

### Files.lines()

*Files.lines()* nos devuelve un *Stream* de líneas de fichero que podemos leer con un bucle normal.

```
try (Stream<String> stream = Files.lines(path)) {  
    stream.forEach(line ->  
        System.out.println(line));  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

*Files.lines()* admite un segundo parámetro para pasar el *encoding* del fichero.

```
Stream<String> stream = Files.lines(path, Charset.defaultCharset())
```

### Files.newBufferedReader()

Con *Files* también podemos crear un *BufferedReader*, como en el primer ejemplo

```
try (BufferedReader reader = Files.newBufferedReader(path)) {  
    reader.lines().forEach(line ->  
        System.out.println(line));  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

podríamos usar *BufferedReader* como en el primer ejemplo, llamando a su método *readLine()*. Pero su método *lines()* nos ofrece también un *Stream* de líneas de fichero, al igual que la clase *Files*.

El método `newBufferedReader()` admite como segundo parámetro el *encoding* del fichero.

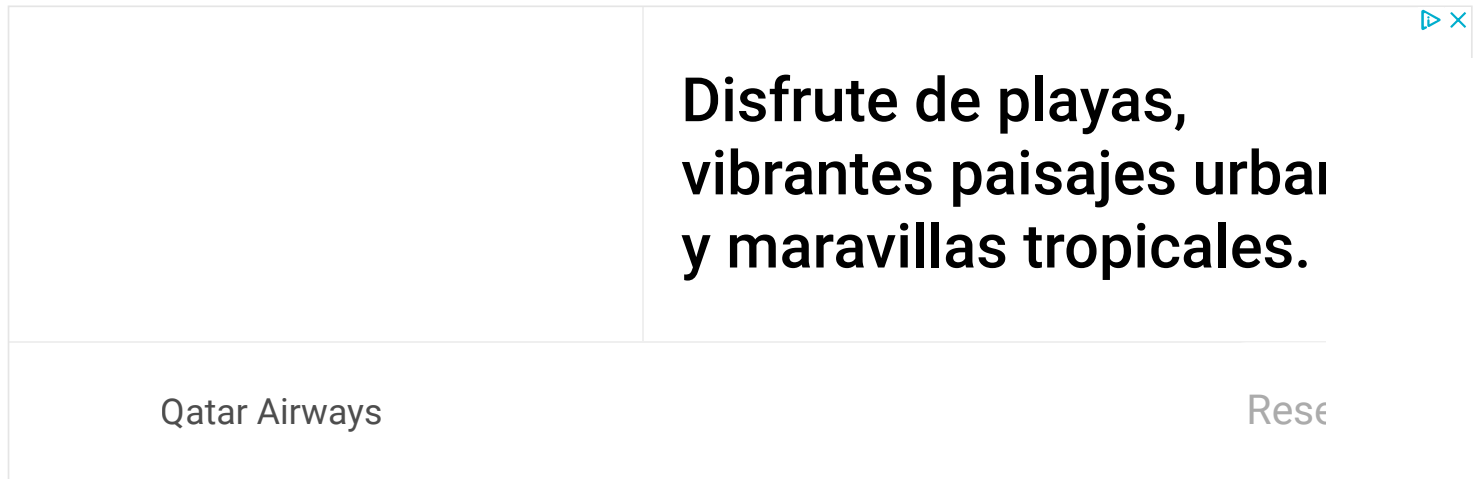
## Escritura de un fichero de texto en java

De la misma forma que en la lectura, tenemos varias formas de escribir en un fichero de texto

### Uso de FileWriter con PrintWriter

En la forma tradicional, se utiliza *FileWriter*. Como esta clase sólo tiene métodos para escribir *bytes*, podemos encapsularla en un *PrintWriter*, que tiene métodos para escribir líneas completas, incluyendo automáticamente el retorno de carro al final.

El siguiente código **escribe un fichero de texto** desde cero. Pone en él 10 líneas



```
import java.io.*;

public class EscribirFichero
{
    public static void main(String[] args)
    {
        try (FileWriter fichero = new FileWriter("c:/prueba.txt"))
        {
            PrintWriter pw = new PrintWriter(fichero);

            for (int i = 0; i < 10; i++)
                pw.println("Linea " + i);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Al igual que en la lectura, poniendo la apertura del fichero en un bloque *try-with-resources*, nos aseguramos que el fichero se cierra al terminar de escribir en él.

Si queremos **añadir al final de un fichero** ya existente, simplemente debemos poner un flag a `true` como segundo parámetro del constructor de **FileWriter**.

```
FileWriter fichero = new FileWriter("c:/prueba.txt", true);
```

### Files.newBufferedWriter()

Al igual que en la lectura de archivos, la clase *Files* nos ofrece opciones para la escritura.

```
<syntaxhighlight lang="java"> String[] lines = new String[] { "line 1", "line 2", "line 2" }; Path path = Paths.get("outputfile.txt");
```

```
try (BufferedWriter br = Files.newBufferedWriter(path,
```

```
    Charset.defaultCharset(), StandardOpenOption.WRITE)) {  
    for (String line : lines) {  
        br.write(line);  
        br.newLine();  
    }  
}
```



```
} catch (Exception e) {
```

```
    e.printStackTrace();
```

```
} </syntaxhighlight>
```

Dentro de la estructura *try-with-resources* usamos `Files.newBufferedWriter()` para obtener un *BufferedWriter*. Como parámetros, pasamos un *path* que obtenemos con `Paths.get()`, el *Charset* (encoding) que queremos que tenga el fichero y el tipo de operación que vamos a hacer con él. `StandardOpenOption.WRITE` escribe en el fichero desde cero, machacándolo si ya existe. Tenemos otras opciones, por ejemplo `StandardOpenOption.APPEND` para añadir sobre lo ya escrito si el fichero existe.

Fíjate que *BufferedWriter* no tiene un método *writeLine()* ni similar, por ello tenemos que escribir los *br.newLine()* explícitamente al final de cada línea. Una alternativa sería, una vez tenemos el *BufferedWriter*, encapsularlo en *PrintWriter*



Disfrute de playas,  
vibrantes paisajes urbanos  
y maravillas tropicales.

```
PrintWriter pw = new PrintWriter(br);  
...  
pw.println(...);
```

## Leer y escribir ficheros binarios

Al igual que con ficheros de texto, Java nos ofrece varias formas de leer y escribir archivos binarios. Veamos algunas de ellas.

### FileInputStream y FileOutputStream

En la forma tradicional, se usan *FileInputStream* y *FileOutputStream*. Estos no tienen métodos *readLine()* o *println* como *BufferedReader* o *PrintWriter*. En su lugar, tenemos métodos *read()* y *write()* de array de bytes.

El siguiente ejemplo hace una copia binaria de un fichero a otro, usando *FileInputStream* y *FileOutputStream*. Hay mejores formas de copiar ficheros binarios de un lado a otro, pero lo hacemos de esta forma para explicar cómo se leen y escribe bytes.

```
public class CopiaFicheros {  
    public static void main(String[] args) {
```

```

// Como fichero binario de ejemplo, usamos el mismo fichero .class compilado de esta clase.
copia("target/classes/com/chuidiang/ejemplos/java_nio/CopiaFicheros.class",
      "target/classes/com/chuidiang/ejemplos/java_nio/CopiaFicheros.class.copy");
}

/** Copia de un fichero binario en otro */
public static void copia(String ficheroOriginal, String ficheroCopia) {
    // Se abre el fichero original
    try (FileInputStream fileInput = new FileInputStream(ficheroOriginal)) {

        // Se abre el fichero donde se guardará la copia
        try (FileOutputStream fileOutput = new FileOutputStream(ficheroCopia)) {

            // Necesitamos un array de bytes para guardar lo leído
            byte[] array = new byte[1000];

            // Al leer, no tenemos garantía de leer todo el array, puede haber menos
            // bytes en el fichero. Así que guardamos el número de bytes leídos
            int leídos = fileInput.read(array);

            // Si se ha leído más de 0, se copian los leídos y se lee el siguiente bloque.
            while (leídos > 0) {
                fileOutput.write(array, 0, leídos);
                leídos = fileInput.read(array);
            }
            // Cuando se leen 0 bytes, es que el fichero se ha acabado.
        } catch (Exception e) {
            e.printStackTrace();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Vamos con los detalles de la lectura

## Lectura de fichero binario

En una estructura *try-with-resources* creamos la instancia de *FileInputStream* pasando como parámetro el path/nombre de fichero binario que queremos leer.

El método *fileInput.read()* nos permite leer una serie de bytes del fichero y los guarda en el array de bytes que le pasemos como parámetro. El método devuelve el número de bytes leídos. Imagina, como en el ejemplo, que le pasamos un array de 1000 bytes. Hay tres posibilidades:

- Si el fichero tienes bastantes bytes como para rellenar el array, lo rellenará al completo. En el *return* nos devolverá 1000, que es el tamaño del array.
- Si no tiene bastantes, sólo rellenará hasta donde tenga. En el *return* nos devolverá el número de bytes leídos, que será mayor de 0 y menor de 1000.
- Si no hay ningún byte disponible, no rellenará nada en el array. El método nos devuelve 0.

Así que la forma de leer todo el fichero es meternos en un bucle, ir leyendo y terminar cuando leamos 0 bytes.

Es importante destacar que después de una lectura, en el array de bytes sólo son válidos desde el byte 0 del array *leídos*, es decir, desde el byte 0 y longitud *leídos*.

## Escritura del fichero binario

Para la escritura, simplemente creamos una instancia de *FileOutputStream* pasando como parámetro el path/nombre del fichero que queremos crear. Podemos poner un segundo parámetro *true* si queremos añadir bytes a un fichero existente. Sin este segundo parámetro, el fichero, en caso de existir, se machaca.

Como es habitual, creamos la instancia de *FileOutputStream* dentro de una estructura *try-with-resources* para garantizar que se cierra cuando terminamos.

## Recuerdos para toda la vida

Explore templos antiguos y sumérjase en la impresion  
belleza.

Qatar Airways

Re

Una vez abierto, usamos el método `write()` para escribir bytes en el fichero. Hay tres métodos `write()` disponibles:

- Para escribir un byte únicamente `write(byte)`
- Para escribir un array de bytes completo `write(byte[])`
- Para escribir parte de un array de bytes `write(byte[], posicionInicial, longitud)`

En nuestro caso usamos el tercero, porque como hemos comentado en la lectura, no tenemos garantía de que todos los bytes de nuestro array sean del fichero. En nuestro ejemplo, el array de bytes es el que hemos leído del fichero, la posición inicial siempre será 0 y la longitud coincide con el *leídos* que nos ha devuelto `read()`.

## File vs Buffered

Si usamos sólo **FileInputStream**, **FileOutputStream**, **FileReader** o **FileWriter**, cada vez que hagamos una lectura o escritura, se hará físicamente 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.

Los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream** añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.

- Si vamos escribiendo, se guardará 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.

## Acceso aleatorio a un fichero

La clase `RandomAccessFile` (<http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html>) de java nos permite acceder para leer o escribir directamente en cualquier posición del fichero, sea binario o de texto.

## Enlaces relacionados

- [Búsqueda de ficheros](#)
- [Ficheros XML](#)
- [La clase File](#)