Grado Superior
Acceso a Datos

Ficheros

Curso 2024/25

Profesor
Jonathan Carrero

Java

Conceptos generales

Definición

Antes de comenzar, mencionar que la lectura y escritura se profundizará el año que viene en la asignatura de Acceso a Datos.

Es una **secuencia de bytes** en un dispositivo de almacenamiento tales como disco duro, CD, DVD, memoria USB, etc.

Se puede **leer** y/o **escribir**.

Se identifica mediante un nombre conocido como `pathname`:

```
/home/jonathan/documents/fichero.txt
```

Definición

La lectura y escritura de ficheros es muy importante porque permite a nuestros programas dialogar e **intercambiar** información con el usuario.

La entrada/salida se realiza mediante el empleo de **flujo** de datos.

Un flujo es una **secuencia ordenada** de datos que se transmite desde una fuente hasta un destino. En Java se denomina `Stream`.

El origen o el destino pueden ser un fichero, un `String`, un dispositivo (teclado, altavoz, pantalla...), etc.

Flujo de datos

La entrada/salida se organiza generalmente mediante **objetos** llamados `Streams`.

Un `stream` es la generalización de un fichero, es decir: su secuencia es ordenada y su origen o destino pueden ser un fichero, un `String`, otro dispositivo, etc.

Para poder usar un `stream` primero hay que **abrirlo**. Esto se hace en el momento de su creación y se **cierra** cuando se deja de utilizar.

Las clases relacionadas con streams se encuentran **definidas** en el paquete `java.io`, abreviatura de Input/Output.

Clasificación de los streams

Por el **tipo** de datos que transportan:

- Binarios: de bytes.
- Caracteres: de texto.

Por el **sentido** del flujo de datos:

- Entrada: los datos fluyen desde el fichero hacia el programa.
- Salida: los datos fluyen desde el programa hacia el fichero.

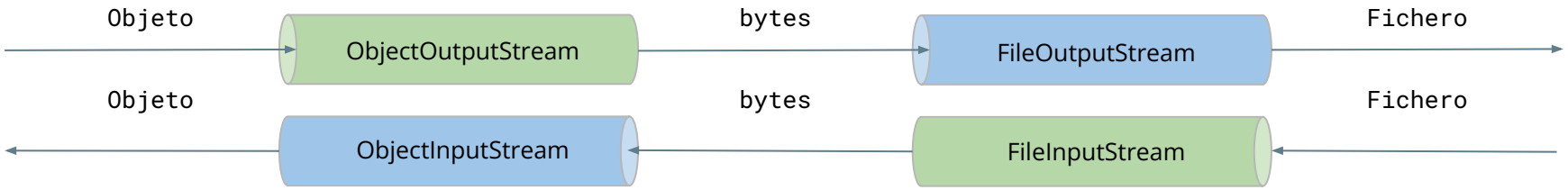
Según su **cercanía** al dispositivo:

- Iniciadores: directamente vuelcan o recogen los datos del dispositivo.
- Filtros: se sitúan entre un stream iniciador y el programa.

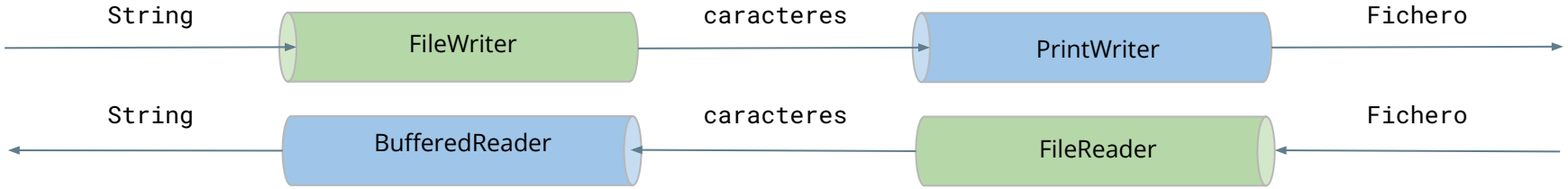
Uso de los streams

Para la leer y escribir de fichero se suele usar un `stream` iniciador y un filtro. Esto nos permite hablar “la misma lengua” que el fichero y comunicarnos con él correctamente.

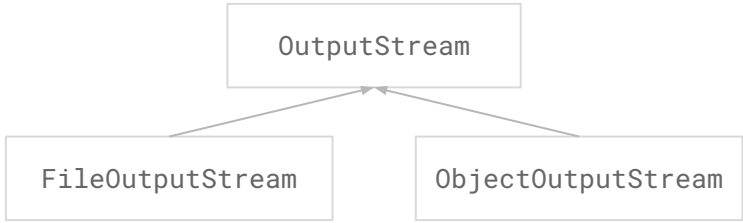
Binario



Texto

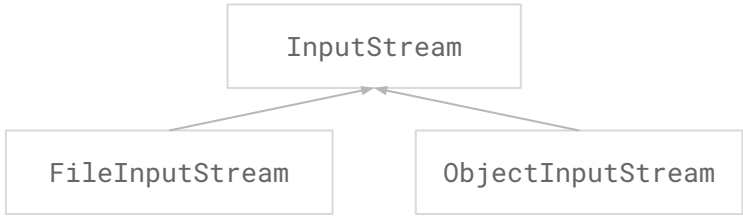


Jerarquía de clases streams binarios



`OutputStream`: escritura de ficheros binarios.

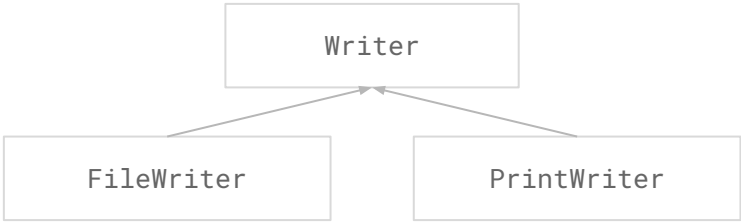
- **`FileOutputStream`**: escribe bytes en un fichero.
- **`ObjectOutputStream`**: convierte objetos y variables en arrays de bytes que pueden ser escritos en un `OutputStream`.



`InputStream`: lectura de ficheros binarios.

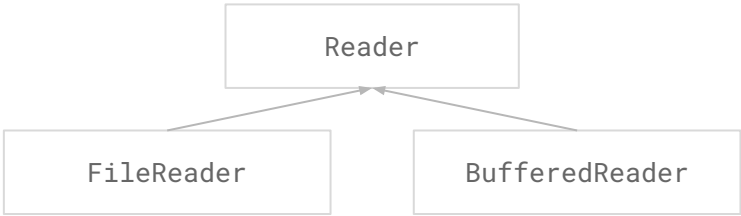
- **`FileInputstream`**: lee bytes de un fichero.
- **`ObjectInputStream`**: convierte en objetos y variables los arrays de bytes leídos de un `InputStream`.

Jerarquía de clases streams de caracteres



`Writer`: escritura de ficheros de texto.

- **`FileWriter`**: escribe texto en un fichero.
- **`PrintWriter`**: permite convertir a texto variables y objetos para escribirlos en un `Writer`.



`Reader`: lectura de ficheros de texto.

- **`FileReader`**: lee texto de un fichero.
- **`BufferedReader`**: lee texto (línea a línea) de un `Reader`.

Objetos stream predefinidos

- **`System.out`**: salida estándar por consola. Es un objeto de la clase `PrintStream` (subclase de `OutputStream`)

- **`System.err`**: salida de error por consola. También es un objeto de la clase `PrintStream`.

- **`System.in`**: entrada estándar por teclado. Es un objeto de la clase `InputStream`.

Java

Ficheros binarios

Clases para el manejo de flujo de bytes

Miscelánea de clases

`FileOutputStream`: permite escribir bytes hacia un fichero binario (ejemplo página 222).

`BufferedOutputStream`: permite escribir información a otro `OutputStream` utilizando un buffer interno que mejora el rendimiento (ejemplo página 222).

`DataOutputStream`: define algunos métodos que permiten escribir datos de tipo primitivo desde un `OutputStream` (ejemplo página 222).

`ByteArrayOutputStream`: permite usar un array de bytes como un `OutputStream` (ejemplo página 223).

Miscelánea de clases

`PrintStream`: permite escribir los datos de un stream en otro automáticamente y sin necesidad de invocar al método `flush()` (ejemplo página 223).

`FileInputStream`: permite leer bytes desde un fichero binario (ejemplo página 223).

`BufferedInputStream`: permite leer información de otro `InputStream` utilizando un buffer que mejora el rendimiento (ejemplo página 224).

`DataInputStream`: define métodos que permiten leer un tipo de datos primitivo desde un `InputStream` (ejemplo página 224).

Miscelánea de clases

`ByteArrayInputStream`: permite usar un array de bytes como `InputStream` (ejemplo página 224).

`SequenceInputStream`: flujo de entrada que puede combinar varios `InputStream` en uno (ejemplo página 225).

Java

Ficheros de texto

Clases para el manejo de flujo de caracteres

Miscelánea de clases

`FileWriter`: permite escribir caracteres hacia un fichero de texto (ejemplo página 226).

`BufferedWriter`: permite escribir información de otro `Writer` utilizando un buffer interno que mejora el rendimiento (ejemplo en página 226).

`PrintWriter`: es la implementación de la clase `Writer`. Permite escribir una cadena de caracteres en cualquier `OutputStream` (ejemplo en página 227).

`StringWriter`: flujo de caracteres que recoge su salida de un buffer y que puede utilizarse para construir un `String` (ejemplo en página 227).

Miscelánea de clases

`FileReader`: flujo de caracteres para la lectura de ficheros de texto (ejemplo página 228).

`BufferedReader`: permite leer información de otro `Reader` utilizando un buffer interno que mejora el rendimiento (ejemplo página 228).

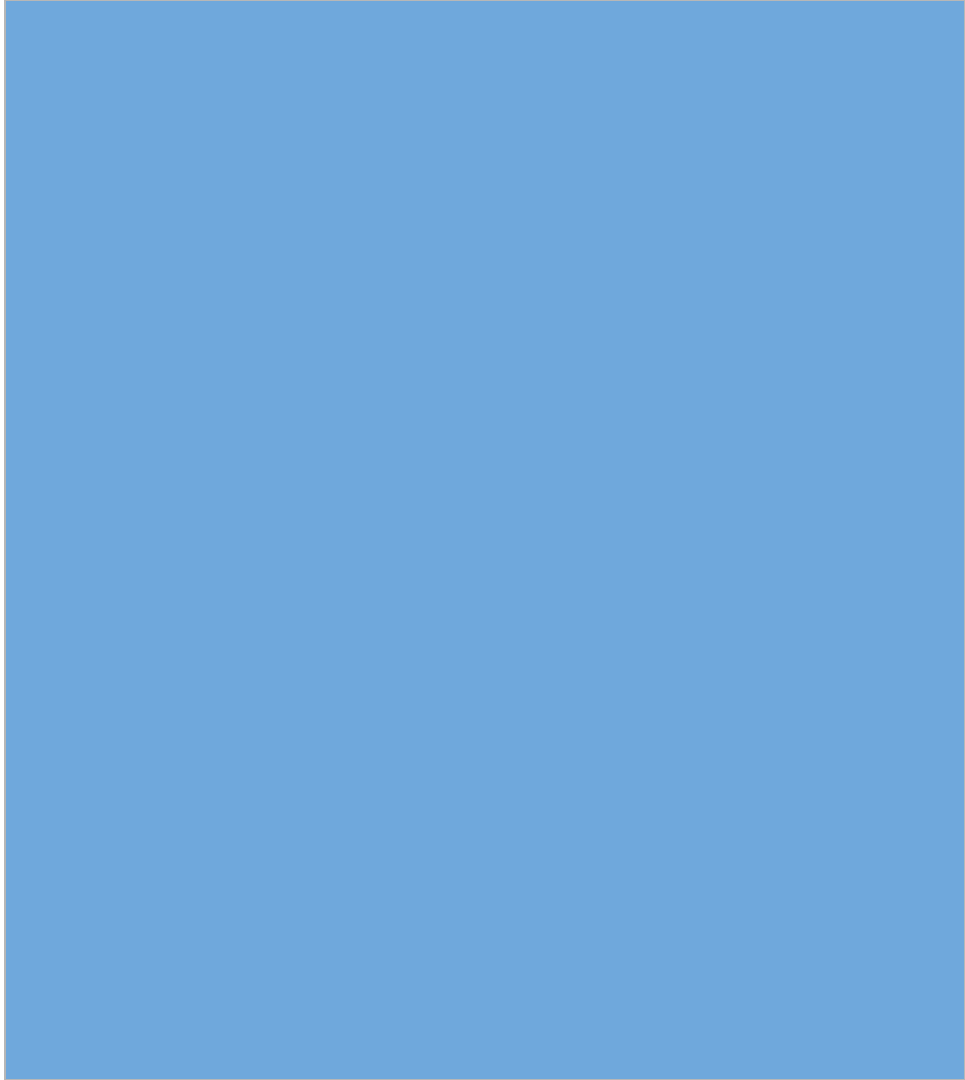
`LineNumberReader`: tipo especial de `BufferedReader` que lleva un conteo del número de línea que se está leyendo en cada momento (ejemplo página 228).

`CharArrayReader`: recoge de un buffer los datos provenientes de un array de caracteres (ejemplo página 228).

`StringReader`: recoge de un buffer los datos provenientes de un `String` (ejemplo página 229).

Java

Lectura y escritura en ficheros de texto con la clase Scanner



Lectura con la clase Scanner

La clase `Scanner` (paquete `java.util`) permite leer número y texto de un fichero de texto y de otras fuentes.

Permite la lectura de ficheros línea a línea (muy similar a `BufferedReader`).

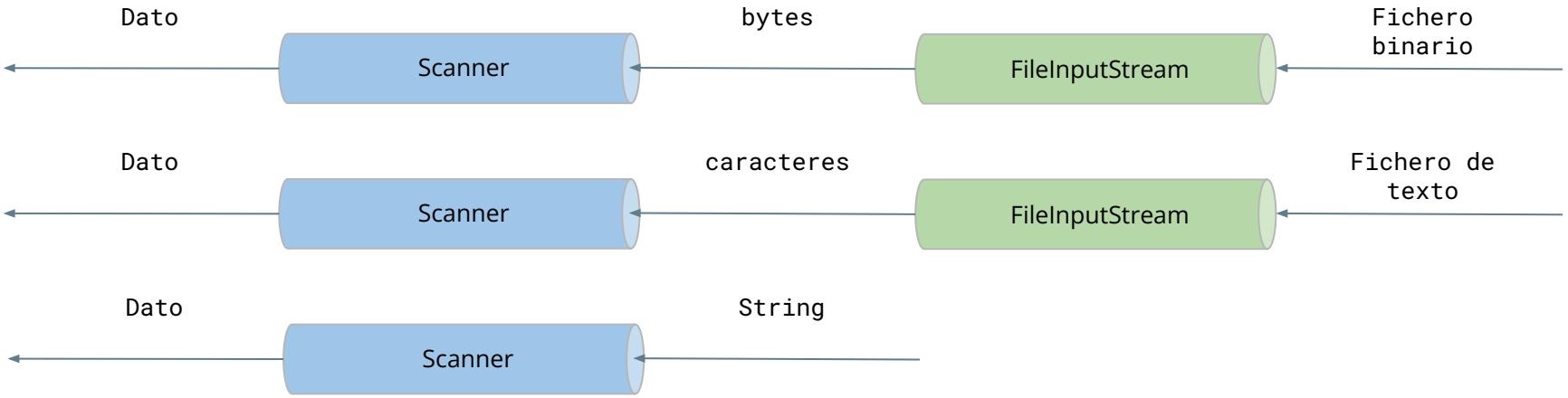
Permite la lectura sencilla de números y palabras separados por algún **separador** específico. El separador por defecto es cualquier espacio en blanco (espacio, salto de línea, tabulador, etc.).

Permite reconocer patrones de texto conocidos como **expresiones regulares**.

Tarea: ¿qué son las expresiones regulares?

Lectura con la clase Scanner

Además, la clase Scanner puede leer de un `InputStream` y de un `FileReader`.



Principales operaciones de la clase Scanner

Descripción	Declaración
Constructor. Requiere un `InputStream`.	`Scanner(InputStream source)`
Constructor. Requiere un objeto que implemente `Readable` (por ejemplo un `FileReader`).	`Scanner(Readable source)`
Constructor. Requiere un `String`.	`Scanner(String source)`
Cerrar.	`void close()`
Leer una línea.	`String nextLine()`
Indica si quedan líneas por leer.	`boolean hasNextLine()`
Lee un booleano.	`boolean nextBoolean()`

Principales operaciones de la clase Scanner

Descripción	Declaración
Indica si es posible leer una palabra que se interprete como un `booleano/double/int`.	`boolean hasNextBoolean(), hasNextDouble(), hasNextInt()`
Leer una palabra.	`String next()`
Indica si quedan más palabras o datos por leer.	`boolean hasNext()`
Leer un `double`.	`double nextDouble()`
Leer un `int`.	`int nextInt()`
Cambia el delimitador que separa los ítems.	`Scanner useDelimiter(String pattern)`

Excepciones de la clase Scanner

El uso de estas operaciones puede generar algunas excepciones tales como:

- `NoSuchElementException`: no quedan más palabras.
- `IllegalStateException`: el scanner está cerrado.
- `InputMismatchException`: el dato leído no es del tipo esperado.

Tarea: haz un esquema de las clases vistas para el manejo de ficheros binarios y de ficheros de caracteres. Además, enumera las principales operaciones para todas ellas.

Clase Scanner, ejemplo

```
public static void main() {
	final String nomFich = "datos.txt";
	Scanner in = null;

	try {
		// Abre el fichero
		in = new Scanner(new FileReader(nomFich));
		// Lee el fichero palabra a palabra
		while (in.hasNext()) {
			// Lee primera palabra
			String palabra = in.next();
			System.out.println("Palabra:" + palabra);
			// Lee números
			while (in.hasNextDouble()) {
				// Lee un double
				double d = in.nextDouble();
				System.out.println("Número: " + d);
			}
		}
```

Clase Scanner, ejemplo

```
	} catch (FileNotFoundException e) {
		System.out.println("Error abriendo el fichero " + nomFich);
	} finally {
		if (in != null)
			in.close();
	}
}
```


¿Qué muestra por pantalla?

Java

Actualización de apuntes

Vuelta al pasado

Antes de comenzar con esta nueva actualización del manejo de ficheros en Java, dedica el tiempo necesario para leer el temario de ficheros de la asignatura de Programación.

Si se presenta alguna duda, busca en Internet y debate los conceptos vistos con tus compañeros. Genera ejemplos con alguna IA y entiende qué es lo que hacen.

Mira algunos de los ejemplos que hicimos en clase, como por ejemplo cómo funcionaba el flujo de ejecución y qué clases relativas a los ficheros se estaban utilizando en las prácticas o en el proyecto de Simon Dice.

En otras palabras; quiero que tengas al día todo lo relativo a ficheros en Java antes de continuar. Al menos, todo lo que vimos el año pasado.

Clase File

Observa el ejemplo del paquete `ejemplo1` proporcionado por el profesor. Este es un ejemplo sencillo en el que escribimos y leemos de fichero. Observa que disponemos de una serie de métodos que resultan muy útiles para trabajar con ficheros (líneas 40-47).

Al mismo tiempo, date cuenta de que el fichero `ejemplo.txt` se ha creado dentro del proyecto. En otras palabras; si no indicamos una ruta concreta, tomará por defecto la ruta en la que se encuentre el proyecto.

Tarea: no ejecutes el proyecto. Piensa y escribe todo lo que se mostrará por pantalla.

Tarea: ¿qué tendrías que hacer si quisieras que el fichero `ejemplo.txt` se crease dentro de la carpeta `src`?

La interfaz Path

La interfaz `Path` proporciona una forma más moderna y flexible de trabajar con rutas de ficheros en Java. Junto con la clase `Files`, puedes realizar diversas operaciones como la creación, lectura y escritura de ficheros. Aquí hay un ejemplo de cómo utilizar `Path` y `Files` para escribir en un fichero.

Tarea: observa el ejemplo del paquete `ejemplo2` y, antes de ejecutarlo, deduce qué se imprimirá por pantalla. ¿Dónde crees que creará el fichero `example.txt`?

Java

Escritura en archivos

Diferentes formas de escribir

Como ya sabes, en Java hay muchas maneras de escribir datos en un archivo, y ciertas implementaciones pueden ser más adecuadas para sus necesidades que otras, dependiendo de la cantidad y el tipo de datos que esté escribiendo en un archivo. En las siguientes diapositivas vamos a ver cuatro de ellas.

1. La clase `Files`: normalmente para una ejecución simple y única, a la vez que sencilla. Es la que debe utilizarse a partir de Java 11.
2. La clase `FileWriter`: la manera más clásica de escribir desde la versión 1.1 de Java. Es la que utilizamos el año pasado en el proyecto de Simon Dice.
3. La clase `BufferedWriter`: si tu aplicación requiere escribir en un archivo muchas veces, entonces es recomendable usar esta clase, la cual ayudará a eliminar la sobrecarga de operaciones de escritura lentas.
4. La clase `FileOutputStream` y `BufferedOutputStream`: si estás escribiendo datos no textuales, entonces podrías considerar usar estas dos clases.

FileOutputStream y BufferedOutputStream

La clase `FileOutputStream` puede emplearse para escribir flujos de datos basados en bytes, como imágenes o datos Unicode, y otros tipos de datos que no se pueden representar como texto. También es posible escribir datos textuales simplemente convirtiendo la cadena de texto a bytes.

Del mismo modo, la clase `BufferedOutputStream` añade un mecanismo de almacenamiento en búfer al `OutputStream` subyacente para mejorar la eficiencia.

No hay reglas establecidas sobre cuándo utilizar un `BufferedOutputStream`, por lo que es importante considerar el tamaño del búfer, la velocidad del disco y de la red (si aplica), y cómo estos aspectos pueden afectar las mejoras o desventajas en el rendimiento antes de decidirse por esta opción para escribir en un archivo.

Observa el ejemplo que aparece en el paquete `ejemplo3`.

BufferedWriter I

El `BufferedWriter`, como su nombre indica, emplea una técnica de almacenamiento en búfer. Al llamar al método `write()`, los datos se escriben primero en un búfer interno del `BufferedWriter` y se guardan en el archivo solo cuando este búfer interno se llena por completo. Esta estrategia de almacenamiento en búfer disminuye la cantidad de operaciones de escritura costosas necesarias, por lo que el `BufferedWriter` es especialmente útil para escribir grandes archivos de manera eficiente.

La clase `BufferedWriter` recibe por parámetro un `FileWriter` que recibe el Path y el modo de apertura (`StandardOpenOption`). Profundizaremos más en el modo de apertura cuando lleguemos a la clase `Files`.

BufferedWriter II

Observa el primer ejemplo que aparece en el paquete `ejemplo4`. Vemos cómo se puede utilizar `BufferedWriter` que, si consultamos la documentación, recibe en la constructora algo de tipo `Writer`. Recuerda, tal y como vimos el año pasado, que `Writer` es una clase abstracta y que `FileWriter` la implementa (por eso en el código, `BufferedWriter` está recibiendo como parámetro algo de tipo `FileWriter`). Después, simplemente se llama al método `writer()`.

Observa el segundo ejemplo que aparece en el paquete `ejemplo4`. Aquí utilizamos `BufferedWriter` con `Path` y `Files`. Por supuesto, la utilización de clases para los ficheros no es exclusiva, por lo que tenemos muchas maneras de hacer las cosas. En este caso, creamos el fichero con `Path` en lugar de utilizar `FileWriter`. Luego, utilizamos `Files` para crear un `BufferedWriter` gracias al método *`newBufferedWriter()`*, el cual recibe el path, y eso que hemos dicho que veremos a continuación. Por último, utilizando el objeto creado, le decimos que queremos escribir en él.

Tarea: ¿por qué el método *`newBufferedWriter()`* está en cursiva?

FileWriter

La clase `FileWriter` ha estado presente en Java desde la versión 1.1, proporcionando una forma conveniente de escribir archivos de texto. Su sintaxis es clara y fácil de entender, entre otras cosas porque no utiliza un buffer... normalmente. Es la que utilizamos en Simon Dice.

Observa los dos ejemplos que aparecen en el paquete `ejemplo5`. En el primero de ellos únicamente hacemos uso de la clase FileWriter. Por el contrario, en el segundo de los ejemplos, hacemos uso de PrintWriter.

Esta clase crea un buffer a través del `FileWriter`, que permite extender los métodos del `FileWriter` por otros similares a los que tenemos en la salida de pantalla. El constructor recibe el `FileWriter` como parámetro. Entre las funciones que tenemos en el `PrintWriter`, las más comunes son `print()` y `println()`. Observa que el `PrintWriter` recibe como parámetro un objeto `FileWriter` en la línea 12.

Files I

Para escribir un pequeño archivo simple, la clase `Files` tiene un enfoque muy intuitivo utilizando el método `write()`. En este método, pasa los datos codificados como bytes y una ruta al archivo, y la clase `Files` maneja el proceso de escritura de archivos.

Decir que en Java 7, necesitábamos convertir un `String` en un `byte[]` antes de escribir.

```
String content = "...";

Files.write(path, content.getBytes(StandardCharsets.UTF_8),
                                   StandardOpenOption.CREATE,
                                   StandardOpenOption.APPEND);
```

Files II

En Java 11 podemos usar `Files.writeString()` para escribir directamente en el archivo.

```
private static void appendToFileJava11(Path path, String content) throws IOException {

      // default StandardCharsets.UTF_8
      Files.writeString(path, content,
                        StandardOpenOption.CREATE,
                        StandardOpenOption.APPEND);
}
```

Files III

`StandardOpenOption` proporciona una forma flexible y poderosa de controlar cómo se abren y manipulan los archivos en Java.

Al comprender y utilizar adecuadamente estas opciones, puedes manejar operaciones de E/S de archivos de manera más eficiente, segura y adaptada a las necesidades específicas de tu aplicación.

A continuación, vamos a ver una serie de ejemplos para comprender mejor cómo utilizar `Files`, sus ventajas y las diferentes opciones.

Files IV

Opción	Description	Uso común
CREATE	Crear el archivo si no existe.	Crear o abrir un archivo para escritura.
CREATE_NEW	Crear un archivo nuevo. Falla si el archivo ya existe.	Asegurarse de crear un archivo nuevo sin sobrescribir.
APPEND	Añade contenido al final del archivo.	Agregar datos a un archivo existente sin borrar el contenido.
TRUNCATE_EXISTING	Trunca el archivo a cero bytes si existe.	Sobrescribir completamente el contenido existente.
READ	Abre el archivo para lectura.	Leer datos de un archivo.
WRITE	Abre el archivo para escritura.	Escribir datos en un archivo.
DELETE_ON_CLOSE	Elimina el archivo al cerrar el canal.	Manejar archivos temporales que deben eliminarse después.

Files V

Consideraciones importantes:

- **Orden de las opciones:** algunas opciones pueden afectar el comportamiento de otras. Por ejemplo, si usas `CREATE_NEW` y `APPEND` juntos, `CREATE_NEW` intentará crear un archivo nuevo y fallará si ya existe, mientras que `APPEND` solo añade contenido si el archivo ya existe.
- **Compatibilidad de opciones:** no todas las opciones son compatibles entre sí. Es importante entender qué hace cada opción y cómo interactúan cuando se combinan.
- **Manejo de excepciones:** al usar opciones como `CREATE_NEW`, es fundamental manejar excepciones como `FileAlreadyExistsException` para evitar que el programa falle inesperadamente.
- **Rendimiento:** opciones como `APPEND` y `TRUNCATE_EXISTING` pueden tener impactos en el rendimiento dependiendo de la operación. Por ejemplo, `APPEND` puede ser más eficiente para añadir datos a un archivo grande sin tener que reescribir todo el contenido.

Files VI

Aquí tienes algunas ventajas de utilizar la clase `Files`:

- **Operaciones más sencillas:** La clase `Files` ofrece métodos simplificados para la creación, escritura, lectura y manipulación de archivos y directorios.
- **Mejor manejo de excepciones:** Utiliza excepciones más específicas como IOException, lo que permite un manejo de errores más preciso.
- **Compatibilidad con Streams:** Integra bien con las API de Streams de Java, permitiendo operaciones más complejas y funcionales sobre los contenidos de archivos.
- **Operaciones atómicas:** Ofrece métodos para realizar operaciones de E/S de manera atómica, lo que reduce el riesgo de condiciones de carrera en aplicaciones concurrentes.

Tarea: Para todo el código que veamos a continuación, crea ejemplos para que funcionen (basta que meterlos dentro de un método `main()`). Haz pruebas con cada uno de ellos y entiende cómo funcionan.

StandardOpenOption.CREATE y APPEND

Tarea: ¿Qué hace el ejemplo de la clase `EscribirConCreateAppend`?

StandardOpenOption.CREATE_NEW

Tarea: ¿Qué hace el ejemplo de la clase `CrearNuevoConCreateNew`? ¿Qué pasará al ejecutarlo dos veces? Busca más información sobre la palabra reservada `intanceof` (línea 19). Además, fíjate en que aquí controlamos la excepción a la que hacíamos mención hace algunas diapositivas en el punto **Manejo de excepciones**.

StandardOpenOption.DELETE_ON_CLOSE

Tarea: ¿Qué hace el ejemplo de la clase `ArchivoTemporal`? Busca más información sobre el método `createTempFile()` de la clase `Files`.

StandardOpenOption.READ y WRITE

Tarea: ¿Qué hace el ejemplo de la clase `LeerEscribir`? Fíjate en que debemos asegurarnos de que el archivo existe y, que si no es así, entonces lo creemos. Al mismo tiempo, fíjate en que, en realidad, `READ` y `WRITE` están implícitamente actuando en los métodos `readString()` y `writeString()` respectivamente, por eso están comentados en las líneas 22 y 26, porque en realidad no hace falta. En otras palabras; no es necesario indicar un `StandardOpenOption.READ`cuando utilizamos el `readString()`, y lo mismo para cuando queremos escribir.

StandardOpenOption.TRUNCATE_EXISTING

Tarea: ¿Qué hace el ejemplo de la clase `TruncarYEscribir`? ¿Qué ocurre si lo ejecutas varias veces?

Para finalizar, busca información sobre los métodos de la clase `Files`. Mira algunos de los que hemos utilizado o investiga acerca de otros. Fíjate bien en la descripción de cada método, los parámetros que reciben y el tipo de retorno que tienen.

Java

Lectura de archivos

Diferentes formas de leer

Hemos visto diferentes formas de escribir y ahora vamos a ver diferentes formas de leer. En realidad, ya hemos visto cómo leer con `Files`. Adicionalmente, decir que uno de los métodos más utilizados es `readAllLines()`, que ya te puedes imaginar lo que hace (si has hecho las tareas anteriores).

Destacar que, lamentablemente, este método no está optimizado para archivos de gran tamaño.

```
public static void main(String[] args) {
    Path path = Path.of("ejemploFilesLectura.txt");
    try {
        List<String> lines = Files.readAllLines(path);
        for (String line : lines) {
            System.out.println(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Leer con la clase BufferedReader I

La clase `BufferedReader` es eficiente para leer archivos de texto línea por línea en Java gracias a su almacenamiento en búfer. Es recomendable cerrar los recursos una vez utilizados, ya sea de manera explícita o con un `try-catch`.

```
public static void main(String[] args) {
    Path path = Path.of("LecturaBufferedReader.txt");

    try (FileReader fileReader = new FileReader(path.toFile());
            BufferedReader br = new BufferedReader(fileReader)) {

        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Leer con la clase BufferedReader II

Sin embargo, para leer archivos completos de una sola vez, otras clases como `Files` pueden ser más eficientes, ya que `BufferedReader` puede generar sobrecarga al realizar lecturas en buffer. Por lo tanto, es importante considerar el caso de uso específico al elegir el enfoque de lectura más adecuado. Algunos pequeños comentarios sobre lo que hace el ejemplo anterior:

- Interfaz `Path`: Para representar el fichero que se quiere leer.
- Clase `BufferedReader`: Crea un buffer a través del `FileReader`, que permite leer más de un carácter. El constructor recibe el `FileReader` como parámetro.
- Utiliza el método del `BufferedReader` llamada `readLine()`, la cual devuelve la siguiente línea de texto si existe. Si no existe, devuelve `null`.
- Teniendo en cuenta el funcionamiento de `readLine()`, se puede leer todo el fichero utilizando un bucle `while`.

Leer con la clase Scanner I

Qué podríamos decir de la tan famosa clase `Scanner` que no sepamos...

La clase `Scanner` nos proporciona una manera de leer desde archivos pieza por pieza. El `Scanner` funciona separando el contenido de un archivo en piezas utilizando un delimitador, por lo que es ideal para leer archivos con contenido que está separado por algún valor constante. Esto podría ser un archivo común separado por comas, por ejemplo, pero el `Scanner` admite cualquier valor para el delimitador.

Leer con la clase Scanner II

La clase `Scanner` ofrece una forma conveniente de leer datos de entrada, incluidos archivos, y puede ser útil cuando necesitas procesar datos que están estructurados de cierta manera dentro del archivo.

```
public class Main {
    public static void main(String[] args) {
        Path path = Path.of("LecturaScanner.txt");

        try (Scanner scanner = new Scanner(path)) {
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Grado Superior
Acceso a Datos

Ficheros

Curso 2024/25

Profesor
Jonathan Carrero