



Java



Tema 7:

Lectura y escritura de informacion

Ficheros

1



Contenidos

- ❖ Clase Paths
- ❖ Clase Files
- ❖ Ficheros de texto
- ❖ Ficheros binarios
- ❖ Control excepciones

2



Ficheros

- ❑ Un fichero permite almacenar datos en memoria secundaria para utilizarlos en el futuro
- ❑ Un fichero puede ser
 - ❑ de **entrada** (**input**), si lo abrimos y leemos sus datos
 - ❑ de **salida** (**output**), si lo abrimos ó creamos, y escribimos
- ❑ **Tipos** de ficheros según formato de su contenido:
 - ❑ ficheros **binarios**
 - ❑ ficheros **de texto**

3



Clase **Paths** e interface **Path**

Esta clase **Paths** se asocia con la ubicación física (ruta) de un fichero, o directorio.

Se encuentra en el package **java.nio.file**

```
Path archivo = Paths.get("archivo.txt");  
Path archivo = Paths.get("c:\\dir\\archivo.txt");  
Path archivo = Paths.get("/home/dir/archivo.txt");
```

Algunos métodos de la interface **Path** que dan acceso a propiedades del archivo/directorio:

```
String nombre      = archivo.getFileName();  
String rutaAbsoluta = archivo.toAbsolutePath();  
File acceso        = archivo.toFile(); //conversion File  
Path ruta          = acceso.toPath();  //conversion a Path
```



Clase Files

Ofrece métodos estáticos para operaciones con ficheros o directorios representados por objetos **Path**. Se encuentra en el package `java.nio.file`

método	Descripción
boolean exists (Path) boolean notexists (Path)	Verificar si existe o no el fichero
boolean isReadable (Path)	Tiene permisos de lectura
boolean isWritable (Path)	Tiene permisos de escritura
boolean isExecutable (Path)	Tiene permisos de ejecución
long size (Path)	Tamaño en bytes
void delete (Path)	Elimina el fichero o directorio
Path copy (Path origen, Path destino)	Copia fichero o directorio
Stream<Path> list (Path directorio)	coleccion de ficheros/directorios del parámetro Path

5



Clase Files: ejemplos

```
Path fichero = Paths.get("archivo.txt");

Boolean existe    = Files.exists(fichero );
boolean esFichero = Files.isWritable(fichero);
long    size      = Files.size(fichero);

Path nuevo = Paths.get("archivo_bis.txt");
Path copia = Files.copy(fichero, nuevo);

Files.delete(fichero);
```

6



Clase Files: listar contenido de directorio

- ❑ Mediante la clase **Files**, se accede a los path de todos los archivos/directorios de un directorio dado, usando método static **list(Path)**
- ❑ Devuelve un objeto **Stream<Path>**, con esos archivos/directorios en objetos Path

```
Path directorio = Paths.get("c:\\directorio");  
  
Stream<Path> datos = Files.list(directorio)
```

- ❑ Lo recorremos mediante un **Iterator<Path>**

```
Iterator<Path> it = datos.iterator();  
while (it.hasNext()) {  
    Path fichero=it.next();  
    System.out.println(fichero.getFileName());  
}
```



Nueva sintaxis try

- ❑ Nueva sintaxis de try conocida como *"try-with-resources"*.
- ❑ Admite entre () la apertura de un recurso (fichero, socket...) que implemente la interfaz *AutoCloseable*.
- ❑ Este recurso será cerrado al terminar, aunque salte una excepción

Nueva sintaxis de try. Java invoca al método .close() que libera recursos

```
try(Stream<Path> ficheros = Files.list(directorio)){  
    Iterator<Path> it = ficheros.iterator();  
    while (it.hasNext()) {  
        Path fichero=it.next();  
        System.out.println(fichero.getFileName());  
    }  
} catch (IOException ex) {  
    System.out.println("Error al listar directorio");  
}
```



Clase Files: listar contenido de directorio

```
Path dir = Paths.get("c:\\RAQUEL");

try(Stream<Path> ficheros = Files.list(dir)){
    Iterator<Path> it = ficheros.iterator();
    while (it.hasNext()) {
        Path fichero= it.next();
        System.out.println(fichero.getFileName());
    }
} catch (IOException ex) {
    System.out.println("Error al listar directorio");
}
```

9



FICHEROS DE TEXTO

LECTURA

10



Ficheros de texto: lectura

- ❑ Mediante la clase **Files**, se leen todas las líneas del fichero usando método static **lines(Path)**
- ❑ Devuelve objeto **Stream<String>**, con esas líneas.

```
Path archivo = Paths.get("archivo.txt");  
Stream<String> datos = Files.lines(archivo);
```

- ❑ Lo recorreremos mediante un **Iterator<>**

```
Iterator<String> it = datos.iterator();  
while (it.hasNext()) {  
    String linea= it.next();  
    System.out.println(linea);  
}
```

11



Ficheros de texto: lectura

- ❑ Nueva sintaxis de try conocida como *"try-with-resources"*.
- ❑ Admite entre () la apertura de un recurso (fichero, socket...) que implemente la interfaz *AutoCloseable*.
- ❑ Este recurso será cerrado al terminar, aunque salte una excepción

Nueva sintaxis de try. Java invoca al método .close() que libera recursos

```
Path archivo = Paths.get("archivo.txt");  
try (Stream<String> datos = Files.lines(archivo)) {  
    Iterator<String> it = datos.iterator();  
    while (it.hasNext()) {  
        String linea = it.next();  
        System.out.println(linea);  
    }  
} catch (IOException ex) {  
    System.out.println("Error en la lectura");  
}
```

12



Ficheros de texto: lectura

- ❑ Podemos indicar el conjunto de caracteres ([Charset](#)) que debe reconocer, mediante el segundo parámetro del método `lines()`
- ❑ La constante `StandardCharsets.ISO_8859_1` representa al alfabeto latino (reconoce letra ñ, acentos ...)

```
Stream<String> datos  
= Files.lines(archivo, StandardCharsets.ISO_8859_1);
```

13



FICHEROS DE TEXTO: ESCRITURA

14



Ficheros de texto: clase **BufferedWriter**

- ❑ Se crea usando método **newBufferedWriter()** de la clase **Files**

```
Path fichero = Paths.get("archivo.txt");  
BufferedWriter bw = Files.newBufferedWriter(fichero);
```

15



Ficheros de texto: clase **BufferedWriter**

- ❑ Permite otro parámetro, para el **conjunto de caracteres** que reconoce:
 - ❑ **StandardCharsets.ISO_8859_1**. Representa al alfabeto latino (incluye letra ñ y los acentos).

```
BufferedWriter bw = Files.newBufferedWriter(fichero,  
                                           StandardCharsets.ISO_8859_1);
```

16



Ficheros de texto: clase `BufferedWriter`

- ❑ Permite otro parámetro, para el **comportamiento ante la existencia o no del archivo**
 - ❑ `StandardOpenOption.CREATE`. Si no existe lo crea
 - ❑ `StandardOpenOption.CREATE_NEW`, salta error si ya existia
 - ❑ `StandardOpenOption.APPEND`, añade al final
 - ❑ `StandardOpenOption.TRUNCATE_EXISTING`, debe existir y lo vacía antes de escribir
- ❑ Se pueden incluir varias opciones

```
BufferedWriter bw = Files.newBufferedWriter(fichero,  
                                           StandardOpenOption.CREATE,  
                                           StandardOpenOption.APPEND);
```

17



Ficheros de texto: ejemplo

```
Path archivo = Paths.get("Datos.txt");
```

```
try (BufferedWriter out =  
    Files.newBufferedWriter(archivo,  
                             StandardCharsets.ISO_8859_1,  
                             StandardOpenOption.CREATE)) {  
    out.write("línea 1");  
    out.newLine(); //salto de línea  
    out.write("línea 2");  
    out.newLine();  
} catch (IOException e) {  
    System.out.println("Error al abrir");  
}
```

18



FICHEROS BINARIOS

19



Ficheros binarios

- ☐ Los ficheros binarios guardan la información no en modo texto sino en código binario.
- ☐ Mediante archivos binarios podemos guardar directamente un objeto sin necesidad de darle formato texto.
- ☐ Para que un objeto pueda ser introducido en un archivo binario la clase debe implementar la **interface Serializable** (que no define ningún método)

20



FICHEROS BINARIOS

ESCRITURA

21



Ficheros binarios: clase **OutputStream**

- ❑ Se crea usando método **newOutputStream()** de la clase **Files**. Crea un archivo, y devuelve un Stream para poder escribir en el archivo

```
OutputStream fo = Files.newOutputStream("datos.dat");
```

- ❑ Añade a un archivo que existe

```
OutputStream fo =  
    Files.newOutputStream("datos.dat", APPEND);
```

- ❑ Añade a un archivo, lo crea si no existe

```
OutputStream fo =  
    Files.newOutputStream("datos.dat", CREATE, APPEND);
```

22



Ficheros binarios: clase **ObjectOutputStream**

- ❑ Se crea a partir de un OutputStream

```
ObjectOutputStream salida =  
    new ObjectOutputStream(fo);
```

- ❑ La clase debe implementar la interfaz **Serializable**

```
public class Empleado implements Serializable{  
    .....}
```

- ❑ Métodos para escribir/guardar objetos en el archivo

```
Empleado emp = new Empleado("Pepe", 50);  
salida.writeObject(emp);           //guarda objeto emp  
  
salida.writeObject(new LocalDate()); //guarda objeto fecha  
salida.writeObject("edad");         //guarda objeto String  
salida.writeInt(555555);             //guarda un int
```

23



FICHEROS BINARIOS

LECTURA



Ficheros binarios: clase **InputStream**

- ❑ Lee, byte a byte, del archivo pasado por parámetro

```
InputStream flectura =  
    Files.newInputStream("datos.dat");  
  
int c = flectura.read();
```


25



Ficheros binarios: clase **ObjectInputStream**

- ❑ Se crea a partir de un InputStream

```
ObjectInputStream salida =  
    new ObjectInputStream(flectura);
```



- ❑ Tiene métodos para leer objetos del archivo binario. Hay que hacer un **casting** (conversión) de lo leído. Debemos conocer que hay escrito en el archivo

```
Empleado emp = (Empleado) salida.readObject();  
LocalDate fecha = (LocalDate) salida.readObject();
```

26



Ficheros binarios: ejemplo

```
//código en Coche.java
import java.io.Serializable
Public class Coche implements Serializable{ ...
*****
try ( ObjectInputStream entrada = new ObjectInputStream(
        Files.newInputStream("objetos.dat"));
    ObjectOutputStream salida = new ObjectOutputStream(
        Files.newOutputStream("objetos.dat")) )
{
    salida.writeObject(new Coche("4444-FDR", "Seat Ibiza"));
    salida.writeObject(new Coche("8888-HHH", "FORD FIESTA"));
    Coche c1 = (Coche) entrada.readObject();
    Coche c2 = (Coche) entrada.readObject();
    System.out.println(c1);
    System.out.println(c2);

} catch (IOException ex) {
    System.out.println("Excepcion de IO" + ex.getMessage());
}
}
```

27



Control de excepciones

- ❑ En la apertura y cierre de ficheros pueden suceder situaciones anómalas. Para controlarlas se deben recoger las **excepciones de tipo checked con try-catch**. En los ejemplos del proyecto **EjemploArchivos** se pueden observar.

28