JAVA.NIO

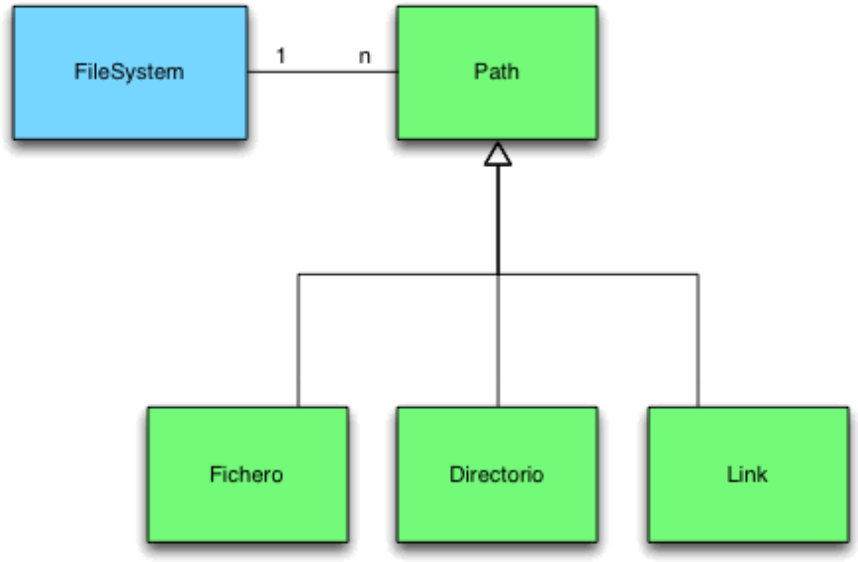
Java NIO.

El concepto de FileSystem define un sistema de ficheros completo. Mientras que por otro lado el concepto de Path hace referencia a un directorio, fichero o link que tengamos dentro de nuestro sistema de ficheros.

- Java.nio define interfaces y clases para que la máquina virtual Java tenga acceso a archivos, atributos de archivos y sistemas de archivos. Aunque dicho API comprende numerosas clases, sólo existen unas pocas de ellas que sirven de puntos de entrada al API, lo que simplifica considerablemente su manejo.

- Java NIO: canales y búferes

- En la API de IO estándar, trabajas con secuencias de bytes y secuencias de caracteres. En NIO, trabaja con **canales y búferes**. Los datos siempre se leen de un canal a un búfer, o se escriben desde un búfer a un canal.
- Java NIO contiene el concepto de "selectores". Un selector es un objeto que puede gestionar múltiples canales para eventos (como: conexión abierta, datos recibidos, etc.). Un Selector permite que un solo hilo maneje múltiples canales.



Java NIO Path

La interfaz java.nio.file.Path representa un path y las clases que implementen esta interfaz puede utilizarse para localizar ficheros en el sistema de ficheros .
Una ruta puede señalar a un archivo o un directorio. Un camino puede ser absoluto o relativo. Una ruta absoluta contiene la ruta completa desde la raíz del sistema de archivos hasta el archivo o directorio al que apunta. Una ruta relativa contiene la ruta al archivo o directorio relativo a alguna otra ruta.

La forma más sencilla de construir un objeto que cumpla la interfaz Path es a partir de la clase java.nio.file.Paths, que tiene métodos estáticos que retornan objetos Path a partir de una representación tipo String del path deseado, por ejemplo:

Path p = Paths.get("/home/ad/mi_fichero");
Por supuesto, no es necesario que los ficheros existan de verdad en el disco duro para que se puedan crear los objetos Path correspondientes: La representación y manejo de paths en Java no está restringida por la existencia de esos ficheros o directorios en el sistema de ficheros.

El interfaz Path declara numerosos métodos que resultan muy útiles para el manejo de paths, como por ejemplo, obtener el nombre corto de un fichero, obtener el directorio que lo contiene, resolver paths relativos, etc.

Una instancia de tipo Path refleja el sistema nombrado del sistema operativo subyacente, por lo que objetos path de diferentes sistemas operativos no pueden ser comparados fácilmente entre sí.

- Operaciones con Path
- Recuperar partes de una ruta
- Eliminar redundancias de una ruta
- Convertir una ruta
- Unir dos rutas
- Crear una ruta relativa a otra dada
- Comparar dos rutas

```
import java.nio.file.Path;

import java.nio.file.Paths;

public class PathEjamplo {

public static void main(String args[]) {

   Path path = Paths.get("C:/Users/alumno/PathEjemplo");

   System.out.println(" path = " + path);

   System.out.println(" is absoute ? = " + path.isAbsolute());
```

```
    System.out.println(" file short name = " + path.getFileName());

    System.out.println(" parent = " + path.getParent());

    System.out.println(" uri = " + path.toUri());

    path = Paths.get("/home/PathEjemplo");

    }

}
```

Clases Java NIO FILES

La clase **java.nio.file.Files** es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite **manejar ficheros reales del disco desde Java**.
Esta clase tiene métodos estáticos para el manejo de ficheros, los métodos de la clase Files trabajan sobre objetos Path.
Las operaciones principales a realizar con archivos y directorios son:
Verificación de existencia y accesibilidad

- Borrar un archivo o directorio
- Copiar un archivo o directorio
- Mover un archivo o directorio

Ejemplo 1: Existencia y comprobación de permisos

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
public class FileEjemplo {
public static void main(String args[]) {
    Path path = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
    System.out.println(" path = " + path);
    System.out.println(" exists = " + Files.exists(path));
    System.out.println(" readable = " + Files.isReadable(path));
    System.out.println(" writeable = " + Files.isWritable(path));
    System.out.println(" executeable = " + Files.isExecutable(path))
```

Ejemplo 2: Creación y borrado de directorios

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
```

// Crea un nuevo fichero o directorio o lo borra si ya existe

```
public class FileEjemplo2 {
public static void main(String args[]) {
   Path path = Paths.get("C:\\Users\\alumno\\prueba");
   try {
   if (Files.exists(path))
       Files.delete(path);
       else
       Files.createFile(path);
    } catch (IOException e) {
    System.err.println(e);
     System.exit(1);
   }
}
}
```

El método **delete(Path)** borra el fichero o directorio o lanza una excepción si el borrado falla. El siguiente ejemplo muestra cómo capturar y gestionar las excepciones que pueden producirse en el borrado Si el fichero o directorio no existe la excepción que se produce es **NoSuchFileException**. Los sucesivos catch permiten determinar por qué ha fallado el borrado:

```
try {
   Files.delete(path);
} catch (NoSuchFileException x) {
   System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
   System.err.format("%s not empty%n", path);
} catch (IOException x) {
   // File permission problems are caught here.
   System.err.println(x);
}
```

El **metodo deleteIfExists(Path)** también borra el fichero o directorio, pero no lanza ningún error en caso de que el fichero o directorio no exista.

Ejemplo 3 crear un directorio

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.io.IOException;
// crea una nueva carpeta
public class FileEjemplo4
{
        public static void main(String args[]) {
        Path path = Paths.get("C:\\Users\\alumno\\newdir");
        try {
            Path newDir = Files.createDirectory(path);
```

```
            } catch(FileAlreadyExistsException e){
   // el directorio ya existe
} catch (IOException e) {
   //error I/O
           e.printStackTrace();
}
}
}
```

Ejemplo 4: copiar directorios

Se puede copiar un archivo o directorio usando el método copy(Path, Path, CopyOption...). La copia falla si el archivo de destino existe, a menos que se especifique la opción. **REPLACE_EXISTING.**

Se puede copiar directorios Aunque, los archivos dentro del directorio no se copian, por lo que el nuevo directorio está vacío incluso cuando el directorio original contiene archivos.

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.io.IOException;
//import java.nio.file.FileAlreadyExistsException; //en caso de querer sobrescribir el directorio
destino
public class FileEjemplo5 {
public static void main(String args[]) {
   Path sourcePath = Paths.get("C:\\Users\\alumno\\origen");
   Path destinationPath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino");
   try {
   Files.copy(sourcePath, destinationPath);
                //Files.copy(sourcePath, destinationPath,
StandardCopyOption.REPLACE_EXISTING);
   } catch (FileAlreadyExistsException e) {
   System.out.println("el fichero existe");
    } catch (IOException e) {
    e.printStackTrace();
     }
}
}
```

Ejemplo 5: Copiar ficheros

```
import java.io.IOException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class FileEjemplo7 {
public static void main(String args[]) {
    Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
    Path destinationPath =Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");
    try {
       Files.copy(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
      } catch (FileAlreadyExistsException e) {
       System.out.println("el destino existe");
     } catch (IOException e) {
     e.printStackTrace();
  }
 }
}
```

Ejemplo 6: Mover ficheros y directorios, cambiando el nombre

```
import java.io.IOException;
import java.nio.file.FileAlreadyExistsException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
public class FileEjemplo7 {
public static void main(String args[]) {
Path sourcePath = Paths.get("C:\\Users\\alumno\\FileEjamplo\\hola.txt");
Path destinationPath =
Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\OtroNombre.txt");
try {
Files.move(sourcePath, destinationPath, StandardCopyOption.REPLACE_EXISTING);
} catch (FileAlreadyExistsException e) {
System.out.println("el destino existe");
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Modos de acceso, el parámetro OpenOptions

A la hora de utilizar un fichero en Java se puede restringir el acceso que tenemos al mismo desde el propio lenguaje, haciendo más estrictos los permisos de acceso que dicho fichero ya tiene en el sistema de ficheros.

Por ejemplo, si el usuario tiene permisos de lectura y escritura sobre un fichero, un programa Java que solo quiera leerlo puede abrir el fichero solo en modo lectura, lo que ayudará a evitar bugs desde el propio lenguaje.
A tal efecto, en java se definen una serie de modos de acceso a un fichero a través del parámetro **OpenOptions**. La forma más cómoda de utilizar este parámetro es a través del enum StandardOpenOptions que puede tomar los siguientes valores (hay más):

- WRITE: habilita la escritura en el fichero
- APPEND: todo lo escrito al fichero se hará al final del mismo
- CREATE_NEW: crea un fichero nuevo y lanza una excepción si ya existía
- CREATE: crea el fichero si no existe y simplemente lo abre si ya existía
- TRUNCATE_EXISTING: si el fichero existe, y tiene contenido, se ignora su contenido para sobreescribirlo desde el principio.

Los métodos que se muestran en las siguientes ejemplos utilizan este parámetro, en la descripción de cada método en el API se explica cual es el comportamiento por defecto en caso de no utilizarse este parámetro.

Escritura desde arrays de bytes

La escritura a ficheros mediante arrays es la forma más sencilla (y limitada) de escritura de ficheros, y se realiza mediante el método java.nio.file.Files.write().

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
public class FileEjemplo10 {
   public static void main(String args[]) {
      Path inputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt");
      Path outputFile = Paths.get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt");
      try {
         byte[] contents = Files.readAllBytes(inputFile);
         Files.write(outputFile, contents, StandardOpenOption.WRITE,
         StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
     } catch (IOException e) {
         System.err.println(" ERROR : " + e);
         System.exit(1);
    }
  }
}
```

Escritura desde buffers

La escritura desde buffers resulta mucho más eficiente que utilizando arrays de bytes para ficheros grandes.
El siguiente programa Java copia ficheros, accediendo al fichero original una vez por línea y escribiendo en el fichero destino una línea cada vez. Utiliza las clases de java.io: BufferedReader y BufferedWriter:

```
import java .nio. file. Path;
import java .nio. file. Paths ;
import java .nio. file. Files ;
import java .io. IOException;
import java .nio. charset . Charset ;
import java .io. BufferedReader;
import java .io. BufferedWriter;
import java .nio. file. StandardOpenOption;
public class FileEjemplo11 {
// Copy a file
public static void main( String args []) {
Path input = Paths . get( "C:\\Users\\alumno\\FileEjamplo\\origen\\hola.txt") ;
Path output = Paths . get("C:\\Users\\alumno\\FileEjamplo\\destino\\hola.txt") ;
try {
      BufferedReader inputReader = Files . newBufferedReader(input , Charset .
      defaultCharset());
       BufferedWriter outputWriter = Files . newBufferedWriter(output , Charset .
       defaultCharset() ,  StandardOpenOption. WRITE , StandardOpenOption.
       CREATE ,  StandardOpenOption. TRUNCATE_EXISTING);
       String line;
       while ( ( line = inputReader. readLine ()) != null ) {
            outputWriter. write (line , 0, line. length ());
            outputWriter. newLine () ;
       }
       inputReader. close ();
       outputWriter. close ();
       } catch ( IOException e) {
       System .err. println (" ERROR : " + e);
       System . exit (1);
      }
   }
}
```